ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Ingeniería Informática + ADE

Universidad de Granada (UGR)

Autor: Ismael Sallami Moreno

Asignatura: Sistemas Concurrentes y Distribuidos

Índice

1 Teoria

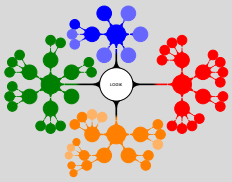
1.1. Tema 1

Tema 1
Sistemas Concurrentes y Distribuidos
Introducción

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 20 septiembre 2024

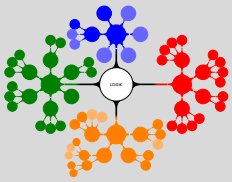
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



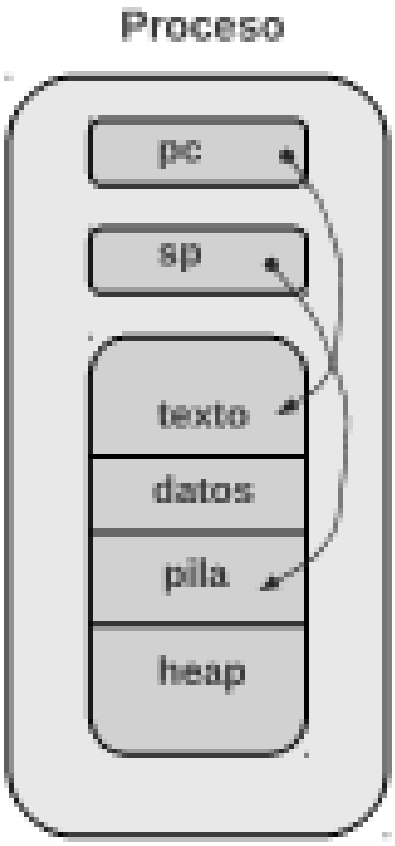
Concurrencia: programa y programación concurrente

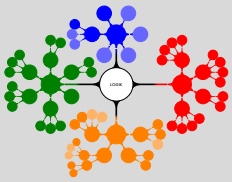
Programa Concurrente

1. Programa secuencial
   - Ejecución lineal de instrucciones.
2. Programa Concurrente
   - Múltiples unidades de ejecución independientes (llamadas procesos).
   - Los procesos cooperan para realizar tareas esenciales.
3. Proceso
   - Definición: Entidad de software abstracta, dinámica y activa.
   - Estado: No sólo las instrucciones; incluye su estado actual y la capacidad de interactuar con el medio ambiente.
4. Estado del proceso
   - Valores en registros (Contador de Programa - PC, Puntero de Pila - SP, Memoria Heap).
   - Acceso a recursos como archivos y dispositivos.
   - El estado debe estar protegido de otros procesos concurrentes.
5. Gestión de la concurrencia
   - Esencial para evitar el acceso incontrolado entre procesos.



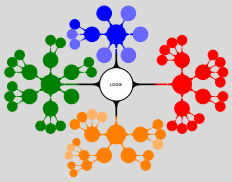
Proceso





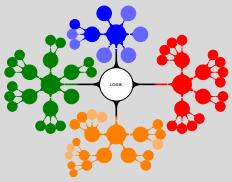
Conceptos básicos de procesos y concurrencia

1. Estructura de memoria del proceso
   - Dividido en zonas:
     - Instrucciones: Secuencia a ejecutar.
     - Área de datos: Para variables globales/estáticas.
     - Pila: Para variables locales y parámetros de procedimientos.
     - Memoria Heap: Variables dinámicas no estáticas.
2. Programas Concurrentes vs Secuenciales
   - Concurrencia: Los flujos de control se intercalan en menos núcleos.
   - Preserva el paralelismo lógico independientemente del número de núcleos.
   - Mejora la eficiencia al permitir la ejecución simultánea de múltiples hilos de control.
   - Reduce los retrasos en los cálculos causados por las operaciones de E/S.
3. Concurrencia en Simulación
   - Simula sistemas del mundo real de forma más natural.
   - Las actividades del mundo real pueden modelarse mejor con procesos concurrentes independientes.
4. Definición de concurrencia
   - El potencial de paralelismo en el código, independientemente de las limitaciones de hardware (número de núcleos/procesadores).



Conceptos clave de la programación concurrente

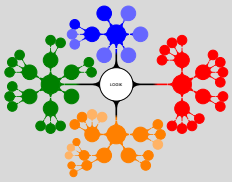
1. Modelo Abstracto de Computación
   - Expresa el paralelismo potencial a un alto nivel de abstracción.
   - Independiente de la arquitectura hardware del ordenador.
2. Concurrencia en la programación
   - Su objetivo es simplificar la sincronización y la comunicación entre los procesos.
   - Permite que el código se ejecute en diferentes arquitecturas.
   - Fomenta programas portables e independientes de la arquitectura.
3. Beneficios del modelo abstracto
   - Herramientas para diseñar y razonar sobre la concurrencia.
   - Simplifica el lenguaje de programación con primitivas de sincronización/comunicación de alto nivel.
   - Evita llamadas al sistema de bajo nivel o instrucciones específicas de la máquina.
4. Cinco axiomas de la programación concurrente
   - (i)Atomicidad e Intercalación de Instrucciones
   - (ii)Consistencia de datos después del acceso concurrente
   - (iii)Irrepetibilidad de las secuencias de instrucciones
   - (iv)Independencia de la velocidad de proceso
   - (v)Hipótesis de progreso finito



(i)Atomicidad e intercalación de instrucciones

Concepto de Sentencia atómica(indivisible)

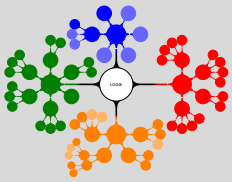
- Atomicidad: Cada instrucción de máquina o ensamblador se ejecuta hasta su finalización sin interrupción, lo que garantiza la coherencia en la ejecución concurrente.
- Concurrencia: Los programas escritos para múltiples procesos (paralelos o de tiempo compartido) producen conjuntos de instrucciones atómicas intercaladas.
- Entrelazamiento: Las secuencias intercaladas de instrucciones de dos procesos (por ejemplo, I1x y I2x) definen el comportamiento observable del programa.
- No determinismo: La secuencia de instrucciones intercaladas es impredecible, lo que pone de relieve el no determinismo en la programación concurrente.
- Independencia del hardware: Este modelo de concurrencia se aplica a nivel lógico, independientemente del hardware que ejecute el programa.



Ejemplos de sentencias atómicas

Como ejemplo de instrucciones atómicas podemos citar muchas de las instrucciones máquina del repertorio de un procesador, por ejemplo las tres siguientes:

- Leer una celda de memoria (variable de tipo simple **x**) y cargar su valor en un registro (**r**) del procesador : `LOAD x`
- Incrementar el valor de un registro (**r**) u otras operaciones aritméticas sobre registros del procesador: `ADD r, a`
- Escribir el valor de un registro (**r**) en una celda de memoria: `STORE x`



Sentencias no atómicas

La mayoría de las sentencias de los lenguajes de programación de alto nivel son típicamente no atómicas, por ejemplo, la sentencia:

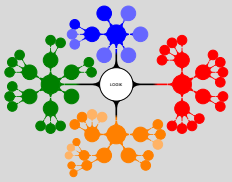
`x:= x + 1;` (incrementa el valor de la variable entera en 1 unidad)

El compilador utiliza una secuencia de tres sentencias:

1. Leer el valor de **x** y cargarlo en un registro **r** del procesador
2. Incrementar en una unidad el valor almacenado en el registro **r**
3. Escribir el valor del regsitro **r** en la variable **x**

El valor que toma la variable **x** justo al terminar la ejecución de la sentencia dependerá de que haya o no otras sentencias ejecutándose a la vez y tratando de escribir simultáneamente en la variable **x**

Decimos en este caso que existe *indeterminación*: no se puede predecir el estado final del proceso a partir de su estado inicial



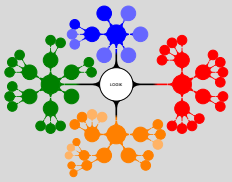
(i) Entrelazamiento de instrucciones atómicas

Programa concurrente C compuesto por 2 procesos secuenciales: P_A y P_B que se ejecutan a la vez

La ejecución de C puede dar lugar a cualquiera de los posibles entrelazamientos de sentencias atómicas de P_A y P_B:

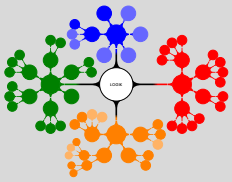
Pr.	Posibles secuencias de instrucciones atómicas
P_A	$A_1\ A_2\ A_3\ A_4\ A_5$
P_B	$B_1\ B_2\ B_3\ B_4\ B_5$
C	$A_1\ A_2\ A_3\ A_4\ A_5\ B_1\ B_2\ B_3\ B_4\ B_5$
C	$B_1\ B_2\ B_3\ B_4\ B_5\ A_1\ A_2\ A_3\ A_4\ A_5$
C	$A_1\ B_1\ A_2\ B_2\ \ldots$
C	$B_1\ B_2\ A_1\ B_3\ B_4\ A_2\ \ldots$
C	$\ldots$

Las secuencia se va ordenando a medida que acaba cada una de las sentencias atómicas que la componen (que es cuando tienen efecto)



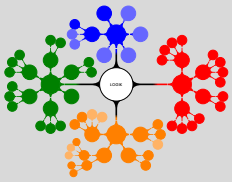
Modelo de programación basado en el entrelazamiento de instrucciones atómicas

- Este modelo es una abstracción que estudia las secuencias de ejecución de procesos concurrentes.
- Solo se consideran las características relevantes para el resultado del programa.
- Detalles ignorados:
  - Estado de memoria asignado
  - Registros de cada proceso
  - Costos de cambios de contexto
  - Políticas de planificación
  - Diferencias de velocidad en multiprocesadores y monoprocesadores



(ii) Consistencia de los datos tras el acceso simultáneo

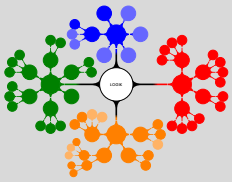
- Instrucciones atómicas: Cuando dos instrucciones atómicas acceden a la misma dirección de memoria, el resultado debe ser consistente, independientemente de si se ejecutan en paralelismo real o o intercambiándolas en un orden impredecible.
- Consistencia de datos: Después de que ambos procesos terminen, la memoria debe permanecer en un estado válido según el tipo de datos de la variable, asegurando que no haya corrupción de datos.
- Resultados impredecibles: El valor final de una variable compartida no es predecible pero permanece consistente con su rango definido.
- Soporte de Hardware: La consistencia de la memoria está garantizada por el árbitro del bus de memoria del sistema, evitando la corrupción de datos por accesos simultáneos.



(iii) Irrepetibilidad de las secuencias de instrucciones

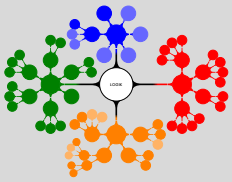
- Trazas: Los programas concurrentes generan un gran número de posibles secuencias de entrelazamiento de instrucciones, por lo que es poco probable que dos ejecuciones sigan el mismo camino.
- Errores transitorios: Estos errores aparecen en algunas trazas pero no en otras, lo que dificulta su detección y corrección.
- Reto de depuración: La imprevisibilidad de las secuencias de ejecución complica la depuración y el análisis de corrección en programas concurrentes.
- Solución: Los métodos formales basados en la lógica matemática son necesarios para verificar la corrección de los programas y eliminar los errores transitorios en el software concurrente.

  Historia o traza de un programa concurrente: Secuencia de estados $s_0 \rightarrow s_1 \rightarrow \ \ldots \rightarrow s_n$, producida por una secuencia concreta de entrelazamiento



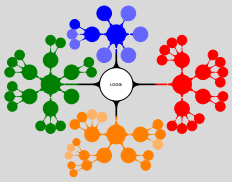
(iv) Independencia de la velocidad del proceso

- Independencia de la velocidad de ejecución: La corrección en programas concurrentes no debe depender de la velocidad relativa de los procesos. A evitar:
  - Falta de Portabilidad: Los programas pueden fallar en diferentes plataformas si se hacen suposiciones sobre la velocidad de ejecución.
  - Condiciones de carrera: El acceso a variables compartidas puede producir resultados impredecibles e incorrectos basados en el orden y la velocidad de ejecución.
- Ejemplo de condición de carrera: Dos procesos (P1 y P2) modificando la misma variable pueden producir resultados impredecibles dependiendo del orden de ejecución del proceso.
- Excepción de los sistemas en tiempo real: En las aplicaciones de tiempo real, la velocidad y el orden de ejecución de los procesos son críticos, y se asignan niveles de prioridad a los procesos.
- Suposición de tiempo finito: Todos los procesos deben completarse en un tiempo finito para garantizar que se mantienen las propiedades de corrección, como la propiedad de *vivacidad* de los procesos.



(v) Hipótesis del progreso finito

- Progreso global: Si al menos un proceso está listo para ejecutarse, eventualmente se le debe permitir ejecutarse, asegurando que el programa no entre en punto muerto o interbloqueo (*deadlock*).
- Progreso local: Una vez que un proceso comienza a ejecutar una sección de código, debe terminar dicha sección.
- Hipótesis de progreso finito: Ningún proceso debe detenerse indefinidamente debido a condiciones internas (como valores en contadores o registros) durante la ejecución del programa. Todo proceso debe seguir progresando durante la ejecución del programa.



Notaciones para expresar ejecución concurrente

Propuestas iniciales: no separan la definición de los procesos de su sincronización

Propuestas posteriores: separan ambos conceptos e imponen una estructura al programa concurrente, diferente de uno secuencial

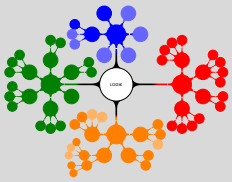
Declaración de procesos: rutinas específicas de programación concurrente $\Rightarrow$ Estructura del programa concurrente más clara

Sistemas Estáticos:

- Número de procesos fijado en el fuente del programa
- Los procesos se activan al lanzar el programa
- Ejemplo: Message Passing Interface (MPI-1)

Sistemas Dinámicos:

- Número variable de procesos/hebras que se pueden activar en cualquier momento de la ejecución
- Ejemplos: OpenMP, PThreads, Java Threads, MPI-2

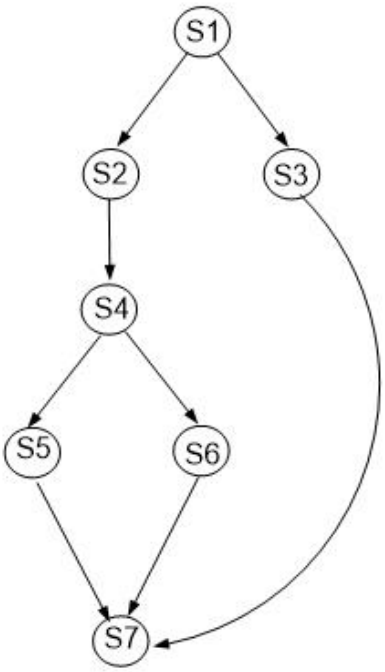


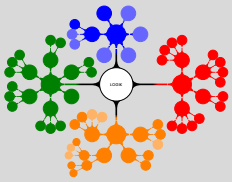
Grafo de sincronización

El Grafo de Sincronización es un Grafo Dirigido Acíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa (actividad):

Dadas dos actividades, S_1 y S_2, una arista (flecha) desde S_1 hacia S_2 significa que S_2 no puede comenzar su ejecución hasta que S_2 haya finalizado

Muestra las restricciones de precedencia que determinan cuándo una actividad puede empezar en un programa





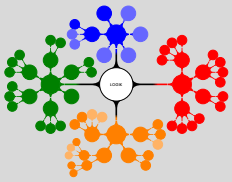
Definición estática de procesos

```
var ... \\variables compartidas

process Uno;
var ... \\variables locales
begin
  ... \\codigo
end;
process Dos;
var ... \\variables locales
begin
  ... \\codigo
end;
... \\otros procesos
```

Se *lanza* la ejecución concurrente de ambos procesos en 1 bloque del programa:

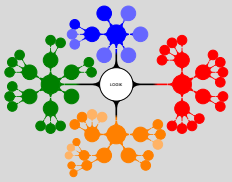
```
cobegin  Uno || Dos coend;
```

El programa acaba cuando acaban de ejecutarse todas las instrucciones de 2 los procesos. Las variables compartidas se inicializan antes de comenzar la ejecución concurrente de los procesos.



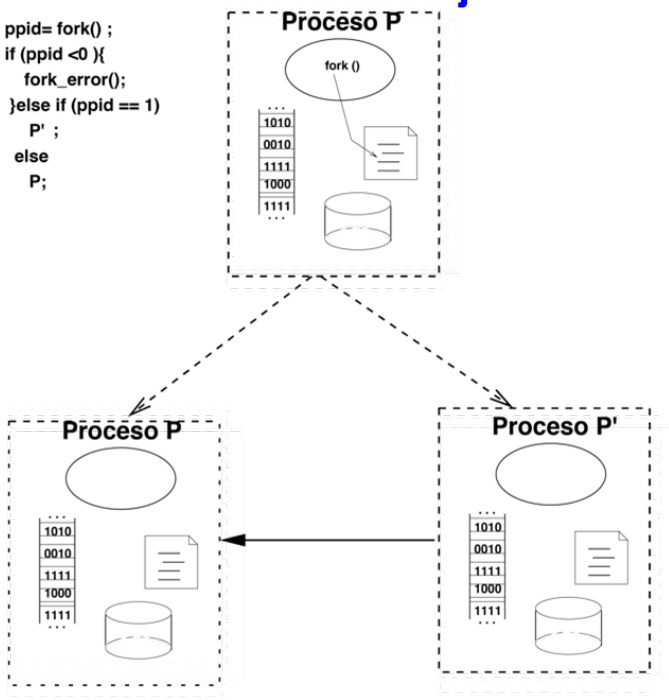
Definición estática de vectores de procesos

```
var ... \\variables compartidas

process NomP[ind : a.. b];
var ... \\variables locales
begin
  ... \\codigo
  ...\\ (ind vale a, a+1, ... b)
end;
... \\otros procesos
```

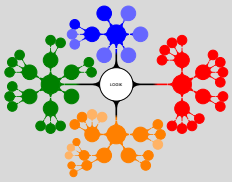
Creación de procesos no estructurada: fork-join

```
process P2;
 begin
  D;
 end;
process P1;
 begin
  A;
  fork P2;
  B;
  join P2;
  C;
 end;
```



fork: sentencia que especifica que la rutina nombrada puede comenzar su ejecución, al mismo tiempo que comienza la sentencia siguiente (bifurcación)

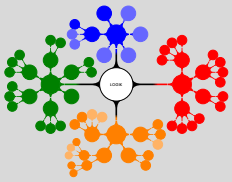
join: sentencia que espera la terminación de la rutina nombrada, antes de comenzar la sentencia siguiente (unión)



Introducción a los conceptos de *exclusión mutua* y *sincronización*

No todas las secuencias de entrelazamiento de las instrucciones de los procesos a que da lugar la ejecución de un programa concurrente son aceptables

- Los procesos no suelen ejecutarse de una forma totalmente independiente, sino que que **colaboran** entre ellos
- **Condición de sincronización:** restricción en el orden en que se pueden entremezclar las instrucciones que generan los procesos de un programa
- **Exclusión mutua:** se da en secuencias finitas de intrucciones del código de un programa, que han de ejecutarse de principio a fin por un único proceso, que no puede ser desplazado del procesador mientras ejecuta esta sección crítica de instrucciones

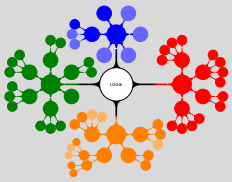


Condición de sincronización

En un programa concurrente, una condición de sincronización establece para asegurarnos de que todas las trazas del programa son correctas

- Suele ocurrir cuando, en un punto concreto de su ejecución, uno o varios procesos deben esperar a que se cumpla una determinada condición global (depende de varios procesos)

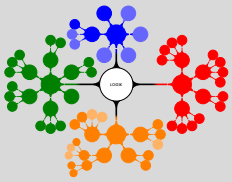
Un ejemplo sencillo de **condición de sincronización**: en el ejemplo del productor-consumidor con 1 variable compartida, el productor espera a que el consumidor haya leído el valor escrito en la variable antes de volver a escribirla.



El paradigma del *Productor Consumidor*

Dos procesos cooperantes en los cuales uno de ellos (*productor*) genera una secuencia de valores (p.ej.enteros) y el otro (*consumidor*) utiliza cada uno de estos valores

```
var x : integer ; { contiene cada valor producido }
{ Proceso productor: calcula 'x' }
process Productor ;
var a : integer ; { no compartida }
begin
 while true do begin
 { calcular un valor }
  a := ProducirValor() ;
  { escribir en mem. compartida }
  x := a ; { sentencia E }
 end
end
process Consumidor ; { Proceso Consumidor: lee 'x' }
var b : integer ; { no compartida }
begin
 while true do begin
 { leer de mem. compartida }
  b := x ; { sentencia L }
  { utilizar el valor leido }
  UsarValor(b) ;
 end
end
```

Trazas incorrectas de un programa concurrente

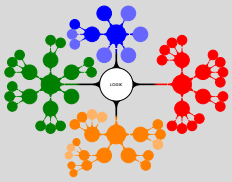
Los procesos sólo *funcionan* como se espera si el orden en el que se entrelazan las sentencias elementales etiquetadas como E (escritura) y L (lectura) es: E, L, E, L, E, L, ...

Trazas incorrecetas del programa *Productor Consumidor*:

- **L, E, L, E,** . . . es incorrecta: se hace una lectura de **x** previa a cualquier escritura (se lee valor indeterminado)
- **E, L, E, E, L,** . . . es incorrecta: hay dos escrituras sin ninguna lectura entre ellas (se produce un valor que no se lee)
- **E, L, L, E, L,** . . . es incorrecta (para el código del P/C anterior): hay dos lecturas de un mismo valor que, por tanto, es usado dos veces

La secuencia válida asegura la condición de sincronización:

- *Consumidor* no lee hasta que *Productor* escribe un nuevo valor en x (cada valor producido es usado una sola vez)
- *Productor* no escribe un nuevo valor hasta que *Consumidor* lea el último valor almacenado en **x** (ningún valor producido se pierde)

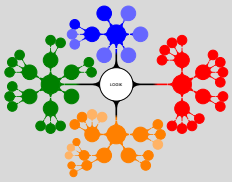


Instrucciones compuestas e instrucciones atómicas

En pseudo-código, podemos convertir las sentencias compuestas a atómicas: indicando que se deben de ejecutar sin ser interrumpidas, usando los caracteres $<$ y $>$:

```
{ instr. compuestas (no atómicas) }
begin
x := 0 ;
cobegin
x:= x+1 ;
x:= x-1 ;
coend
end
////////////////////////////
{ instr. atómicas }
begin
x := 0 ;
cobegin
< x:= x+1 > ;
< x:= x-1 > ;
coend
end
```

- En el primer código, al acabar, **x** puede tener un valor cualquiera del conjunto $\{-1, 0, 1\}$
- En el segundo código **x** finaliza siempre con el valor 0

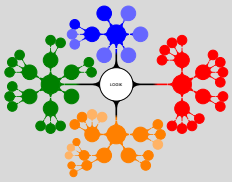


Concepto de corrección de un programa concurrente

Propiedad de un programa concurrente: Algo que se puede afirmar del programa que es cierto para todas las posibles trazas del programa

Hay 2 tipos de propiedades:

- Propiedad de seguridad (safety).
- Propiedad de vivacidad (liveness).



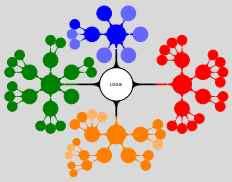
Propiedad de *seguridad*

Condiciones que deben cumplirse en cada instante de la traza del programa; son del tipo: *nunca pasará nada malo*

- Requeridas en especificaciones estáticas del programa
- Son fáciles de demostrar y para cumplirlas se suelen impedir determinadas posibles trazas

Ejemplos:

- *Exclusión mutua*: 2 procesos nunca entrelazan ciertas subsecuencias de operaciones
- *Ausencia Interbloqueo* (Deadlock-freedom): Nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá
- *Propiedad de seguridad en el Productor-Consumidor*: El consumidor debe consumir todos los datos producidos por el productor en el orden en que se van produciendo



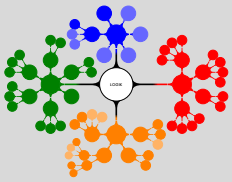
Propiedades de *vivacidad*

Son propiedades que deben cumplirse en algún momento futuro y no especificado del programa; son del tipo: *realmente sucede (o va a suceder) algo bueno*

- Son propiedades dinámicas, más difíciles de demostrar que las propiedades de seguridad

Ejemplos:

- *Ausencia de inanición* (starvation-freedom): Un proceso o grupo de procesos del programa no puede ser indefinidamente pospuesto. Asegura que, en algún momento, podrá avanzar
- *Equidad* (fairness): Tipo particular de propiedad de vivacidad. Un proceso que pueda progresar debe hacerlo con justicia relativa con respecto a los demás procesos del programa. Más ligado a la implementación y a veces se incumple: existen distintos grados



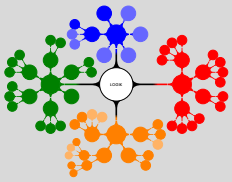
Pruebas simples de programas concurrentes

¿Cómo demostrar que un programa cumple una determinada propiedad ?

- *Posibilidad*: realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad
- *Problema*: Sólo permite considerar un número finito y muy limitado de trazas del programa y no demuestra la ausencia de casos indeseables (en alguna traza no explorada)
- Ejemplo: Comprobar que el proceso P produce al final `x == 3`:

```
process P ;
var x : integer := 0 ;
cobegin
x = x+1 ; x = x+2 ;
coend
```

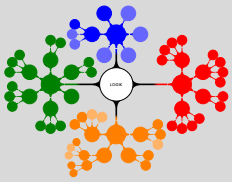
(hay varias trazas que llevan al resultado: x==1 o x==2, y estas historias podrían ocurrir en algunas ejecuciones, por tanto, produciendo un resultado no-determinado)



Enfoque operacional: análisis exhaustivo

- *Enfoque operacional*: Análisis exhaustivo de casos. Se comprueba la corrección de todas las posibles trazas
- *Problema*: Su utilidad está muy limitada cuando se aplica a los programas concurrentes, ya que el número de entrelazamientos crece exponencialmente con el número de instrucciones de los procesos

Para el sencillo programa P, formado por 2 procesos y 3 sentencias atómicas por proceso, tendríamos que estudiar 20 historias diferentes (=*permutaciones con repetición de 6 elementos con 2 grupos de 3 elementos*)



Verificación: enfoque axiomático

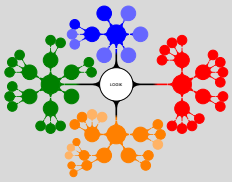
Se define un sistema lógico formal que permite establecer propiedades de programas en base a axiomas y reglas de inferencia

- Se usan fórmulas lógicas (llamados: *asertos* o *predicados*) para caracterizar un conjunto de estados
- Las sentencias atómicas actúan como transformadores de tales asertos. Los teoremas se escriben de la forma:

$$\{P\}\ S\ \{Q\}$$

Interpretación de un teorema (llamado también: *triple*) en esta lógica: "*Si la ejecución de la sentencia* **S** *comienza en algún estado en el que es verdadero el aserto* **P** *(precondición), entonces el aserto* **Q** *(poscondición) será verdadero en el estado resultante*"

Menor Complejidad que el enfoque operacional: El trabajo que conlleva la prueba de corrección es proporcional al número de sentencias atómicas en el programa



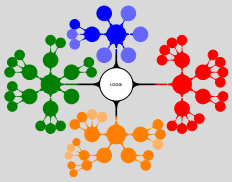
Verificación de programas

Notación: *Triples* de Hoare

- Se introduce la notación de *especificación de corrección parcial* para especificar lo que hace un programa

$$\{P\}\ C\ \{Q\}$$

  - C es un programa del lenguaje cuyos programas están siendo especificados
  - P y Q son asertos definidos con las variables programa que representa C
  - Las variables libres de P y Q son del programa o son variables lógicas
- Los asertos caracterizan los estados aceptables del programa:
  - V caracteriza a todos los estados del programa; sin embargo, F no caracteriza a ninguno
  - Los *triples* ({P} C {Q}) son los teoremas (= *proposiciones demostrables*) con la lógica que estamos tratando



Significado de la notación de Hoare

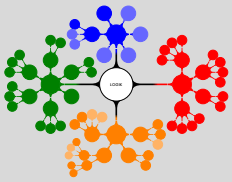
{P} C {Q} es cierto si

- siempre que C es ejecutado en un estado que
- satisface P y *si* la ejecución de C termina,
- el estado en que C termina satisface Q

Ejemplo: $\{X == 1\}\ X = X + 1\ \{X == 2\}$

- el aserto P dice que el valor de X es 1
- el aserto Q dice que el valor de X es 2
- C es la sentencia de asignación $X = X + 1$

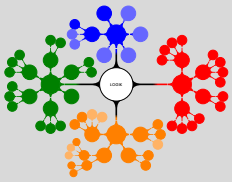
- $\{X == 1\}\ X = X + 1\ \{X == 2\}$ es cierto
- $\{X == 1\}\ X = X + 1\ \{X == 3\}$ es falso
- $\{X == 1\}$ *WHILE T DO NULL* $\{Y == 3\}$ ¡es cierto!



La estructura de las demostraciones

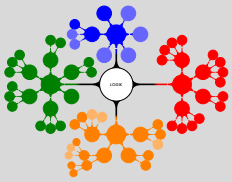
Una demostración es una secuencia de líneas (o *triples*) de la forma {P} *C* {Q}, cada una de los cuales es un *axioma* o deriva de los anteriores aplicando una *regla de inferencia* de la lógica

- Una demostración consiste en una secuencia de *líneas*:
  - como la definición de $()^2$ del ejemplo siguiente
- Cada una de las líneas es una instancia de un *axioma*
- o se deriva de las líneas anteriores por medio de una *regla de inferencia*
- La sentencia de la última línea de una demostración es lo que se quiere demostrar
  - como ocurre con: $(\mathtt{X}+1)^2 == \mathtt{X}^2 + 2 \times \mathtt{X} + 1$



Ejemplo de demostración

- Ejemplo de una demostración formal:
  1. $(X+1)^2 == (X+1) \times (X+1)$
     Definición de $()^2$
  2. $(X+1) \times (X+1) == (X+1) \times X + (X+1) \times 1$
     Distributiva izquierda de $\times$ con $+$
  3. $(X+1)^2 == (X+1) \times X + (X+1) \times 1$
     Sustituir línea 2 en 1
  4. $(X+1) \times 1 == X+1$ Ley identidad para 1
  5. $(X+1) \times X == X \times X + 1 \times X$
     Distributiva derecha de $\times$ con $+$
  6. $(X+1)^2 == X \times X + 1 \times X + X + 1$
     Sustituir líneas 4 y 5 en 3
  7. $1 \times X == X$ Ley identidad para 1
  8. $(X+1)^2 == X \times X + X + X + 1$
     Sustituir línea 7 en 6
  9. $X \times X == X^2$ Definición de $()^2$
  10. $X + X == 2 \times X$ 2==1+1, ley distributiva
  11. $(X+1)^2 == X^2 + 2 \times X + 1$
      Sustituir líneas 9 y 10 en 8

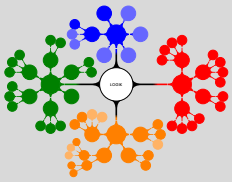


Propiedades de seguridad y complección

- Un aserto {P} caracteriza un estado aceptable del programa, es decir, un estado que podría ser alcanzado por el programa si sus variables tomaran unos determinados valores
- Concepto de Interpretación de las fórmulas de un SLF:

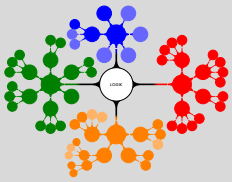
$$\text{asertos} \rightarrow \{V, F\}$$

  - Fórmula satisfascible
  - Fórmula válida
  - el aserto { V } caracteriza a todos los estados del programa (o *precondición más débil*)
  - el aserto { F } no se cumple por parte de ningún estado del programa (o *precondición más fuerte*)
- *SLF* seguro: $\text{asertos} \subseteq \{\text{hechos ciertos}\}$ – expresados como fórmulas de la lógica
- *SLF* completo: $\{\text{hechos ciertos}\} \subseteq \{\text{asertos}\}$
- Los sistemas lógicos para demostración de programas no suelen poseer la propiedad de complección, pero a efectos prácticos es suficiente con la de complección relativa (todas las proposiciones son demostrables excepto si contienen expresiones aritméticas)



Fórmulas proposicionales válidas (tautologías)

- Leyes distributivas:
  - $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
  - $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
- Leyes de De Morgan:
  - $\neg(P \wedge Q) = \neg P \vee \neg Q$
  - $\neg(P \vee Q) = \neg P \wedge \neg Q$
- Eliminación-*And*: $(P \wedge Q) \rightarrow P$
- Eliminación-*Or*: $P \rightarrow (P \vee Q)$

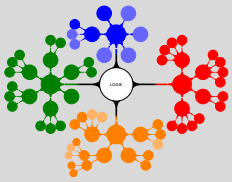


Axiomas y reglas de inferencia

Sirven para poder llevar a cabo las demostraciones de programas, ya que cada línea de la demostración es o bien un axioma o se deriva de la anterior mediante una regla de inferencia

- Axioma: fórmulas que sabemos que son ciertas en cualquier estado del programa
- Regla de inferencia: para derivar fórmulas ciertas a partir de los axiomas o de otras que se han demostrado ciertos previamente

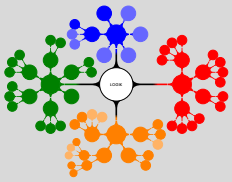
Notación de una Regla de inferencia:
(*nombre de la regla*) $\frac{H_1, H_2, \ldots, H_n}{C}$
Un aserto (o *teorema*) es una fórmula con una interpretación cierta que pertenece al dominio de los hechos que queremos probar (o *dominio del discurso*)
Los asertos de nuestra lógica coinciden con las líneas o sentencias lógicas de las que se compone la demostración de un programa



Axiomas y reglas de inferencia–II

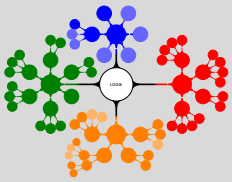
1. Axioma de la sentencia nula: { P } NULL { P}: si el aserto es cierto antes de ejecutarse esta sentencia, tendrá la misma interpretación cuando la sentencia termine
2. Sustitución textual: $\{P_e^x\}$ es el resultado de sustituir la expresión *e* en cualquier aparición de la variable *x* en P
3. Axioma de asignación: $\{P_e^x\}x = e\{P\}$: una asignación cambia solo el valor de la *variable objetivo*, el resto de las variables conservan los mismos valores.
   Ejemplos:
   - A $\{V\}\, x = 5\, \{x == 5\} \equiv \{P\}$ es un *aserto* pues: $P_5^x \equiv \{x == 5\}_5^x \equiv \{5 == 5\} \equiv V$
   - B $\{x > 0\}\, x = x + 1\, \{x > 1\} \equiv \{P\}$ es un aserto pues: $P_{x+1}^x \equiv \{x > 1\}_{x+1}^x \equiv \{x + 1 > 1\} \equiv \{x > 0\}$

   Reglas para *conectar* los triples en las demostraciones:
4. Regla de la consecuencia (1): $\dfrac{\{P\}S\{Q\},\ \{Q\} \rightarrow \{R\}}{\{P\}S\{R\}}$



Axiomas y reglas de inferencia–II

1. Regla de la consecuencia (2): $\frac{\{R\} \rightarrow \{P\}, \{P\} S \{Q\}}{\{R\} S \{Q\}}$
2. Regla de la composición: $\frac{\{P\} S_1 \{Q\}, \{Q\} S_2 \{R\}}{\{P\} S_1 ; S_2 \{R\}}$: para obtener la pre y pos-condición de 2 sentencias juntas
3. Regla del IF: $\frac{\{P\} \wedge \{B\} S_1 \{Q\}, \ \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \mathit{if}\ B\ \mathit{then}\ S_1\ \ \mathit{else}\ S_2\ \mathit{fi} \{Q\}}$
4. Regla de la iteración: $\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\} \mathit{while}\ B\ \mathit{do}\ S\ \ \mathit{enddo} \{I \wedge \neg B\}}$: podrá iterar un número arbitrario de veces (incluso 0); el *invariante* I se satisface antes y después de cada iteración



Verificación de sentencias concurrentes

- La ejecución concurrente de los procesos hace que el SLF deje de ser seguro: problema de la *interferencia*

```
y=0; z=0;
cobegin
 x= y+z || y=1; z=2
coend;
```

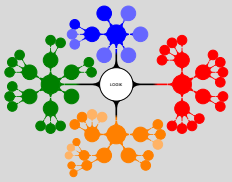
estados del programa	valores de x
x==y==z==0	0
x==y==1, z==0	1
x==3, y==1, z==2	3
-	2

¿Cómo evitar la interferencia?

- Evaluación de expresiones que no hacen referencia a variables modificadas concurrentemente

```
 {x==0; y==0}
  x= 0; y=0;
  cobegin x= x+1 || y= y+1 coend;
  z=x+y
{x==1, y==1, z==2}
```

- ¡Condición demasiado restrictiva!: casi ningún programa la cumpliría.



Verificación de sentencias concurrentes-II

Acción atómica elemental $< \ldots >$

Propiedad como máximo una vez: " *La evaluación de una expresión por un proceso concurrente se hace de forma atómica si las variables que comparte son leídas o escritas por un único proceso*"

Ejemplo 1: *Evaluación de expresiones*

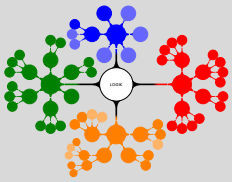
1. $x = 0; y = 0;$ *cobegin* $x = y + 1 || y = x + 1$ *coend*; ¡Incorrecta!
2. $x = 0; y = 0; z = Z;$ *cobegin* $x = y + 1 || y = z + 1$ *coend*;
3. $x = 0; y = 0; z = Z;$ *cobegin* $x = z + 1 || y = x + 1$ *coend*;

No interferen. entre un aserto y una instrucción atómica *a*:

$$\{C \wedge \text{pre}(a)\}\, a\, \{C\}$$

Ejemplo de no-interferencia:

1. $\{y = 0, z = Z\} < y = z + 1 >$ no interfiere con la poscondición de $< x = y + 1 > \{x = Y + 1, z = Z\}$
2. $\{x = 0, z = Z\} < x = y + 1 >$ no interfiere con la poscondición de $< y = z + 1 > \{y = Z + 1, z = Z\}$



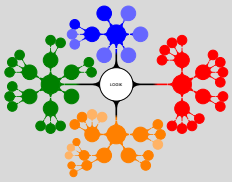
Regla de inferencia de la composición concurrente de sentencias

Si $\{P_i\}\ S_i\ \{Q_i\}$ no interfieren, entonces se puede demostrar el triple:

$$\{P_1 \wedge P_2 \ldots \wedge P_n\}$$
$$\text{COBEGIN}$$
$$S_1 \| S_2 \| \ldots \| S_n$$
$$\text{COEND}$$
$$\{Q_1 \wedge Q_2 \ldots \wedge Q_n\}$$

Si los asertos de los procesos no se invalidan entre sí, entonces su composición concurrente transforma la conjunción de sus precondiciones en la conjunción de sus postcondiciones:

$$\{x == 0\}$$
$$\text{COBEGIN}$$
$$< x = x + 1; > \ \| \ < x = x + 2 >$$
$$\text{COEND}$$
$$\{x == 3\}$$



Regla de inferencia de la composición concurrente de sentencias -II

Traza de la demostración libre de interferencias:

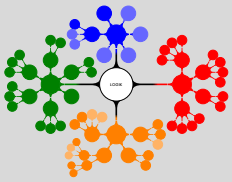
$$\{x == 0\}$$
$$\text{COBEGIN}$$
$$\{x == 0 \vee x == 2\} \qquad \{x == 0 \vee x == 1\}$$
$$< x = x + 1; > \;\|\; < x = x + 2 >$$
$$\{x == 1 \vee x == 3\} \qquad \{x == 2 \vee x == 3\}$$
$$\text{COEND}$$
$$\{x == 3\}$$

Aplicando la *regla de la no interferencia* a la expresión de asertos concurrentes anterior, la precondición de la sentencia `cobegin` ha de ser equivalente a la conjunción de las precondiciones de sus sentencias componentes

$(x == 0 \vee x == 2) \wedge (x == 0 \vee x == 1) \equiv (x == 0)$

Así mismo ocurre con la poscondición de `coend`

$(x == 1 \vee x == 3) \wedge (x == 2 \vee x == 3) \equiv (x == 3)$



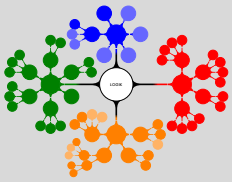
Invariante global

Invariante global: se trata de un predicado que referencia variables globales del programa que se demuestra cierto en el estado inicial de cada proceso y se mantiene cierto ante cualquier *asignación* dentro de los procesos
Ejemplo de invariante global:

- En una solución correcta del Productor-Consumidor, un invariante global sería:

$$consumidos \leq producidos \leq consumidos + 1$$

- Condición de invariante global (IG): Dado un aserto I definido a partir de las *variables compartidas* entre los procesos de un programa concurrente, puede ser considerado un IG válido si y sólo si:
  - Es cierto para los valores iniciales de las variables
  - Se mantiene cierto después de la ejecución de cada instrucción atómica *a* del programa: $\{I \wedge pre(a)\}\ a\ \{I\}$



Bibliografía

Para más información, ejercicios, bibliografía adicional, se puede consultar:

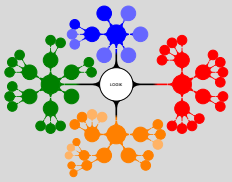
1.1. Conceptos básicos y Motivación, Palma (2003): capítulo 1

1.2. Modelo abstracto y Consideraciones sobre el hardware, Ben-Ari (2006), capítulo 2. Andrews (2000), Palma (2003): capítulo 1

1.3. Exclusión mutua y sincronización, Palma (2003), capítulo 1.

1.4. Propiedades de los Sistemas Concurrentes, Palma (2003), Capel (2022): capítulo 1

1.5. Verificación de Programas concurrentes, Andrews (2000), capítulo 2. Capel (2022): capítulo 1



1.2. Tema 2

1.3. Tema 2.1

Tema 2

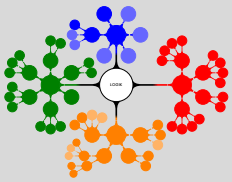
Sincronización en memoria compartida

SCD para GIIM

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 11 Octubre, 2024

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



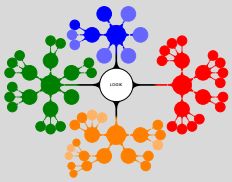
Limitaciones de los semáforos

Hay algunos inconvenientes de usar mecanismos como los semáforos:

- Basados en variables globales
- El uso y función de las variables (*protegidas* de los semáforos)
- Las operaciones se encuentran dispersas y no protegidas

Por tanto, es necesario:

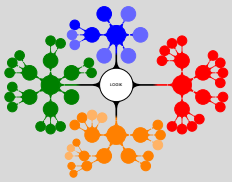
- un nuevo *mecanismo* de programación que permita la encapsulación de la información y de la sincronización entre procesos,
- programar las operaciones de sincronización (`wait`, `signal` `post`...) dentro de bloques o procedimientos que se ejecuten con instrucciones atómicas



Estructura y funcionalidad de un monitor

C.A.R. Hoare, en 1974, idea el concepto de Monitor, que es un mecanismo de programación de alto nivel que permite definir objetos abstractos compartidos entre los procesos concurrentes.

Las variables permanentes y los procedimientos de los monitores garantizan acceso en exclusión mutua y encapsulación de su sincronización.

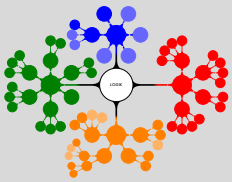


Características generales de los monitores

Concepto de monitor

"Módulo" de las aplicaciones concurrentes

- Modularidad en el desarrollo de programas y aplicaciones
- Programa= {Monitores, Procesos}
- Estructuración en el acceso a tipos de datos, variables compartidas, etc.
- Capacidad de modelado de interacciones cooperativas y competitivas entre procesos concurrentes, lo más general posible
- Ocultación a los procesos de las operaciones de sincronización sobre datos compartidos
- Reusabilidad basada en parametrización de los módulos monitor
- Verificación mediante reglas más simples que las de los semáforos



Semántica de los componentes de un monitor

Exclusión mutua en el acceso a los procedimientos/métodos del módulo monitor

Variables permanentes: son el estado interno del monitor

Procedimientos: modifican el estado interno (garantizando la exclusión mutua durante dicho cambio)

Código de inicialización: fija el estado interno inicial

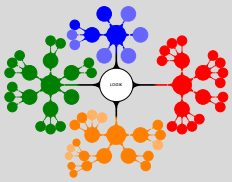


Diagrama de los componentes del monitor

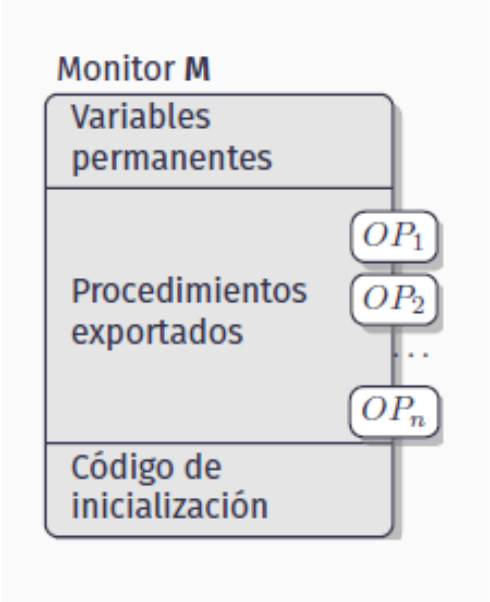
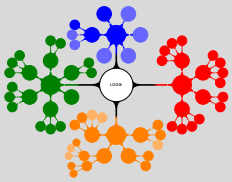


Figure: Un monitor se puede visualizar como aparece en el diagrama



Concepto de *monitor* III

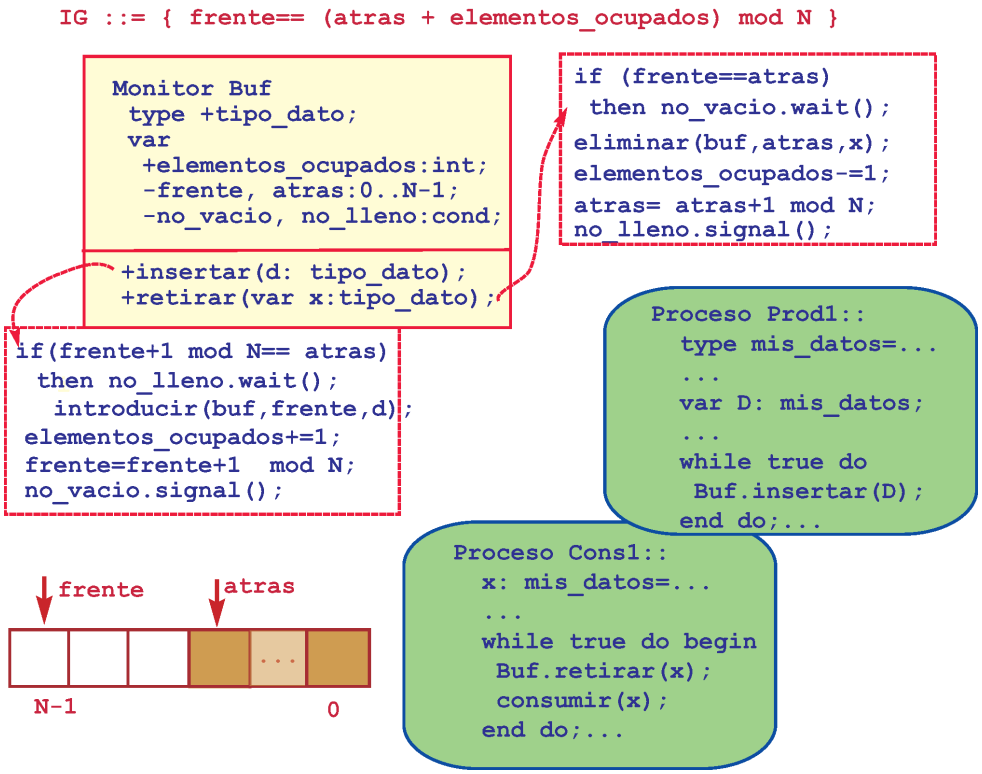
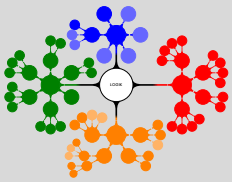
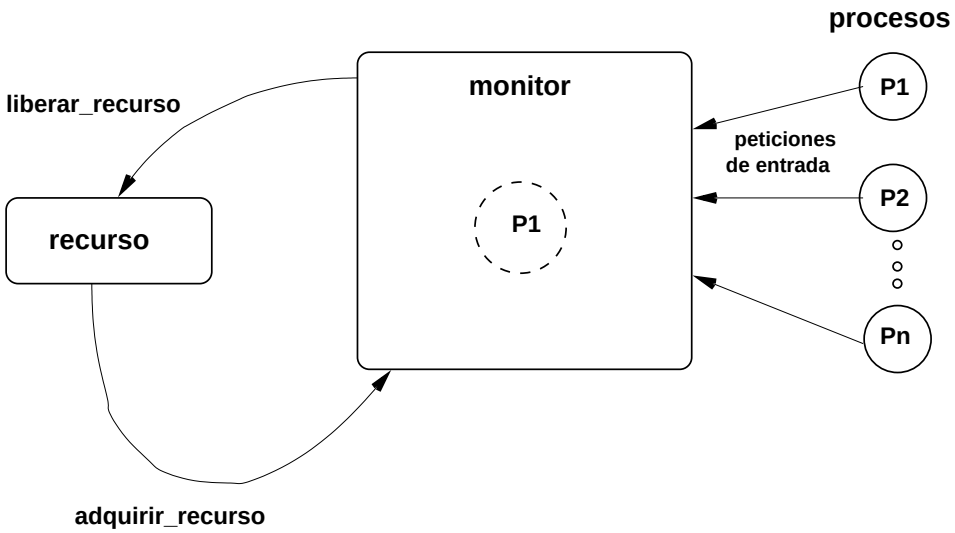


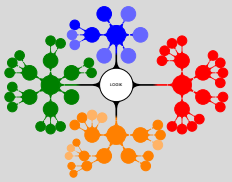
Figure: Representación de gráfica de un módulo monitor



Características de la programación con *monitores*

- Centralización de recursos críticos
- Sólo 1 procedimiento ejecutado por un solo proceso
- Los procedimientos pueden interrumpirse
- Posibilidad de ejecución concurrente de monitores no-relacionados

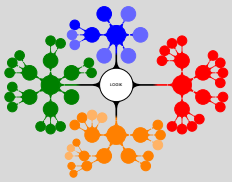




Instanciación de los monitores

Los monitores son “objetos” pasivos

Objetivos de la instanciación de los monitores

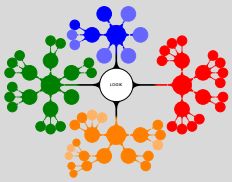


Instanciación de los monitores-II

- Cada instancia tiene sus variables permanentes propias
- La E.M. ocurre en cada instancia por separado

Condición para que un lenguaje de programación pueda compilar monitores instanciables:

El código de los procedimientos de los monitores ha de ser *reentrante*

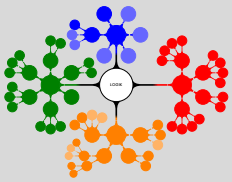


Instanciación de los monitores-III

```
class monitor VariableProtegida(entr,salid : integer);
var x, inc : integer;
// incremento(), valor(); son los procedimientos llamables
procedure incremento( );
begin
  x := x+inc ;
end;
procedure valor(var v : integer);
begin
  v := x ;
end;
begin
x:= entr ; inc := salid ;
end
```

`mv1` y `mv2` pueden ser compartidas por varios procesos concurrentes sin que se produzca *interferencia*.

```
var mv1 : VariableProtegida(0,1);
mv2 : VariableProtegida(10,4); i1,i2 : integer ;
begin
mv1.incremento() ;
mv1.valor(i1) ; { i1==1 } //permanentes (x,inc) distintas
mv2.incremento() ; //para cada instancia del monitor
mv2.valor(i2) ; { i2==14 }
end
```

Cola del monitor para exclusión mutua

El control de la exclusión mutua se basa en la existencia de una *cola FIFO de entrada al monitor* proceso que ejecute una llamada a uno de sus procedimientos, entrará en el monitor

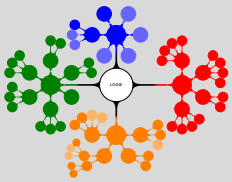


Diagrama de estados de un proceso

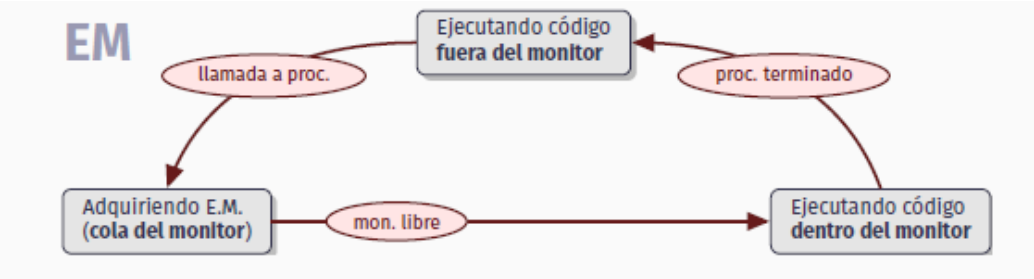
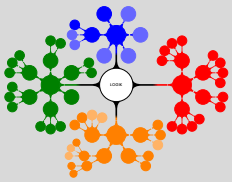
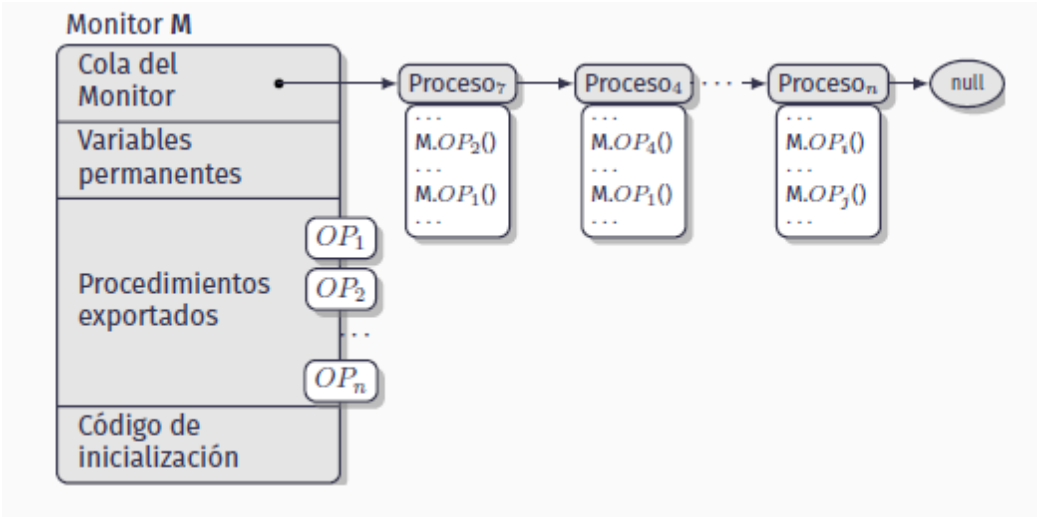


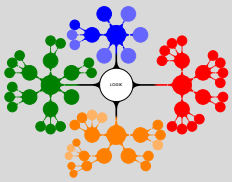
Figure: Posibles estados de los procesos y las transiciones entre dichos estados



Estado del monitor



El estado del monitor incluye la cola de procesos esperando a comenzar a ejecutar el código del mismo



Operaciones de sincronización en monitores

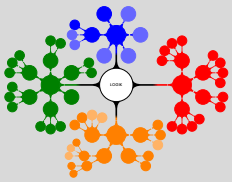
Para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que sea cierta determinada condición

En semáforos existe:

- La posibilidad de bloqueo (`sem_wait`) y activación (`sem_signal`)
- Un valor entero (el valor de la variable protegida “s” del semáforo)

En monitores, sin embargo, se tiene:

- Sólo se dispone de sentencias de bloqueo y activación
- No hay variable, ni algún valor protegido, asociada a 1 variable condición



Operaciones de sincronización en monitores-II

Encapsulación de la sincronización

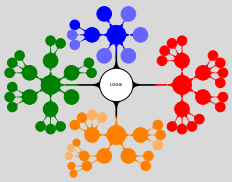
- Explícitamente programada dentro de los procedimientos
- TDA `cond`
- Variables condición (sólo declararlas, no inicializarlas)
- Se define 1 variable condición, por cada una de las condiciones para esperar, en el monitor

`c.wait()`: bloquea siempre. El desbloqueo de los procesos se produce en orden FIFO	`c.signal()`: si la cola de `c` no está vacía, desbloquea al primer proceso de la cola

- Las variables condición pertenecen a un tipo abstracto y opaco de datos La representación interna de las variables condición no es accesible al programador de monitores
- La cola de procesos bloqueados asociada a 1 variable condición es FIFO

`c.queue()`:

función lógica que devuelve `true` si hay algún proceso esperando en la cola de cond, y `false` en caso contrario



El TDA *cond* de los monitores

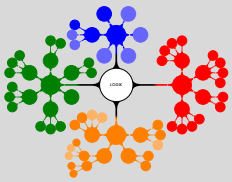
- La exclusión mutua se levanta como consecuencia de ejecutar `c.wait()` cuidado: deja "abierto" el monitor antes de bloquear al proceso
- Se cede el acceso del monitor al proceso señalado (`c.signal()` con semántica desplazante)
- La semántica desplazante de las señales evita un robo de señal (que se *cuele* un tercer proceso en el monitor)

Comportamiento de los procesos después de ejecutar las operaciones de sincronización

- Después de ejecutar `c.wait()`
- Después de ejecutar `c.signal()`

  No pueden programarse operaciones `c.wait()` indebidas, ni tampoco omitirse las operaciones `c.signal` necesarias

- Operaciones `c.queue` y `c.signal_all`
- No es segura la simulación de `c.signal_all` utilizando `c.queue` y señales desplazantes



Estado de un monitor con varias colas

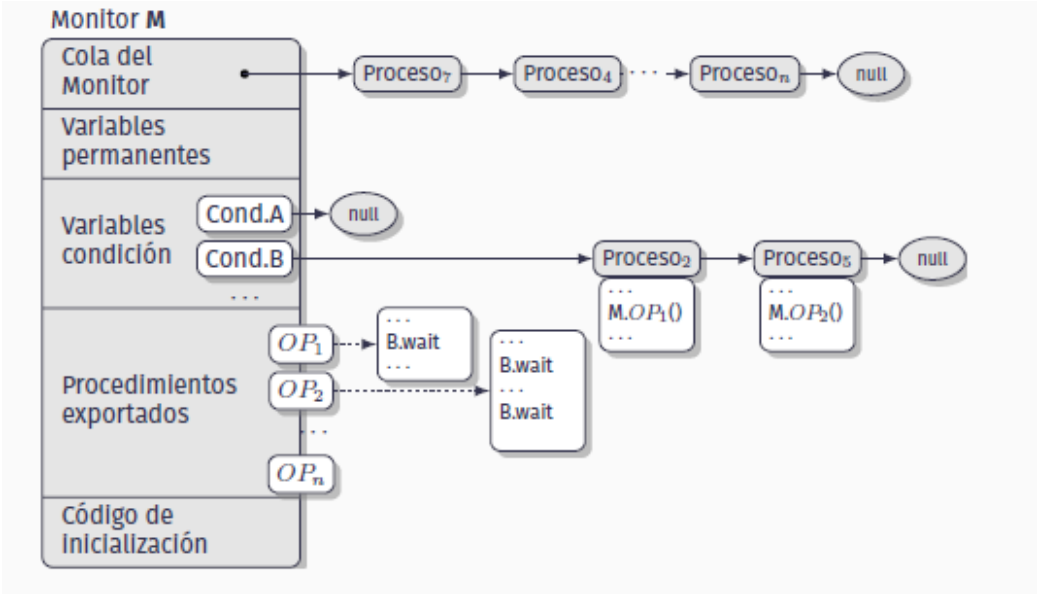
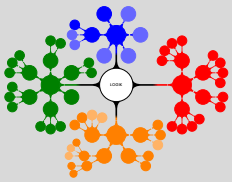


Figure: Los procesos 2 y 5 ejecutan las operaciones 1 y 2, ambas producen esperas de la condición B



Verificación de programas con monitores

El programador no puede conocer a priori la traza concreta de llamadas a los procedimientos del monitor por parte de los procesos del programa concurrente.

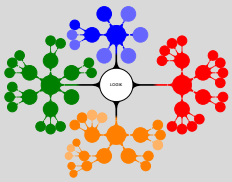
Monitor Buf;

Process P1 (consumidor) || Process P2 (consumidor) || Process P3(productor)
==>Trazas:
P3::Buf.insertar(x);P3::Buf.insertar(x');P2::Buf.retirar(x);P1::Buf.retirar(x'); ...

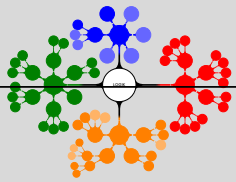
P3::Buf.insertar(x);P2::Buf.retirar(x);P1::Buf.retirar(x') [bloqueado]; P3::Buf.insertar(x');...
...

Invariante de un monitor (IM):

- Es una propiedad que el monitor cumple siempre, pero específico de cada monitor diseñado por un programador



Verificación de programas con monitores-II



Esquema general de demostración de programas con monitores:

-Probar la corrección parcial de los procesos secuenciales

-Comprobar que el invariante de cada uno de los monitores del programa se mantienen: incialmente y antes/después de ejecutar cada pr

-Aplicar la regla de la concurrencia

Características de un IM:

- Su *valor de verdad* depende de los valores de las variables permanentes
- Debe ser cierto en cualquier estado del programa concurrente, excepto cuando un proceso está ejecutando código del monitor

Verificación de los monitores

- Axiomas iniciales

  La demostración de corrección se basa en:

> **Invariante del monitor**(IM)
> relación constante entre los valores permitidos de las variables permanentes del monitor

Ejemplo (del Monitor Buf)`elementos_ocupados` $\leq$ `N`:

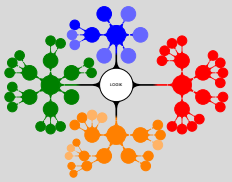
Axioma (inicialización variables)

{V} inicializacion variables permanentes {IM}

La inicialización se lleva a cabo dentro del cuerpo `begin ... end` del monitor

El invariante del monitor debe ser cierto en su estado inicial, justo después de la inicialización de las variables permanentes

Inicialmente: `elementos_ocupados = 0` $\leq$ `N` : dado que `N` $>$ `0`



Verificación de los monitores-II

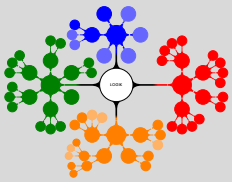
El invariante del monitor debe ser cierto:

- antes y después de cada llamada a un procedimiento del monitor

Axioma (procedimientos del monitor)

$$\{\text{IN} \wedge \text{IM}\}\ \text{procedimiento}_i\ \{\text{OUT} \wedge \text{IM}\}$$

- `IN` satisfecho por los p. `in,in/out`
- `OUT` satisfecho por los p. `out,in/out`

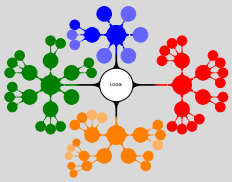


Axiomas de operaciones de sincronización con semántica desplazante

Axioma (operacion `c.wait()`)

$$\{IM \wedge L\}\ c.wait()\ \{C \wedge L\}$$

- El proceso que ocasiona la ejecución de `c.wait` se bloquea y deja libre el monitor
- Entra en la cola FIFO asociada a `c`
- Las demostraciones de corrección no tienen en cuenta la obligación de que el proceso termine de ejecutar `c.wait()`

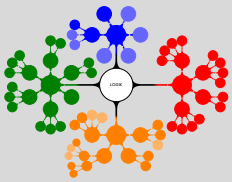


Axiomas de operaciones de sincronización con semántica desplazante-II

Axioma (operación `c.signal()`)

$$\{\neg \text{vacio(c)} \wedge L \wedge C\}\ \text{c.signal()}\{IM \wedge L\}$$

- No tiene efecto si la cola `c` está vacia
- Se supone semántica desplazante: el monitor mantiene el estado expresado por C
- Los axiomas no tienen en cuenta el orden de desbloqueo de los procesos

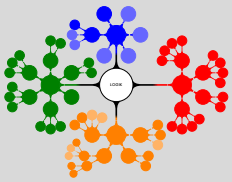


Ejemplo: verificación con señales desplazantes

Regla verificación del IF: ambas ramas con misma postcondición $\{s > 0\}$

```
Monitor Semaforo;
 var s: integer; {IM: s ≥ 0}
condición de sincronización: {s > 0}
     c: cond;
   procedure P;
     begin
      {IM}
      if s=0 then
        {s = 0 ∧ IM}
        c.wait;
      {s>0}
      else
      {s>0}
        null;
      {s>0}
      endif;
      {s > 0}
        s:= s-1
      {s ≥ 0 → IM}
     end;
```

```
procedure V;
 begin
  {IM}
     s:= s+1;
    {s>0}
          c.signal;
  {s≥ 0 → IM}
  end;
  begin
    {TRUE}
    s:= 0;
 {s ≥ 0} → {IM}
    end;
```

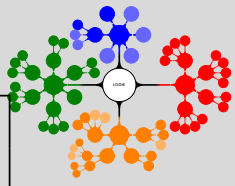



Verificación de un monitor con señales desplazantes-II

Ejercicio: Semáforo de Habermann

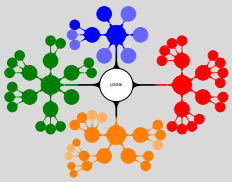
$n_p \leq n_a$, ya que primero ha de incrementarse n_a
$n_p \leq n_v$, ya que n_a no puede superar a n_v para incrementar n_p
$n_p \geq min(n_a, n_v)$, condición no necesaria, pero deseable

```
Monitor Semaforo;
  var na, np, nv:int
      c: cond;

   procedure P;
   begin
    na:= na+1;
    if(na > nv)
      then c.wait();
    np:= np+1;
   end;

   procedure V;
   begin
    nv:= nv+1;
    if(na > np)
      then c.signal;
   end;

              na:=0;
              nv:=0;
              np:=0;
```

Verificación del Monitor “Semáforo de Habermann”

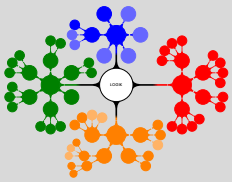
```
Monitor Semaforo;
  var na, np, nv:int
      c: cond;
   procedure P();                    procedure V();
   begin{np==min(na,nv)}             begin {np==min(na,nv)}
     na:= na+1;                        nv:= nv+1;
    {np==min(na-1,nv)}                {np==min(na,nv-1)}
     if(na > nv)                       if(na > np)
        then                              then
{(na>nv) and np===min(na-1,nv)}=>    {(na>np) and np===min(na,nv-1)}=>
np== min(na,nv)                      na>np and np== nv-1
           c.wait();                         c.signal;
       {na>np and np===nv-1}              {np==min(na,nv)}
      else null                          end;
       { np<nv and np===na-1}
      endif
    {np+1===min(na,nv)}
     np:= np+1;
    {np===min(na,nv)}
   end;
{V}  na:=0;    nv:=0;    np:=0;    {np===min(na,nv)}
```

Regla de la concurrencia para la verificación de programas con monitores

$\{P_i\}\, S_i\, \{Q_i\},\ 1 \leq i \leq n$
ninguna variable libre en P_i o en Q_i es modificada por $S_j,\ i \neq j$
todas las variables en IM_k son locales al monitor m_k

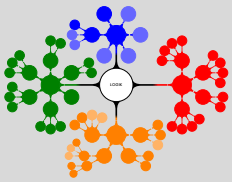
$\{IM_1 \wedge \ldots\ IM_m \wedge P_1 \wedge \ldots \wedge P_n\}$
cobegin $S_1 \parallel S_2 \parallel \ \ldots\ \parallel S_n$ coend
$\{IM_1 \wedge \ldots\ IM_m \wedge Q_1 \wedge \ldots \wedge Q_n\}$



Patrones de uso de monitores

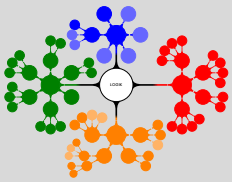
Se estudian a continuación los patrones de solución para tres problemas sencillos, típicos de la Programación Concurrente

- Espera única (EU): un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (ocurre típicamente cuando un proceso debe leer una variable escrita por otro proceso, el primero se suele denominar Consumidor y el segundo Productor)
- Exclusión mutua(EM): acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos
- Problema del Productor/Consumidor(PC): similar a la espera única, pero de forma repetida en un bucle (un proceso Productor escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso Consumidor)



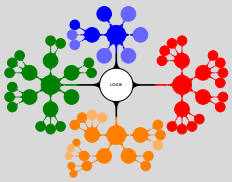
Uso del monitor de EU

```
//variables compartidas
var x: integer; //contiene cada valor producido
process Productor;//escribe x
  var a:integer ;
  begin
   a:=ProducirValor();
   x:=a; //sentencia E
   EU.notificar();//sentencia N
process Consumidor//lee x
  var b:integer;
  begin
  EU.esperar();//sentencia W
  b:=x;//sentencia L
  EU.termina();
  UsarValor(b) ;
end
```

Monitor para Espera Única (EU)

```
monitor EU;
var terminado:boolean;
//terminado==true si terminado, si no: terminado==false
    leer_autorizado: boolean //variable auxiliar
    cola:condition;
    //cola consumidor esperando a que terminado==true
    export esperar, notificar;
  //Invariante: terminado= false => lee= false
  procedure esperar();//para llamar antes de sentencia  L
  begin
   if (not terminado) then //si no se ha terminado W
       cola.wait();//esperar hasta que termine
   //Condición de sincronización terminado= true
   leer_autorizado:= true;//funciona solo suponiendo
       semantica desplazante de la senial!!!
  end;
  procedure termina();
  begin
    leer_autorizado:= false; terminado:= false;
  end;
  procedure notificar();//para llamar después de E
  begin
    terminado:=true;//Condición de sincronización
    cola.signal();//reactivar al otro proceso, si esa
        esperando
  end
```

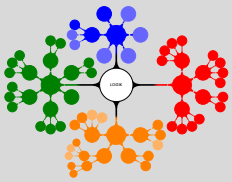



Verificación del monitor EU

$$\textit{Invariante}: \ \textit{terminado} = \textit{false} \ \Rightarrow \ \textit{leer\_autorizado} = \textit{false}$$

```
procedure esperar();
{Invariante, terminado= false, leer_autorizado= false}
if (not terminado) then
  begin
    {terminado= false, leer_autorizado= false, Invariante}
    cola.wait();
  endif;
  {Condicion de sincronizacion: terminado= true}
  leer_autorizado:= true
  {Invariante}
 end;
```

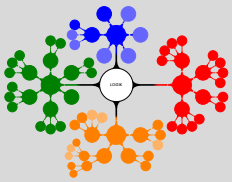
```
procedure notificar();
begin
  {terminado= false, leer_autorizado=false, Invariante}
  terminado:= true;
  {Condicion de sincronizacion: terminado= true}
  cola.signal();
  {Invariante}
end;
```

Uso y propiedades del monitor EM

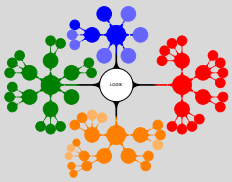
El monitor puede ser utilizado por n procesos concurrentes:

```
process Usuario[ i : 0..n ]
  begin
   while true do begin
    EM.entrar();//esperar SC libre, registrar SC ocupada
    ..... //sección crítica (SC)
    EM.salir(); //registrar SC libre, señalar
    ..... //otras actividades (RS)
  end
end
```

Patrón de Exclusión Mutua (EM)

```
monitor EM ;
  var ocupada:boolean;
      //ocupada==true hay un proceso en SC, si no: ocupada
          ==false
      cola:condition;
      //cola de procesos esperando a que ocupada==false
      export entrar,salir;
      //nombra procedimientos públicos
  procedure entrar();//protocolo de entrada (sentencia E)
   begin
    if ocupada then//si hay un proceso en la SC
        cola.wait();//esperar hasta que termine
    ocupada:=true;//indicar que la SC está ocupada
   end
  procedure salir();//protocolo de salida (sentencia S)
   begin
    ocupada := false;//marcar la SC como libre
    //si al menos un proceso espera, reactivar al primero
    cola.signal();
   end
  begin//inicializacion:
   ocupada:=false;//al inicio no hay procesos en SC
  end
```

Invariante del monitor

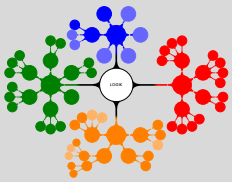
El invariante del monitor, es la conjunción de estas dos condiciones:

- $$IM :: \ ocupada == false \Leftrightarrow num\_sc == 0 \wedge$$
$$0 \leq num\_sc \leq 1$$

  es decir, no puede ejecutar más de 1 proceso la sección crítica.

Demostración del IM:

- Al inicio, IM es cierto (la sección crítica está vacía, luego $num\_sc == 0$ y ocupada == false).
- Demostración de los procedimientos (`entrar()`, `salir()`)del monitor:
  - Declarar una variable permanente ficticia `num_sc` (inicialmente== 0)
  - Modificar el IM como: $libre + num\_sc == 1$ (suponemos la aritmetización de las constantes lógicas)

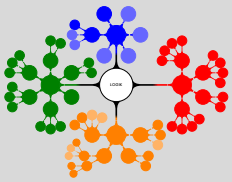


Sincronización tipo Productor/Consumidor

El problema del Productor-Consumidor con las lecturas y escrituras (E y L) repetidas en un bucle puede solucionarse con el monitor PC, muy sencillo, que encapsula el valor compartido

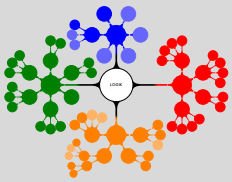
- El procedimiento `escribir` escribe el parámetro en la variable compartida
- El procedimiento `leer`, lee el valor que hay en la variable

```
process Productor;//calcula x
  var a:integer;
  begin
   while true do begin
    a:=ProducirValor();
    PC.escribir(a);//copia a en valor
   end
end
process Consumidor //lee x
  var b : integer ;
  begin
   while true do begin
    PC.leer(b); //copia valor en b
    UsarValor(b);
   end
end
```

Monitor Productor/Consumidor

```
Monitor PC ;
 var valor_com:integer;//valor compartido
     pendiente:boolean;//true: valor escrito y no leido
     cola_prod:condition;
     //espera productor hasta que pendiente == false
     cola_cons:condition;
     //espera consumidor hasta que pendiente == true
  procedure escribir( v:integer );//sentencia E
   begin
    if pendiente then
      cola_prod.wait();
      valor_com:=v;
      pendiente:=true;
      cola_cons.signal();
   end;
  function leer():integer;//sentencia L
   begin
   if (not pendiente) then
     cola_cons.wait();
     result:=valor_com;
     pendiente:=false;
     cola_prod.signal();
   end;
   begin // inicializacion }
     pendiente := false ;
   end;
```

Verificación del monitor

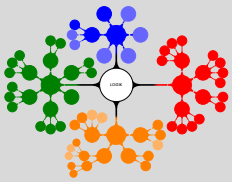
Al igual que en los otros casos, podemos verificar que el monitor funciona bien:

- #E = número de llamadas a escribir completadas
- #L = número de llamadas a leer completadas
- El invariante del monitor (IM) es:

$$\#E - \#L = \begin{cases} 0 \; si\; pendiente == false \\ 1 \; si\; pendiente == true \end{cases}$$

Demostración del IM:

- Se demuestra igual que en el caso del Patrón EM, sustituyendo:
  - $\#E - \#L == num\_sec$ y
  - *pendiente* == *NOT libre*



Señales con prioridad

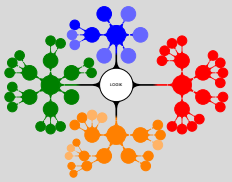
- La semántica de las señales no contempla el desbloqueo de los procesos según un orden prioritario

  `c.wait(prioridad)` bloquea a los procesos en la cola `c`, pero ordenándolos con respecto al valor del argumento prioridad

Despertador que recuerda los tiempos indicados por sus usuarios. Varios de ellos pueden indicar la misma hora

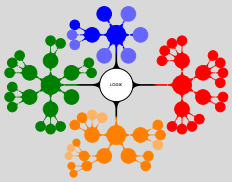
```
procedure tick();//cableada a INT CLK
  begin
    ahora:= ahora +1;
    despertar.signal();
  end;
begin
  ahora:= 0;
end;
```

`ahora` es una variable del sistema que se obtiene procesando el resultado de ejecutar la llamada `clock()` del sistema operativo



Señales con prioridad-II

```
monitor despertador;
  var
    ahora: Long_integer;
    despertar: cond; --prioritaria

  procedure despiertame(n: integer);
    var alarma: Long_integer;
    begin
      alarma:= ahora + n;
      while ahora< alarma do
        despertar.wait(n);
      end do;
      despertar.signal();
    end;
```

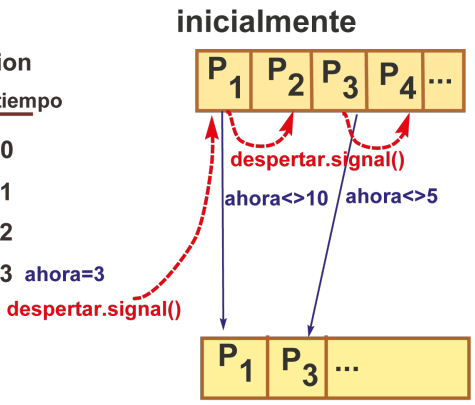
Señales con prioridad-III

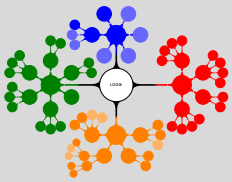
- Simulación mediante señales FIFO con semántica desplazante

```
Monitor despertador[d:tipo_enumerado]
 var
  ahora:int;
  despertar:cond;
 procedure despiertame(n:int)
  begin
   alarma:= ahora + n;
   while(ahora< alarma) do
    begin
     despertar.signal();
     despertar.wait();
    end;
   despertar.signal;
  end;
 procedure tick
   begin
    ahora:= ahora + 1;
    despertar.signal();
   end;
 begin
  ahora:= 0;
 end;
```

Escenario de ejecucion

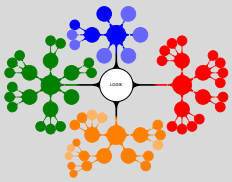
id	instruccion	tiempo
P1	despiertame(10)	0
P2	despiertame(2)	1
P3	despiertame(3)	2
P4	despiertame(0)	3 ahora=3





Todos los mecanismos de señalación alternativos de los monitores

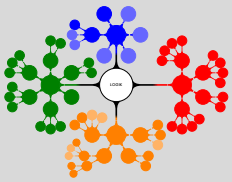
SA	señales automáticas	señal implícita
1 SC	señalar y continuar	señal explícita, no desplazante
2 SS	señalar y salir	señal explícita, desplazante, el proceso sale del monitor
3 SE	señalar y esperar	señal explícita, desplazante, el proceso señalador espera en la cola de entrada al monitor
4 SU	señales urgentes	señal explícita, desplazante, el proceso señalador espera en la cola de *procesos urgentes*



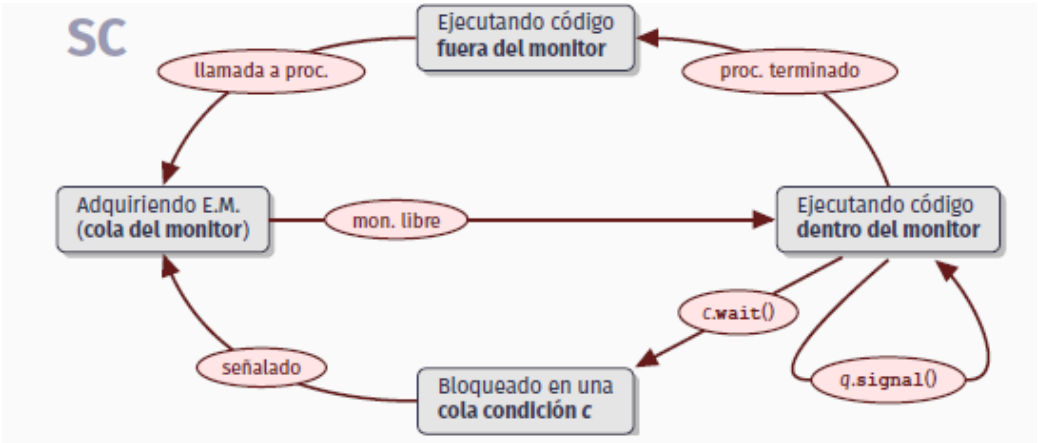
Posibles semánticas de las señales de los monitores

1. [SC] El proceso señalador continua su ejecución tras la operación `signal`

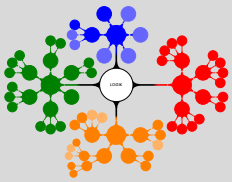
El proceso señalado espera bloqueado hasta que puede adquirir la E.M. de nuevo, semántica de señal:*SC: señalar y continuar*



Señalar y continuar (SC): diagrama de estados del proceso



- Señalador: continúa inmeditamente la ejecución de código del procedimiento del monitor tras `signal`
- Señalado: abandona la cola condición y espera en la cola del monitor hasta readquirir la E.M. y ejecutar código tras la operación `wait`

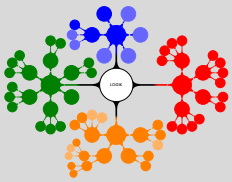


Señalar y continuar (SC): características

El proceso señalador continúa su ejecución dentro del monitor después del signal

- El proceso señalado abandonará después la cola condición y espera en la cola del monitor para readquirir la E.M.
- Tanto el señalador como otros procesos pueden hacer falsa la condición despues de que el señalado abandone la cola condición
- Por tanto, no se puede garantizar que la condición de sincronización asociada a **cond** sea cierta al volver **cond**.wait() y, lógicamente, es necesario volver a comprobarla entonces
- Esta semántica obliga a programar la operación wait en un bucle, de la siguiente manera:

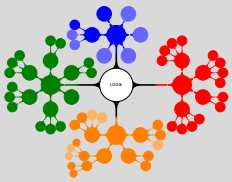
```
while not condicion_logica_sincronizacion do
        cond.wait() ;
```

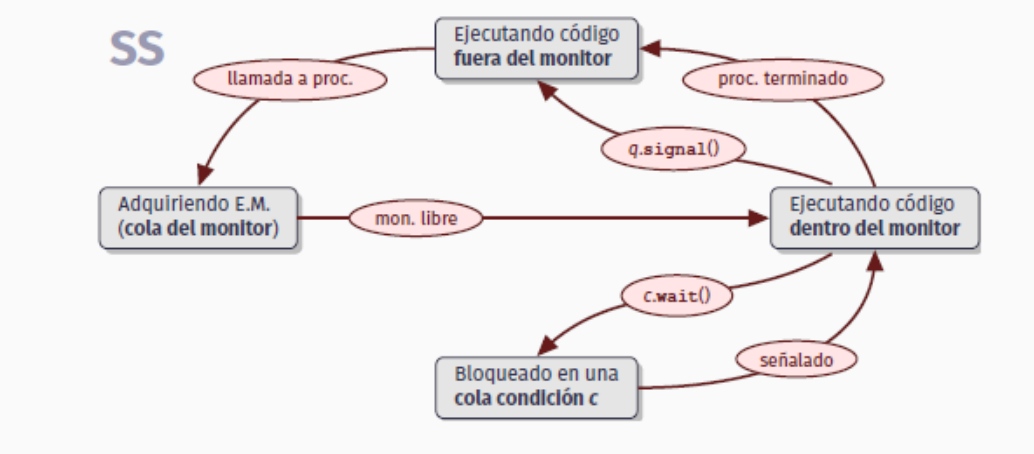
Señalar y Salir

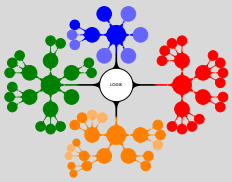
[SS] El proceso señalado se reactiva inmediatamente

2. El proceso señalador abandona el monitor tras hacer `signal`, sin ejecutar el código que haya después de dicho `signal`, la semática de señal:*SS: señalar y salir* admite varias implementaciones.



Señalar y salir (SS): diagrama de estados del proceso

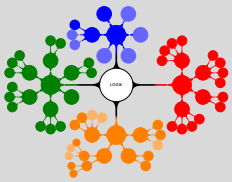




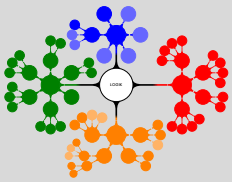
Señalar y salir (SS): características

El proceso señalador sale del monitor después de ejecutar cond.`signal()`

- Si hay código tras signal, no se ejecuta. El proceso señalado reanuda inmediatamente la ejecución de código del monitor.
- En ese caso, la operación signal conlleva:
  - Liberar al proceso señalado
  - Terminación del procedimiento del monitor que estaba ejecutando el proceso señalador porque se le obliga a salir del monitor (señales SS)
  - La condición de sincronización se mantiene hasta que el señalado continua su ejecución.
  - Esta semántica condiciona el estilo de programación:
    - operación signal como última instrucción de los procedimientos
    - o antes de programar una operación c.wait()

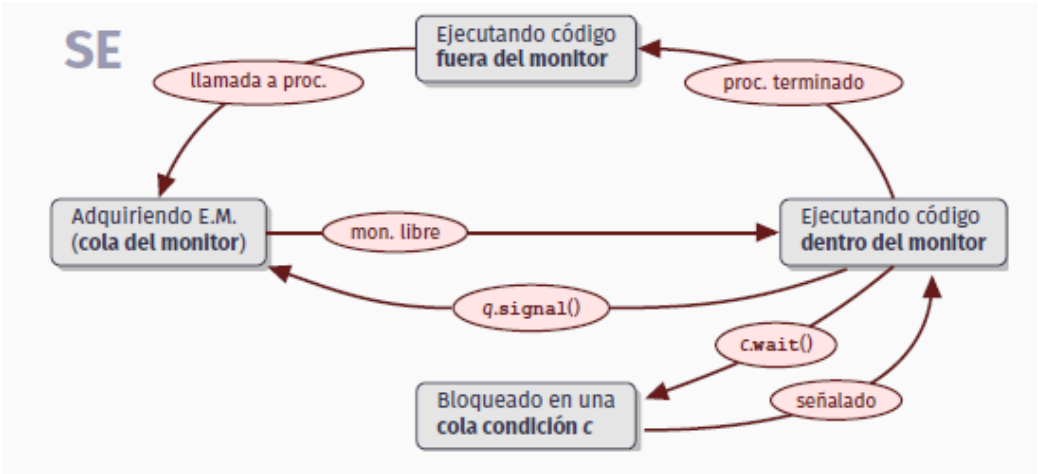


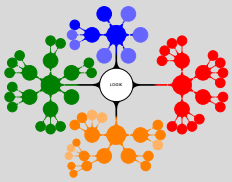
Señalar y esperar (SE)



3. **[SE] El proceso señalador se bloquea en la cola del monitor justo después de ejecutar** `signal()`
   - El proceso señalado entra de forma inmediata en el monitor
   - Está asegurada el cumplimiento de la condición de sincronización
     - El proceso señalador entra en la cola de procesos del monitor
     - Puede considerarse una semántica injusta respecto del progreso del proceso señalador

Señalar y esperar (SE): diagrama de estados del proceso

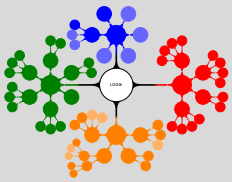




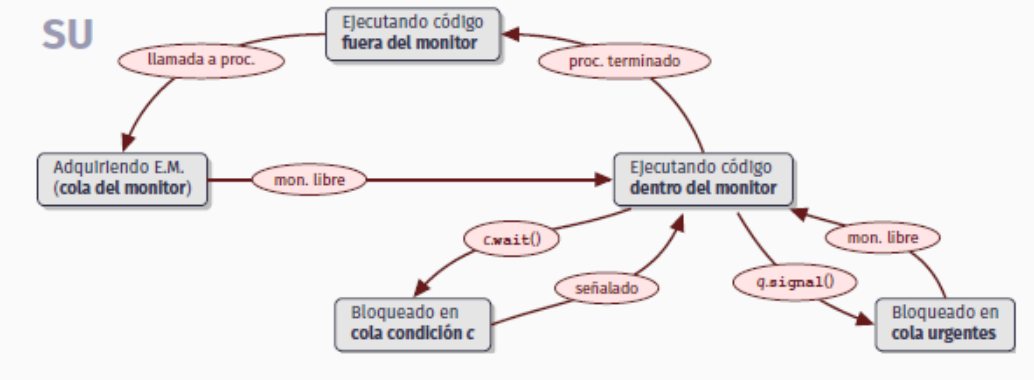
Señalar y espera urgente (SU): características

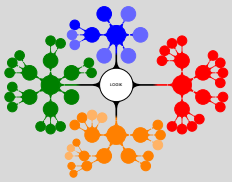
4. **[SU] Es similar la semántica SE, pero se intenta corregir el problema de falta de equidad de los señaladores**

   - El proceso señalador se bloquea justo después de ejecutar la operación `signal()`
     1. El proceso señalado entra de forma inmediata en el monitor
     2. El proceso señalador entra en una nueva cola de procesos del monitor prioritarios o cola de *procesos urgentes*
     3. Los procesos de la cola de procesos urgentes tienen preferencia cuando se queda libre el monitor



Señalar y espera urgente (SU): diagrama de estados del proceso





Procesos en la cola de urgentes

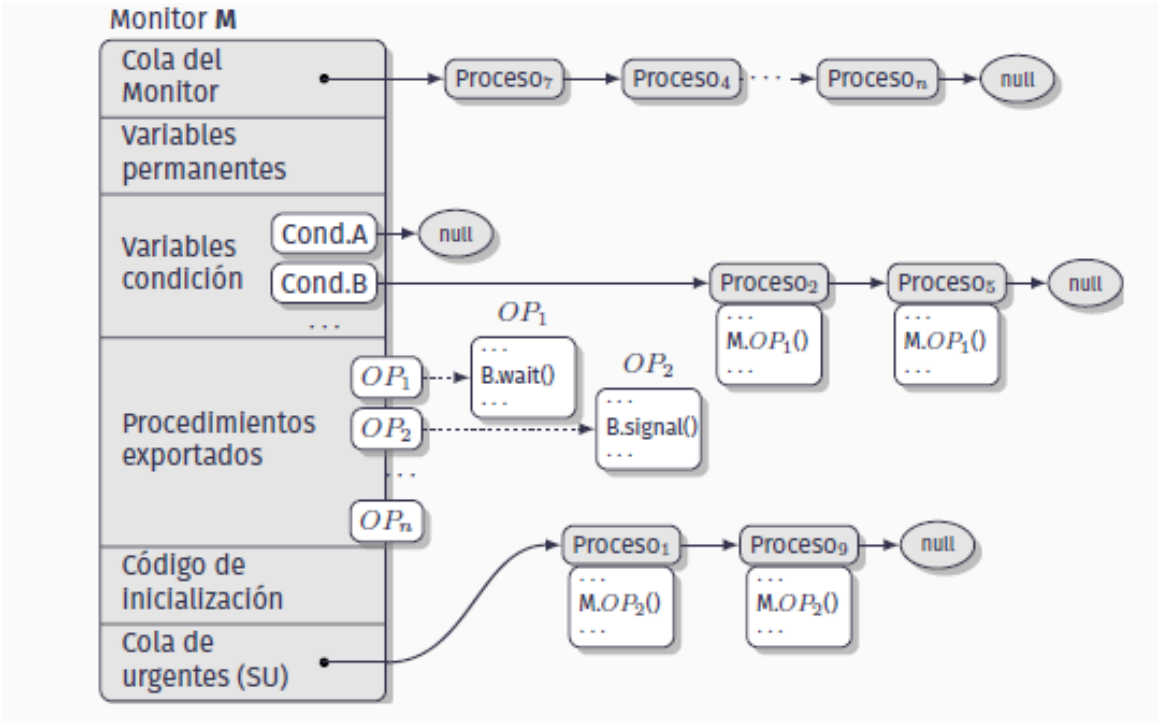
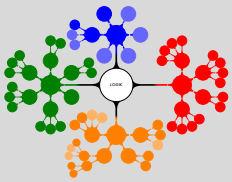


Figure: El proceso 1 y el 9 han ejecutado la op.2, que hace signal de la **cond**. B.



Análisis comparativo de las diferentes semánticas de señales

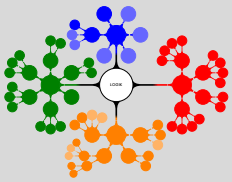
Potencia expresiva: todas las semánticas son capaces de resolver los mismos problemas

Facilidad de uso:

La semántica SS condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos

Eficiencia:

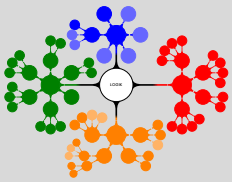
- Las semánticas SE y SU resultan ineficientes cuando no hay código tras `signal`
- La semántica SC también es un poco ineficiente al obligar a usar un bucle de comprobación para evitar el *robo de señal*



Comparativa entre las distintas semánticas de señales mediante un ejemplo

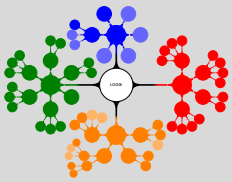
Barrera parcial

- El monitor tiene un único procedimiento público llamado *cita*
- Hay p procesos ejecutando un bucle indefinido, en cada iteración realizan una actividad de duración arbitraria y después llaman a *cita*
- Ningún proceso termina *cita* antes de que haya al menos n de ellos que la hayan iniciado (donde 1 < n < p). Después de esperar en cita, pero antes de terminarla, el proceso imprime un mensaje
- Cada vez que un grupo de n procesos llegan a la cita, esos n procesos imprimen su mensaje antes de que lo haga ningún otro proceso que haya llegado después de todos ellos a dicha cita (que sea del siguiente grupo de n)



Una posible implementación del monitor

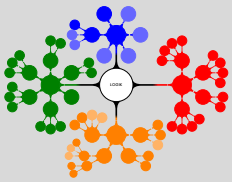
```
Monitor BP //monitor Barrera Parcial }
  var cola : condition;//procesos esperando contador==n
      contador : integer;//numero de procesos ejecutando
          cita
procedure cita() ;
  begin
   contador := contador+1;//registrar un proceso mas en el
        estado "ejecutando cita"
   if (contador<n) then
       cola.wait();//esperar a que haya n procesos en el
           estado "ejecutando cita"
      else begin //si ya hay n procesos ejecutando la cita
         for i := 1 to n-1 do //para cada uno de estos
           cola.signal();//despertalo
         contador := 0;//volver a poner el contador a 0
      end
      print("salgo de cita");//mensaje de salida
  end
begin//inicializacion del monitor
contador := 0 ; { inicialmente, no hay procesos en cita }
```

Comportamiento del monitor *barrera parcial* para las distintas semánticas de señales

Si llamamos último al último proceso en llegar a la cita de cada grupo de n (el que observa contador==n), ocurrirá lo siguiente:

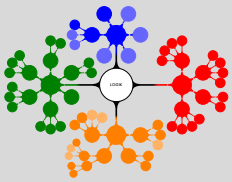
- *Señalar y Continuar:* Los n - 1 procesos señalados abandonan el `wait`, pero pasan a la cola del monitor. Los procesos del grupo de la siguiente cita se podrían adelantar.
- *Señalar y Salir:* en este caso, el último proceso abandona el monitor , no pone `contador` a 0.



Comportamiento del monitor *barrera parcial* para las distintas semánticas de señales-II

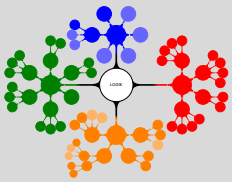
...

- *Señalar y Esperar:* Similar a la semántica SS, no es correcta tampoco esta solución.
- *Señalar y Espera Urgente:* el último proceso ejecuta los `signals` a los `n-1` procesos restantes. Entre cada dos de ellos, espera en la *cola de urgentes* a que el proceso recién señalado abandone el monitor.
- Cuando no quedan procesos bloqueados que señalar, el último proceso vuelve a entrar en el monitor y ejecuta la instrucción `contador:=0`, antes que entre otro proceso al monitor.
- La semántica SU de señales hace que esta solución sea correcta



Implementación alternativa del monitor *barrera parcial*

```
Monitor BP
var cola:{\bf cond}ition;//procesos esperando contador==n
    contador:integer;//numero de procesos esperando en la
        cola
  procedure cita() ;
   begin
    contador:=contador+1;//registrar un proceso mas
        esperando
    if (contador<n) then//todavia no hay n procesos:
      cola.wait();//esperar a que los haya
    contador:=contador-1;//registrar un proceso menos
        esperando
    print("salgo de la cita");//mensaje de salida
    if (contador>0) then//si hay otros procesos en la cola
      cola.signal();//despertar al siguiente
   end
begin//inicialización:
  contador:=0;//inicialmente, no hay procesos en la cola
end;
```

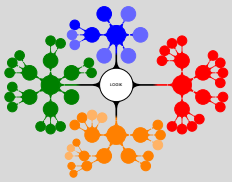



Comportamiento de la versión alternativa

Analizamos el comportamiento en todas las semánticas:

- No funciona con la semántica SC.
- Sí funciona con el resto de semánticas (SE,SS,SU).

En general, hay que ser cuidadoso con la semántica en uso, especialmente si el monitor tiene código tras `signal()`. Generalmente, la semántica SC puede complicar mucho los diseños



Reglas de verificación de las señales no-desplazantes SC

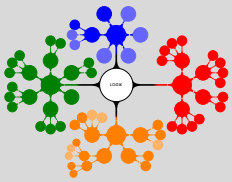
Axioma de la operación `c.wait()`

$$\{\texttt{IM} \wedge \texttt{L}\}\ \texttt{c.wait()}\ \{\texttt{IM} \wedge \texttt{L}\}$$

- La regla permite demostrar la corrección parcial de los programas independientemente de la planificación de los procesos de la cola `c`
- No demuestra, por tanto, por tanto ni la propiedad de vivacidad, ni detecta bloqueos

Axioma de las operaciones `c.signal(), c.signal_all()`

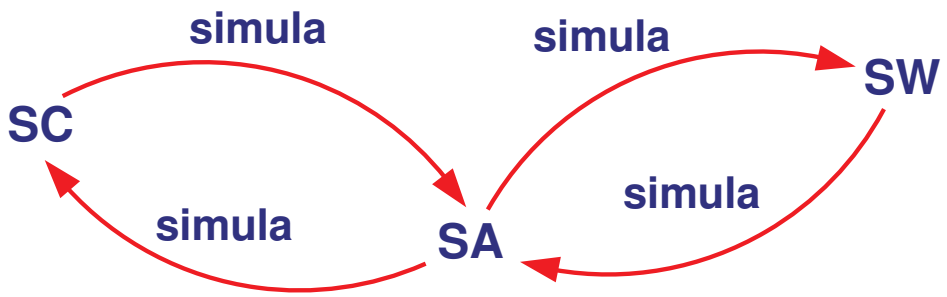
$$\{\texttt{P}\}\ \texttt{c.signal()}\ \{\texttt{P}\}$$

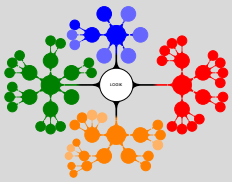


Intercambio de señales en programas que usan monitores

condiciones que se han de cumplir para poder intercambiar SW y SC sin modificar adicionalmente el código del monitor:

1. Sólo se ha exigir como post**cond**ición de `c.wait()` el Invariante del Monitor
2. Después de una llamada a la operación `c.signal()` se ha de *salir* del monitor
3. No se puede utilizar `c.signal_all()`: difusión de una señal a un grupo de procesos

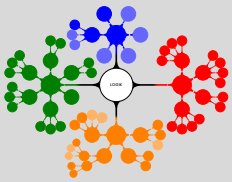




Implementaciones alternativas de un semáforo con monitores

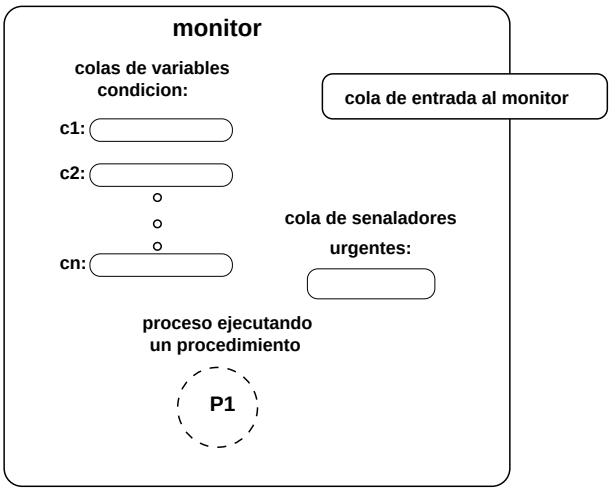
```
Monitor semaforo_FIFO1;
 {IM: s>=0}
var c: cond;
    s: int;
procedure P;
 begin
  if(s=0) then
      c.wait();
      {s > 0}
   s:=s-1;
 end;

procedure V;
 begin
  s:= s+1;
  c.signal;
 end;
begin
  s:=0;
end;
```

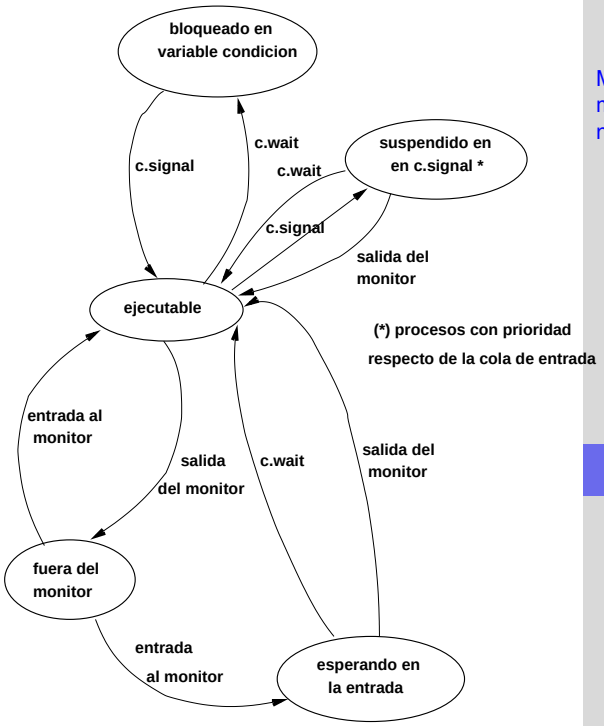
```
Monitor semaforo_FIFO2;
 {IM: s>=0}
var c: cond;
    s: int;
procedure P;
 begin
  while(s=0) do
      c.wait();
      {s>= 0}
   end do;
   {s > 0}
   s:=s-1;
 end;

procedure V;
 begin
  c.signal;
  s:= s+1;
 end;
begin
  s:=0;
end;
```

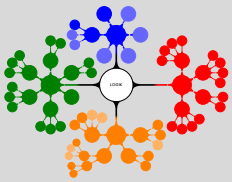



Implementación de los monitores

Esquema de implementación de un monitor con señales SU



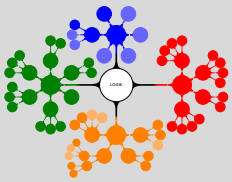




Implementación de los monitores con semáforos

Concepto fundamental:

- Cola de entrada al monitor: controlada por el semáforo mutex
- Cola de procesos urgentes: controlada por el semáforo next
- Número de procesos *urgentes* en cola: se contabiliza en la variable next_count
- Colas de procesos bloqueados en cada condición: controladas por el semáforo asociado a cada condición x_sem y el número de procesos en cada cola se contabiliza en una variable asociada a cada condición (x_sem_count)

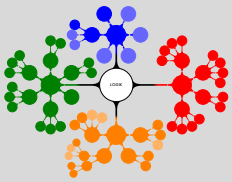


Implementación de los monitores con semáforos-II

Semáforo mutex para implementar la exclusión mutua del monitor

```
procedure P1(...)
begin
sem_wait(mutex);
{ cuerpo del procedimiento }
sem_signal(mutex);
end
```

```
//inicializacion
mutex := 1 ;
```

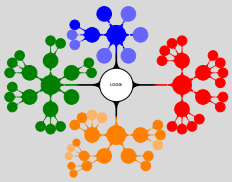



Implementación de los monitores con semáforos-III

Semáforo next para implementar la cola de *urgentes* y next_count para contar los procesos en esa cola

```
procedure P1(...)
begin
sem_wait(mutex);
{ cuerpo del procedimiento }
if (next_count > 0) then
      sem_signal(next);
   else sem_signal(mutex);
end;
```

```
//inicializacion
next := 0 ;
next_count :=0 ;
```

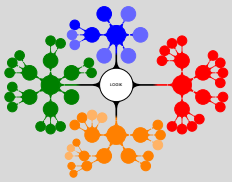



Implementación de los monitores con semáforos-IV

Para implementar las variables condición: semáforo x_sem para cada una y una variable para contar los procesos bloqueados en su cola asociada: x_sem_count

```
void entrada(){
//implementacion de entrada al monitor
sem_wait(mutex); //entre al monitor o se queda bloqueado
    en la cola de entrada
}
```

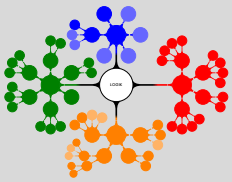
```
void x_wait(Semaphore x_sem, unsigned x_sem_count){
//implementacion de x.wait()
x_sem_count := x_sem_count + 1 ;//cuenta 1 proceso mas
    bloqueado en la cola de condicion "x"
if (next_count <> 0) then//hay procesos señaladores
    esperando
    sem_signal(next); //desbloquea un proceso señalador
   else
    sem_signal(mutex);  //deja libre el monitor para que
        entre otro proceso
sem_wait(x_sem);//se bloquea esperando la certeza de
    condicion "x"
x_sem_count := x_sem_count - 1;//cuenta 1 proceso menos
    bloqueado en la condicion "x"
}
```

Implementación de los monitores con semáforos-V

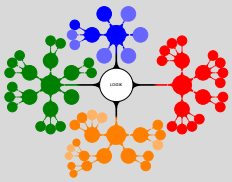
```
void x_signal(Semaphore x_sem, unsigned x_sem_count){
//implementacion de x.signal()
if (x_sem_count <> 0) then
  begin //hay procesos bloqueados esperando la condicion "
      x"
    next_count := next_count + 1;//cuenta 1 proceso mas en
        la cola de señaladores
    sem_signal(x_sem); //desbloquea 1 proceso esperando:
        la condicion "x" es cierta ahora
    sem_wait(next); //entra en la cola de señaladores
    next_count := next_count - 1;//cuenta 1 proceso
        señalador menos en cola de señaladores
  end
}
```

```
void salida(){
//implementacion de la salida del monitor
if (next_count <> 0) then//hay procesos senialadores
    esperando
     sem_signal(next); //desbloquea un proceso senialador
  else
     sem_signal(mutex);  //libera la exclusion mutua del
         monitor
}
```

Ejemplo de uso de la implementación del Monitor

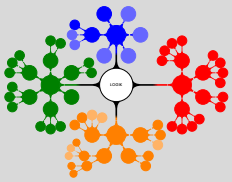
```
Semaphore puede_escribir= 0, puede_leer= 0;
//Monitor PC ;
int valor_com,puede_escribir_count,puede_leer_count;
boolean pendiente;
  procedure escribir(puede_escribir,puede_escribir_count);
   begin entrada();
    if pendiente then
      x_wait(puede_escribir, puede_escribir_count);
    valor_com:=v; pendiente:=true;
    x_signal(puede_leer, puede_leer_count);
    salida();
   end;
  function leer(puede_leer,puede_leer_count):integer;
   begin entrada();
   if (not pendiente) then
     x_wait(puede_leer, puede_leer_count);
   result:=valor_com; pendiente:=false ;
   x_signal(puede_escribir, puede_escribir_count);
   salida();
  end;
  begin entrada();
    pendiente := false ;
    valor_com=0;puede_escribir_count=0;puede_leer_count=0;
    salida();
  end;
```

Bibliografía

Para más información, ejercicios, además de la bibliografía fundamental, se puede consultar:

- Concurrent Programming, Andrews (1991), capítulo 6.
- Programación Concurrente y en Tiempo Real, Capel (2022), capítulo 2.



1.4. Tema 2.2

Tema 2

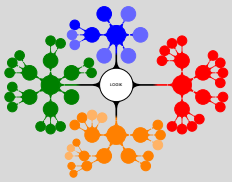
Sincronización en memoria compartida

SCD para GIIM

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 11 Octubre, 2024

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



Introducción al problema del “exclusión mutua”

Introducción histórica al problema de la exclusión mutua

1962 Dekker propone el problema de la exclusion mutua para multiprocesadores:

"disenar un protocolo que garantice el acceso mutuamente excluyente, sin que exista interbloqueo, a una seccion critica por parte de un determinado numero de procesos que compiten por entrar a dicha seccion..."

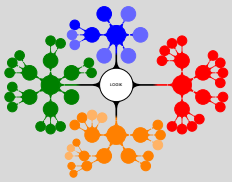
1965 Dijkstra propone una solucion segura, libre de interbloqueo, pero que puede producir inanicion

1966 Knuth propone una solucion sin inanicion; garantiza retraso limitado de los procesos, pero no FIFO

1974 Lamport: permite a los procesos "detenerse" en la ejecucion del protocolo de adquisicion, solapar las operaciones de lectura con la escritura y retraso FIFO de los procesos que ya esperan entrar

1981 Peterson propone una solucion equitativa para 2 y "n" procesos; garantiza el retraso cuadratico de los procesos, es la solucion mas simple hasta fecha para multiprocesadores

1983 Algoritmos totalmente distribuidos que resuelven el problema para multicomputadores; Ricart-Aggrawala, Suzuki-Kasami

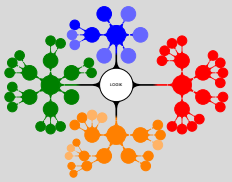


Solución al problema con bucles de espera activa

- Los procesos iteran en un bucle vacío hasta que la entrada en Sección Crítica (SC) sea segura
- Aceptable si el sistema/aplicación no tuviera muchos procesos

Condiciones de Dijkstra para obtener una solución *parcialmente correcta* al problema de exclusión mutua:

1. No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por el multiprocesador
2. Ni tampoco acerca de la velocidad de ejecución de los procesos, excepto que no es cero (*Progreso Finito*)
3. Cuando un proceso se encuentra ejecutando código fuera de la sección crítica no puede impedir a los otros procesos entrar en ésta
4. La sección crítica siempre será alcanzada por alguno de los procesos que esperan entrar



Método de refinamiento sucesivo

Esquema de "corrutinas": no se cumple la tercera propiedad de Dijkstra!

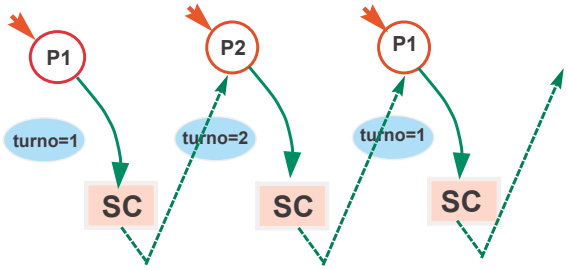
1A Etapa

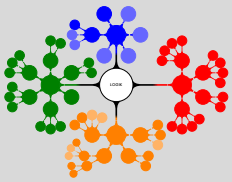
Proceso P1

```
while true do
 begin
  <<resto instrucciones>>

  while turno <> 1 do
    nothing;
  enddo;

  <<seccion critica>>
  turno:= 2;
 end
enddo;
```

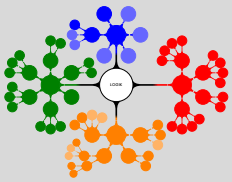
Proceso P2

```
while true do
 begin
  <<resto instrucciones>>

  while turno <> 2 do
    nothing;
  enddo;

  <<seccion critica>>
  turno:= 1;
 end
enddo;
```





Método de refinamiento sucesivo –II



2A Etapa

Proceso P1

```
while true do
 begin
  <<resto instrucciones>>
  while c2=0 do
    nothing;
  enddo;
  c1:= 0;
  <<seccion critica>>
  c1:= 1;
 end
enddo;
```

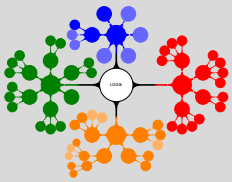
Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  while c1= 0 do
    nothing;
  enddo;
  c2:= 0;
  <<seccion critica>>
  c2:= 1;
 end
enddo;
```

La salida de la espera activa y el cambio de la clave no se realizan atomicamente

Problema!: no se cumple la propiedad de seguridad del algoritmo.

Método de refinamiento sucesivo –III



3A Etapa

Adelantando la asignacion de la clave la solucion es segura

Proceso P1

```
while true do
 begin
  <<resto instrucciones>>
  c1:= 0;

  while c2=0 do
    nothing;
  enddo;

  <<seccion critica>>
  c1:= 1;

 end
enddo;
```

Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  c2:= 0;

  while c1= 0 do
    nothing;
  enddo;

  <<seccion critica>>
  c2:= 1;
 end
enddo;
```

Nuevo problema!: el proceso que modifica la clave no sabe si el otro hace lo mismo concurrentemente con el

Método de refinamiento sucesivo –IV

4A Etapa:

Proceso P1

```
while true do
 begin
  <<resto instrucciones>>
  c1:= 0;

  while c2=0 do
     begin
      c1:= 1;
      while c2= 0 do
        nothing;
       enddo;
      c1:= 0;
     end
  enddo;
  <<seccion critica>>
  c1:= 1;
 end
enddo;
```

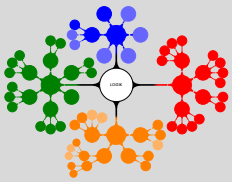
Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  c2:= 0;

  while c1= 0 do
     begin
      c2:= 1;
      while c1= 0 do
        nothing;
      enddo;
      c2:= 0;
     end
  enddo;
  <<seccion critica>>
  c2:= 1;
 end
enddo;
```

Para indicar que intenta entrar en S.C., cambia su clave

Comprueba la clave del otro; la vuelve a cambiar si el otro tambien intenta entrar,

Método de refinamiento sucesivo –V



5A Etapa: Algoritmo de Dekker

Proceso P1

```
while true do
 begin
  <<resto instrucciones>>
  c1:= 0;
  while c2= 0 do
    if turno= 2 then
     begin
      c1:= 1;
      while turno= 2 do
        nothing;
      enddo;
      c1:= 0;
     end
    endif
  enddo;
  <<seccion critica>>
  turno:= 2;
  c1:= 1;
 end
enddo;
```

El proceso intenta entrar en S.C.

comprueba la clave del otro

si no tiene el turno hace espera activa, despues de cambiar su clave

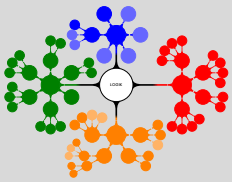
Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  c2:= 0;
  while c1= 0 do
    if turno= 1 then
     begin
      c2:= 1;
      while turno= 1 do
        nothing;
      enddo;
      c2:= 0;
     end
    endif
  enddo;
  <<seccion critica>>
  turno:= 1;
  c2:= 1;
 end
enddo;
```

Método de refinamiento sucesivo: verificación de propiedades

Verificación de propiedades de seguridad Exclusión mutua:

1. P_i entra en sección crítica sólo si $c[j] == 1$
2. P_i comprueba la clave del otro, $c[j]$, sólo después de asignar su propia clave Luego, cuando P_i entra se cumple $c[j] == 1 \wedge c[i] == 0$



Método de refinamiento sucesivo: verificación de propiedades –II

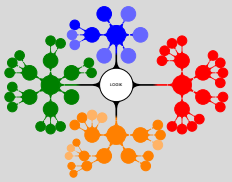
Verificación de propiedades de seguridad

Alcanzabilidad de la sección crítica:

Si P_i y P_j intentan entrar en sección crítica y `turno == i`:

1. si P_i encuentra la clave del otro c[j] == 1, entonces P_i entra;
2. si no, dependerá de quien tenga el turno:
   1. si `turno == i` espera que P_j cambie su clave y, después, entra
   2. si `turno == j` cambia su clave a 1 y se queda en espera activa

Discusión sobre la *equidad* de la solución dada por el Algoritmo de Dekker



Generalización a N procesos: Algoritmo de Dijkstra

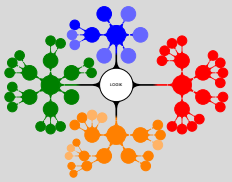
```
c: array[0..n-1] of (pasivo,solicitando,en_SC)
turno: 0.. n-1;
repeat
  repeat
    E2:c[i]:= solicitando;
    while turno <> i do
    E3:if c[turno]= pasivo
         then turno:= i
       endif;
    enddo;
    E4:c[i]:= en_SC;
    j:= 0;
    while(j<n) and (j=i or c[j] <> en_SC) do
      j:= j+1;
    enddo;
  until j>= n;
    <<Seccion Critica>>
E1:c[i]:= pasivo;
    <<Resto de instrucciones>>
until false
```

1A barrera detiene a los procesos si el que posee el turno no esta pasivo

La comprobacion del estado del que tiene el turno y el cambio de este no se hace atomicamente

Si un grupo de procesos se ve obligado a ciclar nuevamente, el que posee el turno no puede estar pasivo

2A barrera asegura que se cumple la propiedad de seguridad



Verificación de las propiedades del A. Dijkstra

Verificación de las propiedades de seguridad

Exclusión mutua:

demostración similar a la del A. Dekker

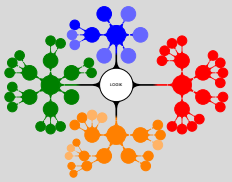
Alcanzabilidad de la sección crítica:

1. `turno` es una variable compartida, mantendrá el valor `i` del último P_i que lo asigne

2. Sean $\{P_l \ldots P_i \ldots P_m\}$ tales que `c[i]` $=$ `en_SC` y `turno ==k` con $l \leq k \leq m$, entonces

   P_k entrará en su sección en tiempo finito y el resto

   $P_i : l \leq i \leq m \wedge i \neq k$ se quedará ciclando en el primer bucle (1A) del protocolo



Verificación de las propiedades de vivacidad del A. Dijkstra

El A. Dijkstra satisface seguridad pero no evita el peligro de inanición de los procesos del programa

posición/acción	c[1]	c[2]	c[3]	turno
Inic.:P_1, P_2, P_3 en E1	pasivo	pasivo	pasivo	3
$P_1 : E_1 \rightarrow E_2$	solicitando	pasivo	pasivo	3
$P_2 : E_1 \rightarrow E_2$	solicitando	solicitando	pasivo	3
$P_1 : E_2 \rightarrow E_3$	solicitando	solicitando	pasivo	3
$P_2 : E_2 \rightarrow E_3$	solicitando	solicitando	pasivo	3
$P_1 : E_3 \rightarrow E_4$	en_SC	solicitando	pasivo	3
$P_2 : E_3 \rightarrow E_4$	en_SC	en_SC	pasivo	3
...	...	...	...	...

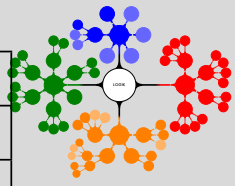
```
c: array[0..n-1] of (pasivo,solicitando,en_SC)
turno: 0.. n-1;
repeat
  repeat
    E2:c[i]:= solicitando;
    while turno <> i do
    E3:if c[turno]= pasivo
         then turno:= i
       endif;
    enddo;
    E4:c[i]:= en_SC;
    j:= 0;
    while(j<n) and (j=i or c[j] <> en_SC) do
      j:= j+1;
    enddo;
  until j>= n;
    <<Seccion Critica>>
E1:c[i]:= pasivo;
  <<Resto de instrucciones>>
until false
```

1A barrera detiene a los procesos si el que posee el turno no esta pasivo

La comprobacion del estado del que tiene el turno y el cambio de este no se hace atomicamente

Si un grupo de procesos se ve obligado a ciclar nuevamente, el que posee el turno no puede estar pasivo

2A barrera asegura que se cumple la propiedad de seguridad



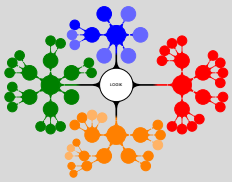
Algoritmo de Knuth para N procesos

```
                c: array[0..n-1] of (pasivo,solicitando,en_SC)
                turno: 0.. n-1;
repeat
  repeat
   E0: c[i]:= solicitando;
       j:= turno; --variable local
   E1: while j <> i do
        if c[j] <> pasivo then j:= turno
          else j:= (j-1) MOD n
        endif;
       enddo;
   E2: c[i]:= en_SC;
       k:= 0
       while (k<n) and (k=i pr c[k]<>en_SC) do
         k:= k+1;
       enddo;
  until k>= n;
   E3: turno:= i;
   << Seccion Critica >>
       turno:= (i-1) MOD n;
   E4: c[i]:= pasivo;
   E5: <<resto de instrucciones>>
until false;
```

1A barrera detiene a los procesos si el que posee el turno no esta pasivo

2A barrera asegura que se cumple la propiedad de seguridad

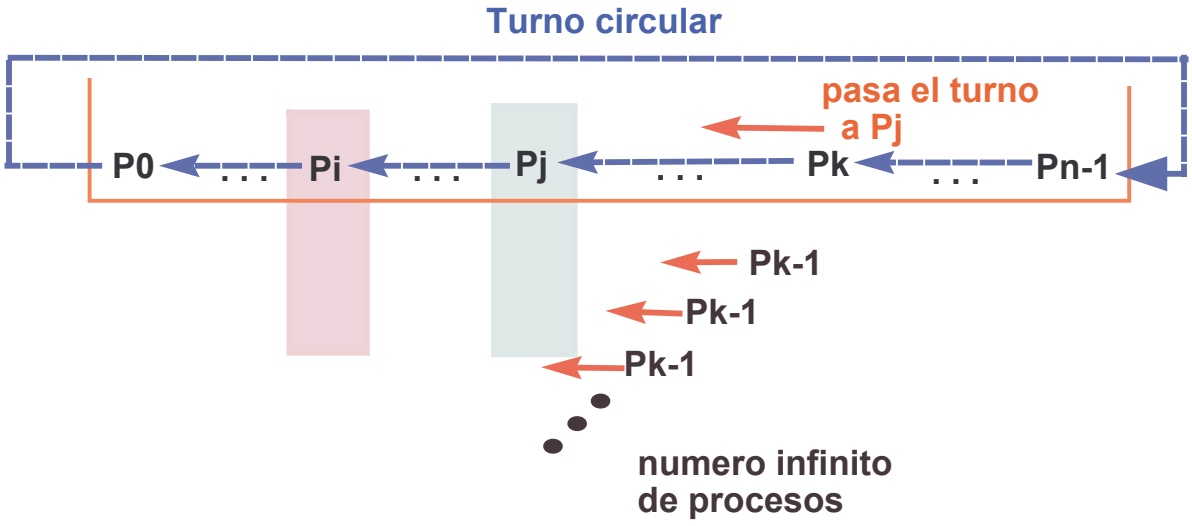
Si un grupo de procesos se ve obligado a ciclar nuevamente, el que posee el turno no puede estar pasivo

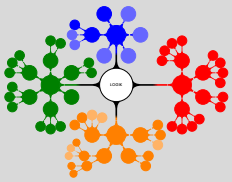


Algoritmo de Knuth para N procesos –II

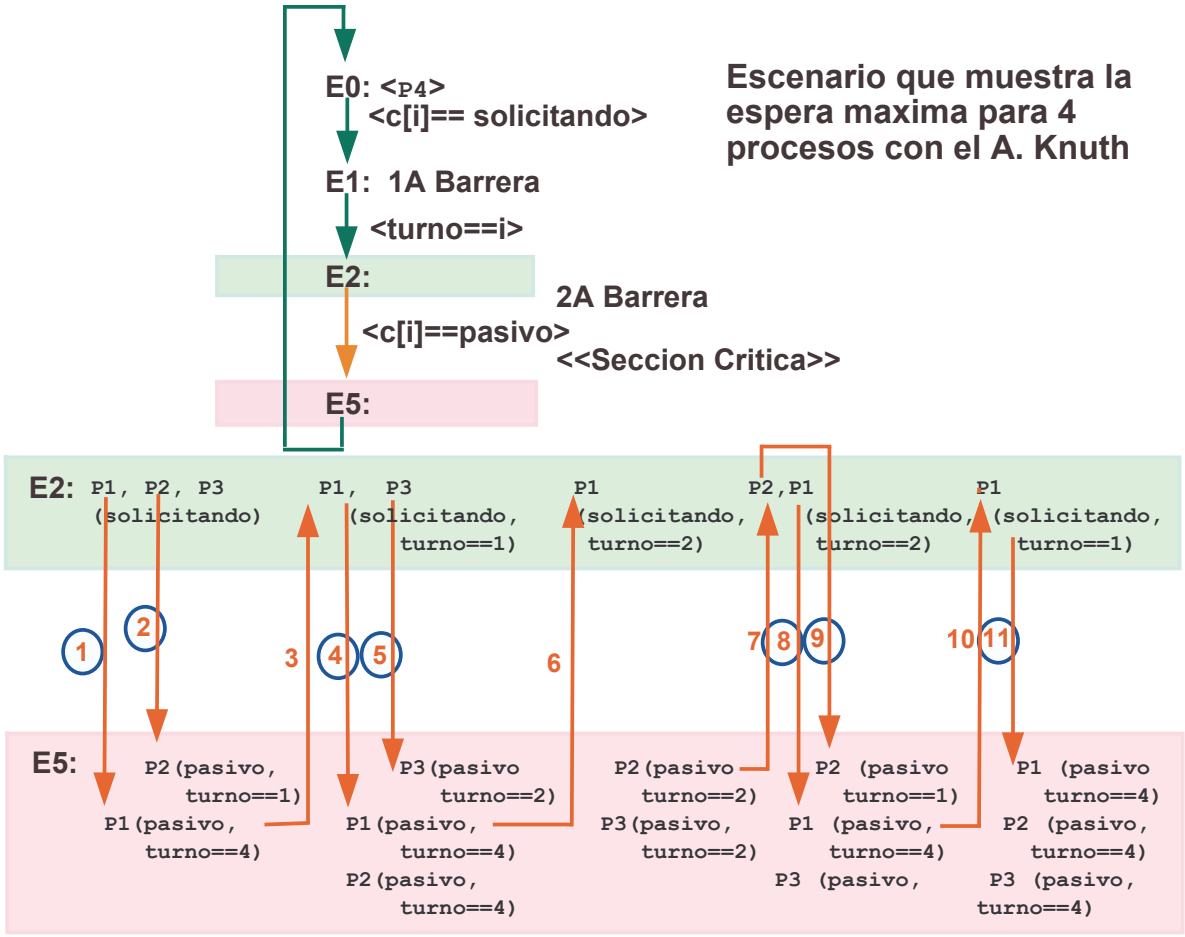
Imposibilidad de la inanición de los procesos si se supone que existe un número finito de ellos en el algoritmo

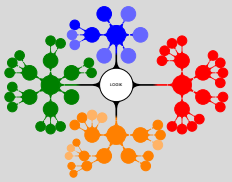
Escenario de inanicion: Pj se adelanta continuamente a Pi





Algoritmo de Knuth para N procesos –III

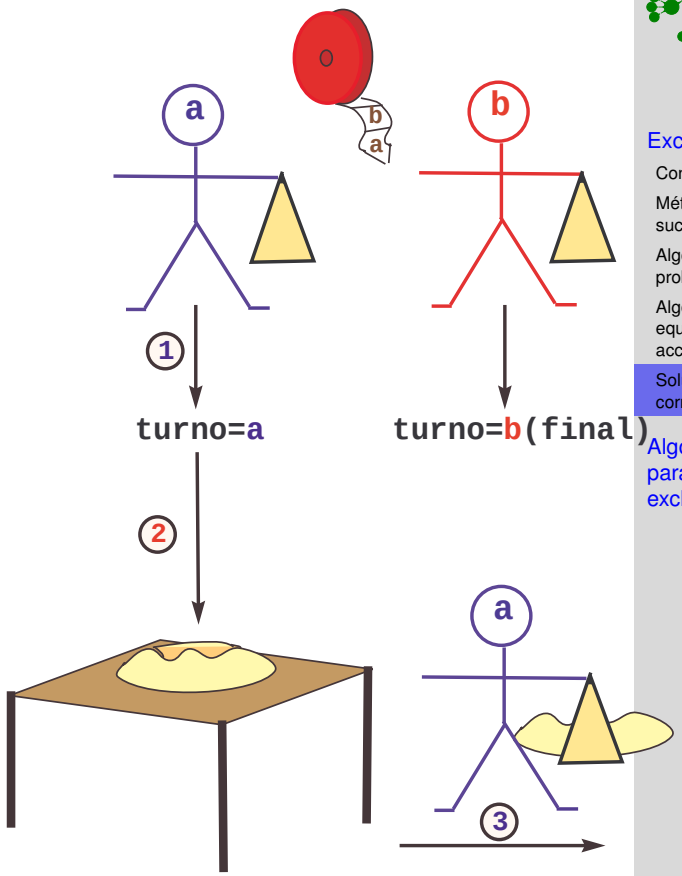


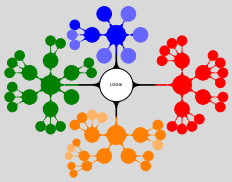


Algoritmo de Peterson

Solución para *2* procesos

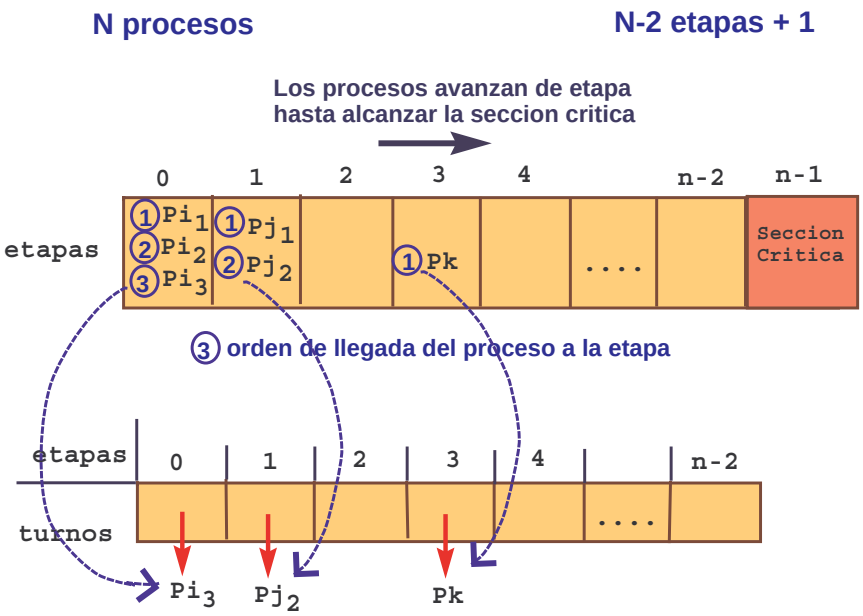
```
var
  solicitado: array[0..1] of boolean;
  turno: 0..1;

Pi::=
 ...
 solicitado[i]:= true; --j=2, i=1
 turno:= i;
 while (solicitado[j] and turno=i) do
  nothing;
 enddo;
 <<seccion critica>>
 solicitado[i]:= false;
 ...
```





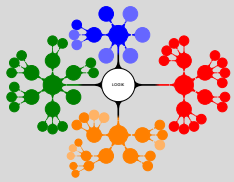
Algoritmo de Peterson-II

Solución para *N* procesos - Idea general:



Variables compartidas entre los procesos:

```
type etapas= -1..n-2; var c: array[0..n-1] of etapas;
    procesos= 0..n-1;     turno: array[0..n-2] of procesos;
```

Algoritmo de Peterson-III

Solución para *N* procesos

```
Pi
        . . .
while true do
 begin
 <<resto de instrucciones>>
for j=0 to n-2 do
 begin
  c[i]:= j;
  turno[j]:= i;
  while ( (Exists k <> i: c[k] >= j) && turno[j] = i) do
     nothing;
 end;
enddo;

c[i]:= n-1; -- metainstruccion
 <<seccion critica>>
c[i}:= -1;
```

Variables compartidas entre los procesos:

```
var c: array[0..n-1] of -1..n-2;
    turno: array[0..n-2] of 0..n-1;
```

El proceso en etapa j

Bucle de asignacion de etapas a procesos

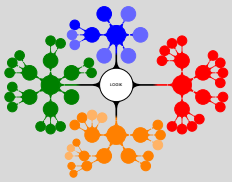
El ultimo en llegar se queda con el turno

Indica que se ha llegado a la S.C.; es una etapa ficticia

Etapa inicial de los procesos

(Exists k <> i: c[k] >= j) && turno[j] = i

No se cumple la condicion si
(a) el proceso esta en la etapa mas avanzada
o (b) ha llegado otro proceso despues a la etapa



Verificación de las propiedades del Algoritmo de Peterson

Se dice que `P_i` precede a un proceso `P_j` sii `c[i] > c[j]`

L1 Un proceso que precede a todos los demás puede avanzar al menos una etapa

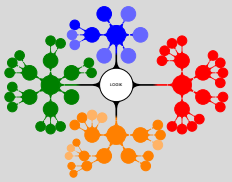
(Exists k <> i: c[k] >= j) && turno[j] = i

Ya que no se cumple la condicion del segundo bucle si:

(a) el proceso esta en la etapa mas avanzada

o (b) ha llegado otro proceso despues a la etapa

- El proceso P_i puede ser adelantado en la etapa siguiente
- Podrían llegar más de un proceso a la etapa `j`
- Pero siempre se cumplirá que P_i avanzará

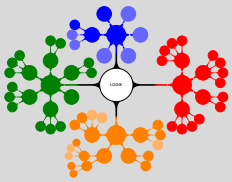


Verificación de las propiedades del Algoritmo de Peterson-II

L2 Un proceso que pasa de la etapa `j` a la `j+1` ha de verificar alguna de estas condiciones:

1. *Precede a todos los demás*
2. *No estaba solo en la etapa j*

- La condición (1) nos sitúa en las condiciones de aplicar el **Lema 1**
- Si (2) $\Rightarrow$ `turno[j] <> i`, luego al proceso se le unió otro
  - Podría suceder que, justo cuando el proceso vaya a avanzar de etapa
  - porque se cumple la condición (1), se le una otro proceso a su etapa.
  - Pero también se cumplirá



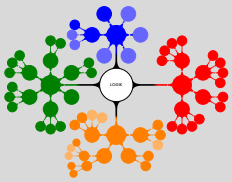
Verificación de las propiedades del Algoritmo de Peterson-III

L3 Si existe al menos dos procesos en la etapa `j`, entonces existe al menos un procesoen cada una de las etapas anteriores

- *La demostración se hace por inducción sobre la variable que representa la etapa j*

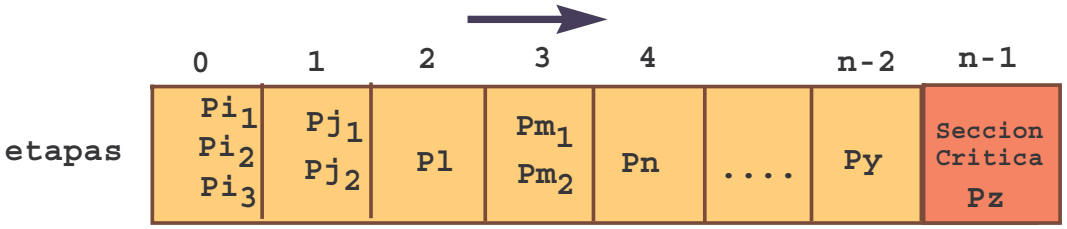
L4 El número máximo de procesos que puede haber en la etapa `j` es `n-j`, con $0 \leq j \leq n-2$

- *La demostración se hace aplicando el* **Lema 3**
- *Por tanto, a la etapa* `n-2` *llegarán como máximo 2 procesos*

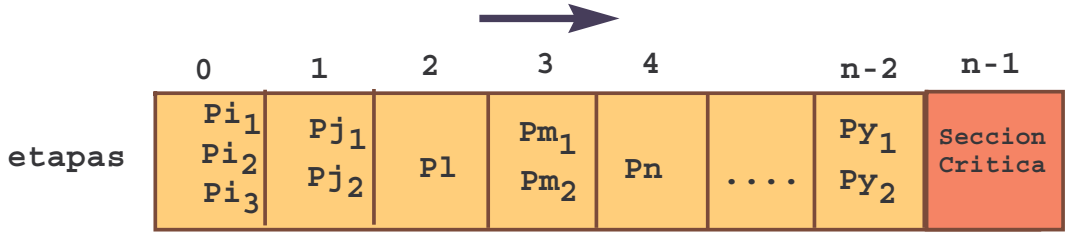


Verificación de las propiedades de seguridad del A. Peterson-IV

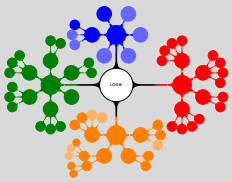
El algoritmo de Peterson cumple con la exclusion mutua en el acceso a la seccion critica



Py no puede avanzar a la siguiente etapa mientras la seccion critica este ocupada (segun las condiciones del Lema 2)

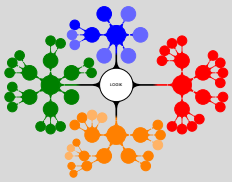


Segun el Lema 2, solo 1 de los 2 procesos en la etapa (n-2) podra avanzar a la seccion critica



Algoritmo de Peterson-IV

```
var c: array[0..N-1] of -1..N-2;
   turno: array[0..N-2] of 0..N-1;
while (true) do
  begin
   Resto de las instrucciones;
   (1) for j=0 TO N-2 do
   (2)   begin
   (3)     c[i]:= j;
   (4)     turno[j]:= i;
   (5)     for k=0 TO N-1 do
   (6)       begin
   (7)         if (k=i) then continue;
   (8)         while(c[k]>=j and turno[j]=i) do
   (9)          nothing;
   (10)         enddo;
   (11)      end;
   (12)   end;
   (13) c[i]:= n-1; /*meta-instruccion*/
   (14) <seccion critica>
   (15) c[i]:= -1
  end
```

Verificación de propiedades concurrentes adicionales

I. Alcanzabilidad de la SC:

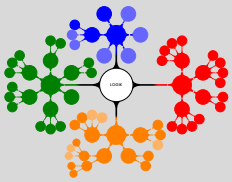
demostración por *reducción al absurdo*

- Todos los procesos se quedan bloqueados al llegar a una etapa y no avanzan más (hipótesis de incorrección)
  1. El proceso precede a los demás $\Rightarrow$ contradice el **Lema 1**
  2. Si no, el proceso llega a una etapa ocupada al menos por otro proceso $\Rightarrow$ se contradice el **Lema 2**

II. Propiedad de equidad

demostración:

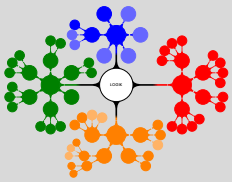
- El número máximo de turnos que un proceso cualquiera tendría que esperar con el algoritmo de Peterson es de
  $r(n) = n - 1 + r(n-1) = \frac{n \times (n-1)}{2}$ turnos



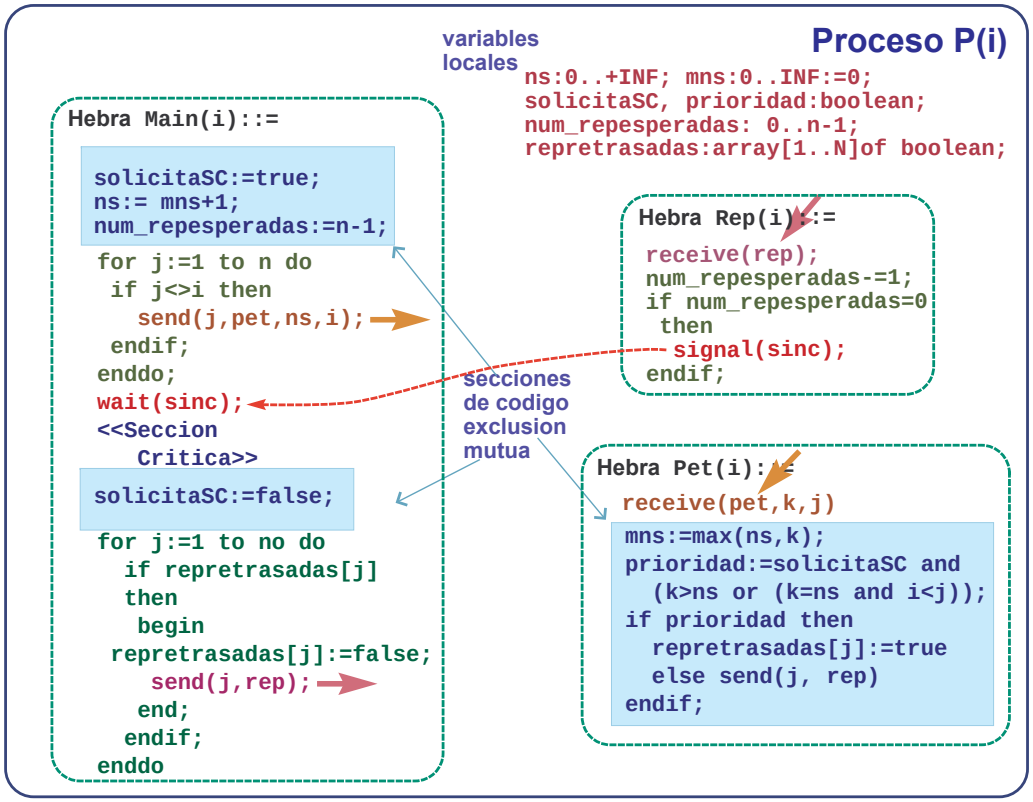
Problemática para la implementación de los algoritmos en sistemas distribuidos

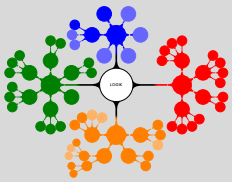
Miscelánea

- Los algoritmos no pueden utilizar sincronización global entre los procesos sólo utilizando variables en memoria compartida
- Se utilizan operaciones atómicas de paso de mensajes para sincronizarlos
- La red de comunicaciones ha de cumplir las siguientes condiciones:
  1. Red de comunicaciones completamente conectada
  2. Transmisión de mensajes sin errores
  3. Retraso variable en la entrega de mensajes con tiempo acotado
  4. Posibles *desencuenciamientos* en la entrega de mensajes en transmisión

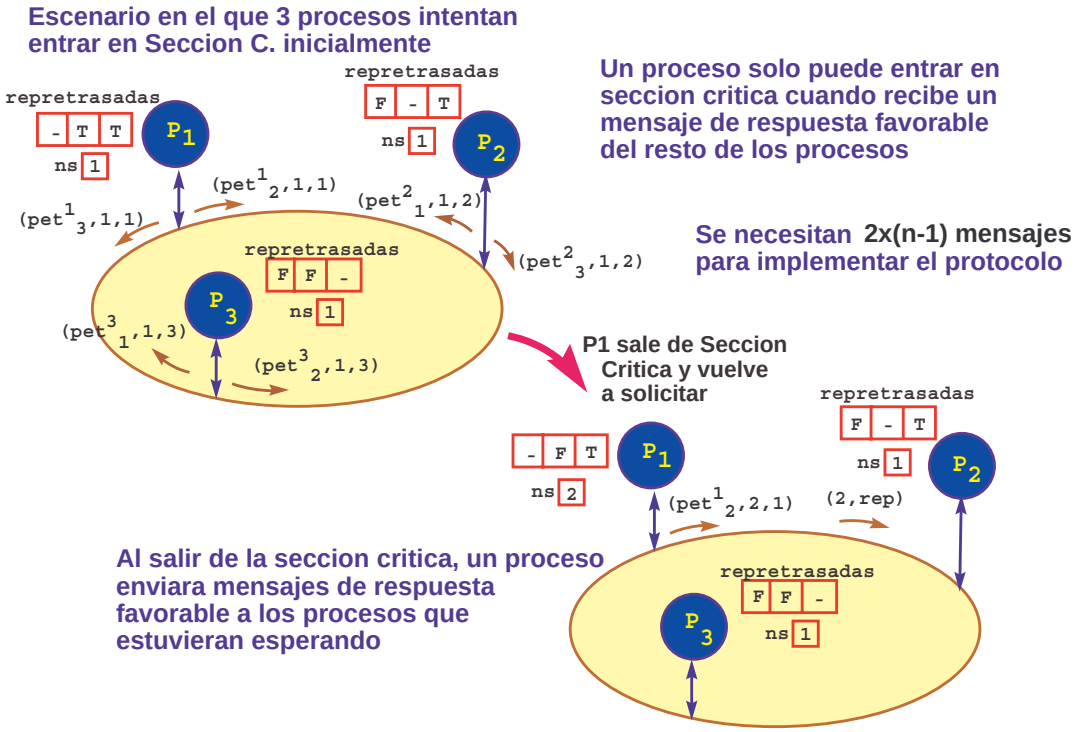


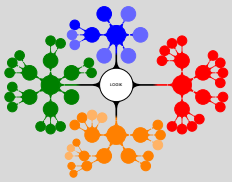
Algoritmo de Ricart-Agrawala





Algoritmo de Ricart-Agrawala-II





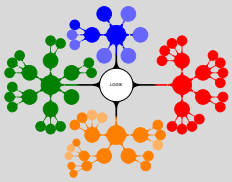
Verificación de las propiedades del algoritmo de Ricart-Agrawala

Exclusión mutua entre procesos al acceder a la sección critica

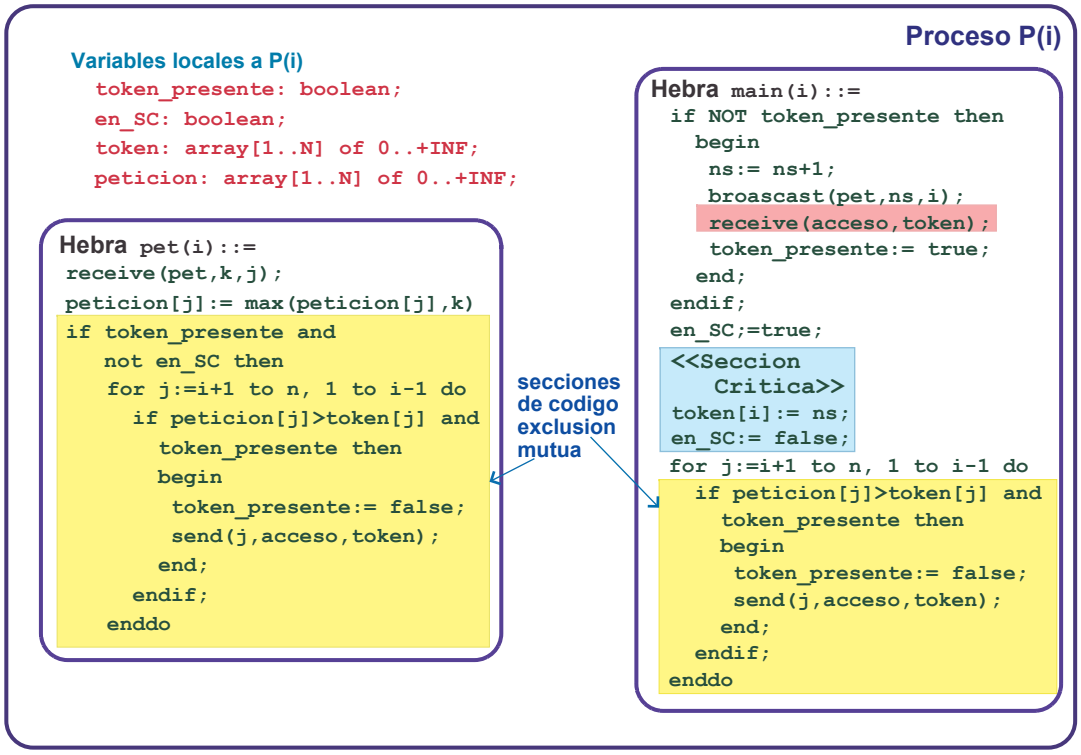
- P_i y P_j consiguen entrar a la vez en S.C. (hipótesis de incorrección)
- $\Leftrightarrow$ $P_{i,j}$ ha transmitido su petición a $P_{j,i}$, recibiendo contestación favorable
  1. P_i ha enviado contestación favorable antes de generar su número de secuencia $\Rightarrow$ P_j pospone la respuesta
  2. P_j ha enviado contestación favorable antes de generar su número de secuencia $\Rightarrow$ P_i pospone la respuesta
  3. Después de generar su número de secuencia es imposible que cada proceso envie contestación favorable al otro

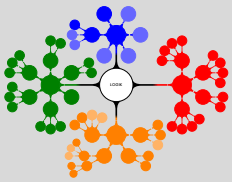
Alcanzabilidad de la sección crítica

Debido a la ordenación total de las peticiones, es imposible que se retrase indefinidamente la contestación favorable a algún proceso



Algoritmo de Suzuki-Kasami-Ricart-Agrawala





Verificación del algoritmo de Suzuki-Kasami-Ricart-Agrawala

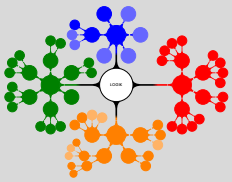
- El que todo proceso acceda en exclusión mutua es equivalente a demostrar el siguiente aserto: "globalmente, el número de variables *token_presente= true* es identicamente igual a la unidad"
  1. El aserto anterior se satisface inicialmente
  2. El aserto anterior se cumple cada vez que e transmite el token. El protocolo necesita **n** mensajes.

Alcanzabilidad de la sección crítica

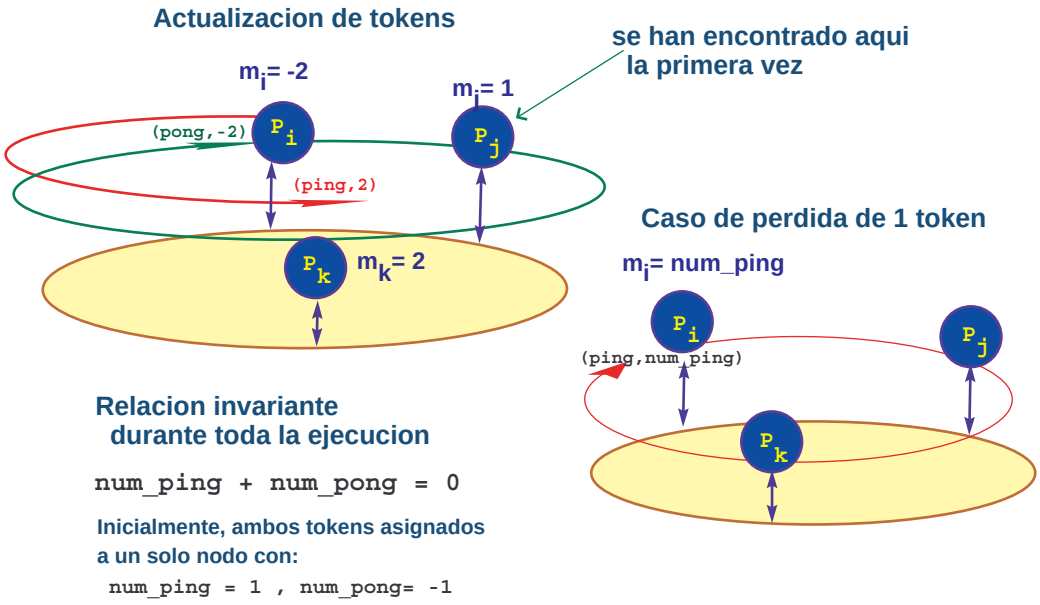
- Si ningún proceso posee el token, en un momento de la ejecución del algoritmo, este ha de estar necesariamene en transmisión

Propiedad de equidad

- Se obliga a que P_j transmita el token al primero que lo solicitó en el orden $\{j+1, j+2, \ldots, n, n-1, \ldots\}$
- ¿ Qué pasa si se pierden mensajes?



Algoritmo de regeneración de token de Misra

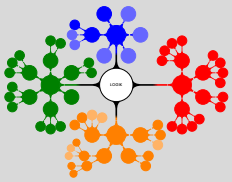


Relacion invariante durante toda la ejecucion

`num_ping + num_pong = 0`

Inicialmente, ambos tokens asignados a un solo nodo con:

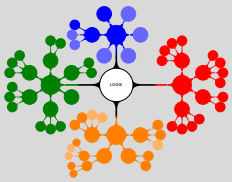
`num_ping = 1 , num_pong= -1`



Algoritmo de regeneración de token de Misra-II

Proceso (i)

```
while true do      variable local  m_i : int:=0;
  Seleccionar
    Cuando recibido(ping,num_ping) hacer
     if m_i = num_ping then
      begin -- se ha perdido pong, hay que recuperarlo
        num_ping := (num_ping+1)mod n+1;
        num_pong := -num_ping;
      end
     else mi:= num_ping;
    endif;
    endhacer;
    Cuando recibido(pong,num_pong) hacer
     if m_i = num_pong then
      begin -- se ha perdido ping, hay que recuperarlo
        num_pong := -((-num_pong+1)mod n+1;)
        num_ping := -num_pong;
      end
     else mi:= num_pong;
    endif;
    endhacer;
    Cuando se encuentran(ping,pong) hacer
      begin -- |num_ping|=|num_pong|,llevan el "num.colisiones"
        num_ping := (num_ping+1)mod n+1;
        num_pong := -num_ping;
      end
    endhacer;
  endseleccionar;
enddo;
```

Bibliografía

Para más información, ejercicios, bibliografía adicional, o "simplemente inspiración" sobre la temática, se puede consultar:

2.1. Capel-Rodríguez Valenzuela (2012) capítulo 2

Michel Raynal (1986). Algorithms for mutual exclusion. Cambridge: MIT Press.

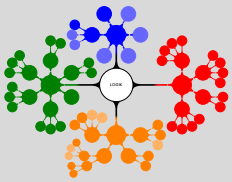
Andrews (2000) capítulo 5,

Ben Ari (2006) capítulo 7

Michel Raynal (2013). Distributed algorithms for message-passing systems. Springer.

“Concurrency”: `https://web.archive.org/web/20060128114620/http://vl.fmnet.info/concurrent/`

“Concurrency Talk”: `http://shairosenfeld.com/concurrency.html`



1.5. Mapa de Ideas Tema 1 y 2

Conceptos
SCD - Primer Parcial
Temas 1 y 2

Ismael Sallami Moreno

November 2024

Contents

1 Tema 1: Introducción a la Programación Concurrente

1.1 Axiomas de la Programción Concurrente

- Atomicidad y entrelazamientos de las ejecuciones atómicas.
- Consistencia de los datos tras un acceso concurrente
- Irrepetibilidad de las secuencias de instrucciones.
- Independencia de la velocidad de los procesos.
- Hipótesis de Progreso Finito.

1.2 Propiedad de los Sistemas Concurrentes

- Propiedad de seguridad (safety). Afirma un estado inalcanzable.
- Propiedad de vivacidad (liveness). En un momento se alcanzará un estado deseado.

1.3 Demostración de programas. Razonamientos.

- Razonamiento operacional.
- Razonamiento asertivo.

1.4 Axiomas

- Axioma de la sentencia nula. $\{P\}null\{P\}$
- Axioma de la asignación. $\{P_e^x\}$

1.5 Reglas

- **Regla de la consecuencia (1).**

$$\frac{\{P\}S\{Q\} \rightarrow \{R\}}{\{P\}S\{R\}} \tag{1}$$

  Es decir, siempre podemos hacer más débil la poscondición de un triple, de forma que este siga siendo cierto.

- **Regla de la consecuencia (2).**

$$\frac{\{R\} \rightarrow \{P\}, \{P\}S\{Q\}}{\{R\}S\{Q\}} \tag{2}$$

  Es decir, siempre podemos hacer más fuerte la precondición de un triple, manteniendo su veracidad.

- **Regla de la composición.**

$$\frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}} \tag{3}$$

Es decir, podemos condensar dos triples en uno, siempre y cuando la poscondición de uno sea la precondición del otro.

- **Regla de la conjunción.**

$$\frac{\{P_1\}S\{Q_1\}, \{P_2\}S\{Q_2\}}{\{P_1 \wedge P_2\}S\{Q_1 \wedge Q_2\}} \tag{4}$$

Es decir, si tenemos distintas pre y poscondiciones para una misma instrucción, podemos juntarlas todas en conjunción.

- **Regla del if.**

$$\frac{\{P \wedge B\}S_1\{Q\}, \{P \wedge \neg B\}S_2\{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2\{Q\}} \tag{5}$$

De esta forma, siempre que queramos probar que una tripleta de la forma

$$\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2\{Q\} \tag{6}$$

es cierta, tendremos que probar que las tripletas

$$\{P \wedge B\}S_1\{Q\} \quad \text{y} \quad \{P \wedge \neg B\}S_2\{Q\} \tag{7}$$

son ciertas.

- **Regla de la iteración.** Supongamos que una sentencia `while` puede iterar un número arbitrario de veces (incluso 0), tenemos que:

$$\{I \wedge B\}S\{I\} \tag{8}$$

$$\{I\} \text{ while } B \text{ do } S \text{ end } \{I \wedge \neg B\} \tag{9}$$

Donde a la proposición I la llamaremos invariante.

1.6 No Interferencia

$$NI(A, a) \equiv \{A \wedge pre(a)\}a\{A\}$$

1.7 Regla de la composición concurrente segura de procesos

Previamente debemos de comprobar que se cumple la regla de la NI.

$$\frac{\{P_i\}S_i\{Q_i\} \quad \text{son triples libres de interferencia} \quad 1 \leq i \leq n}{\{P_1 \wedge P_2 ... \wedge P_n\} \quad \text{cobegin} \quad (S_1 \parallel S_2 \parallel .... \parallel S_n) \quad \text{coend} \quad \{Q_1 \wedge Q_2 ... \wedge Q_n\}}$$

1.8 Verificación usando invariantes globales

- 1º Paso: Debemos de demostrar que el invariantes es cierto al inicio del programa.
- 2º Paso: Debemos de ir comprobando línea por línea en el código que el invariante se cumple antes y después de cada instrucción.

1.9 Demostrar que un programa acaba

- Demostrar que las variables adoptan valores diferentes a lo largo de la ejecución del programa.
- Identificar los valores como miembros del vector variante (vector que obtiene los valores posibles del programa), en caso contrario, notar que el programa no terminará.
- Si la salida esta en el vector variante, razonar que se alcanzará este estado.

2 Tema 2: Monitores y Exclusión Mutua

2.1 Dijkstra

2.1.1 Condiciones de Dijkstra

- No hacer suposiciones sobre el número de procesos soportados por el procesador.
- No hacer suposiciones sobre la velocidad de ejecución de los procesos.
- Si un proceso no esta en la sección crítica no puede impedir a otros procesos entrar en ella.
- Propiedad de alcanzabilidad: se garantiza que vaya a entrar un proceso de los que quieran entrar.

2.1.2 Algoritmo

```
var
    c : array[0..n-1] of (pasivo, solicitando, en_SC);
    turno : 0..n-1;

Process Pi();
begin
    while true do
    begin
        { Acceso a la secci n  cr tica }
        repeat
            c[i] := solicitando;
            { A }
            while turno <> i do
            begin
                if c[turno] = pasivo then
                    turno := i;
            end do
            c[i] := en_SC;
            { B }
            j := 0;
            while (j < n) and (j = i or c[j] <> en_SC) do
            begin
                j := j + 1;
```

```
24              end do
25          until j >= n
26          { Secci n  cr tica }
27          c[i] := pasivo;
28      end do
29  end
```

Listing 1: Algoritmo de exclusión mutua

2.2 Método de refinamiento sucesivo

- Primera etapa: entrada a la SC condicionada por una variable turno.
- Segunda etapa: Asociar a cada proceso una clave que lo defina.
  - Estado pasivo: no intenta acceder, representado con un 0.
  - ... con un 1.
- Tercera etapa: Cambiar el valor de la clave a 0 antes de que entre en la SC.
- Cuarta etapa: Comprueba que si el otro proceso también ha cambiado el valor de la clave a 0, este pasa a poner dicho valor a 1.

2.3 Algoritmo de Dekker

- Se puede considerar como una 5º etapa del método de refinamiento sucesivo.
- Mezcla entre las condiciones 1 y 4 del algoritmo de Dijkstra.
- Propiedades de corrección:
  - EM.
  - Alcanzabilidad en la SC.
  - Vivacidad: se garantiza usando Dekker, pero puede causar inanición.
  - Equidad: depende del hardware.

2.4 Algoritmo de Knuth

Descripción

El algoritmo de Knuth utiliza un arreglo de dos enteros y un indicador global para gestionar la entrada a la sección crítica. La clave es que cada proceso debe verificar su turno antes de ingresar.

Código

```
var
    turno : 0..1;                    { Indica el turno }
    desea : array[0..1] of boolean; { Deseo de entrar de cada
       proceso }

Process Pi (i : 0..1);
begin
    while true do
    begin
        desea[i] := true;            { El proceso i desea entrar }
        while turno <> i do
        begin
            if not desea[1-i] then  { Verifica si el otro
               proceso no desea entrar }
                turno := i;          { Cambia el turno a i }
        end do;
        { Secci n  cr tica }
        desea[i] := false;           { Sale de la secci n
           cr tica }
    end do;
end;
```

Listing 2: Algoritmo de Knuth

2.5 Algoritmo de Peterson

Descripción

El algoritmo de Peterson es simple y elegante, combinando un indicador de turno y un arreglo de booleanos para garantizar la exclusión mutua y evitar interbloqueos.

Código

```
var
    turno : 0..1;                    { Indica el turno }
    desea : array[0..1] of boolean; { Deseo de entrar de cada
       proceso }

Process Pi (i : 0..1);
begin
    while true do
    begin
        desea[i] := true;            { El proceso i desea entrar }
        turno := 1 - i;              { Cede el turno al otro
           proceso }
        while desea[1-i] and (turno = 1 - i) do
            skip;                    { Espera activa si el otro
               desea entrar }
        { Secci n  cr tica }
```

```
14          desea[i] := false;          { Sale de la secci n
               cr tica }
15      end do;
16  end;
```

Listing 3: Algoritmo de Peterson

2.6 Anotaciones Generales

- Dekker: para 2 procesos, pero no garantiza que no haya inanición.
- Dijkstra es para n procesos, pero no garantiza que no haya inanición.
- Knuth: Si garantiza que no haya inanición, pero es de orden de 2^n.
- Peterson: similar al anterior, pero de orde de n^2.

1.6. Tema 3

Tema 3

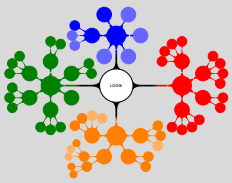
Sistemas basados en paso de mensajes

Soluciones software para desarrollar programas distribuidos

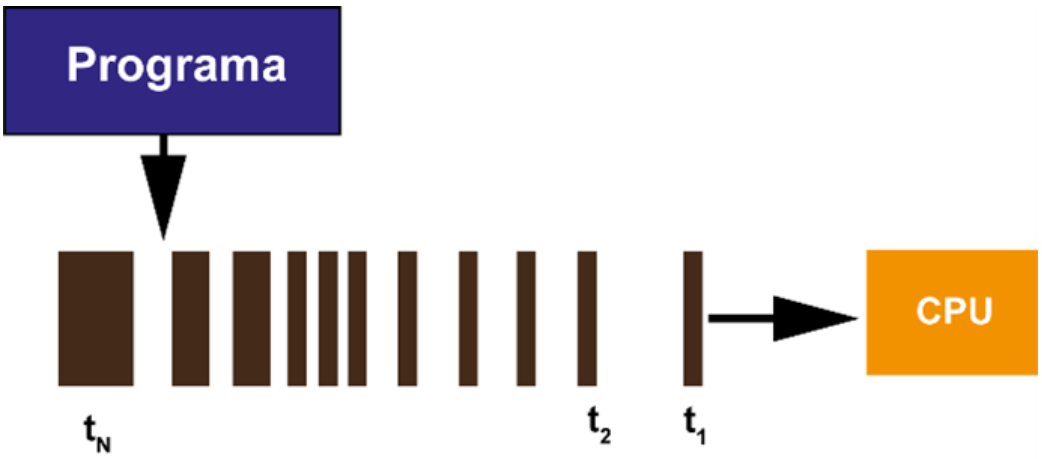
Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 15 de noviembre de 2024

Manuel I. Capel
manuelcapel@ugr.es
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

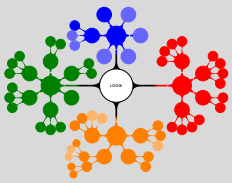


Motivación



Computador monoprocesador clásico: las instrucciones son ejecutadas 1 a 1 en la CPU

- Problemas que reducen la eficiencia del sistema
- Los computadores de programa almacenado: no pueden producirse simultáneamente una búsqueda de instrucciones y una operación de datos



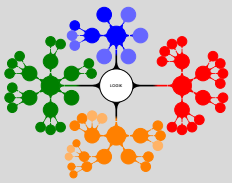
Problemas de computadores con programa único almacenado

Algunos problemas de estas arquitecturas:

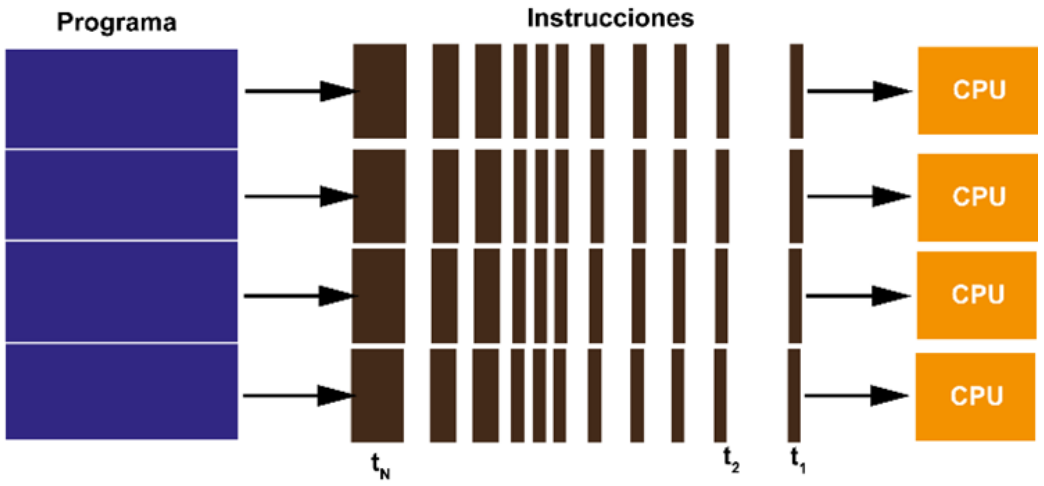
- El cuello de botella "Von Neumann"
- Limitación de velocidad del bus
- Los programas necesitan más tiempo de ejecución

- Los inconvenientes anteriores afectan al rendimiento de la computación
- Además, son más caras de producir

Manuel I. Capel
manuelcapel@ugr.es



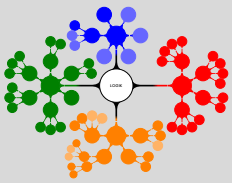
Arquitectura que usa múltiples elementos de cómputo simultáneamente



Las instrucciones son ejecutadas en varios procesadores y los datos se acceden independientemente

Solucionan los problemas de las arquitecturas clásicas de programa único almacenado:

- El sistema consume menos energía,
- Es más barato de producir y
- Proporciona mejor rendimiento porque resuelve el problema del “cuello de botella"



Multiprocesamiento

Idea fundamental

Utilización de varios procesadores o *núcleos* para ejecutar los programas de una misma aplicación

Desde el punto de vista del sistema

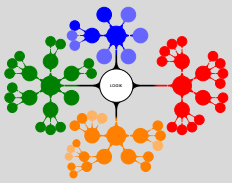
Capacidad para gestionar más de 1 procesador y re-asignar tareas entre tales procesadores durante la ejecución de los programas

- Los procesadores pueden ejecutar 1 sola secuencia o varias secuencias de instrucciones, que se ejecutarán en *contextos* múltiples y simultáneos

	Instrucción única	Múltiples instrucciones
Datos únicos	SISD	MISD
Múltiples datos	SIMD	MIMD

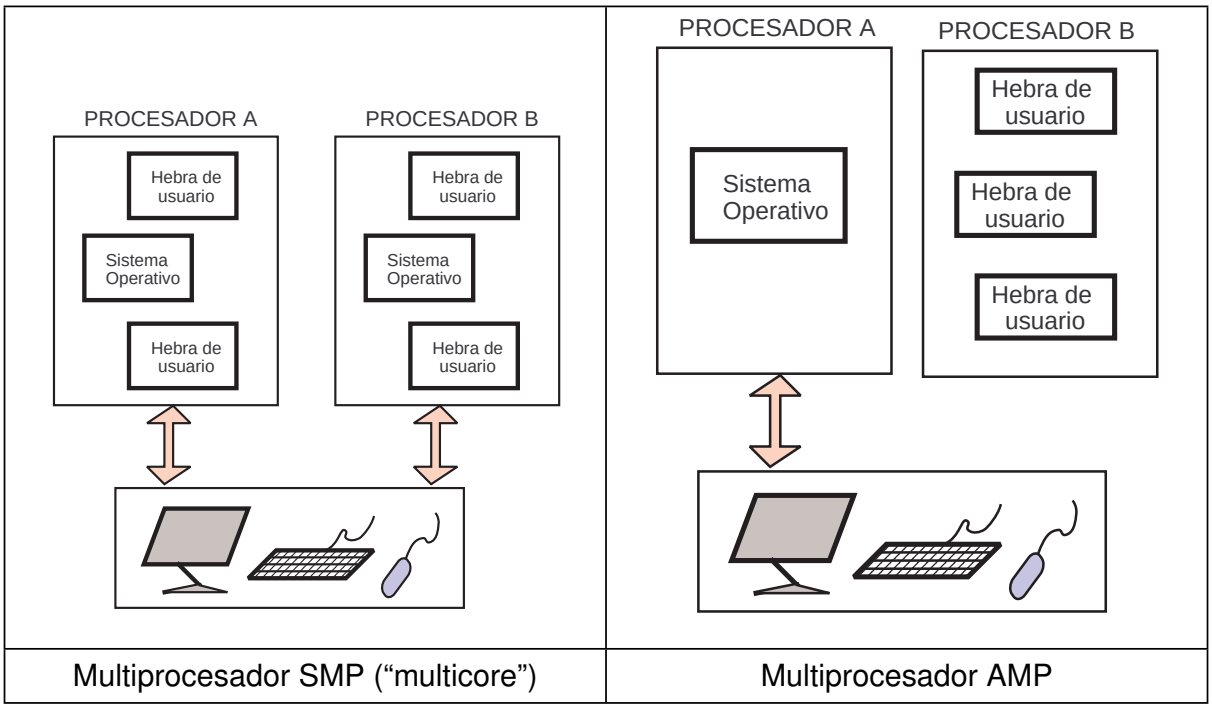
Clasificación de Flynn

Manuel I. Capel
manuelcapel@ugr.es



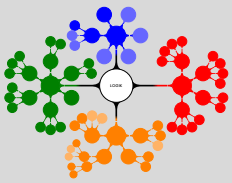
Mecanismos básicos en sistemas *multiprocesadores*

- El multiprocesador que resulta más útil: procesadores individuales que ejecutan sus instrucciones independientemente (modelo MIMD)
  1. Utilizar variables en memoria común a los procesadores
  2. El inconveniente principal es falta de escalabilidad.
  3. Hardware de interconexión procesadores-memorias caro



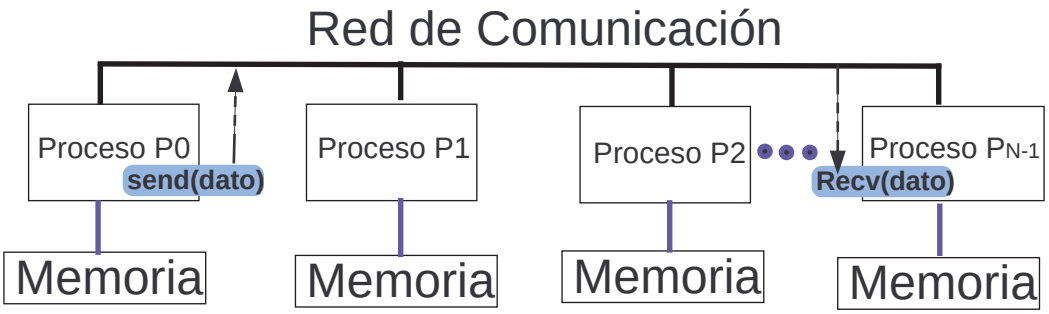
Multiprocesador SMP ("multicore") | Multiprocesador AMP

Manuel I. Capel
manuelcapel@ugr.es



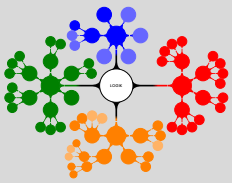
Mecanismos básicos en sistemas *multicomputadores*

- No existe memoria de acceso común a todos los computadores
  1. Son más difíciles de programar.
  2. No suelen presentar el problema de la *escalabilidad*
  3. Se necesita una notación de programación más flexible que la que usamos con multiprocesadores, por ejemplo:
     - diferentes modos de comunicación entre los procesos,
     - *no-determinismo* en la recepción de múltiples comunicaciones síncronas en servidores.



  4. Las primitivas concurrentes clásicas para multiprocesadores que suponen memoria compartida no se pueden utilizar, obviamente.

Manuel I. Capel
manuelcapel@ugr.es

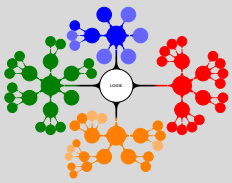


Inconvenientes de la Programación Distribuida y Paralela

Puntos clave a considerar:

- Supone aprender a programar según un paradigma de programación que asume el avance de múltiples líneas de ejecución de instrucciones en los programas y sin variables globales compartidas por los procesos
- La depuración de estos programas se convierte en una tarea muy difícil
- La depuración efectiva requiere acceso a la red de interconexión y a la memoria, tanto a nivel de caché como a nivel de MPs

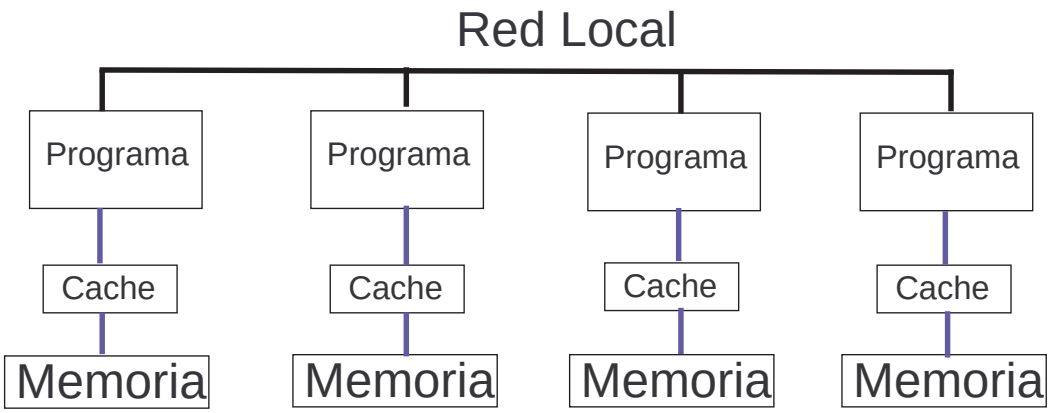
Manuel I. Capel
manuelcapel@ugr.es



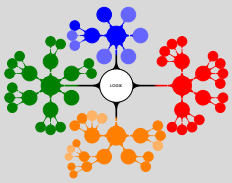
Estilo de multiprogramación SPMD

Idea fundamental

Se utiliza en programación distribuida y paralela como un solución intermedia entre SIMD y MIMD: el código que ejecutan los procesos es idéntico pero tal código actúa sobre diferentes datos



Manuel I. Capel
manuelcapel@ugr.es

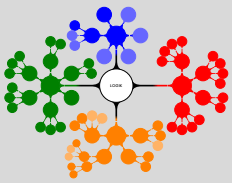


Estilo de multiprogramación SPMD–II

Características

- Variante del modelo general MIMD de Flynn
- No se necesita una arquitectura especial de computador (como ocurre con el SIMD) para programar SPMD
- Los procesadores ejecutan el mismo programa pero de forma independiente
- Posteriormente a la compilación, a cada proceso paralelo del programa se le asignará un valor de identificador distinto
- De esta forma, cada proceso puede ejecutar una parte distinta del programa común

Manuel I. Capel
manuelcapel@ugr.es

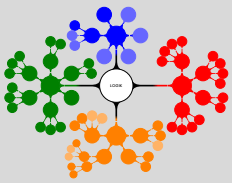


Ejemplo de código programado según un estilo SPMD

```
                Cliente Trabajador 1 Trabajador 2
                  a b e | c d f      | c d f
-------------------------------------------------
a = 3;            3 - - | - - -      | - - -
b = 4;            3 4 - | - - -      | - - -
spmd                    |            |
c = rank();       3 4 - | 1 - -      | 2 - -
d = c + a;        3 4 - | 1 4 -      | 2 5 -
end                     |            |
e = a + d{1};     3 4 7 | 1 4 -      | 2 5 -
c{2} = 5;         3 4 7 | 1 4 -      | 5 5 -
spmd                    |            |
f = c * b;        3 4 7 | 1 4 4      | 5 5 20
end
```

- El valor de las variables definidas en el `cliente` puede ser leído por los `trabajadores`, pero no puede ser cambiado.
- Las variables definidas por los `trabajadores` pueden ser leídas o cambiadas por el `cliente`.

Manuel I. Capel
manuelcapel@ugr.es



Semántica de las operaciones de paso de mensajes

Idea fundamental

Significado (semántica) de las operaciones de comunicación ({`send()`, `receive()`}) entre procesos depende de 2 factores:

1. cumplimiento (o no) de la propiedad de *seguridad*
2. modo de comunicación

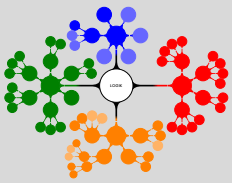
1) Propiedad de seguridad

- No se cumple: se permite alterar un dato después de que la operación de envío (`send(...)`) se ejecute y vuelva, pero antes de que termine la transmisión del valor enviado
- Se cumple: el valor del dato enviado coincide con el del dato recibido posteriormente

2) Modo de comunicación de las operaciones de paso de mensajes

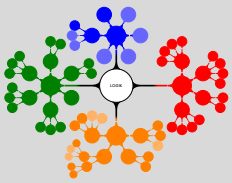
- Operaciones bloqueantes
- Operaciones no-bloqueantes (con *asincronicidad*)

Manuel I. Capel
manuelcapel@ugr.es



Operaciones bloqueantes de paso de mensajes

Manuel I. Capel
manuelcapel@ugr.es



Idea fundamental

La operación de envío (`send(...)`) sólo vuelve cuando termina la transmisión del mensaje

Modo de comunicación	Hardware especializado	Sincronización	Seguridad
Sin búfer	-	Sí (citas)	Sí
Con búfer	Sí No	Relajada Sí	Sí Sí

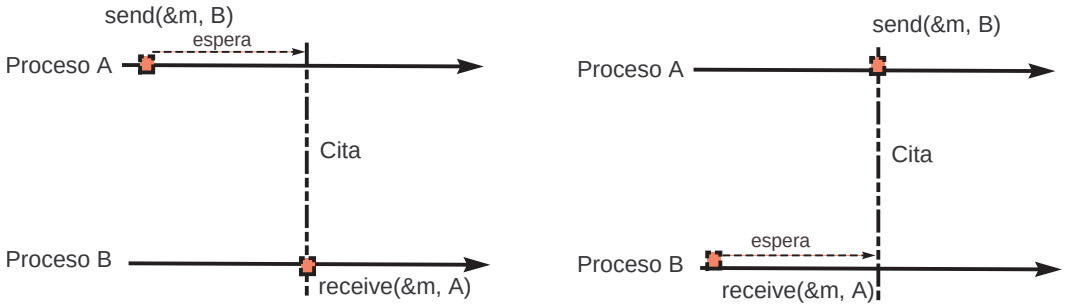
Características de las operaciones bloqueantes

Mecanismo de *citas*

Idea fundamental

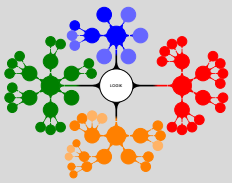
Se trata de una operación de comunicación bloqueante y sin búfer.
Cita: tiene lugar antes de que comience la transmisión de los datos

- El estado del proceso emisor se mantiene hasta que vuelve la operación de recepción en el otro proceso
- Los procesos receptor o emisor pueden sufrir espera ociosa hasta que termina la mencionada cita



Cita entre procesos con comunicación síncrona

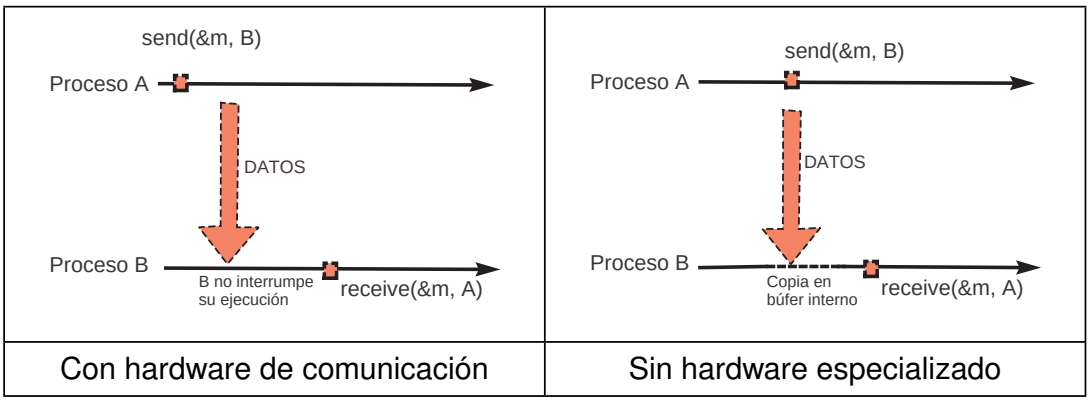
Manuel I. Capel
manuelcapel@ugr.es



Paso de mensajes bloqueante con búfer

Idea fundamental

El proceso emisor del mensaje *vuelve* inmediatamente al ejecutar la operación `send(...)` salvo que el búfer se llene. La operación `receive(...)` no *vuelve* hasta que se han recibido todos los datos en el receptor (si no hay hardware especializado)



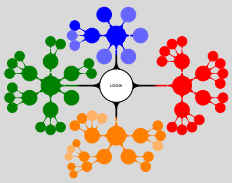
Con hardware de comunicación	Sin hardware especializado

Implementación del paso de mensajes bloqueante con búfer

Variantes

Hardware especializado	Sí	No
Bloqueo del receptor	No (transferencia inmediata)	Sí (transferencia sincronizada)

Manuel I. Capel
manuelcapel@ugr.es



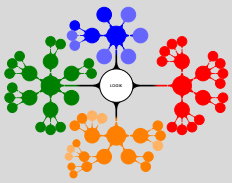
Causas de ineficiencia en la implementación del paso de mensajes bloqueante sin hardware especializado

Esquema de paso de mensajes	Motivo de la ineficiencia
Síncrono (citas)	Espera ociosa de 1 proceso
Mediante búfer	Sobrecarga por gestión del búfer

Posible solución:

- Definir operaciones send y receive que no se bloqueen
- $\Rightarrow$ Dejar en la responsabilidad del programador el asegurar que no se alteren los datos de los programas mientras están siendo transmitidos

Manuel I. Capel
manuelcapel@ugr.es



Operaciones no-bloqueantes

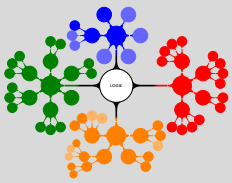
Idea fundamental

El cumplimiento de la propiedad de seguridad en los programas distribuidos con paso de mensajes se convierte en responsabilidad exclusiva del programador.

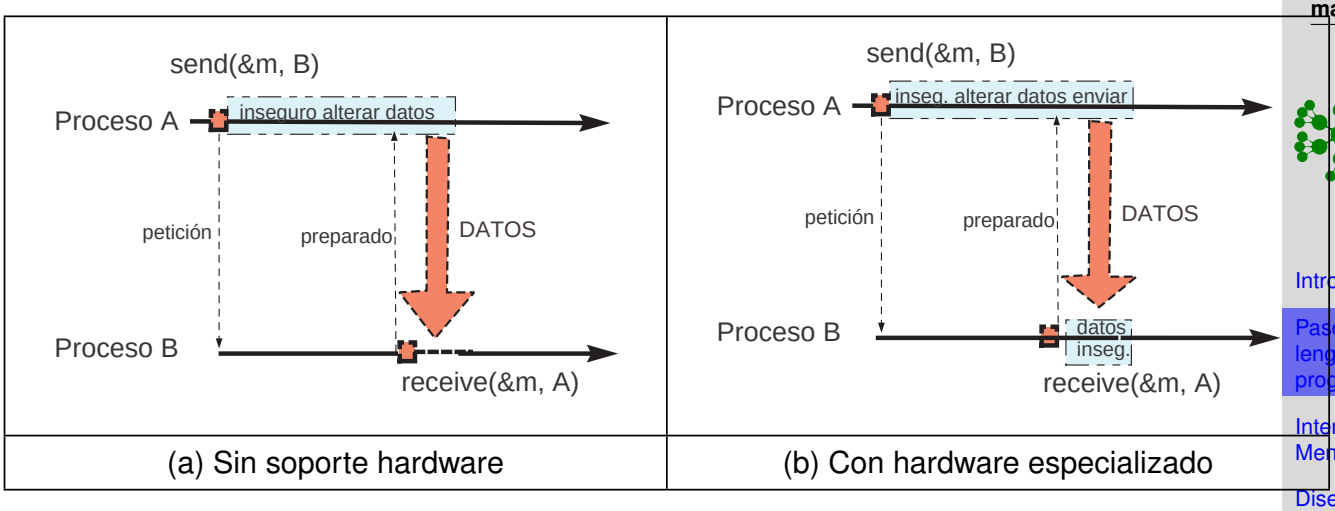
Características

- El tiempo de transmisión de los datos (emisor/`send()` – receptor/`send()`) no es *despreciable*, en general
- Las operaciones de envío se ejecutan y *vuelven* inmediatamente, antes incluso de que sea seguro modificar los datos que están en transmisión.
- En los lenguajes, se necesitan operaciones de *comprobación de estado* de los datos transmitidos
- La operación `receive(...)` vuelve o no lo hace, antes de terminarse la transmisión, dependiendo de si existe o no hardware especializado, respectivamente.

Manuel I. Capel
manuelcapel@ugr.es



Paso de mensajes no bloqueante



(a) Sin soporte hardware | (b) Con hardware especializado

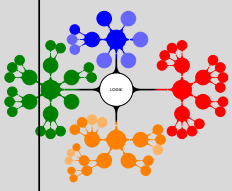
Paso de mensajes no-bloqueante sin búfer

Existe una operación de comprobación que indicaría cuándo es seguro acceder a los datos, que están siendo transmitidos, por parte del proceso receptor (caso b)

Paso de mensajes no-bloqueante con búfer

En este caso se reduce el tiempo de espera respecto del caso anterior porque la operación `receive()` provoca la transferencia inmediata de datos del búfer a la memoria propia del proceso receptor.

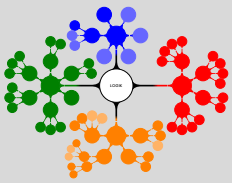
Manuel I. Capel
manuelcapel@ugr.es



Bibliotecas de paso de mensajes y patrones de interacción

- Modelo SPMD: **Message Passing Interface** (MPI)
  - Biblioteca de funciones para C, más el archivo de cabecera `mpi.h`
  - Biblioteca de funciones para FORTRAN, junto con el archivo de cabecera `mpif.h`
  - Órdenes para compilación: `mpicc, mpif77`: versiones de las órdenes habituales (`cc, f77`)
  - Órdenes específicas para ejecución de aplicaciones paralelas: `mpirun`
  - Herramientas para monitorización y depuración de programas paralelos
- No es la única biblioteca para la programación de aplicaciones paralelas y distribuidas (PVM, NX en el Intel Paragon, MPL en el IBM SP2, etc.), pero sí la más utilizada en la actualidad
- Se dispone de funciones de comunicación punto-a-punto, así como operaciones colectivas para involucrar a un grupo de procesos
- Los procesos pueden agruparse y formar *comunicadores* para permitir la definición del ámbito de las operaciones colectivas y un diseño modular de los programas distribuidos

Manuel I. Capel
manuelcapel@ugr.es



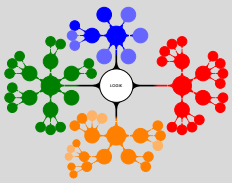
Bibliotecas de paso de mensajes y patrones de interacción-II

Ejemplo de programa que utiliza el *binding* de MPI para C

```
# include "mpi.h"
# include <iostream>
using namespace std;
main (int argc, char **argv) {
int nproc; /*Numero de procesos */
int yo; /* Mi direccion: 0<=yo<=(nproc-1)*/
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &yo);
/* CUERPO DEL PROGRAMA */
cout<<"Soy el proceso " <<yo<<" de "
<<nproc<<endl;
MPI_Finalize(); }
```

- Valor devuelto es `MPI_SUCCESS`: la función se ha realizado con éxito
- `MPI_COMM_WORLD` se refiere al comunicador universal, es decir, que incluye a todos los procesos de nuestro programa
- Se pueden definir otros comunicadores. Todas las funciones de MPI necesitan como argumento un comunicador

Manuel I. Capel
manuelcapel@ugr.es

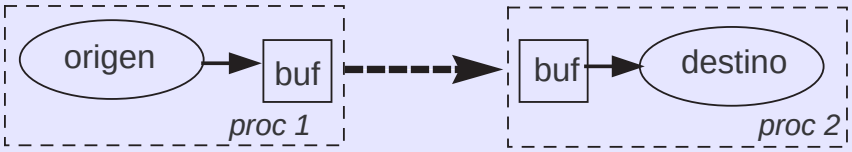


Operaciones de paso de mensajes utilizando MPI

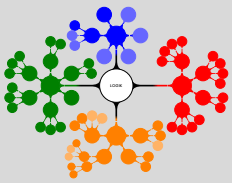
Mensaje de MPI

Bloque de datos trasferido entre procesadores y consiste en:

1. *Envoltorio* del mensaje:
   - destino / origen
   - etiqueta
   - comunicador



2. *Cuerpo* del mensaje:
   - búfer
   - contador
   - tipo de datos



Operaciones de paso de mensajes utilizando MPI-II

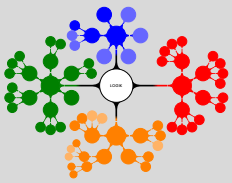
```
int MPI_Send(void*bufer, int cont, MPI_Datatype t_datos,
             int destino, int etiqueta, MPI_Comm comunicador)
```

Campos de una operación MPI de envío con buffer

```
int MPI_Recv(void*bufer, int cont, MPI_Datatype t_datos,
             int origen, int etiqueta, MPI_Comm comunicador,
             MPI_Status *estado)
```

Campos de una operación MPI de recepción con buffer

Manuel I. Capel
manuelcapel@ugr.es



Semántica de las operaciones de paso de mensajes bloqueantes

Concordancia entre mensajes

- El *envoltorio* (origen, etiqueta, comunicador) del mensaje enviado ha de coincidir con el envoltorio del mensaje recibido.

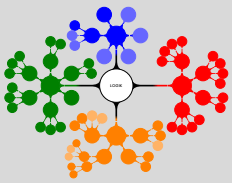
Bloqueo de las operaciones de comunicación

- Las operaciones: `MPI_Send`, `MPI_Ssend`, `MPI_Recv` se suspenden (no “vuelven”) hasta que se *completan*

Completación de las operaciones

- `MPI_Send`: cuando termina de copiar el mensaje en el búfer `RMI`
- `MPI_Ssend`: cuando se produce la *cita* con el proceso que llama a `MPI_Recv` y hay concordancia entre los *envoltorios* en cada parte de la comunicación
- `MPI_Recv`: existe ya un mensaje pendiente *conforme* (concordancia y compatibilidad de tipos) a la declaración de parámetros de esta operación de comunicación

Manuel I. Capel
manuelcapel@ugr.es



Semántica de las operaciones de paso de mensajes bloqueantes-II

Situaciones de error

- El tamaño del mensaje recibido es mayor que el esperado (programa aborta)
- Los tipos de datos declarados en el emisor y en el receptor son incompatibles (resultado indefinido)

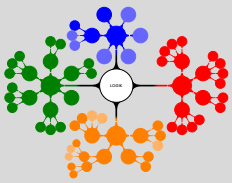
Sustitución de comodines

- Se puede sustituir `MPI_ANY_SOURCE` en el campo *origen* y `MPI_ANY_TAG` en el campo *etiqueta* sin que se produzca error en la operación de comunicación
- Sin embargo, el campo *comunicador* carece de *comodín*

Obtener información de los mensajes recibidos

- estado/tamaño: `MPI_Get_Count(`*`status,t_datos,cont`*`)`
- estado: campo `MPI_Status` (último parámetro de la orden `MPI_Recv`)

Manuel I. Capel
manuelcapel@ugr.es



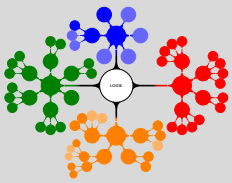
Comunicación no bloqueante en `MPI`

Idea fundamental de estas operaciones

- Evitar situaciones de *interbloqueo* de procesos debidas a una *incorrecta secuenciación* en el orden de ejecución de las operaciones de envío y recepción por parte de los procesos que intervienen en una comunicación
- Impedir el bloqueo de los procesos emisores de mensajes debido al desbordamiento del búfer interno al recibir

Operaciones `MPI` de envío/recepción no bloqueantes

- `MPI_Isend`: Inicia envío, pero vuelve de la llamada antes de comenzar a copiar el mensaje en el buffer
- `MPI_Irecv` : Inicia recepción pero vuelve de la llamada antes de comenzar a recibir ningún mensaje
- `MPI_Test(MPI_Request*r,int*flag,MPI_Status*s)` Chequea si la operación no bloqueante (identificada por el argumento `r`) ha finalizado: argumento `flag >0`



Comunicación no bloqueante en MPI-II

Sondeo de estado de un mensaje

```
MPI_Iprobe(int origen, int etiqueta, MPI_Comm comunicador,
           int*flag, MPI_Status*estado)
```

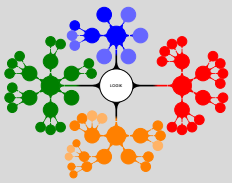
Si hay mensaje pendiente (flag $>$ 0), entonces hay que recibirlo con una llamada a MPI_Recv. Comprobación existencia de mensaje no bloqueante.

```
MPI_Probe(int origen, int etiqueta, MPI_Comm comunicador,
          MPI_Status*estado)
```

Comprobación existencia de mensaje bloqueante. Esperar un mensaje sin conocer su procedencia, etiqueta o tamaño.

Característica	MPI_Probe	MPI_IProbe
Bloqueante	Si	No
Comportamiento	Espera hasta que haya un mensaje disponible	Vuelve inmediatamente con el estado del mensaje
Eficiencia	Puede causar ineficiencia en ausencia de mensajes	Permite trabajar en paralelo mientras se espera

Manuel I. Capel
manuelcapel@ugr.es



Comunicación no bloqueante en MPI-III

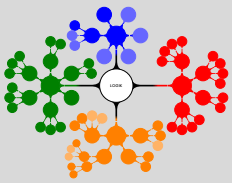
- Ejemplo de uso con `MPI_Iprobe`

```
int flag; MPI_Status status;
MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &flag, &status);
if (flag) {
    printf("Hay un mensaje disponible de origen %d
        con etiqueta %d.\n",
           status.MPI_SOURCE, status.MPI_TAG);
}
```

- Ejemplo de uso con `MPI_Probe`

```
MPI_Status status;
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD
    , &status);
printf("Se detectó un mensaje de origen %d con
    etiqueta %d.\n",
       status.MPI_SOURCE, status.MPI_TAG);
```


Manuel I. Capel
manuelcapel@ugr.es



Comunicación no bloqueante en MPI-IV

Esperar la completación de una operación

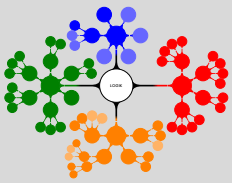
```
MPI_Wait(MPI_Request*solicitud, MPI_Status*estado)
```

Bloquea al proceso que la llama hasta que la operación identificada por `solicitud` termina de forma segura

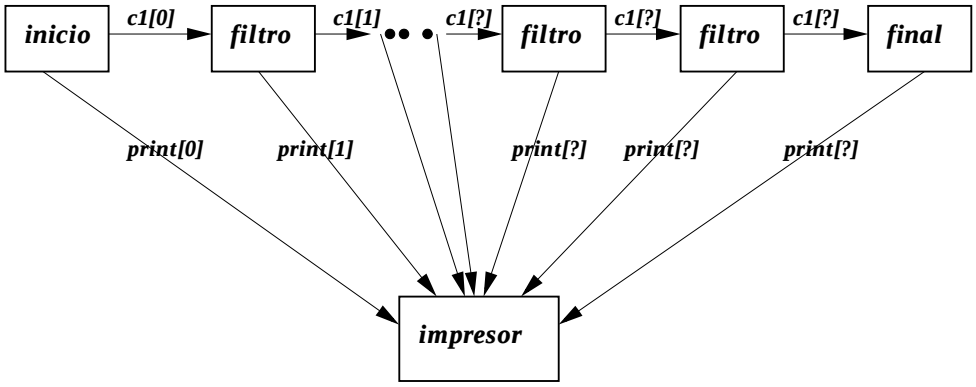
```
MPI_Request_free(MPI_Request*solicitud)
```

Libera al objeto `solicitud` de forma explícita

Manuel I. Capel
manuelcapel@ugr.es

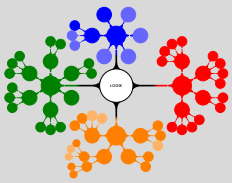


Ejemplo: *pipeline* de procesos que genera números primos



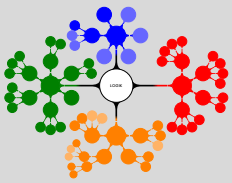
Generación de la serie de números primos con la *criba de Erastótenes*

Manuel I. Capel
manuelcapel@ugr.es



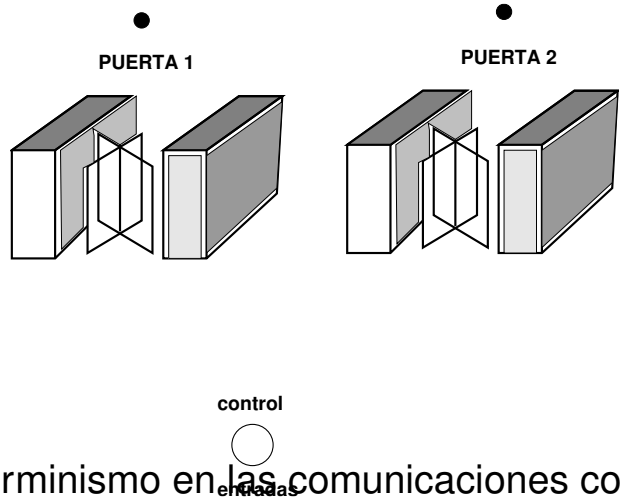
Esquema de la solución con MPI

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (rank==0){ x=2; /* primer numero primo*/ i=1;
     MPI_Send(&x,1,MPI_INT,size-1,0,MPI_COMM_WORLD);
     while(!fin){
       i+=2; x=i;
       MPI_Send(&x,1,MPI_INT,rank+1,0,MPI_COMM_WORLD);
       MPI_Irecv(&x,1,MPI_INT,size-1,MPI_ANY_TAG,
            MPI_COMM_WORLD, &request);
       ....} }
     else if (rank == size-1){/*este es el impresor*/
     ... else{/*representa a los procesos filtros*/
      MPI_Recv(&valor,1,MPI_INT,rank-1,0,MPI_COMM_WORLD,&
          status);
      MPI_Send(&valor,1,MPI_INT,size-1,0,MPI_COMM_WORLD);
         while(!fin){
      MPI_Recv(&x,1,MPI_INT,rank-1,0,MPI_COMM_WORLD,&
          status);
           if (rank<(size-2))
           if (x%valor!=0)
           MPI_Send(&x,1,MPI_INT,rank+1,0,MPI_COMM_WORLD);
           MPI_Irecv(&x,1,MPI_INT,size-1,MPI_ANY_TAG,
               MPI_COMM_WORLD,&request);
           if (x==size-1) fin=true; } }
 MPI_Finalize();
```

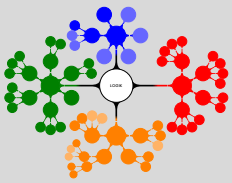
Tipos de procesos

- **Filtros**: son procesos transformadores de datos.
- **Clientes**: son procesos desencadenantes de algo.
- **Servidores**: son procesos reactivos.
- **Pares**: procesos idénticos que cooperan para resolver 1 problema.



Ejemplo: no determinismo en las comunicaciones con paso de mensajes

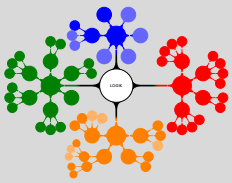
Manuel I. Capel
manuelcapel@ugr.es



Modelos y lenguajes de programación distribuida

- Programación de procesos servidores con paso de mensajes síncrono:
  1. Inadecuación de las órdenes condicionales deterministas (`if, switch, ...`) de los lenguajes secuenciales para implementar servidores
  2. Sentencias no-deterministas en los lenguajes de programación facilitan la implementación de sistemas que reaccionan frente estímulos procedentes de su entorno.
- Solución incorrecta al problema del museo si se suponen operaciones bloqueantes de paso de mensajes:

```
Process P1::                Process P2::              Process P3::
                                                       /*Controlador*/
for(i=1;i<20i++){           for(i=1;i<20;i++){
 send(P3,1);                 send(p3,1);              for (j=1;j<20;j++){
 /*pasa 1 persona*/         /*pasa 1 persona*/        receive(P1, temp);
 }                           }                        cont=cont + temp;
                                                      receive(P2, temp);
                                                      cont=cont + temp;}
                                                     printf("%d", cont);
main(){ cobegin P1;P2;P3 coend; }
```

Paradigma cliente/servidor de programación distribuida

Ideas fundamentales

-Comunicación muchos-a-uno,
-Cada comunicación es un *par*: (datos de entrada, resultados)
-Selección no-determinista entre varias comunicaciones posibles, en lado del servidor

Características

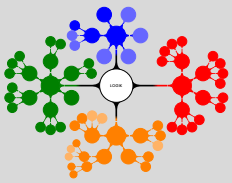
- Proceso cliente: solicita 1 servicio enviando un mensaje al servidor. Los procesos clientes tienen un carácter activo, ya que envían mensajes solicitando un servicio.
- Proceso servidor: tiene un carácter pasivo; recibe una petición de servicio de los clientes, devuelve un mensaje con los posibles resultados.

Simulación *no segura* utilizando paso síncrono de mensajes:

```
proceso cliente[i]
 *[TRUE ->

  Send(servidor, peticion())
  Receive(servidor,respuesta)]
```

```
proceso servidor
*[ (i:0..n) condicion(i);
cliente(i)::Receive(peticion());
  realizar.servicio
  Send(cliente[i],resultado)]
```


Manuel I. Capel
manuelcapel@ugr.es

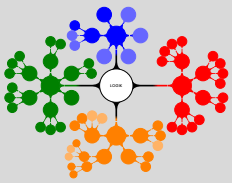


Órdenes con guarda

Sentencia “orden con guarda"

- Sirve para construir una sentencia (**Select**) de los lenguajes de programación distribuida que evita el bloqueo de servidores si sólo se dispone de paso de mensajes síncrono desde los procesos clientes
- Pasan a ser las sentencias componentes básicas de las construcciones alternativas y repetitivas de los lenguajes concurrentes del tipo anterior (paradigma cliente-servidor e invocaciones remotas)

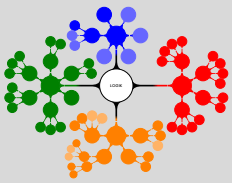
```
<orden.alternativa>::= if <conjunto.ordenes.con.guarda> fi
<orden.repetitiva>::= do <conjunto.ordenes.con.guarda> do
<conjunto.ordenes.con.guarda>::= <orden.con.guarda>{[]<
    orden.con.guarda>}
<orden.con.guarda>::= <guarda> -> <lista.sentencias>
<guarda>::= <expresion.booleana> |
            <expresion.booleana>;receive(<argumentos>);|
            receive(<argumentos>)
}
```

Ejemplo: solución correcta al problema del museo

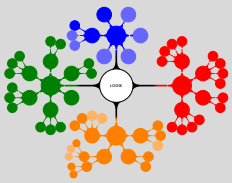
```
PUERTA(i:1..2)::
{  int s=0;
  do
   if (s<HORA.CIERRE && PERSONA())->
       {send(CONTROL, S()); //envia una se\~nal de entrada
            de persona
        DELAY.UNTIL(s+1);//espera hasta el siguiente
            instante
        s:=s+1;// cuenta un nuevo tick de reloj
        }
     []
       (s<HORA.CIERRE && NOT PERSONA())->
        DELAY.UNTIL(s+1);//espera hasta el siguiente
            instante
        s:=s+1;]//cuenta otro tick
     []
        TRUE->DELAY.UNTIL (TIME() + 16*3600);
        //es la hora de cierre del museo; hay
        //que esperar 16 horas para activar el controlador
            .
        // TIME() devuelve una cuenta en segundos.
  fi
  do;
  send(CONTROL, Start();}
...
```


Manuel I. Capel
manuelcapel@ugr.es



Ejemplo: solución correcta al problema del museo-II

```
CONTROL::
{  int cont= 0;
  if  receive(PUERTA(1), Start());
      //desde cualquiera de los sensores
    [] //de las puertas se arranca
      receive(PUERTA(2), Start());//el controlador
  fi
  do
     []*[(j:1..2) receive(PUERTA(j), S())-> cont:= cont
          +1];
 //cuenta una persona mas, porque ha recibido la se\~nal
 //de cualquiera de las 2 puertas (no se puede saber cual)
  od;
  printf("numero de personas",%d, cont));
}

main(){
  cobegin PUERTA;CONTROL coend;}
```

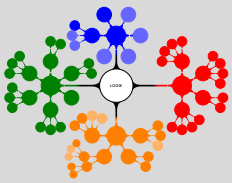



Introducción a la sentencia (Select) (o espera selectiva)

Ideas fundamentales

- Se trata de una versión de la *orden alternativa* (selección no determinista entre un conjunto de órdenes con guarda) que poseen algunos lenguajes de programación distribuida (Ada, SR, ...)
- Mantiene un modo síncrono de comunicación (paso de mensajes bloqueante) cliente/servidor pero el servidor podría mantener varias comunicaciones preparadas para recibir, y recibir el mensaje de una de ellas
- Resuelve el problema de la recepción de una entre varias señales pendientes sin dependencias del orden temporal de envío (*problema del museo*)

Manuel I. Capel
manuelcapel@ugr.es



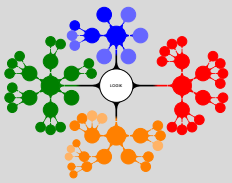
Instrucción de espera selectiva: Select

Semántica de la instrucción Select:

- Cada bloque que comienza en `when` se denomina una *alternativa* ($\equiv$ *orden guardada*)
- En cada alternativa, el texto desde `when` hasta `do` se denomina *guarda* de dicha alternativa, que puede incluir una expresión lógica ($\texttt{condicion}_i$) y una orden de recibir mensajes
- Las instrucciones `receive` nombran a otros procesos del programa concurrente ($\texttt{proceso}_i$), y cada uno referencia una variable local ($\texttt{variable}_i$), donde eventualmente se recibirá un valor del proceso emisor asociado.

```
select
  when condicion1 receive( variable1, proceso1 ) do
        sentencias1
  when condicion2 receive( variable2, proceso2 ) do
        sentencias2
 ...
  when condicionn receive( variablen, proceson ) do
        sentenciasn
end
```


Manuel I. Capel
manuelcapel@ugr.es



Sintaxis de las guardas

Guarda de una orden Select:

- La expresión lógica puede omitirse:

```
when receive( mensaje, proceso ) do
sentencias
```

o bien, equivalentemente:

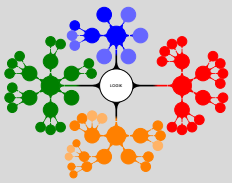
```
when (true) receive( mensaje, proceso ) do
        sentencias
```

- La sentencia receive puede no aparecer:

```
when condicion do
        sentencias
```

entonces, decimos que ésta es una *guarda* sin sentencia de entrada

Manuel I. Capel
manuelcapel@ugr.es



Evaluación y ejecución de las *guardas*

Guarda ejecutable durante la ejecución de un proceso `P`:

- La condición de la guarda se evalúa en ese momento a `true`
- Si tiene sentencia de entrada, entonces el proceso origen nombrado ya ha iniciado en ese momento una sentencia `send` (de cualquier tipo) con destino al proceso `P`, que concuerda con el `receive`

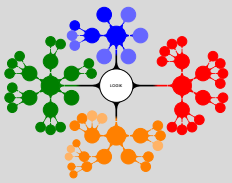
Guarda potencialmente ejecutable durante la ejecución de un proceso `P`:

- La condición de la guarda se evalúa a `true`
- Contiene una sentencia de entrada que nombra a un proceso que aún no ha iniciado la ajecución de una sentencia `send` hacia el proceso `P`

Guarda no ejecutable:

Se refiere al caso restante de evaluación de una guarda, es decir, cuando la condición de la guarda se evalúa como `false`

Manuel I. Capel
manuelcapel@ugr.es



Evaluación de las guardas de una instrucción Select:

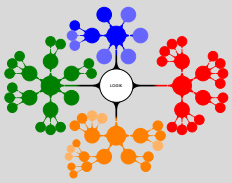
Evaluación de condiciones y estado de los procesos emisores:

Se produce cuando el flujo de control llega a una instrucción **select**, de esta forma se clasifican las guardas y se selecciona una alternativa para su ejecución

Determinación de la alternativa a ejecutar:

1. Si hay guardas ejecutables con sentencia de entrada, se selecciona aquella cuyo `send` se inició antes (esto garantiza a veces la equidad)
2. Si hay guardas ejecutables, pero ninguna tiene una sentencia de entrada, se selecciona *no determinísticamente* una cualquiera
3. Si no hay ninguna guarda ejecutable, pero sí hay guardas potencialmente ejecutables, la instrucción *espera* (suspende el proceso) hasta que alguno de los procesos nombrados en esas guardas inicie un `send`; en ese momento acabará la espera y se seleccionará la guarda correspondiente a ese proceso
4. Si no hay guardas ejecutables ni potencialmente ejecutables, no se puede seleccionar ninguna guarda y el programa suele producir un error o levantar una excepción

Manuel I. Capel
manuelcapel@ugr.es

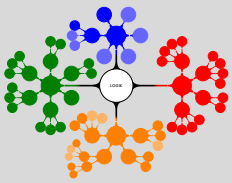


Ejecución de la instrucción select-II

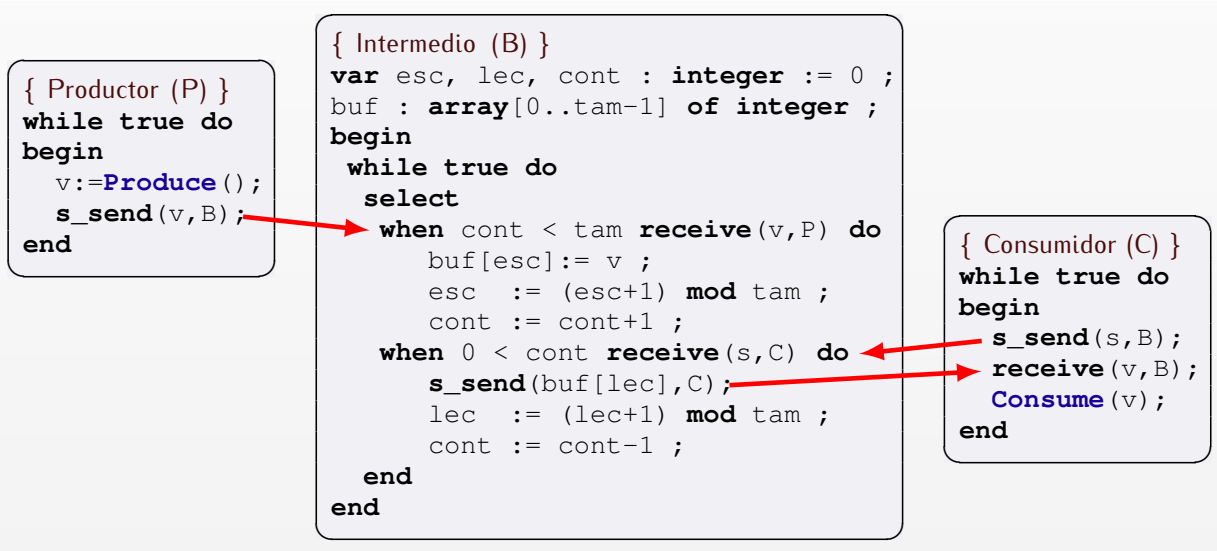
Consideraciones importantes:

- Hay que tener en cuenta que la ejecución de una instrucción `select` conlleva potencialmente esperas y, por tanto, se pueden producir esperas indefinidas (interbloqueo)
- Existe un instrucción **select** con prioridad: la selección de la alternativa a ejecutar dejaría de ser *no determinística*, porque el orden en el que aparecen las alternativas establecerá la prioridad de ejecución de cada una
- Para implementar un proceso servidor, la instrucción **select** ha de programarse dentro de un bucle: en cada iteración se vuelve a evaluar las guardas y se selecciona no deterministícamente 1 de ellas para su ejecución

Manuel I. Capel
manuelcapel@ugr.es



Ejemplo: productor-consumidor con búfer FIFO



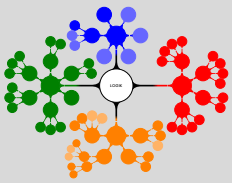
```
{ Productor (P) }
while true do
begin
  v:=Produce();
  s_send(v,B);
end
```

```
{ Intermedio (B) }
var esc, lec, cont : integer := 0 ;
buf : array[0..tam-1] of integer ;
begin
 while true do
  select
   when cont < tam receive(v,P) do
      buf[esc]:= v ;
      esc  := (esc+1) mod tam ;
      cont := cont+1 ;
   when 0 < cont receive(s,C) do
      s_send(buf[lec],C);
      lec  := (lec+1) mod tam ;
      cont := cont-1 ;
  end
end
```

```
{ Consumidor (C) }
while true do
begin
  s_send(s,B);
  receive(v,B);
  Consume(v);
end
```

No se conoce de antemano el orden de las peticiones de inserción/extracción. Las guardas garantizan la propiedad de seguridad en el acceso concurrente al búfer.

Manuel I. Capel
manuelcapel@ugr.es



Instrucción Select con guardas indexadas

Sintaxis

A veces es necesario replicar una alternativa. En estos casos se puede usar una sintaxis que evita reescribir el código muchas veces, con esta sintaxis:

```
for indice := inicial to final
        when condicion receive( mensaje, proceso ) do
                sentencias (indice);
```

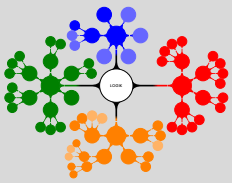
Tanto la condición, como el mensaje, el proceso o las sentencias componentes pueden contener referencias a la variable indice

Equivalencia:

La construcción con índices es equivalente a:

```
when condicion receive( mensaje, proceso ) do
        sentencias { se sustituye indice por inicial }
when condicion receive( mensaje, proceso ) do
        sentencias { se sustituye indice por inicial + 1 }
  ...
when condicion receive( mensaje, proceso ) do
```


Manuel I. Capel
manuelcapel@ugr.es



Ejemplo: Select con guardas indexadas

`suma` es un vector de n enteros, y `fuente[0]`, `fuente[1]`, etc... son n procesos:

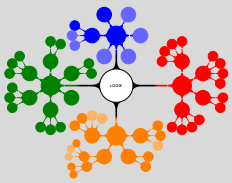
```
for i := 0 to n-1
        when suma[i] < 100 receive( numero, fuente[i] ) do
          suma[i] := suma[i] + numero ;
```

Lo cual es equivalente a:

```
when suma[0] < 100 receive( numero, fuente[0] ) do
        suma[0] := suma[0] + numero ;
when suma[1] < 100 receive( numero, fuente[1] ) do
        suma[1] := suma[1] + numero ;
...
when suma[n-1] < 100 receive( numero, fuente[n ? 1] ) do
        suma[n-1] := suma[n-1] + numero ;
```

En un **select** se pueden combinar una o varias alternativas indexadas con alternativas normales no indexadas.
Resultado de la orden: se recibirá 1 número de un proceso `fuente[i]` si `suma[i]` es menor que 100 (antes de recibir)

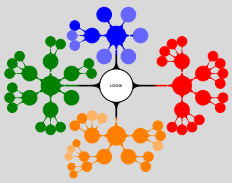
Manuel I. Capel
manuelcapel@ugr.es



Select con sentencia else

Se trata de programar un proceso servidor que sume los números recibidos desde *n* procesos, que se ejecutan cada uno continuamente, hasta que cada suma iguale o supere el valor 100. Las guardas son siempre *ejecutables* hasta llegar a la situación de terminación.

```
var suma : array[0..n-1] of integer := (0,0,...,0) ;
continuar : boolean := true ;
numero : integer ;
  begin
    while continuar do begin
      select
        for i := 0 to n - 1
          when suma[i] < 100 receive( numero, emisor[i] )
              do
           suma[i] := suma[i]+numero ; { sumar }
           continuar := true ; {metainstrucción: no es
               necesaria pero se incluye para clarificar}
          end
          else continuar:= false;
    end
  end
```

Bibliografía

Para más información, ejercicios, bibliografía adicional:

Capítulo 3: Sistemas basados en paso de mensajes de "Programación Concurrente y de Tiempo Real". M.I.Capel(2022), Garceta (Madrid).

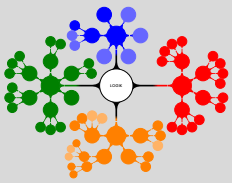
"Paso de mensajes síncrono/asíncrono":
`https://en.wikipedia.org/wiki/Message_passing#Synchronous_versus_asynchronous_message_passing`

"Citas- como mecanismo distribuido síncrono":
`http://www.dalnefre.com/wp/2010/07/message-passing-part-1-synchronous-rendezvous/`

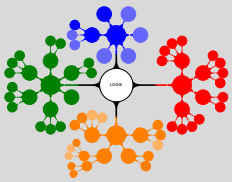
"Canales `core.async` de Clojure":
`https://clojure.org/news/2013/06/28/clojure-clore-async-channels`

"Sentencia de elección no determinística":
`https://www.quora.com/Network-Programming/Network-Programming-How-is-select-implemented`

Manuel I. Capel
manuelcapel@ugr.es



1.7. Tema 4



Tema 4

Introducción a los Sistemas de Tiempo Real

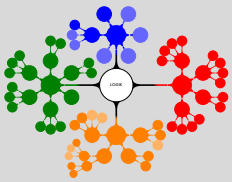
desarrollo de programas con tareas de tiempo real

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 22 de noviembre de 2024

Manuel I. Capel
manuelcapel@ugr.es
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

Manuel I. Capel
manuelcapel@ugr.es

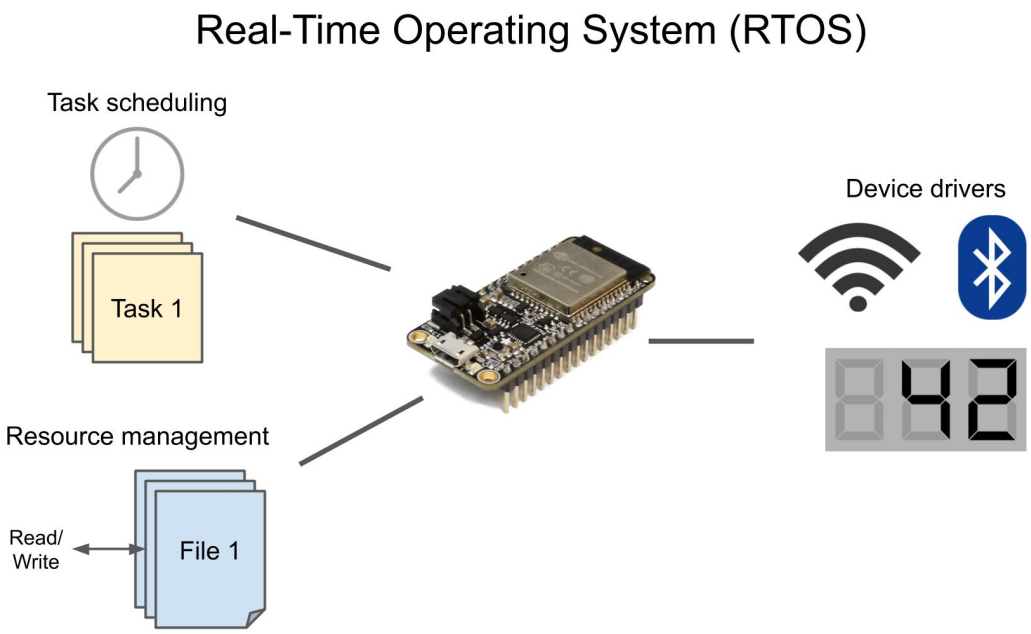


Confusión conceptual acerca de qué es un STR

- Los STR se confunden con sistemas:
  (a) *en línea*, (b)*interactivos*, (c) *rápidos*
- Propiedades definitorias de un STR :
  1. *Reactividad*
  2. *Determinismo*
  3. *Responsividad*
  4. *Confiabilidad*

Un STR es aquél en el que la respuesta correcta a un cálculo realizado por el programa no sólo depende de que efectivamente *haga lo que tenga que hacer*, sino también de cuándo esté disponible dicho resultado.

Definición en el ámbito de los Sistemas Operativos



Sistemas operativos de de tiempo real

Sistema operativo de TR es aquel que tiene la capacidad para suministrar un nivel de servicio requerido en un tiempo limitado y especificado a priori.

Manuel I. Capel
manuelcapel@ugr.es

Clasificación de las aplicaciones de tiempo real basada en varios criterios

- Atendiendo a la *criticidad* de los STR:

Denominación	Ejemplo	Complementos
1. Misión crítica	Control de aterrizaje	Tolerancia a fallos
2. Estrictos	Reservas de vuelos	Calidad de respuesta
3. Permisivos	Adquisición datos meteorológicos	Medidas de fiabilidad

Español	Inglés
Misión Crítica	Hard Real-time Task
Estrictos	Firm Real-time Task
Permisivos	Soft Real-Time Task, Systems with time criticality

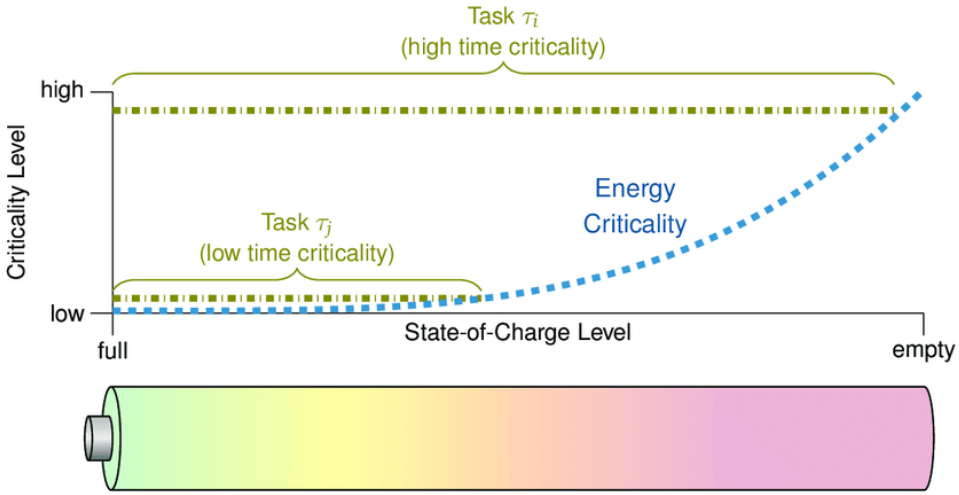
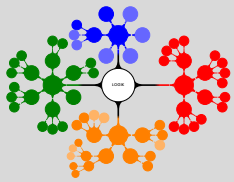
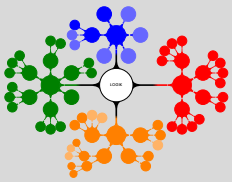


Ilustración de 2 tareas con diferentes niveles de criticidad temporal

Manuel I. Capel
manuelcapel@ugr.es

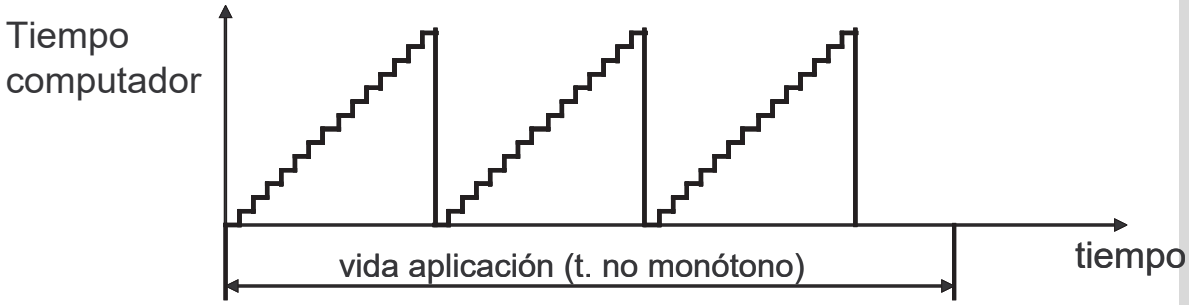


Manuel I. Capel
manuelcapel@ugr.es



Tipos de medidas del tiempo de interés en STR

1. Tiempo absoluto:
   - Sistemas de referencia locales,
   - Astronómicos (`UT0`),
   - Atómicos (`IAT`),
   - Tiempo Coordinado Universal, desde 1972 (`UTC`),
   - Satelital (`GPS`).
2. Intervalos o tiempo relativo:

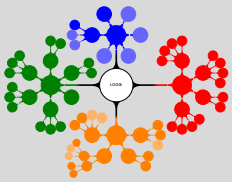


Representación del tiempo no-monótono en un reloj de tiempo real

El tiempo de GPS (GPST) es una escala de tiempo continua (sin segundos intercalares) definida por el segmento de control GPS sobre la base de un conjunto de relojes atómicos en las estaciones de monitorización y a bordo de los satélites. Comenzó a las 0h UTC (medianoche) del 5 al 6 de enero de 1980.

El UTC se obtiene a partir del Tiempo Atómico Internacional (IAT), este estándar se calcula a partir de una media ponderada de las señales de los relojes atómicos, localizados en cerca de 70 laboratorios nacionales de todo el mundo. Se retrasa con

Manuel I. Capel
manuelcapel@ugr.es



Medida del tiempo

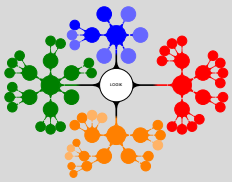
- Concepto de *reloj de tiempo real* : $\{\texttt{oscilador}, \texttt{contador}, \texttt{software}\}$
- Características más importantes:
  - Precisión (granularidad)
  - Tiempo de desbordamiento
  - Intervalo
- Escalas temporales:
  - Tiempo monótono /No monótono
  - Absolutas/no absolutas

Precisión	Intervalo
100 ns.	Hasta 429,5 s.
1 μs.	Hasta 71,58 m.
100 μs.	Hasta 119,3 h.
1 ms.	Hasta 49,71 días
1 s.	Hasta 136,18 años

Intervalo vs.precisión para un contador de 32 bits

Temporizadores

Manuel I. Capel
manuelcapel@ugr.es

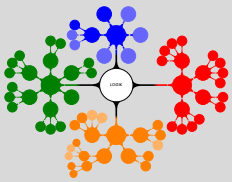


Ideas fundamentales

- Se activan con una llamada (operación) de sistema operativo, **no son instrucciones de los lenguajes de programación**
- Tipo de reloj especializado de los sistemas operativos
- Elementos para programar:
  - Tiempo de *arranque*
  - Tiempo de *parada*

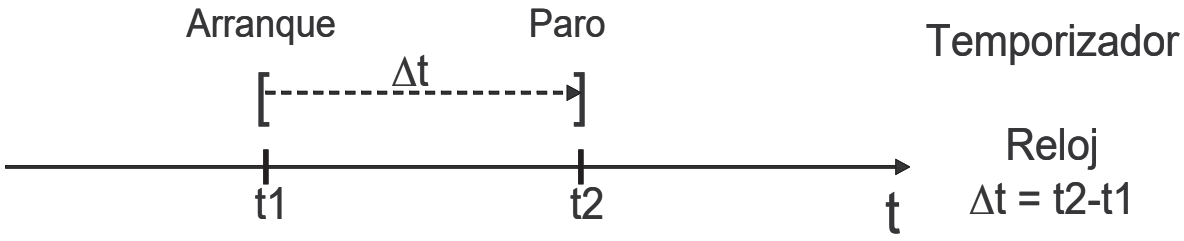
Tipos de temporizadores

Manuel I. Capel
manuelcapel@ugr.es



Características fundamentales:

- De 1 solo disparo
- Periódicos
- Pueden presentar *deriva*



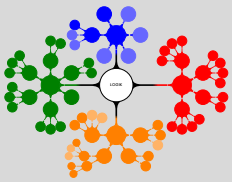
Representación del tiempo no-monótono en un reloj de tiempo real

Retardos en las tareas:

- Se pueden programar con *temporizadores* o con instrucciones de los lenguajes de programación
- Suspenden, al menos, hasta la duración especificada

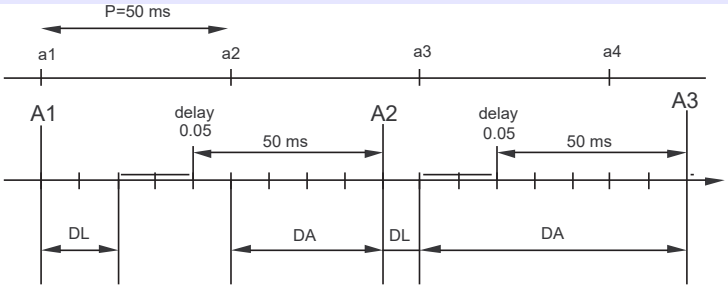
```
tarea T(milliseconds t_computo_p){
  ...
  t_computo= t_computo_p; //p.e.: 100 milisegundos
  ...

  do {

    // accion a realizar simulada
    sleep_for(T_computo);//afectada de deriva local

  while (true);
  }
```


Manuel I. Capel
manuelcapel@ugr.es

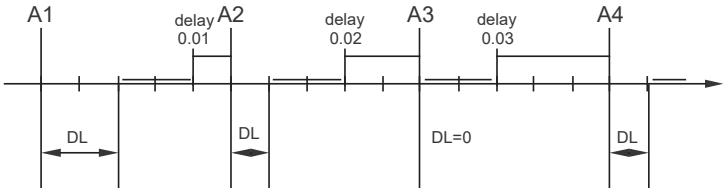


Deriva acumulativa y activación periódica de las tareas

- Problema de la *deriva acumulativa* de las tareas
- Resultaría imposible programar una tarea que se active transcurrido un periodo exacto

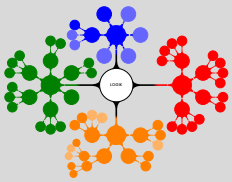


Primera aproximación



Aproximación buena

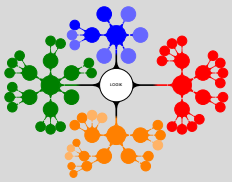
Manuel I. Capel
manuelcapel@ugr.es



Eliminación de la deriva acumulativa

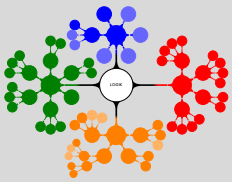
```
tarea Periodica(){
  t_ciclo= 100; //milisegundos
  siguiente_instante= clock()::now();//milisegundos
                                     //funcion del sistema
  do {
    // accion a realizar

    siguiente_instante += t_ciclo;
    sleep_until(siguiente_instante);

  } while (true);
  }
```


Manuel I. Capel
manuelcapel@ugr.es



Tareas y recursos

Manuel I. Capel
manuelcapel@ugr.es



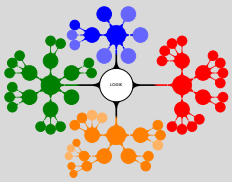
Tipos de elementos de un STR

- Tarea ($\equiv$ "proceso")
  - Sujeta a restricciones de tiempo, definidas por *atributos temporales*
  - Concepto de *activación* de una tarea de TR
  - *Instante de activación*
- Recurso (necesarios para ejecución de las tareas):
  - *Activo*: procesador, red de comunicaciones, ...
  - *Pasivo*: datos, memoria, discos, ...

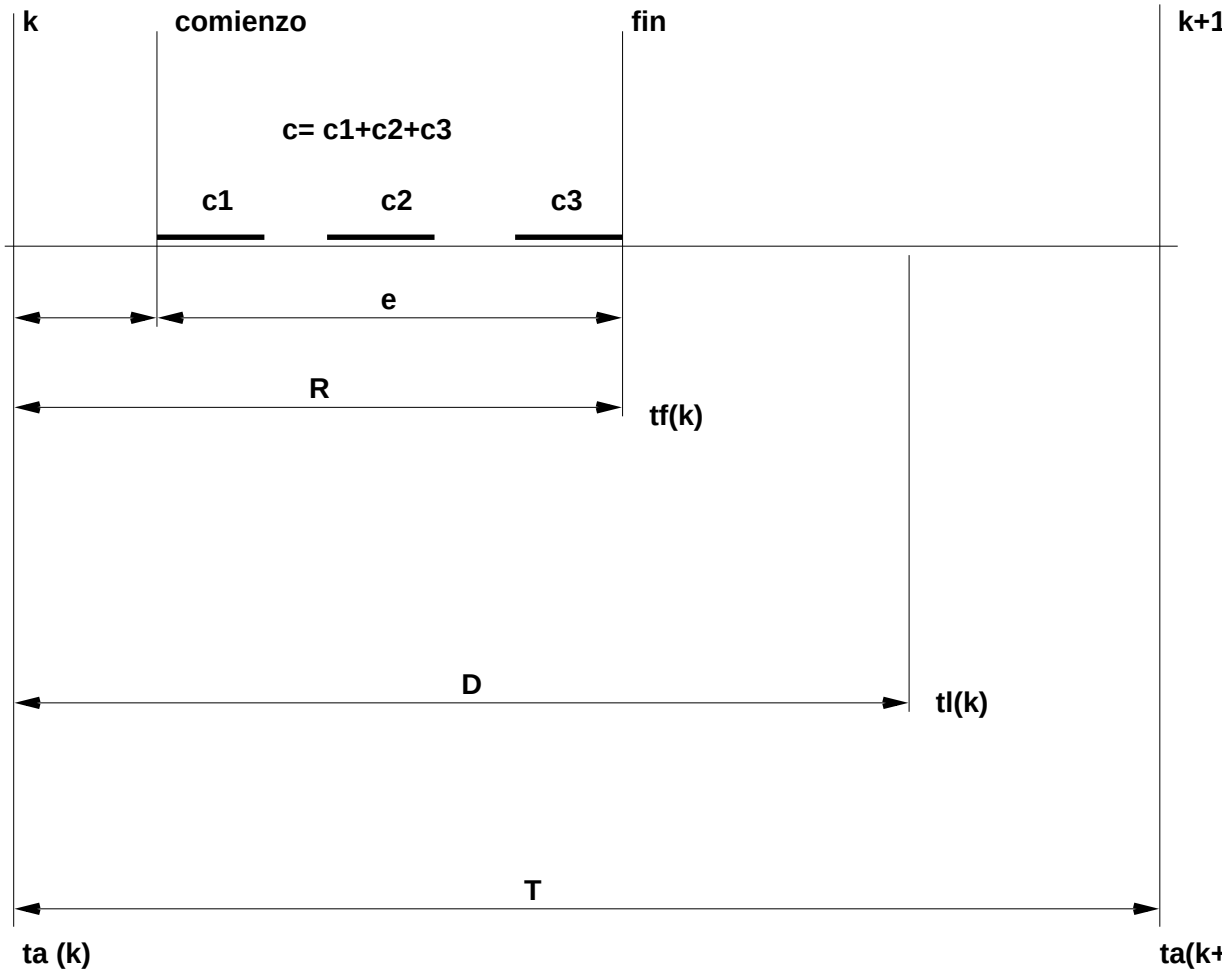
Atributos temporales principales de una tarea

- *Tiempo de cómputo máximo o de ejecución (C_i)*
- *Tiempo de respuesta (R_i)*
- *Plazo de respuesta máximo ("deadline")(D_i)*
- *Periodo (T_i)*

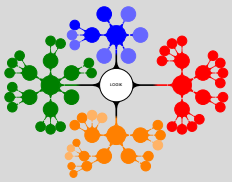
Manuel I. Capel
manuelcapel@ugr.es



Atributos temporales principales de una tarea-II

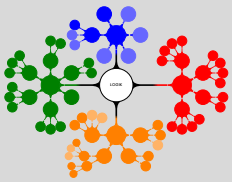


Manuel I. Capel
manuelcapel@ugr.es

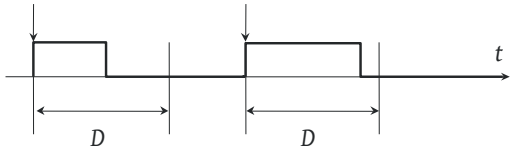


Tipos de tareas

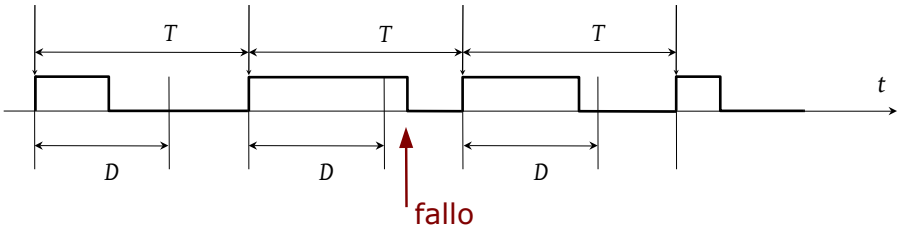
Manuel I. Capel
manuelcapel@ugr.es



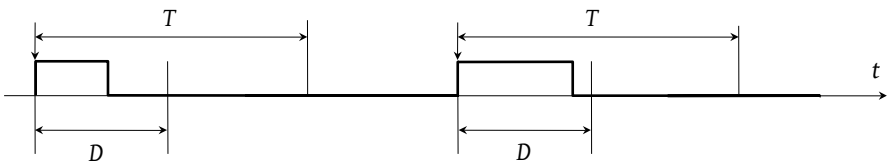
- **Aperiódica:** activación en instantes arbitrarios.



- **Periódica:** repetitiva, T es el **período** (tiempo exacto entre act.)



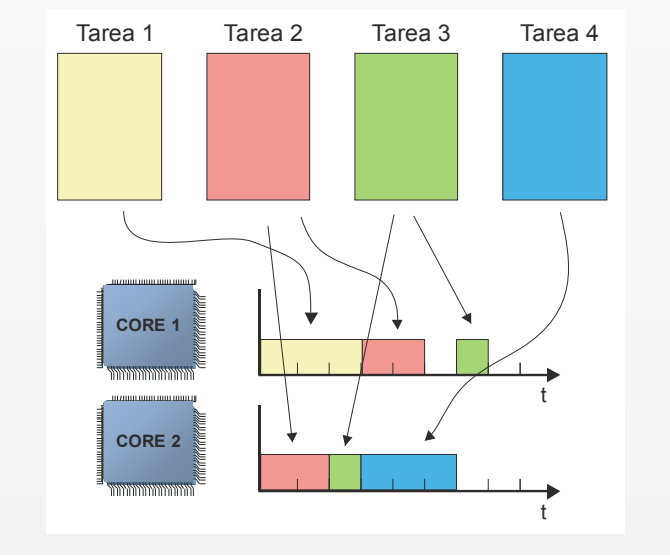
- **Esporádica:** repetitiva, T es intervalo mínimo entre activaciones



Planificación de tareas

Planificación

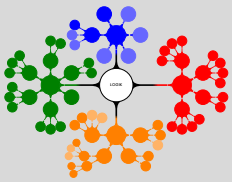
- Asignación de tareas a los recursos activos del sistema para garantizar la ejecución completa de cada tarea sujeto a un conjunto de restricciones definidas



Planificación de tareas de TR

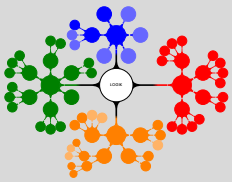
Se ha de garantizar la ejecución completa de cada tarea antes de que expire su plazo de respuesta máximo(D)

Manuel I. Capel
manuelcapel@ugr.es



Problemática de la planificación de tareas de tiempo–real

Manuel I. Capel
manuelcapel@ugr.es

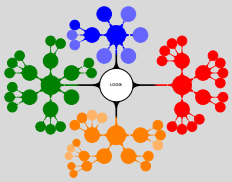


Problema central a resolver:

- **Calcular a priori si, dado un conjunto arbitrario de tareas, estas pueden ejecutarse completamente antes de que termine el plazo de respuesta máximo de cada una de ellas, en cada una de sus activaciones**
- El cálculo anterior para un conjunto de tareas, que se pueden interrumpir unas a otras varias veces durante su ejecución, no tiene una solución matemática sencilla
- Se puede “resolver"de forma *estática*
- Para resolverlo de forma *dinámica* se utiliza un *modelo simple* de tareas de TR

Planificación cíclica

Manuel I. Capel
manuelcapel@ugr.es

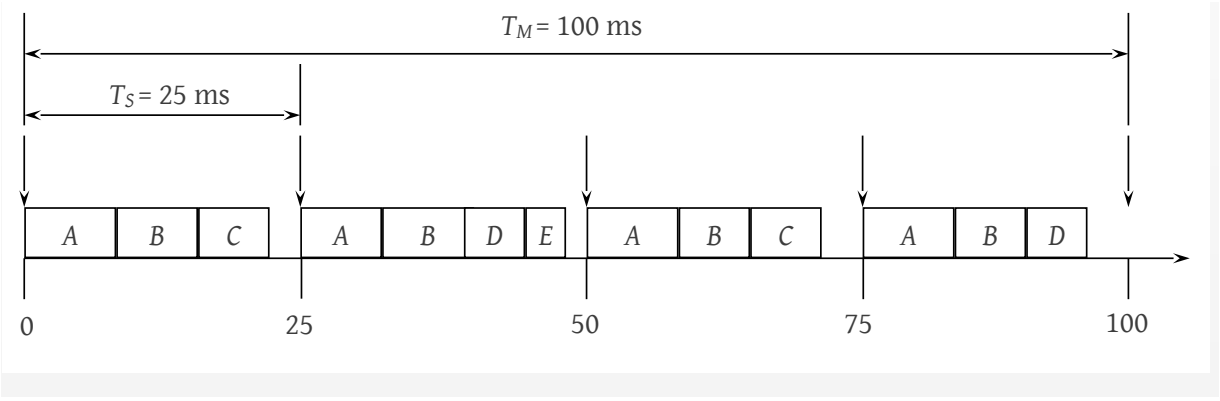


Ejecutivo cíclico

- La planificación cíclica se basa en una estructura de activación fija de las tareas periódicas
- El plan principal es un ciclo que tiene una duración constante denominado hiperperiodo (T_M):
  - $T_M = mcm(T_1, T_2, ..., T_n)$ (mínimo común múltiplo de los periodos de las tareas)
  - La unidad que cuenta el tiempo es siempre **entera** (p.e., ticks de reloj de tiempo real del computador)
  - El ciclo principal, que se repite siempre, se descompone en varios ciclos secundarios de igual duración todos ellos (T_S)

Ejecutivo cíclico

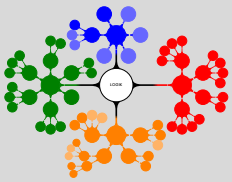
Tarea	T_i	C_i
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2



Suponemos que $D_i = T_i$ para cada tarea τ_i

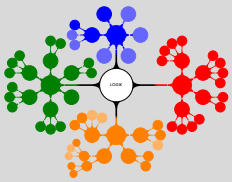
$$T_M = mcm(25, 50, 100)$$

Manuel I. Capel
manuelcapel@ugr.es



Planificación cíclica

Manuel I. Capel
manuelcapel@ugr.es



Ideas fundamentales

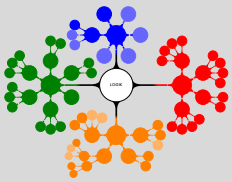
- Plan principal de ejecución de las tareas explícito y *fuera de línea* (“off line”), es decir,
  sin acceso concurrente al procesador según la prioridad de cada tarea
- Establece de una vez cómo se entrelazan las tareas en cualquier ejecución del programa, pero puede variar la ordenación relativa de las tareas en cada *ciclo secundario*
- Cada tarea se ejecuta completamente 1 sola vez dentro de cada repetición de su *periodo*
- Si una tarea se inicia dentro de una iteración del ciclo secundario, ha de terminar antes del final de dicha iteración

Implementación de la planificación cíclica

```
void EjecutivoCiclico()
{
const milliseconds tmp_secundario (25);
const int nciclos = 4 , // número de ciclos secundarios
siguiente_instante= clock::now();
int frame = 0 ; // número del siguiente ciclo secundario
  while( true ){
  for(frame= 1; i<=nciclos; frame++){ // ejecuta la tareas
       del ciclo secundario actual
    switch( frame )
    { case 0 : A(); B(); C(); break ;
      case 1 : A(); B(); D(); E(); break ;
      case 2 : A(); B(); C(); break ;
      case 3 : A(); B(); D(); break ;
    }
     siguiente_instante += tmp_secundario;
    //pasar al siguiente ciclo secundario
    sleep_until(siguiente_instante);
    // espera inicio siguiente periodo de 25 miliseg.
   }//for
  } //while
}
```

- Cada iteración del bucle `for` ejecuta un ciclo secundario
- Cada 4 iteraciones del bucle ejecutan un ciclo principal

Manuel I. Capel
manuelcapel@ugr.es



Diseño del ejecutivo cíclico

Obtención del *buen valor* de T_S:

- Restricciones a tener en cuenta:
  - El periodo del ciclo secundario será un divisor del periodo del ciclo principal:
    existirá un entero k tal que: $T_M = k \cdot T_S$
  - $T_S \geq$ el tiempo de cómputo máximo (Cw) de cualquier tarea,

$$máximo\,(C_1, C_2, ..., C_n) \leq T_S$$

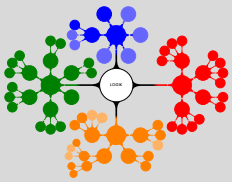
- Sugerencias para conseguirlo:
  - El ciclo secundario debe ser menor o igual que el mínimo plazo de respuesta máximo (D) de todas las tareas:

$$T_S \leq mínimo\,(D_1, D_2, ..., D_n)$$

Acotación del tamaño del ciclo secundario:

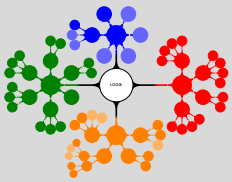
$$máximo\,(C_1, C_2, ..., C_n) \leq T_S \leq mínimo\,(D_1, D_2, ..., D_n)$$

Manuel I. Capel
manuelcapel@ugr.es



Propiedades del ejecutivo cíclico

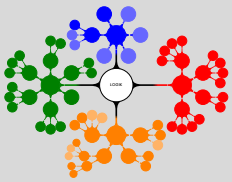
Manuel I. Capel
manuelcapel@ugr.es



- No hay concurrencia en la ejecución:
  - Cada ciclo secundario es una secuencia de llamadas a procedimientos (no se lanza la ejecución de las tareas)
  - No se necesita un núcleo de ejecución multitarea, ni un sistema operativo de tiempo real, etc.
- Los procedimientos acceden secuencialmente a los datos que pudieran compartirse entre las tareas:
  - No se necesitan mecanismos de exclusión mutua como los semáforos o monitores
- No hace falta analizar el comportamiento temporal del programa, es decir, se hace innecesario realizar un análisis de planificabilidad del conjunto de tareas:
  - El sistema es correcto por construcción

Problemas del ejecutivo cíclico

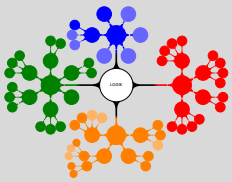
Manuel I. Capel
manuelcapel@ugr.es



- Dificultad para incorporar tareas con periodos largos
- Las tareas esporádicas son difíciles de tratar:
  - Se podría utilizar un servidor de consulta
- El plan cíclico del proyecto es difícil de construir:
  - Si los periodos son de diferentes órdenes de magnitud el número de ciclos secundarios se hace muy grande
  - Puede ser necesario partir una tarea en varios procedimientos
- Es poco flexible y difícil de mantener:
  - Cada vez que se cambia una tarea hay que rehacer toda la planificación

Esquema de planificación de tareas

Manuel I. Capel
manuelcapel@ugr.es

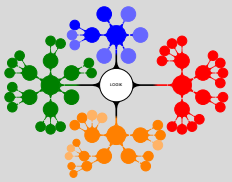


Determinación de la planificabilidad de un conjunto de tareas

- *Algoritmo de planificación*: determina el orden que acceden las tareas a los procesadores
- *Método de análisis de planificabilidad*: se basa en un test que predice si las tareas son planificables conjuntamente, sujetas a las condiciones o restricciones especificadas a priori durante:
  - todos las activaciones posibles de las tareas en su ejecución
  - la ejecución de estas suponiendo la situación más desfavorable o WCET (*Worst Case Execution Time*) en dicha ejecución para cada tarea

Modelo simple de tareas

Manuel I. Capel
manuelcapel@ugr.es

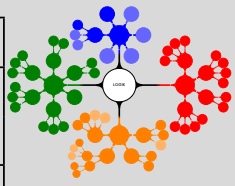


Estructura de las tareas y sus interacciones para resolver el problema de planificabilidad:

- Programa: conjunto fijo de tareas que comparten el tiempo de 1 procesador
- Tareas periódicas, con periodos conocidos
- Independientes entre sí (las tareas no se bloquean por acceder a recursos compartidos)
- Todas las tareas tienen un *plazo de respuesta máximo* (D) que coincide con su periodo (T)
- Los retrasos debidos al sistema (cambios de contexto, etc.) son ignorados
- Los *eventos* de tiempo real no se guardan Tiempo máximo de cómputo de las tareas (C_i): conocido y fijo
- No se contemplan tareas esporádicas o aperiódicas.

Modelo de tareas-II

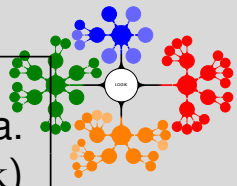
Manuel I. Capel
manuelcapel@ugr.es



Notación	Atributo temporal	Descripción
P	Prioridad	Prioridad asignada a la tarea (si fuera aplicable)
τ	Tarea	Nombre de la tarea
t_a	Tiempo de activación de la tarea (arrival time, request time, release time)	Instante en el que la tarea está para ejecución.
t_s	Tiempo de comienzo (start time)	Instante de tiempo en que comienza realmente su ejecución.
t_f	Tiempo de finalización de la tarea (finishing time)	Instante en el que la tarea finaliza su ejecución.
$t_l,\ d$	Tiempo límite (absolute deadline)	Instante de tiempo límite para la ejecución de la tarea. Es fijo: $t_l(k) = d = t_a + D$

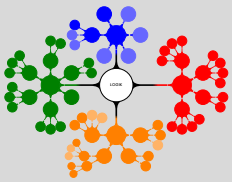
Modelo de tareas-III

Manuel I. Capel
manuelcapel@ugr.es



T	Periodo de ejecución	Intervalo de tiempo entre dos activaciones sucesivas de una tarea periódica. Es un valor fijo, dado por $T = t_a(k+1) - t_a(k)$
J	Latencia	tarea suspendida hasta que se ejecuta: $J(k) = t_s(k) - t_a(k)$
c	Tiempo de cómputo	Tiempo de ejecución de la tarea
C	Cómputo máximo	Tiempo de ejecución de peor caso de la tarea
e	Tiempo de transcurrido	Tiempo transcurrido desde el instante de comienzo hasta la finalización de la tarea Viene dado por: $e(k) = t_f(k) - t_s(k)$
R	Tiempo de respuesta	Tiempo de la tarea hasta completar completamente su ejecución y viene dado: $R(k) = J(k) + e(k)$.

Manuel I. Capel
manuelcapel@ugr.es



Propósito

Planificación (“Task–Scheduling") de Trabajos:

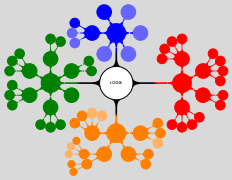
Estudia un conjunto de técnicas de *Programación Entera* para obtener una asignación de recursos y tiempo a actividades de tal modo que se cumplan determinados requisitos de eficiencia.

Para conseguirlo se utiliza una heurística que intenta maximizar una función objetivo: $\mathrm{U(N)}$, denominada *utilización del* procesador para N tareas (de Tiempo-Real en nuestro caso).

Planificación con prioridades

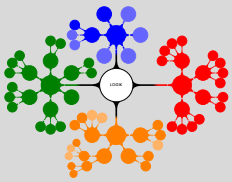
- La planificación con prioridades permite solventar los problemas descritos. Cada tarea tiene asociado un valor entero positivo, llamado prioridad de la tarea:
  - Es un atributo de las tareas normalmente ligado a su importancia relativa en el conjunto de tareas
  - Por convención se asigna números enteros menores a procesos más urgentes
- La prioridad de una tarea la determina sus necesidades temporales; no es importante el rendimiento o comportamiento del sistema
- Una tarea puede estar en varios estados
- Las tareas ejecutables se despachan para su ejecución en orden de prioridad
- No confundir *urgencia* con *criticidad* de una tarea de tiempo real

Manuel I. Capel
manuelcapel@ugr.es



Esquema de planificación de tareas

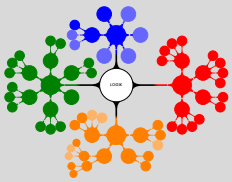
Manuel I. Capel
manuelcapel@ugr.es



1. Algoritmo para ordenar el acceso de las tareas al procesador
2. Un test para predecir el comportamiento del sistema en el *peor caso* de planificación, es decir, cuando exista mayor interferencia entre las tareas
3. Principales características de un esquema de planificación de tareas de TR:
   - Dinámico vs. estático
   - Desplazante (*preemptive*) de tareas menos prioritarias
   - Análisis de las tareas dentro de 1 “ventana temporal"
   - Concepto de **instante crítico**
4. Una posible solución al problema del análisis temporal de las tareas esporádicas y aperiódicas

Planificación con prioridades-II

Manuel I. Capel
manuelcapel@ugr.es



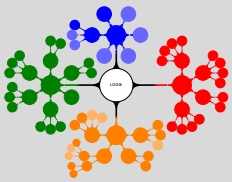
Tipos de esquemas de planificación:

- **Prioridades asignadas estáticamente a las tareas**: Cada tarea tiene asociada una prioridad fija durante toda la ejecución del sistema:
  - *Cadencia monótona* o RMS (Rate Monotonic Scheduling): prioridad a la tarea más frecuente (menor periodo de activación)
  - *Plazo de respuesta monótono* o DMS (Deadline Monotonic Scheduling): prioridad a la tarea más urgente, es decir, con menor plazo de respuesta máximo D_i

Modelo de tareas-IV

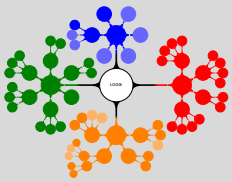
D	Plazo de respuesta máximo(relative deadline)	Define el máximo intervalo de tiempo o máximo tiempo de respuesta
ϕ	Desplazamiento o fase	Tiempo para activarse por primera vez
RJ	Fluctuación relativa o *jitter*(relative release jitter)	Máxima desviación en el tiempo de comienzo entre 2 activac. de una tarea Viene definido por: $\mathrm{RJ} = \max((\mathrm{t_s(k+1)} - t_a(k+1)) - (t_s(k) - t_a(k)))$
H	Holgura (laxity, slack time)	Tiempo de permanecer activa dentro del plazo de respuesta máximo: $\mathrm{H(k)} = \mathrm{t_l} - \mathrm{t_a(k)} - \mathrm{e(k)} = \mathrm{D} - \mathrm{e(k)}$

Manuel I. Capel
manuelcapel@ugr.es



Planificación con prioridades-III

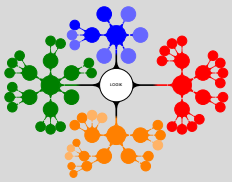
Manuel I. Capel
manuelcapel@ugr.es



Tipos de esquemas de planificación:

- **Prioridades asignadas dinámicamente a las tareas**: Las prioridades de las tareas cambian durante la ejecución del sistema:
  - *Primero la de tiempo límite más cercano* o EDF: (Earliest Deadline First):
    prioridad a la tarea que tenga siguiente plazo de respuesta absoluto (absolute deadline) más próximo a expirar
  - Primero la de menor holgura temporal o LLF (Least Laxity First):
    prioridad a la tarea que le queda menos tiempo durante el que puede permanecer *activa* (antes de que expire su plazo de respuesta máximo)

Manuel I. Capel
manuelcapel@ugr.es



Esquema de planificación de tareas de TR basado en el algoritmo de *cadencia monótona*

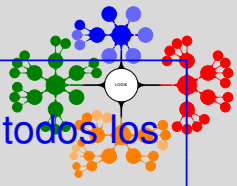
1. Algoritmo de cadencia monótona (*Rate Monotonic Scheduling*)(RMS)
   - Prioridad estáticamente asignada a cada tarea (τ_i), que es tanto mayor cuanto menor es su periodo (T_i)

$$\forall i, j: \; T_i < T_j \Rightarrow P_i > P_j$$

   - No se atiende a la *criticidad* de las tareas, sino a su *urgencia* ($= \frac{1}{T_i}$)
   - Este algoritmo es óptimo entre los algoritmos de asignación estática de prioridades si las tareas son periódicas y el plazo de respuesta coincide con el periodo ($D_i = T_i$)
2. El **test de planificabilidad** se basa en la suma del **Factor de Utilización** del procesador para cada tarea:

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{T_i}\right)$$

Manuel I. Capel
manuelcapel@ugr.es



1. Test de planificabilidad de un conjunto de tareas $\{\tau_1, \tau_2, \ldots \tau_N\}$:

En un sistema de N tareas periódicas, independientes, con prioridades asignadas en orden de su frecuencia (e inverso al periodo), se cumplen todos los tiempos límite de las tareas, para cualquier desfase (ϕ_i) inicial de cada tarea si el factor de*utilización del procesador* ($U(N)$) no supera la *utilización prefijada máxima* ($U_0(N)$) para ese número de tareas N:

$$U = \sum_{i=1}^{N} (\frac{C_i}{T_i}) < U_0 = \quad N \times (2^{\frac{1}{N}} - 1)$$

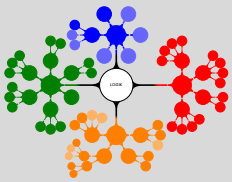
- Un conjunto de tareas de tiempo real pasa el test si el factor de utilización es menor o igual que el factor máximo:

$$U \leq U_0(N)$$

- En este caso el sistema es planificable. En caso contrario, no se puede afirmar nada (es sólo *condición suficiente*).

Inexactitud del test de planificabilidad RM

Manuel I. Capel
manuelcapel@ugr.es

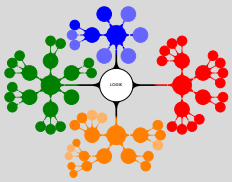


N	límite utilización
1	100
2	82,85 %
3	78,0 %
4	75,7 %
5	74,3 %
10	71,8 %
$\rightarrow \infty$	69,3 %

El valor del límite de utilización máxima $\mathrm{U_0(N)}$ del procesador depende del número de tareas N

Condición suficiente de planificabilidad

Manuel I. Capel
manuelcapel@ugr.es



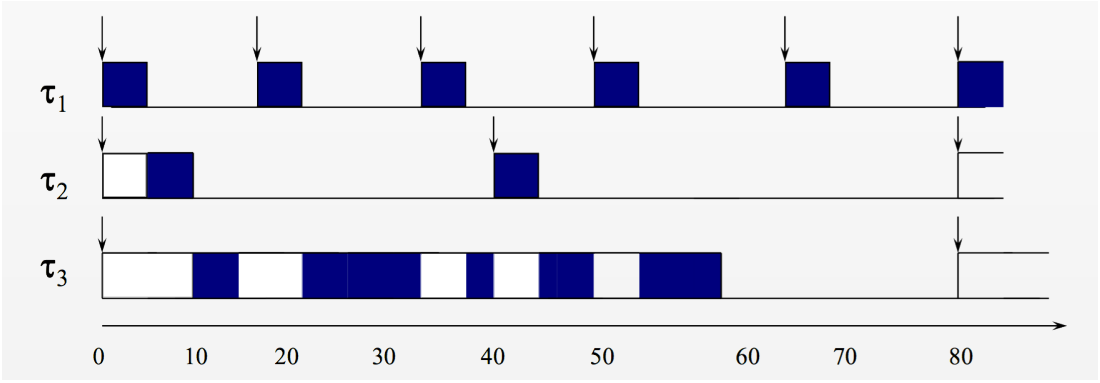
Características del test RMS

- Se dice que el test anterior es sólo una condición suficiente para determinar si un conjunto de tareas es planificable
- Se utilizan diagramas de Gantt con una ventana temporal M_i = **m.c.m.**$\{T_1\ T_2\ ...\ T_i\}$ para comprobar si un conjunto de tareas que no ha pasado el test anterior aún puede resultar planificable
- Los tests de planificabilidad basados en la utilización del procesador no proporcionan información acerca del tiempo de respuesta de la tareas

Ejemplo de RMS con el test de Liu y Layland

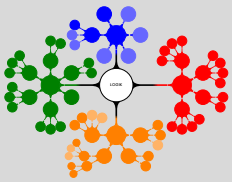
Tarea	C	D	T	C/T
τ_1	4	16	16	0,250
τ_2	5	40	40	0,125
τ_3	32	80	80	0,400

El sistema pasa el test: $U = 0,775 \leq 0,779 = U_0(3)$



(No sería necesario realizar el diagrama de Gantt en este caso)

Manuel I. Capel
manuelcapel@ugr.es



Ejemplo de RMS sin pasar el test de Liu y Layland

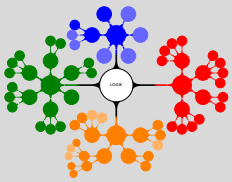
Tareas	C	D	T
Tarea 1	10	30	30
Tarea 2	10	40	40
Tarea 3	10	50	50

Calculamos U y lo comparamos con $U_0(3)$:

$$U = \sum_{i=1}^{N}(\frac{C_i}{T_i}) = \frac{10}{30} + \frac{10}{40} + \frac{10}{50} = 0,783333 \not< 0,779 = U_0(3)$$

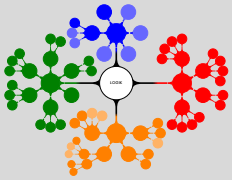
- El sistema no pasa el test
- No podemos decir si es planificable o no lo es
- Intentamos el diagrama de Gantt

Manuel I. Capel
manuelcapel@ugr.es

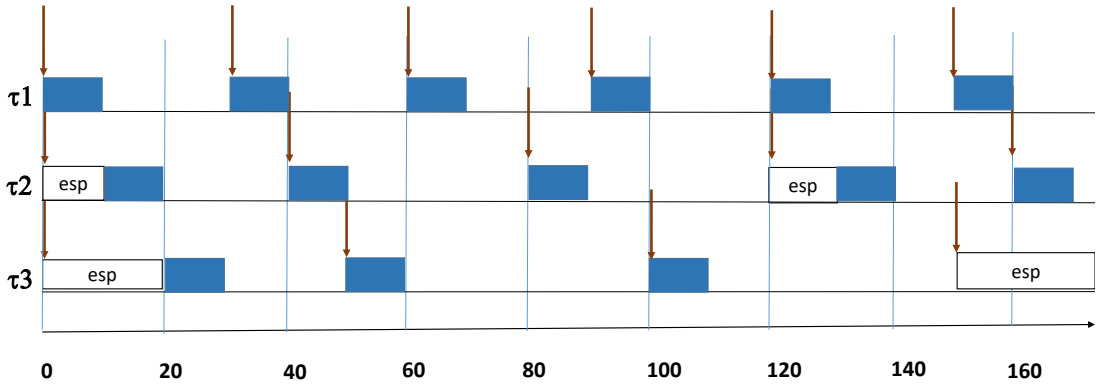


Ejemplo de RMS sin pasar el test de Liu y Layland-II

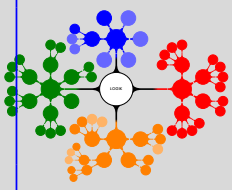
Manuel I. Capel
manuelcapel@ugr.es



- Para analizar la planificabilidad del sistema con dichas restricciones, hay que hacer el cronograma y verificar que:
  - para cada tarea i, se cumple el plazo de respuesta: R < Di
  - esto se debe verificar para un hiperperiodo (después se repite). En el ejemplo, se debería hacer el cronograma completo hasta t = 600 (= mcm{30, 40, 50}), aquí vemos una primera parte (hasta t = 160).



Manuel I. Capel
manuelcapel@ugr.es



Segundo teorema de planificabilidad de un conjunto de tareas $\{\tau_1, \tau_2, \ldots \tau_N\}$: periódicas

En un sistema de N tareas periódicas, independientes, con prioridades asignadas en orden de su frecuencia, se cumplen todos los tiempos límite de las tareas, si cuando se activan todas ellas simultáneamente, cada tarea acaba antes de que expire el tiempo límite que tiene asignado en su primera activación.

Con el test anterior se puede llegar a una utilización del procesador máxima:

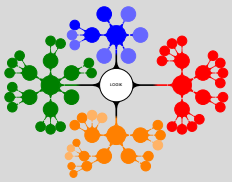
$$\Sigma_{i=1}^{N}(\frac{C_i}{T_i}) \leq 1$$

- La cota máxima que puede alcanzar U es ahora mayor:
  - puede aplicarse a más conjuntos de tareas que con la planificación RMS porque éstas pueden tener mayor factor de utilización de procesador y, sin embargo, pasan el test
- El test ahora es *exacto*: expresa una condición necesaria y suficiente para que un conjunto de tareas sea planificable

Ventajas de los esquemas de planificación estáticos

- Un esquema estático resulta ser más sencillo y eficiente de implementar
- Resulta más sencillo que diseñar un ejecutivo de tareas con tiempos límite (*plazos de respuesta absolutos*) calculados exactamente
- Permite incorporar otros factores que influyen en la planificación de un conjunto de tareas cuando sus prioridades no están asociadas a un tiempo límite
- Durante sobrecarga transitoria, un esquema de planificación estático normalmente resulta ser predicible. Lo cual no ocurre necesariamente si se utiliza un esquema dinámico (EDF)

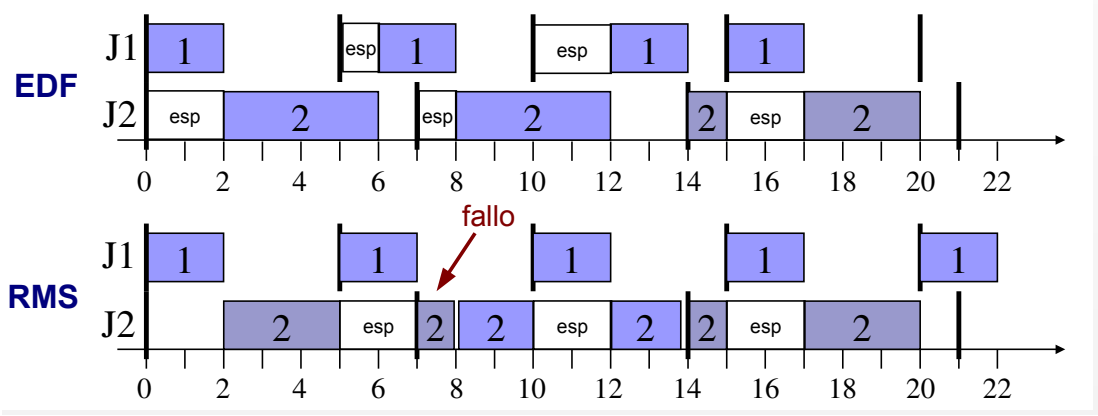
Manuel I. Capel
manuelcapel@ugr.es



Ejemplo de planificación dinámica: EDF

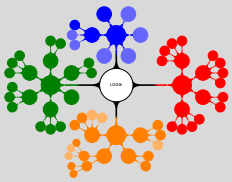
Tarea	C	D	T
χ_1	2	5	5
χ_2	4	7	7

El sistema pasa el segundo test de Liu & Layland (EDF): $U = \frac{2}{5} + \frac{4}{7} = 0{,}971 < 1{,}0$, pero no se cumple la condición que establece el segundo teorema para poder asegurar su planificabilidad



Se verifica que el sistema es planificable con EDF, pero no con RMS

Manuel I. Capel
manuelcapel@ugr.es

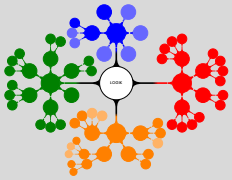


Modelo general de tareas de TR

Estructura general que se puede asumir para resolver el problema de planificabilidad:

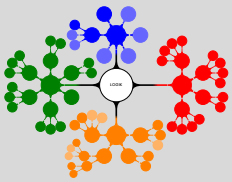
- Programa: conjunto fijo de tareas que comparten el tiempo de 1 procesador, pero pueden aparecer tareas aperiódicas o esporádicas
- Las tareas pueden bloquearse por acceder a recursos compartidos (por ejemplo, comparten semáforos en su código)
- Las tareas tienen un *plazo de respuesta máximo* (D) que, en general, no coincide con su periodo (T) Por ejemplo las tareas esporádicas suelen tener un plazo de respuesta máximo (D) muy corto
- Los retrasos debidos al sistema (cambios de contexto, etc.) son ignorados
- Los *eventos* de tiempo real no se guardan. Tiempo máximo de cómputo de las tareas (C): conocido y fijo.

Manuel I. Capel
manuelcapel@ugr.es



Inversión de prioridad

Manuel I. Capel
manuelcapel@ugr.es

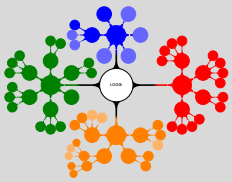


En qué consiste el problema:

- La tarea más prioritaria del conjunto espera durante un tiempo arbitrariamente largo porque se ejecutan continuamente tareas menos prioritarias, mientras la primera se mantiene bloqueada
- Si se produce la denominada *inversión de prioridad*, invalida cualquier previsión sobre la planificación del conjunto de tareas, incluso si han pasado el test RMS
- Se produce debido al esquema estático de asignación de prioridades a las tareas
- La inversión de prioridad no puede ser eliminada completamente, cuando se produce, pero sus efectos adversos sobre la planificación pueden ser minimizados

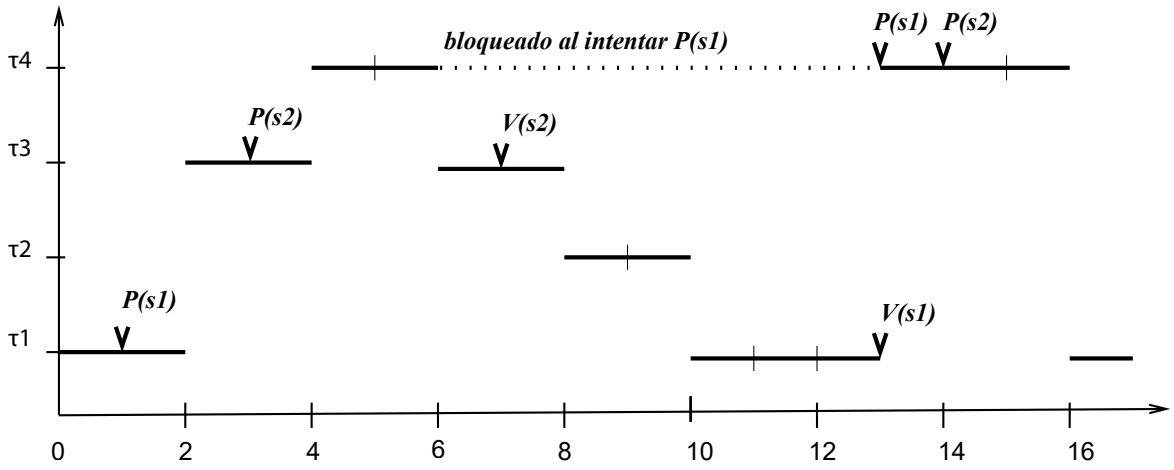
Escenario con Inversión de la prioridad de tareas

Manuel I. Capel
manuelcapel@ugr.es



Ejemplo de "fuerte" inversion de prioridad de la tarea t4

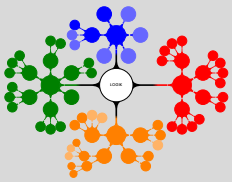
	Ci	Pi	Ii
τ4	5	1	4
τ3	4	2	2
τ2	2	3	2
τ1	6	4	0



Representación de la inversión de prioridad entre tareas periódicas con prioridades estáticas

Sección crítica no expulsable

Manuel I. Capel
manuelcapel@ugr.es



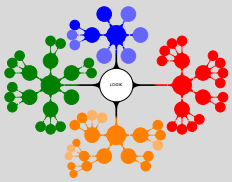
En qué consiste este protocolo:

- Las secciones críticas se ejecutan con una prioridad estática igual a la prioridad máxima del sistema

Características:

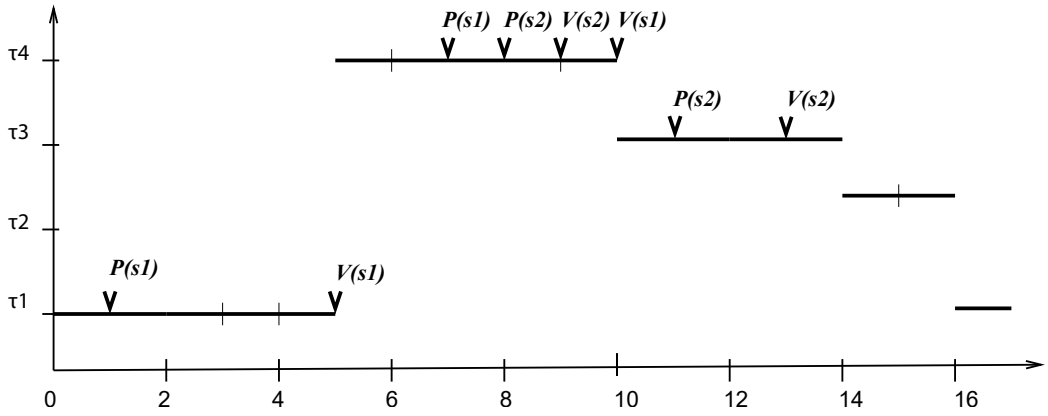
- Es el protocolo más simple para evitar la *inversión de prioridad*
- Puede inducir bloqueos excesivamente largos de las tareas más prioritarias
- Este protocolo puede llegar a interferir en (es decir, impedir) la ejecución de todas las tareas, incluso si no llegan a hacer uso de los recursos compartidos

Manuel I. Capel
manuelcapel@ugr.es



Ejemplo de seccion critica "no−expulsable"

	Ci	Pi	Ii
τ4	5	1	4
τ3	4	2	2
τ2	2	3	2
τ1	6	4	0



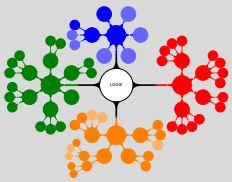
Representación de tareas que usan la sección crítica no expulsable

Nota:

prioridad del sistema= prioridad (τ_4)= 1 (máxima)

Tiempo de bloqueo de la sección crítica no expulsable

Manuel I. Capel
manuelcapel@ugr.es



- Para la tarea τ_i, su tiempo de ejecución de peor caso se incrementará en un factor constante:

$$C_i^* = C_i + B_i$$

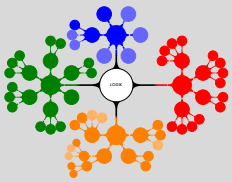
- La tarea más prioritaria τ_i puede ser bloqueada, como máximo, durante la ejecución de 1 sección crítica:

$$B_i = max_{j,\, j>i}(max_k(Dur(s_{jk}))$$

Donde s_{jk} es la sección crítica *k* ejecutada por la tarea τ_j menos prioritaria que τ_i y $Dur(s_{jk})$ es la duración de dicha sección crítica.

Herencia de prioridad

Manuel I. Capel
manuelcapel@ugr.es



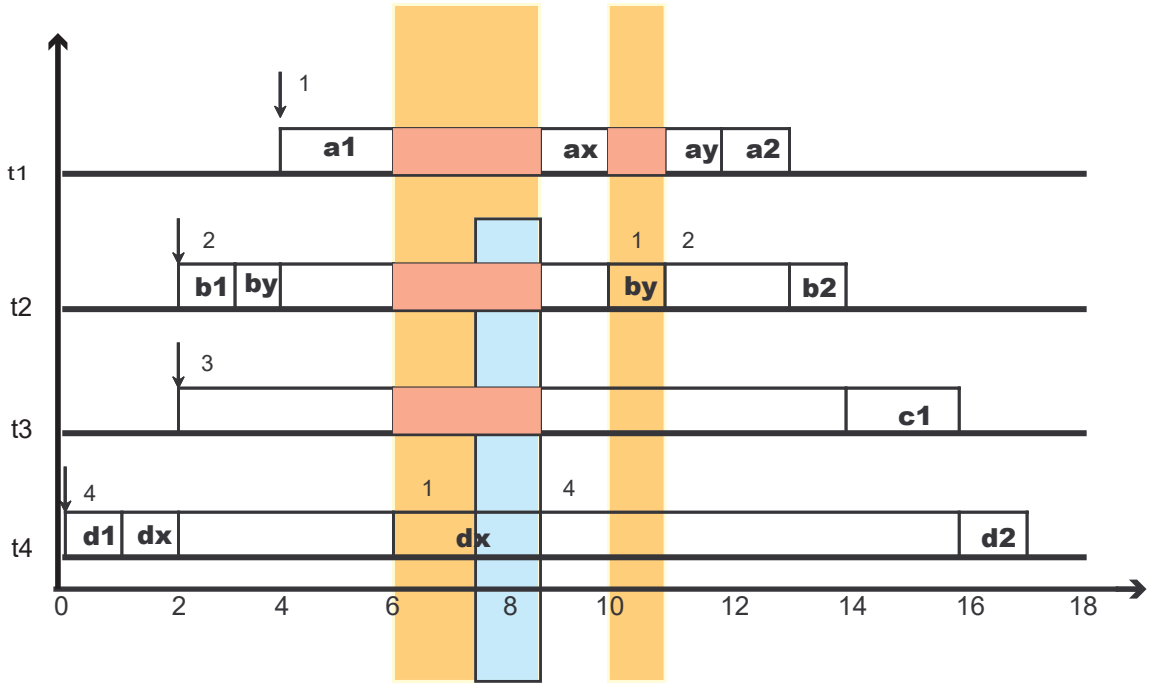
En qué consiste este protocolo:

- La prioridad efectiva de una tarea del programa será el máximo entre su prioridad por defecto y las prioridades de las demás tareas con las que comparte recursos y que tiene bloqueadas en ese momento

Características:

- Las prioridades de las tareas no se mantienen estáticas durante todo el programa
- De esta forma, se consigue minimizar el efecto de la inversión de prioridad y se optimiza el aprovechamiento del tiempo del procesador

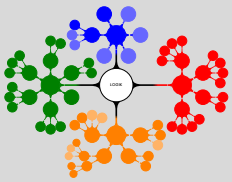
Herencia de prioridad



Tareas con el protocolo de *herencia de prioridad*
Nota: los valores de las prioridades son: $P_i = 1$ (más prioridad)...
$P_i = 4$(menos prioridad)

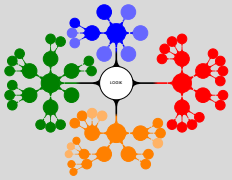
- `t4`: posee una sección crítica (**x**) de 4 unidades de tiempo
- `t1`: posee 2 secciones críticas: (**x**) e (**y**), ambas de 1 unidad temporal
- `t2` sufre un *bloqueo indirecto* y posee una sección (**y**) de 2 unidades temporales

Manuel I. Capel
manuelcapel@ugr.es



Tipos de Bloqueos con el Protocolo de *Herencia de Prioridad*

Manuel I. Capel
manuelcapel@ugr.es



Tipos de bloqueos que se pueden dar con este protocolo:

(a) Bloqueos directos,
(b) Bloqueos indirectos

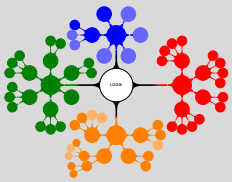
Características de este protocolo

Las tareas sólo pueden verse bloqueadas un número limitado de veces por otras menos prioritarias, en consecuencia:

1. Si una tarea tiene definidas en su código M secciones críticas, entonces el número máximo de veces que puede verse bloqueada durante su ejecución es M.
2. Si hay sólo $N < M$ tareas menos prioritarias, el máximo número de bloqueos que puede experimentar la tarea más prioritaria se reducirá a N.

Cálculo del Factor de Bloqueo

Manuel I. Capel
manuelcapel@ugr.es



El factor de bloqueo vendrá dado como el mínimo entre 2 términos:

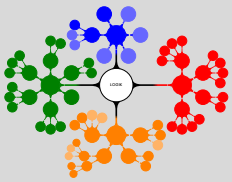
1. B_i^l: bloqueo debido a tareas τ_j menos prioritarias, que acceden a secciones críticas *k* compartidas con tareas más prioritarias [1]:
   - $B_i^l = \sum_{j=i+1}^{n} \max_k[Dur_{j,k} : Limite(S_k) \geq P_i]$
2. B_i^s: bloqueo debido a todas las secciones críticas a las que accede la tarea τ_i,
   - $B_i^s = \sum_{k=1}^{m} \max_{j>i}[Dur_{j,k} : Limite(S_k) \geq P_i]$

   Factor de bloqueo de la tarea τ_i como: $B_i = Min(B_i^l, B_i^s)$.

[1] aunque no necesariamente la comparten con la tarea τ_i que estamos considerando, ya que su prioridad se podría elevar indirectamente.

Protocolos de Techo de Prioridad

Manuel I. Capel
manuelcapel@ugr.es



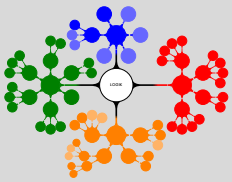
Techo de prioridad de un recurso: es la prioridad de la tarea más prioritaria que puede bloquear al recurso

- Estos protocolos garantizan que una tarea prioritaria sólo puede ser bloqueada como máximo 1 vez por otras de menor prioridad
- Se previenen los interbloqueos
- También los bloqueos transitivos
- Se asegura el acceso en exclusión mutua a los recursos compartidos

Protocolo de techo de prioridad inmediato (PPP)

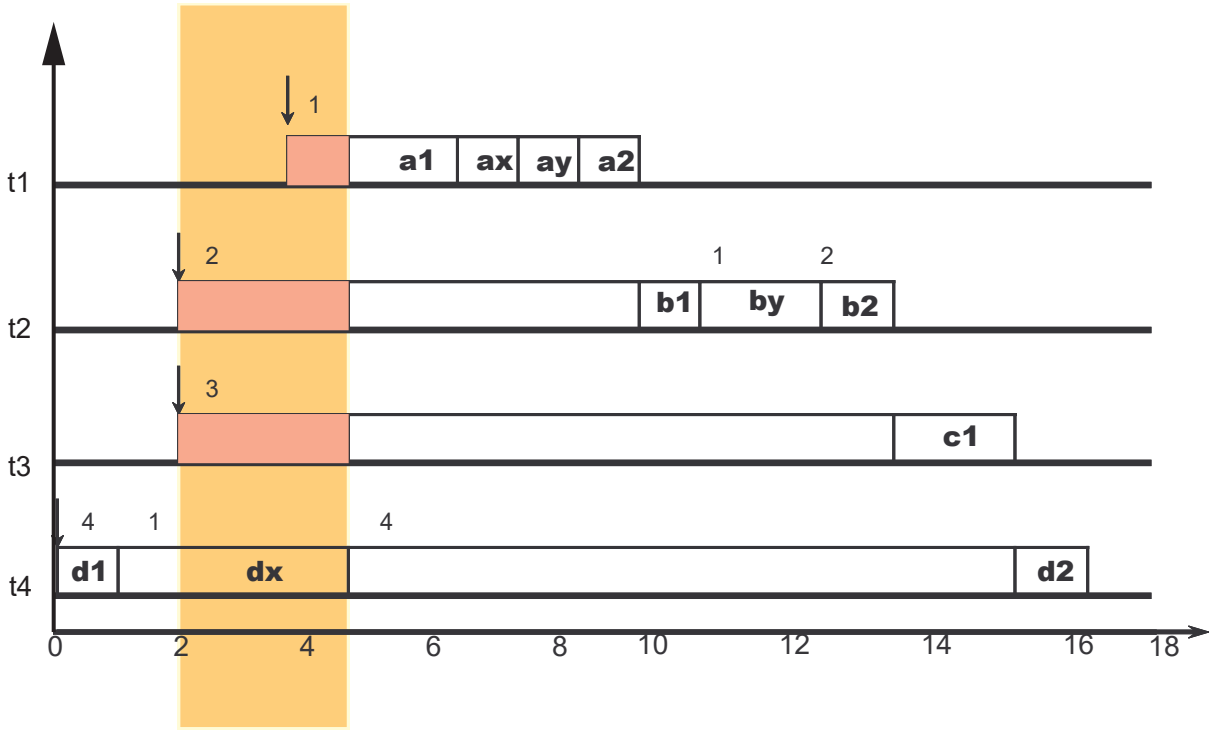
1. Cada tarea tiene una prioridad estática asignada por defecto.
2. Cada recurso tiene un valor de techo de prioridad definido.
3. Una tarea tiene una prioridad dinámica que será el máximo entre su propia prioridad estática inicial y los valores de los techos de prioridad de los recursos que mantenga bloqueados.

Manuel I. Capel
manuelcapel@ugr.es



Protocolo de Techo de prioridad

Manuel I. Capel
manuelcapel@ugr.es



Tareas con el protocolo de *techo de prioridad inmediato*

- `t4`: posee una sección crítica (**x**) de 4 unidades de tiempo

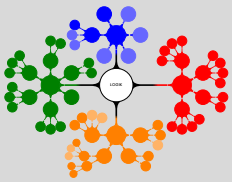
Tiempo de bloqueo

- El valor máximo del tiempo de bloqueo de una tarea es igual a la duración de la sección crítica más larga a las que acceden las tareas de prioridad inferior y que posea un techo de prioridad no inferior a la prioridad de la tarea en cuestión:

$$B_i = Max_{\{j,k\}} \{Dur_{j,k} \mid prio(\tau_j) < prio(\tau_i), techo\_prioridad(S_k) \geq prio(\tau_i)\}$$

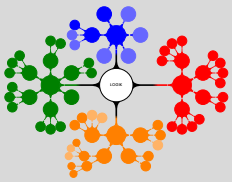
- Con el protocolo PPP, una tarea puede verse bloqueada por otra menos prioritaria, aunque no accedan a recursos comunes

Manuel I. Capel
manuelcapel@ugr.es



Asignación de Prioridades a las Tareas Aperiódicas

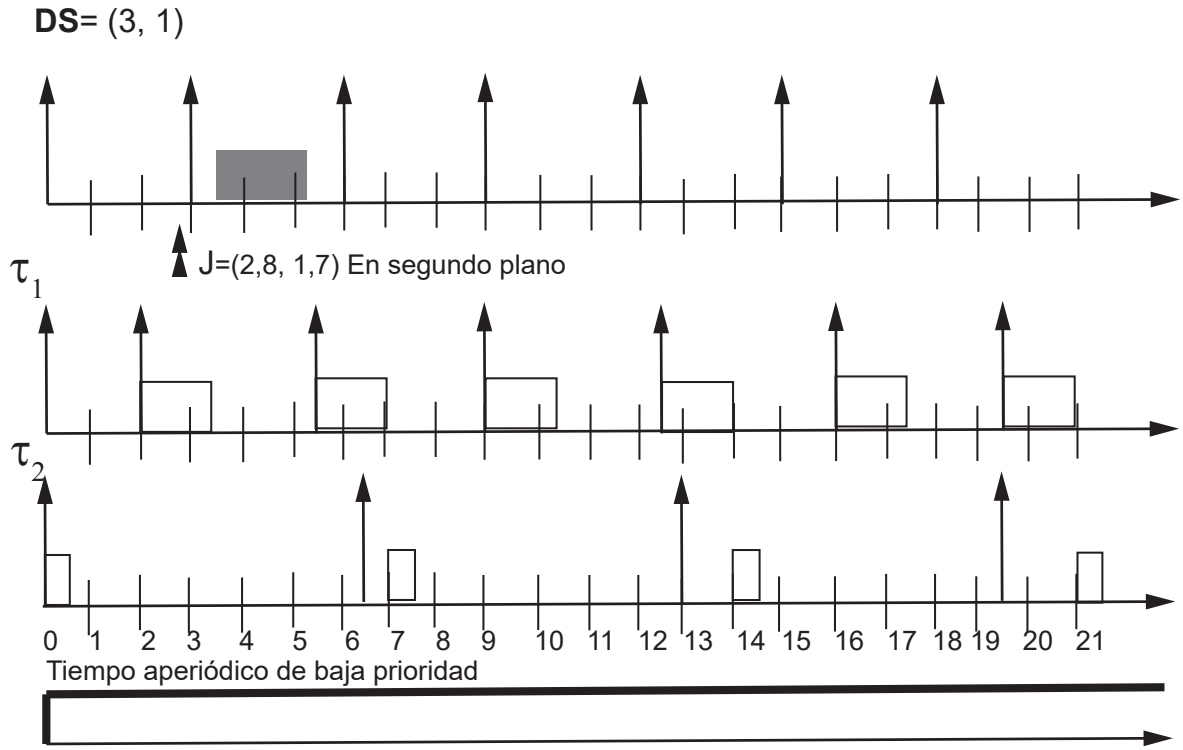
Manuel I. Capel
manuelcapel@ugr.es



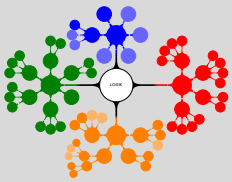
- A las tareas aperiódicas en una aplicación de tiempo real no se les puede asignar una prioridad inferior a la de las tareas de *misión crítica*
- Utilización de un *servidor aperiódico*, que puede ser una tarea real o conceptual, que permita ejecutar a los procesos aperiódicos tan pronto como sea posible

Servicio de aperiódicas con tiempo en segundo plano

- El tiempo aperiódico se incrementa a intervalos regulares
- Tiene la prioridad más baja
- Cuando no hay peticiones aperiódicas se perderá

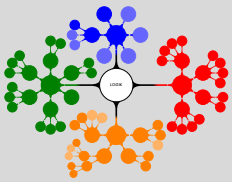


Manuel I. Capel
manuelcapel@ugr.es



Servicio de Aperiódicas con Sondeos

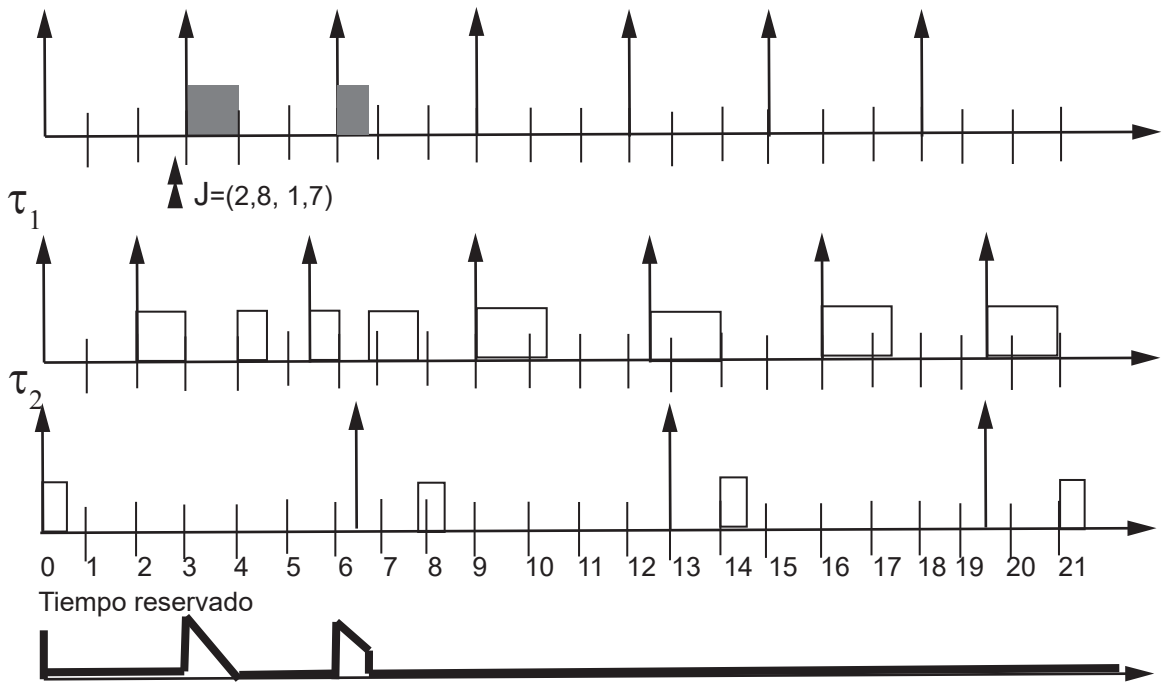
Manuel I. Capel
manuelcapel@ugr.es



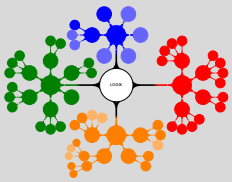
- Se añade una tarea *sondeante* a las tareas periódicas, con periodo T_s y tiempo de ejecución de peor caso: C_s
- Se le asigna la prioridad (que convenga)
- Se aplica el test de planificabilidad de cadencia monótona incluyendo la tarea sondeante:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq (N+1)[2^{\frac{1}{(N+1)}} - 1]$$

Tareas Aperiódicas y Tarea Sondeante



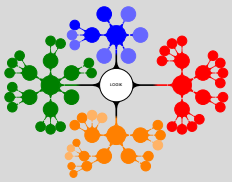
Manuel I. Capel
manuelcapel@ugr.es



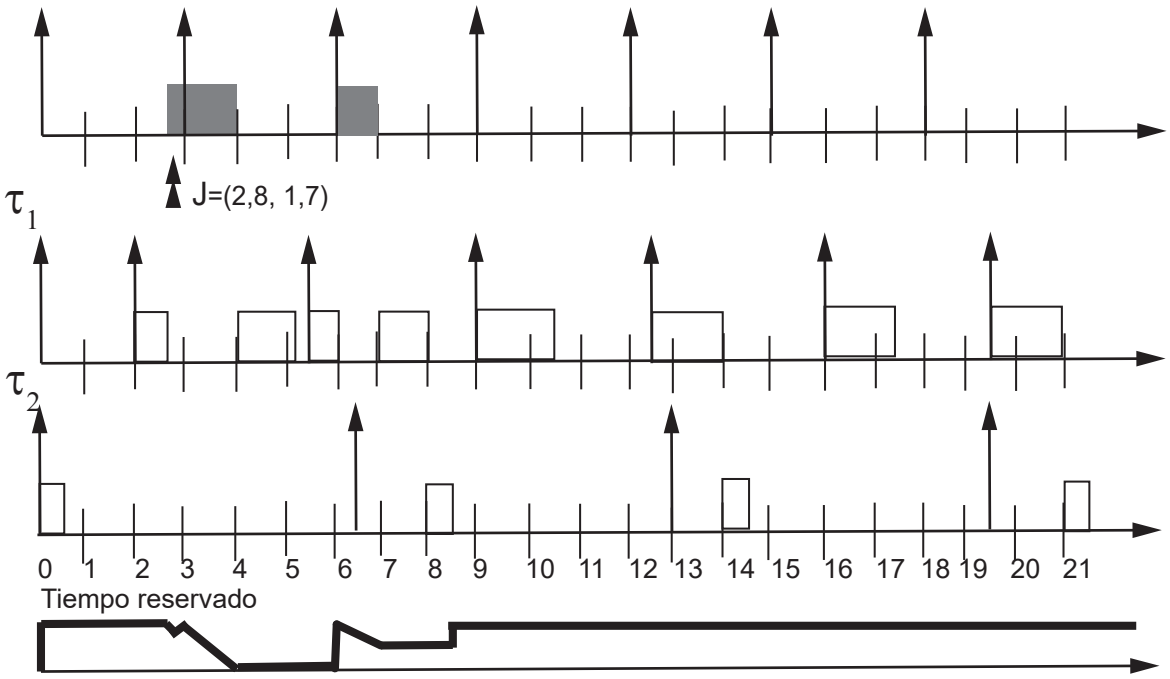
Servidor Diferido

- Preserva el tiempo aperiódico, incluso en ausencia temporal de peticiones de este tipo
- Se asigna un tamaño de servidor C_s que se gasta sólo en atender peticiones periódicas
- El tamaño del servidor se rellena hasta su valor máximo en cada periodo del servidor T_s
- El valor inicial de C_s se determina realizando un análisis previo de planificabilidad del conjunto de tareas
- Se atienden las peticiones aperiódicas a un nivel de prioridad alto siempre que el tiempo de servidor no se haya agotado

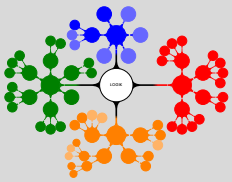
Manuel I. Capel
manuelcapel@ugr.es



Peticiones aperiódicas y Servidor Diferido

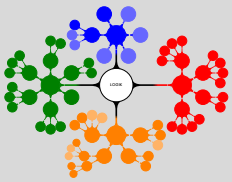


Manuel I. Capel
manuelcapel@ugr.es



Análisis de Planificabilidad con el *Servidor Diferido*

Manuel I. Capel
manuelcapel@ugr.es



- Conjunto de *N* tareas periódicas, $\tau_1 \ldots \tau_N$ y un *servidor diferido* con prioridad más alta.
- Se calcula el *menor límite superior de utilización*:

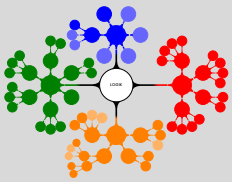
$$U_{mls} = U_s + N\left[\left(\frac{U_s + 2}{2U_s + 1}\right)^{\frac{1}{N}} - 1\right]$$

- Para $N \to \infty$, en el peor caso:

$$\lim_{N \to \infty} U_{mls} = U_s + \ln\left(\frac{U_s + 2}{2U_s + 1}\right)$$

Análisis de Planificabilidad con el *Servidor Diferido*

Manuel I. Capel
manuelcapel@ugr.es



- Dado un conjunto de *N* tareas periódicas y un *servidor diferido* con límites de utilización U_p y U_s, respectivamente, la planificabilidad del conjunto de tareas periódicas está garantizada, con el algoritmo de cadencia monótona, si $U_p + U_s \leq U_{mls}$;

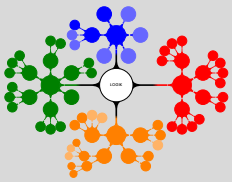
- De forma equivalente, si se cumple la desigualdad siguiente:

$$U_p \leq \ln\left(\frac{U_s + 2}{2U_s + 1}\right)$$

.

Bibliografía

Manuel I. Capel
manuelcapel@ugr.es

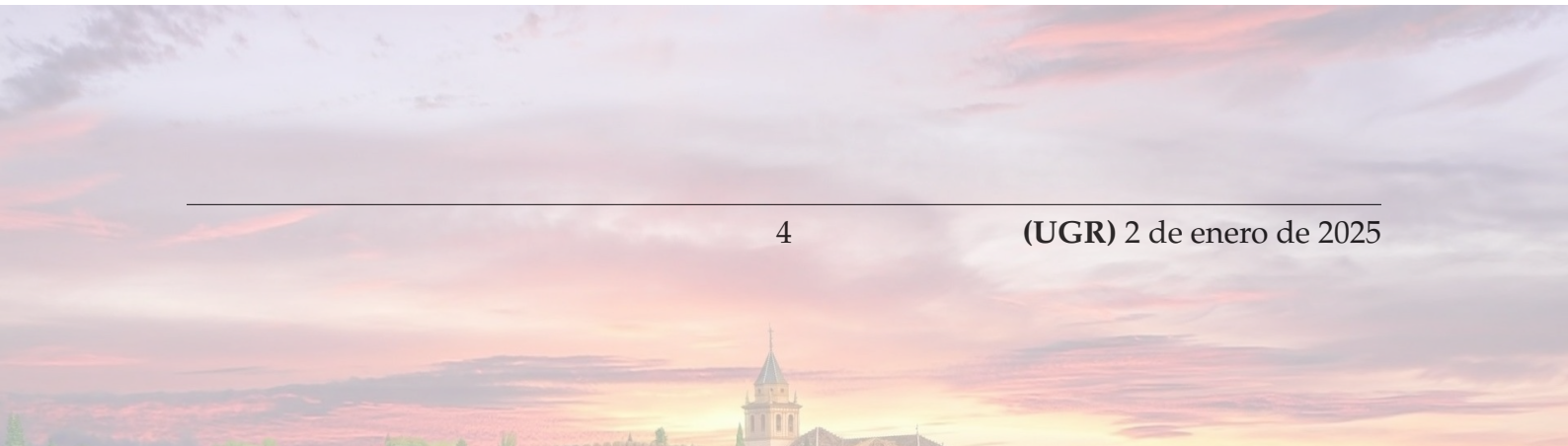


Para más información, ejercicios, bibliografía adicional:

- “Programación Concurrente y de Tiempo Real”: M.I.Capel (2022), Garceta (Madrid). Sistemas de Tiempo Real (capítulo 4)
- “Sistemas de Tiempo Real”: Burns. Addison-Wesley (2003)
- “Hard real-time computing systems: predictable algorithms and applications”: Butazzo. Springer-Verlag (2005)
- “Concurrent and real-time programming in Java”: Wellings. John Wiley (2004)

2 Resúmenes

Índice



1 Tema 3: Sistemas basados en el paso de mensajes

1.1. El cuello de botella de Von Neumann

El **cuello de botella de Von Neumann** se refiere a una limitación inherente en las arquitecturas de computación clásicas basadas en el modelo de programa único almacenado. Este problema surge debido a la dependencia de un único bus para transferir datos e instrucciones entre la memoria y la unidad de procesamiento central (CPU). A continuación, se presentan sus principales características y consecuencias:

- **Limitación de velocidad del bus:** La transferencia de datos entre la memoria y la CPU está restringida por la velocidad máxima del bus, lo que afecta el rendimiento general del sistema.
- **Cuello de botella:** Dado que no es posible realizar simultáneamente una operación de búsqueda de instrucciones y una operación de datos, se genera un retraso en el flujo de ejecución de las instrucciones.
- **Impacto en el rendimiento:** Esta arquitectura requiere más tiempo de ejecución para los programas, lo que reduce la eficiencia del sistema computacional.

El cuello de botella de Von Neumann es una de las razones principales por las cuales las arquitecturas modernas buscan soluciones más eficientes, como los sistemas multiprocesadores o arquitecturas paralelas.

1.2. Clasificación de Flynn

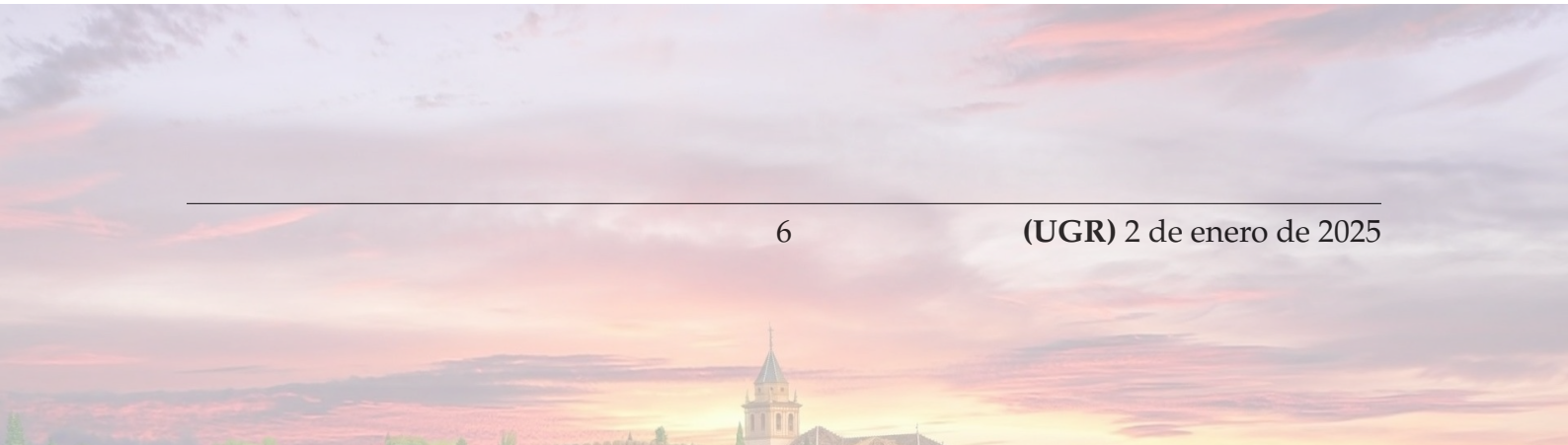
La **clasificación de Flynn** es un esquema que categoriza las arquitecturas de sistemas computacionales en función de cómo manejan las instrucciones y los datos durante la ejecución. Esta clasificación es fundamental para entender los diferentes modelos de procesamiento y se divide en cuatro categorías principales:

- **SISD (Single Instruction, Single Data):**
  - Una sola unidad de control ejecuta una única secuencia de instrucciones.
  - Procesa un único flujo de datos.
  - Ejemplo: computadores monoprocesador tradicionales.
- **SIMD (Single Instruction, Multiple Data):**
  - Una única instrucción se aplica simultáneamente a múltiples flujos de datos.
  - Común en aplicaciones con operaciones repetitivas sobre grandes conjuntos de datos, como gráficos y procesamiento de señales.
  - Ejemplo: procesadores vectoriales y GPU.
- **MISD (Multiple Instruction, Single Data):**

- Múltiples unidades de control ejecutan distintas instrucciones sobre un único flujo de datos.
- Es menos común en sistemas reales debido a limitaciones prácticas.

- **MIMD (Multiple Instruction, Multiple Data):**
  - Múltiples unidades de control ejecutan distintas secuencias de instrucciones sobre múltiples flujos de datos.
  - Flexible y utilizado en sistemas paralelos modernos.
  - Ejemplo: arquitecturas multicore y clusters de computadoras.

Esta clasificación permite identificar las capacidades y limitaciones de las diferentes arquitecturas, sirviendo como guía para diseñar y optimizar sistemas computacionales.



1.3. Diferencias entre Multiprocesador SMP y AMP

Características	SMP (Multiprocesamiento Simétrico)	AMP (Multiprocesamiento Asimétrico)
Arquitectura	Todos los procesadores son iguales y comparten el acceso a la memoria	Un procesador principal controla el sistema y los otros son secundarios
Acceso a memoria	Todos los procesadores tienen acceso compartido a la memoria central	Solo el procesador principal tiene acceso a la memoria central
Control de procesos	Los procesadores pueden trabajar independientemente	El procesador principal gestiona los procesos y asigna tareas a los secundarios
Escalabilidad	Alta escalabilidad; se pueden agregar más procesadores sin mucha complicación	Baja escalabilidad; añadir más procesadores no mejora el rendimiento de forma eficiente
Costo	Más costoso debido a la necesidad de múltiples procesadores iguales y una memoria compartida	Menos costoso, ya que solo se necesita un procesador principal y algunos secundarios
Rendimiento	Mejor rendimiento en tareas paralelas y de procesamiento intensivo	Menor rendimiento en tareas paralelas, debido a la dependencia del procesador principal
Sistemas típicos	Servidores, estaciones de trabajo y sistemas de alta gama	Sistemas embebidos, estaciones de trabajo de menor escala y dispositivos con recursos limitados

Cuadro 1: Diferencias entre Multiprocesador SMP y AMP

1.4. Mecanismo de citas

Se trata de una operación de comunicación NO bloqueante y sin búfer. La cita tiene lugar antes de que comience la transmisión de datos.

1.5. Paso de mensajes bloqueante con búfer

El proceso emisor vuelve inmediatamente al ejecutar la operación de envío, *salvo que el proceso buffer esté lleno*. En este caso, el proceso emisor se bloquea hasta que haya espacio en el buffer. La operación *receive* no vuelve hasta que se han recibido los datos.

1.6. Paso de mensajes no bloqueante con búfer

En este caso se reduce el tiempo de espera debido a que la operación *receive* provoca la transferencia inmediata de datos del búfer a la memoria del proceso receptor.

1.7. MPI

1.7.1. Comunicación bloqueante

Campos de operación MPI de envío con buffer

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int
   dest, int tag, MPI_Comm comm);
```

Campos de operación MPI de recepción con buffer

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
   source, int tag, MPI_Comm comm, MPI_Status *status);
```

- El *origen, tag, comunicador* deben de coincidir en el mensaje enviado y recibido.
- *MPI_Send, MPI_Recv, MPI_Ssend* no vuelven hasta que se completan.

1.7.2. Comunicación No bloqueante

- *MPI_Isend, MPI_Irecv, MPI_Iprobe, MPI_Test* son operaciones no bloqueantes.
  - *MPI_Isend:* Inicia una operación de envío no bloqueante. Permite que el programa continúe ejecutándose mientras se realiza la operación de envío en segundo plano.
  - *MPI_Irecv:* Inicia una operación de recepción no bloqueante. Permite que el programa continúe ejecutándose mientras se realiza la operación de recepción en segundo plano.
  - *MPI_Iprobe:* Permite comprobar de manera no bloqueante si hay un mensaje disponible para ser recibido. Esto es útil para evitar que el programa se bloquee esperando un mensaje.
  - *MPI_Test:* Verifica si una operación no bloqueante ha completado. Esto permite al programa comprobar el estado de las operaciones no bloqueantes y actuar en consecuencia.

1.7.3. Sondeo de Mensajes

- *MPI_Iprobe* y *MPI_Probe* son operaciones de sondeo de mensajes que permiten a un proceso verificar si hay mensajes disponibles para ser recibidos.

- *MPI_Iprobe:* Permite sondear de manera no bloqueante si hay mensajes disponibles para ser recibidos. Devuelve un valor verdadero si hay mensajes pendientes y falso en caso contrario. El valor que modifica es el flag, es decir, si flag>0, hay que recibirlo con MPI_Recv, debido a que se da la existencia de un mensaje no bloqueante.
- *MPI_Probe:* Permite sondear de manera bloqueante si hay mensajes disponibles para ser recibidos. Se bloquea hasta que llega un mensaje.

Diferencias entre MPI_Probe y MPI_IProbe

Característica	**MPI_Probe**	**MPI_IProbe**
Bloqueante	Sí	No
Comportamiento	Espera hasta que haya un mensaje disponible	Vuelve inmediatamente con el estado del mensaje
Eficiencia	Puede causar ineficiencia en ausencia de mensajes mientras se espera	Permite trabajar en paralelo mientras se espera

Cuadro 2: Diferencias entre MPI_Probe y MPI_IProbe

1.8. Explicación del código de la criba de Eratóstenes

Para ello pincha aquí. Además la actividad extra se encuentra aquí.

1.8.1. Explicación del problema del museo

```
    PUERTA(i:1..2)::
{ int s=0;
do
if (s<HORA.CIERRE && PERSONA())->
{send(CONTROL, S()); //envia una se\~nal de entrada
de persona
DELAY.UNTIL(s+1);//espera hasta el siguiente
instante
s:=s+1;// cuenta un nuevo tick de reloj
}
[]
(s<HORA.CIERRE && NOT PERSONA())->
DELAY.UNTIL(s+1);//espera hasta el siguiente
instante
s:=s+1;]//cuenta otro tick
[]
TRUE->DELAY.UNTIL (TIME() + 16*3600);
//es la hora de cierre del museo; hay
//que esperar 16 horas para activar el controlador
.
// TIME() devuelve una cuenta en segundos.
fi
```

```
do;
send(CONTROL, Start();}
CONTROL::
{ int cont= 0;
if receive(PUERTA(1), Start());
//desde cualquiera de los sensores
[] //de las puertas se arranca
receive(PUERTA(2), Start());//el controlador
fi
do
[]*[(j:1..2) receive(PUERTA(j), S())-> cont:= cont
+1];
//cuenta una persona mas, porque ha recibido la se\~nal
//de cualquiera de las 2 puertas (no se puede saber cual)
od;
printf("numero de personas", %d, cont));
}
main(){
cobegin PUERTA;CONTROL coend;}
```

1.9. Orden Select

- versión de orden alternativa no determinista
- modo síncrono de comunicación, paso de mensajes bloqueantes
- resuelve el problema de la recepción de tener varias variables pendientes sin dependencias del orden temporal

1.9.1. Semántica de la orden select

- cada bloque que comienza con *when* se denomina alternativa (orden guarda)
- desde when hasta do se denomina guarda de dicha alternativa
- instrucciones receive nombran a otros procesos del programa concurrente y cada uno referencia a una variable local

```
    select
when condicion1 receive( variable1, proceso1 ) do
sentencias1
when condicion2 receive( variable2, proceso2 ) do
sentencias2
...
when condicionn receive( variablen, proceson ) do
sentenciasn
end
```

1.9.2. Formas de poner la guarda de una orden select

- when receive(mensaje, proceso) do sentencias
- when true receive(mensaje, proceso) do sentencias
- when condicion do sentencias (puede omitirse el receive, decimos que es una guarda sin sentencia de entrada)

1.9.3. Tipos de ejecución de las guardas

- guarda ejecutable: si la condición es verdadera y ya ha iniciado la sentencia send
- guarda potencialmente ejecutable: si la condición es verdadera pero no ha iniciado la sentencia send
- guarda no ejecutable

1.9.4. Determinación de la guarda a ejecutar

- aquella que inicio el send antes
- se selecciona no determinísticamente una cualquiera si no hay guardas con sentencias de entrada
- si hay solo guardas potenciablemente ejecutables, se inicia la guarda cuando esta inicia la sentencia send
- si no hay ningún tipo de guarda se genera una excepción/error.

Hay que tener en cuenta que la ejecución de un instrucción select conlleva esperas, por lo que pueden producirse situaciones de interbloqueo.

Existe una orden select con prioridad, por lo que dejaría de ser no determinística.

Para programar a un servicio servidor se debe de ejecutar el select dentro de un bucle para que se evalúe de nuevo la guarda y se seleccione no determinísticamente a una de ellas para poder ejecutarlas.

1.9.5. Ejemplo: Productor-Consumidor con Búfer FIFO

Este ejemplo ilustra el patrón clásico de **productor-consumidor** utilizando un búfer FIFO intermedio. En este esquema:

- El **productor** genera elementos y los envía al búfer.
- El **intermedio** actúa como un búfer que controla la inserción y extracción de datos, garantizando condiciones de sincronización mediante guardas.
- El **consumidor** extrae elementos del búfer y los procesa.

La clave de este diseño radica en que el intermedio no conoce de antemano el orden de las peticiones de inserción y extracción, pero asegura la propiedad de **seguridad** en el acceso concurrente al búfer.

```
1    { Productor (P) }
2    while true do                          // Inicia un bucle infinito en el
        productor.
3    begin
4        v := Produce();                    // El productor genera un valor 'v
            '.
5        s_send(v, B);                      // El productor envía el valor 'v'
             al búfer 'B'.
6    end
7
8    { Intermedio (B) }
9    var esc, lec, cont: integer := 0;     // Inicializa las variables: 'esc'
         (índice de escritura), 'lec' (índice de lectura) y 'cont' (contador
         de elementos en el búfer).
10   buf: array[0..tam-1] of integer;    // Declara el búfer 'buf' como un
        arreglo de tamaño 'tam', donde se almacenarán los valores producidos
        .
11   begin
12       while true do                      // Inicia un bucle infinito en el
            intermedio (el búfer).
13           select                           // Comienza una selección de
                condiciones para ser ejecutadas.
14               when cont < tam receive(v, P) do       // Si el número de
                    elementos en el búfer es menor que 'tam', hay espacio
                    para almacenar más elementos, El intermedio recibe un
                    valor 'v' del productor (P).
15
16                   buf[esc] := v;         // El valor recibido se almacena
                        en la posición 'esc' del búfer.
17                   esc := (esc + 1) mod tam;  // Se actualiza el índice de
                         escritura 'esc' de manera circular (se vuelve a 0
                        cuando alcanza 'tam').
18                   cont := cont + 1;      // Se incrementa el contador de
                        elementos en el búfer.
19               when 0 < cont receive(s, C) do             // Si el número
                    de elementos en el búfer es mayor que 0, hay datos para
                    consumir, El intermedio recibe una solicitud del
                    consumidor (C).
20
21                   s_send(buf[lec], C);  // El intermedio envía el valor
                        almacenado en 'lec' al consumidor.
22                   lec := (lec + 1) mod tam;  // Se actualiza el índice de
                         lectura 'lec' de manera circular.
23                   cont := cont - 1;      // Se decrementa el contador de
                        elementos en el búfer.
24           end
25   end
26
27   { Consumidor (C) }
28   while true do                          // Inicia un bucle infinito en el
        consumidor.
29   begin
```

```
30          s_send(s, B);                    // El consumidor solicita un valor
                del búfer (envía una solicitud 's' al búfer).
31          receive(v, B);                   // El consumidor recibe el valor '
               v' del búfer.
32          Consume(v);                      // El consumidor procesa el valor
               recibido.
33      end
```

Listing 1: Productor-Consumidor con Búfer FIFO

Breve Explicación

- **Productor (P):** Un bucle infinito genera valores (*Produce*) y los envía al búfer (*s_send(v, B)*).
- **Intermedio (B):** Administra las operaciones de inserción y extracción usando un arreglo cíclico:
  - Inserta valores en el búfer si no está lleno (*cont <tam*).
  - Extrae valores del búfer si no está vacío (*cont >0*).
  - Las guardas en las sentencias *select* controlan estas condiciones.
- **Consumidor (C):** Recibe valores del búfer y los consume (*Consume(v)*).

Este modelo asegura la sincronización entre los procesos mediante las condiciones de las guardas, evitando inconsistencias en el acceso concurrente al búfer.

1.9.6. Select con Guardas Indexadas

Instrucción `Select` con guardas indexadas Cuando es necesario replicar alternativas en una estructura de espera selectiva, se puede utilizar una construcción que evita la redundancia de código. La sintaxis para lograrlo es:

```
for indice := inicial to final
when condicion receive(mensaje, proceso) do
    sentencias(indice);
```

En esta construcción, tanto la condición, el mensaje, el proceso y las sentencias pueden referirse al índice de la iteración. Esta forma compacta es equivalente a expandir explícitamente las alternativas, como se muestra a continuación:

```
when condicion receive(mensaje, proceso) do
    sentencias { se sustituye indice por inicial }
when condicion receive(mensaje, proceso) do
    sentencias { se sustituye indice por inicial + 1 }
...
```

Este mecanismo facilita la creación de programas distribuidos más legibles y eficientes, reduciendo la redundancia en el código fuente.

1.9.7. Select con sentencia else

En este caso es similar al uso del *when*, pero en este caso debemos de hacer uso del *else*, podemos concebir que el uso y desempeño es similar al de in ifelse, pero en este caso es un whenelse. Ejemplo:

```
1    // Declaración de variables globales
2    var suma : array[0..n-1] of integer := (0,0,...,0) ; // Arreglo para
        almacenar las sumas parciales de cada proceso.
3    continuar : boolean := true ;                        // Variable de
        control para el bucle principal.
4    numero : integer ;                                   // Variable para
        recibir el número enviado por un proceso.
5
6    begin
7    while continuar do begin                             // Bucle principal
        que se ejecuta mientras 'continuar' sea verdadero.
8        select                                            // Inicia la
            estructura de selección para manejar múltiples condiciones.
9            for i := 0 to n - 1                          // Itera sobre
                todos los procesos del 0 al n-1.
10           when suma[i] < 100 receive( numero, emisor[i] ) // Si la suma
                parcial del proceso i es menor que 100,
11                                                           // recibe un n
                                                                úmero de
                                                                dicho
                                                                proceso (
                                                                emisor[i]).
12           do
13               suma[i] := suma[i] + numero ;            // Actualiza la
                    suma parcial del proceso i con el número recibido.
14               continuar := true ;                      // Establece '
                    continuar' como verdadero para seguir ejecutando el
                    bucle.
15                                                        // Nota: esta
                                                             instrucción no
                                                              es
                                                             estrictamente
                                                             necesaria,
                                                             pero aclara
16                                                        // que el bucle
                                                             continuará.
17       end
18       else continuar := false;                         // Si no se cumple
            ninguna condición, establece 'continuar' como falso
19                                                        // para salir del
                                                             bucle principal.
20   end
21   end
```

Listing 2: Select con sentencia else

2 Tema 4: Introducción a los Sistemas de Tiempo Real

2.1. Confusión con otros sistemas

Los sistemas de tiempo real (STR) a menudo se confunden con otros tipos de sistemas debido a ciertas similitudes en sus características y comportamientos. A continuación, se explican las razones de estas confusiones:

- **En línea:** Los sistemas en línea están diseñados para estar disponibles y operativos en todo momento, similar a los STR que deben responder a eventos en tiempo real. Sin embargo, la diferencia clave es que los STR tienen restricciones temporales estrictas que deben cumplirse para garantizar el correcto funcionamiento del sistema.
- **Interactivos:** Los sistemas interactivos permiten la interacción directa con los usuarios, lo que puede dar la impresión de que son sistemas de tiempo real. No obstante, los STR no solo requieren interacción, sino que también deben cumplir con plazos específicos para procesar y responder a eventos.
- **Rápidos:** La velocidad de respuesta es una característica común tanto en sistemas rápidos como en STR. Sin embargo, la principal diferencia radica en que los STR no solo deben ser rápidos, sino que deben garantizar que las respuestas ocurran dentro de un tiempo predefinido y crítico para el correcto funcionamiento del sistema.

2.2. Propiedades de los STR

Los Sistemas de Tiempo Real (STR) tienen ciertas propiedades fundamentales que los distinguen de otros sistemas informáticos. Estas propiedades son esenciales para asegurar que los STR cumplan con los requisitos temporales y de fiabilidad en su funcionamiento. Las principales propiedades de los STR son las siguientes:

- **Reactividad:** Los sistemas de tiempo real deben reaccionar de manera adecuada y rápida a los eventos que ocurren en el entorno. La reactividad implica que el sistema debe responder a los eventos en un tiempo predecible, es decir, la respuesta no solo depende de lo que el sistema haga, sino también de cuándo lo haga. Esta propiedad es crucial en aplicaciones como el control de sistemas de vuelo o los sistemas de frenado en vehículos.
- **Determinismo:** Un STR debe ser determinista, lo que significa que dada una entrada específica, el sistema debe producir siempre la misma salida dentro de un tiempo determinado. El comportamiento de un STR es predecible y se puede calcular con antelación. Esto es fundamental en sistemas como los de control de maquinaria, donde la predictibilidad es crucial para garantizar el buen funcionamiento del sistema sin errores inesperados.
- **Responsabilidad:** Esta propiedad hace referencia a la capacidad de un STR para garantizar que las tareas o procesos se completan dentro de sus plazos establecidos. En otras palabras, un STR debe ser capaz de asegurar que las tareas se

ejecuten correctamente y a tiempo, incluso bajo condiciones de carga alta o en situaciones imprevistas. La responsabilidad es vital para garantizar la estabilidad y efectividad del sistema.

- **Confiabilidad:** Los STR deben ser confiables, lo que significa que deben funcionar correctamente y de forma estable durante su operación, incluso ante fallos o errores. La confiabilidad es una propiedad clave para sistemas críticos como los utilizados en aplicaciones médicas, aeronáuticas o automotrices, donde un fallo podría tener consecuencias graves. La fiabilidad implica que los sistemas estén diseñados para manejar fallos de manera segura o minimizar su impacto.

2.3. Definición en el ámbito de los sistemas operativos

Sistema Operativo de Tiempo Real es aquel que tiene la capacidad para suministrar un nivel de servicio requerido en un tiempo limitado y especificado a priori.

2.4. Clasificación de los STR

Los Sistemas de Tiempo Real (STR) se pueden clasificar según el nivel de criticidad temporal que tienen sus tareas, lo cual influye en los requisitos y características de cada sistema. A continuación, se presentan tres categorías basadas en la criticidad de los STR:

2.4.1. Atendiendo a la criticidad de los STR

- **Misión Crítica:** Estos sistemas son los más críticos, ya que cualquier fallo en la ejecución dentro del tiempo establecido puede tener consecuencias graves. Un ejemplo de este tipo de sistema es el *control de aterrizaje* de aeronaves, donde la puntualidad y exactitud son cruciales para la seguridad de la operación. Los complementos importantes en estos sistemas incluyen *tolerancia a fallos*, lo que significa que el sistema debe ser capaz de manejar y recuperarse de errores sin comprometer la seguridad ni la efectividad.

- **Estrictos:** Los sistemas estrictos también requieren una alta precisión temporal, aunque las consecuencias de un fallo no son tan críticas como en los sistemas de misión crítica. Un ejemplo sería el sistema de *reservas de vuelos*, donde las respuestas deben ser rápidas y dentro de los tiempos previstos, pero un pequeño retraso no implica necesariamente un fallo catastrófico. Los complementos asociados a estos sistemas incluyen *calidad de la respuesta*, ya que es importante que la respuesta no solo sea puntual, sino que también sea precisa y eficiente.

- **Permisivos:** Estos sistemas tienen requisitos menos estrictos en cuanto a tiempo, ya que las tareas pueden tolerar ciertos retrasos sin consecuencias graves. Un ejemplo típico sería la *adquisición de datos meteorológicos*, donde los datos pueden ser procesados con algo de retraso sin que afecte gravemente al resultado final. Los complementos de estos sistemas incluyen *medidas de fiabilidad*, lo que significa que, aunque el sistema no necesite ser extremadamente puntual, debe ser confiable y eficiente en su funcionamiento.

2.5. Tipos de medidas del tiempo de interés en STR

Existen dos tipos principales de medidas del tiempo de interés en los Sistemas de Tiempo Real (STR): el **tiempo absoluto** y los **intervalos o tiempo relativo**. A continuación se detallan ambos tipos:

2.5.1. 1. Tiempo Absoluto

El tiempo absoluto se refiere a la medición del tiempo con relación a un sistema de referencia global. Algunos de los sistemas de referencia más comunes incluyen:

- **Sistemas de referencia locales:** Este tipo de medida usa un sistema de tiempo basado en la localización geográfica de un lugar específico. Puede estar basado en relojes locales o sistemas de referencia regionales.
- **Astronómicos (UT0):** Se refiere al tiempo universal basado en la rotación de la Tierra en relación con las estrellas. El *UT0* es un tiempo que considera los movimientos irregulares de la Tierra y es usado como una medida precisa en astronomía.
- **Atómicos (IAT):** El Tiempo Atómico Internacional (IAT) es una medida basada en la oscilación de los átomos de cesio y es utilizado para obtener un estándar muy preciso de medición del tiempo. Este tiempo es utilizado como base para el UTC.
- **Tiempo Coordinado Universal (UTC):** Es el estándar internacional de tiempo utilizado para coordinar el tiempo globalmente. Desde 1972, el UTC ha sido utilizado ampliamente y es una medida combinada basada en el IAT, pero con ajustes ocasionales para sincronizarlo con la rotación de la Tierra.
- **Satelital (GPS):** El tiempo de GPS (GPST) es una escala de tiempo continua que no tiene segundos intercalares. Se define a partir del segmento de control del sistema GPS, el cual se basa en relojes atómicos ubicados en estaciones de monitoreo y en los satélites. Este sistema comenzó a las 00:00 UTC del 5 al 6 de enero de 1980.

2.5.2. 2. Intervalos o Tiempo Relativo

El tiempo relativo se refiere a la medición de intervalos de tiempo entre eventos, y no a un instante específico en un sistema de referencia global. Los intervalos de tiempo pueden no ser monótonos, lo que significa que pueden verse afectados por factores como la deriva o los ajustes de reloj. En STR, esto es importante porque el tiempo computacional puede no ser uniforme o estable debido a la interferencia de diferentes tareas en ejecución.

Por ejemplo, el *Tiempo Computador* puede variar dependiendo de la carga del sistema, lo que hace que el tiempo sea no monótono, y este comportamiento debe ser gestionado adecuadamente para garantizar el cumplimiento de los plazos de las tareas en los STR.

La medida del tiempo en los Sistemas de Tiempo Real (STR) se realiza a través de diferentes tipos de relojes y características que permiten un seguimiento preciso de los eventos y tareas dentro de los sistemas. A continuación se presentan los conceptos clave y características de los relojes de tiempo real:

2.5.3. Concepto de reloj de tiempo real

El reloj de tiempo real puede estar basado en diferentes componentes que permiten su funcionamiento:

- **Oscilador:** Es un dispositivo que genera una señal de frecuencia constante, utilizada para medir el tiempo. Los osciladores suelen ser muy precisos, pero pueden tener limitaciones en cuanto a estabilidad a largo plazo.
- **Contador:** Un contador de tiempo real lleva el registro del tiempo mediante una secuencia de pulsos. Cada pulso representa una unidad de tiempo y el contador se incrementa con cada uno. Este tipo de reloj puede ser más eficiente en cuanto a precisión temporal.
- **Software:** Los relojes de tiempo real basados en software utilizan el sistema operativo y el hardware subyacente para realizar mediciones temporales. Estos pueden estar sujetos a variaciones o retardos dependiendo de las cargas de trabajo del sistema.

2.5.4. Características más importantes de los relojes de tiempo real

Los relojes de tiempo real tienen características fundamentales que permiten su funcionamiento adecuado en los STR:

- **Precisión (granularidad):** La precisión de un reloj de tiempo real se refiere a la mínima unidad de tiempo que puede medir. Esta se mide generalmente en nanosegundos (ns), microsegundos (μs), milisegundos (ms), o segundos (s). Cuanto mayor sea la precisión, mayor será la capacidad del sistema para gestionar tareas con plazos muy estrictos.
- **Tiempo de desbordamiento:** El tiempo de desbordamiento se refiere al límite de tiempo después del cual un contador o reloj vuelve a cero. Este comportamiento es común en relojes de 32 bits, ya que al llegar al límite de valor, el contador se reinicia.
- **Intervalo:** El intervalo se refiere al tiempo transcurrido entre dos eventos consecutivos o dos mediciones del reloj. Este intervalo puede ser afectado por la precisión del reloj y la estabilidad del sistema.

2.5.5. Escalas temporales

Existen varias escalas temporales utilizadas en los STR, dependiendo del tipo de medición y de las necesidades de la aplicación. Algunas de las más comunes son:

- **Tiempo monótono:** Es un tipo de tiempo en el que el reloj avanza de manera continua, sin retroceder o saltar en el tiempo. Es muy útil para aplicaciones que requieren un seguimiento constante del tiempo.
- **Tiempo no monótono:** En este caso, el reloj puede avanzar o retroceder en función de varios factores, como ajustes de los relojes o cambios de sistema. Es común en sistemas que necesitan sincronización con otros sistemas o eventos externos, como los relojes GPS.
- **Tiempo absoluto:** Es un tiempo basado en un sistema de referencia global (por ejemplo, UTC), que no depende de los eventos o el sistema local.
- **Tiempo no absoluto:** El tiempo no absoluto depende del sistema local y puede estar sujeto a variaciones o desincronización respecto a un sistema de referencia global.

2.5.6. Precisión y Intervalo para un contador de 32 bits

A continuación se muestra la relación entre la precisión y el intervalo para un contador de 32 bits. Este tipo de contador tiene una capacidad limitada debido al tamaño del número que puede almacenar (32 bits), lo que implica que el contador se desbordará después de un determinado intervalo.

- **Precisión:** La precisión de un contador de 32 bits puede variar dependiendo de la unidad de medida. A continuación se presentan algunos ejemplos de intervalos de tiempo que un contador de 32 bits puede manejar:
  - 100 ns (nanosegundos) hasta 429,5 segundos.
  - 1 μs (microsegundos) hasta 71,58 minutos.
  - 100 μs (microsegundos) hasta 119,3 horas.
  - 1 ms (milisegundos) hasta 49,71 días.
  - 1 s (segundo) hasta 136,18 años.

  Estos valores indican el intervalo máximo que un contador de 32 bits puede medir antes de que se produzca un desbordamiento. Cuanto mayor sea la precisión, menor será el intervalo que el contador podrá gestionar antes de desbordarse.

2.6. Temporizadores

Los temporizadores son elementos clave en los sistemas operativos y en los sistemas concurrentes y distribuidos, particularmente en sistemas de tiempo real. A continuación, se describen sus características y cómo se utilizan para gestionar el tiempo y las tareas dentro del sistema.

2.6.1. Ideas Fundamentales

Los temporizadores no son instrucciones nativas de los lenguajes de programación, sino que se activan mediante una llamada o operación de sistema operativo. Funcionan como relojes especializados en la gestión de tareas con temporización. Para programar un temporizador, los elementos básicos que se necesitan son:

- **Tiempo de arranque:** El instante en que se inicia el temporizador.
- **Tiempo de parada:** El instante en que se detiene el temporizador.

2.6.2. Tipos de Temporizadores

Existen varios tipos de temporizadores, entre los cuales destacan:

- **De 1 solo disparo:** Se activan una única vez, y luego se desactivan.
- **Periódicos:** Se activan repetidamente a intervalos regulares.
- **Con deriva:** Pueden presentar una pequeña desviación en el tiempo de activación, lo que se denomina deriva acumulativa.

2.6.3. Retardos en las Tareas

Las tareas en sistemas de tiempo real a menudo requieren programar retardos. Para ello, se pueden usar temporizadores o instrucciones específicas de los lenguajes de programación. Un ejemplo de tarea con retardo programado es la siguiente:

```
void tareaT(int t_computo_p) {
    int t_computo = t_computo_p; // Ejemplo: 100 milisegundos
    do {
        // Acción a realizar
        sleep_for(t_computo); // Afectada por la deriva local
    } while (true);
}
```

Aquí, la función `sleep_for` introduce un retardo de `t_computo`, el cual está afectado por la deriva local del sistema.

2.6.4. Deriva Acumulativa y Activación Periódica de las Tareas

Un problema común al trabajar con temporizadores en sistemas de tiempo real es la **deriva acumulativa**. La deriva acumulativa ocurre cuando las tareas no se activan exactamente en los intervalos deseados, lo que puede afectar el comportamiento del sistema, especialmente cuando se trata de tareas periódicas.

Eliminación de la Deriva Acumulativa:

Para mitigar este problema, se puede programar una tarea periódica que se active de manera precisa en cada ciclo, ajustando la hora de activación a medida que se acerca al siguiente ciclo. A continuación, se muestra un ejemplo de cómo programar una tarea periódica en C++:

```cpp
void tareaPeriodica() {
    int t_ciclo = 100; // milisegundos
    auto siguiente_instante = std::chrono::steady_clock::now(); //
        milisegundos
    do {
        // Acción a realizar
        siguiente_instante += std::chrono::milliseconds(t_ciclo);
        std::this_thread::sleep_until(siguiente_instante); // Sin deriva
            acumulativa
    } while (true);
}
```

En este ejemplo, la función `std::this_thread::sleep_until` asegura que la tarea se active exactamente en el siguiente instante deseado, ajustando la activación para evitar la deriva acumulativa.

2.6.5. Conclusión

Los temporizadores son herramientas fundamentales en los sistemas de tiempo real, ya que permiten gestionar tareas con precisión en cuanto al tiempo. Sin embargo, es importante tener en cuenta la deriva acumulativa y utilizar mecanismos adecuados para eliminarla, como se mostró en el ejemplo de la tarea periódica.

2.7. Tareas y Recursos

En los sistemas de tiempo real (STR), las tareas y los recursos son elementos clave para el diseño y la ejecución de procesos que deben cumplir restricciones temporales específicas. A continuación, se detallan los tipos de elementos que componen un STR.

2.7.1. Tipos de Elementos de un STR

- **Tarea (≡ "Proceso"):** Es el componente fundamental en los sistemas de tiempo real. Una tarea está sujeta a restricciones de tiempo definidas por sus atributos temporales, como el tiempo de ejecución y los plazos de respuesta.
- **Recurso:** Son los elementos necesarios para la ejecución de las tareas. Los recursos pueden ser:
  - **Activos:** Procesador, red de comunicaciones, etc.
  - **Pasivos:** Datos, memoria, discos, etc.

2.7.2. Atributos Temporales Principales de una Tarea

Las tareas en los sistemas de tiempo real tienen varios atributos temporales que determinan su comportamiento dentro del sistema. Algunos de los principales son:

- **Tiempo de cómputo máximo o de ejecución (Ci):** El tiempo máximo que una tarea puede tardar en ejecutarse.

- **Tiempo de respuesta (Ri):** El tiempo total que transcurre desde que se solicita la tarea hasta que se completa.
- **Plazo de respuesta máximo ("deadline") (Di):** El límite máximo de tiempo en el que la tarea debe completarse.
- **Período (Ti):** Es el tiempo exacto entre dos activaciones sucesivas de una tarea periódica.

2.7.3. Tipos de Tareas

Las tareas pueden clasificarse según su patrón de activación y periodicidad. Los principales tipos son:

- **Aperiódica:** Estas tareas se activan en instantes arbitrarios. La activación no sigue un patrón fijo.
- **Periódica:** Estas tareas se activan en intervalos regulares, donde T_i representa el período (el tiempo exacto entre dos activaciones sucesivas).
- **Esporádica:** Son tareas repetitivas que tienen un intervalo mínimo T_i entre sus activaciones. No son estrictamente periódicas, pero se repiten con una frecuencia mínima.

2.8. Planificación de Tareas

La planificación de tareas consiste en asignar las tareas a los recursos activos de un sistema, principalmente al procesador o los procesadores, con el objetivo de garantizar la ejecución de todas las tareas de acuerdo con un conjunto de restricciones específicas. En los sistemas de tiempo real, estas restricciones suelen estar asociadas a restricciones temporales, como los plazos límites.

2.8.1. Planificación de Tareas de Tiempo Real

En los sistemas de tiempo real, se debe garantizar que cada tarea se ejecute completamente antes de que expire su plazo de respuesta máximo D_i. El problema central en la planificación de tareas de tiempo real es determinar si, dado un conjunto arbitrario de tareas, estas pueden ejecutarse completamente antes de que termine el plazo de respuesta máximo en cada una de sus activaciones.

El cálculo anterior para un conjunto de tareas que pueden interrumpirse mutuamente varias veces durante su ejecución no tiene una solución matemática sencilla. Sin embargo, este problema puede abordarse de manera estática o dinámica. Una de las soluciones dinámicas es utilizar un modelo simple de tareas de tiempo real.

2.8.2. Planificación Cíclica

La planificación cíclica se basa en una estructura de activación fija de las tareas periódicas. El plan principal es un ciclo que tiene una duración constante denominada hiperperiodo (T_M):

$$T_M = \text{mcm}(T_1, T_2, \ldots, T_n)$$

donde $T_1, T_2, \ldots, T_n$ son los períodos de las tareas. La unidad que cuenta el tiempo es siempre entera (por ejemplo, los ticks de reloj del sistema).

El ciclo principal se descompone en varios ciclos secundarios de igual duración (T_S), y este ciclo se repite de manera continua.

Tarea	T_i	C_i
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

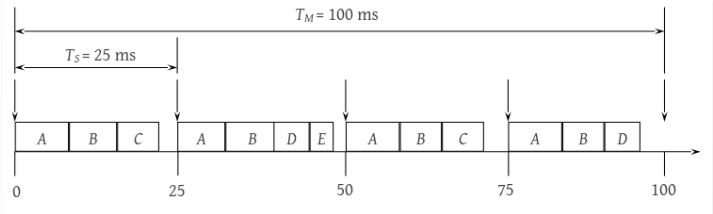


Figura 1: Planificación cíclica

2.8.3. Ideas Fundamentales de la Planificación Cíclica

En la planificación cíclica, el plan principal de ejecución de las tareas es explícito y se define fuera de línea ("off-line"), es decir, sin acceso concurrente al procesador según la prioridad de cada tarea. Las tareas están definidas en un ciclo con una estructura fija.

Algunas características clave de la planificación cíclica son:

- El plan principal establece de antemano cómo se entrelazan las tareas en cualquier ejecución del programa, aunque la ordenación relativa de las tareas puede variar en cada ciclo secundario.
- Cada tarea se ejecuta completamente una sola vez dentro de cada repetición de su período.
- Si una tarea se inicia dentro de una iteración del ciclo secundario, debe terminar antes del final de esa iteración.

2.8.4. Implementación de la Planificación Cíclica

A continuación se muestra un ejemplo de implementación en C++ para la planificación cíclica:

```
1 void EjecutivoCiclico()
2 {
3     const milliseconds tmp_secundario(25);
4     const int nciclos = 4; // Número de ciclos secundarios
5     auto siguiente_instante = clock::now();
```

```
    int frame = 0; // Número del siguiente ciclo secundario

    while (true) {
        for (frame = 1; frame <= nciclos; frame++) { // Ejecuta las tareas
            del ciclo secundario actual
            switch (frame) {
                case 0: A(); B(); C(); break;
                case 1: A(); B(); D(); E(); break;
                case 2: A(); B(); C(); break;
                case 3: A(); B(); D(); break;
            }
            siguiente_instante += tmp_secundario; // Pasar al siguiente
                ciclo secundario
            sleep_until(siguiente_instante); // Esperar inicio siguiente
                periodo de 25 miliseg.
        }
    }
}
```

Cada iteración del bucle `for` ejecuta un ciclo secundario, y cada 4 iteraciones del bucle se ejecuta un ciclo principal.

2.8.5. Diseño del Ejecutivo Cíclico

Para obtener un buen valor de T_S (duración del ciclo secundario), es necesario considerar algunas restricciones:

- El período del ciclo secundario debe ser un divisor del período del ciclo principal, es decir, existe un entero k tal que:

$$T_M = k \cdot T_S$$

- El valor de T_S debe ser al menos el tiempo de cómputo máximo (C_w) de cualquier tarea, es decir, $T_S \geq \text{máx}(C_1, C_2, \ldots, C_n)$.
- T_S debe ser menor o igual que el mínimo plazo de respuesta máximo (D) de todas las tareas, es decir, $T_S \leq \text{mín}(D_1, D_2, \ldots, D_n)$.

Así, el ciclo secundario debe cumplir la siguiente restricción:

$$\text{máx}(C_1, C_2, \ldots, C_n) \leq T_S \leq \text{mín}(D_1, D_2, \ldots, D_n)$$

2.8.6. Propiedades del Ejecutivo Cíclico

Entre las principales propiedades del ejecutivo cíclico destacan:

- No existe concurrencia en la ejecución: Cada ciclo secundario es una secuencia de llamadas a procedimientos, sin necesidad de lanzar la ejecución de las tareas.
- No es necesario un sistema operativo multitarea ni un núcleo de ejecución en tiempo real.

- Los procedimientos acceden secuencialmente a los datos compartidos entre tareas, lo que elimina la necesidad de mecanismos de exclusión mutua (como semáforos o monitores).
- No es necesario realizar un análisis de planificabilidad del conjunto de tareas, ya que el sistema es correcto por construcción.

2.8.7. Problemas del Ejecutivo Cíclico

Sin embargo, la planificación cíclica presenta ciertos problemas, entre los cuales se incluyen:

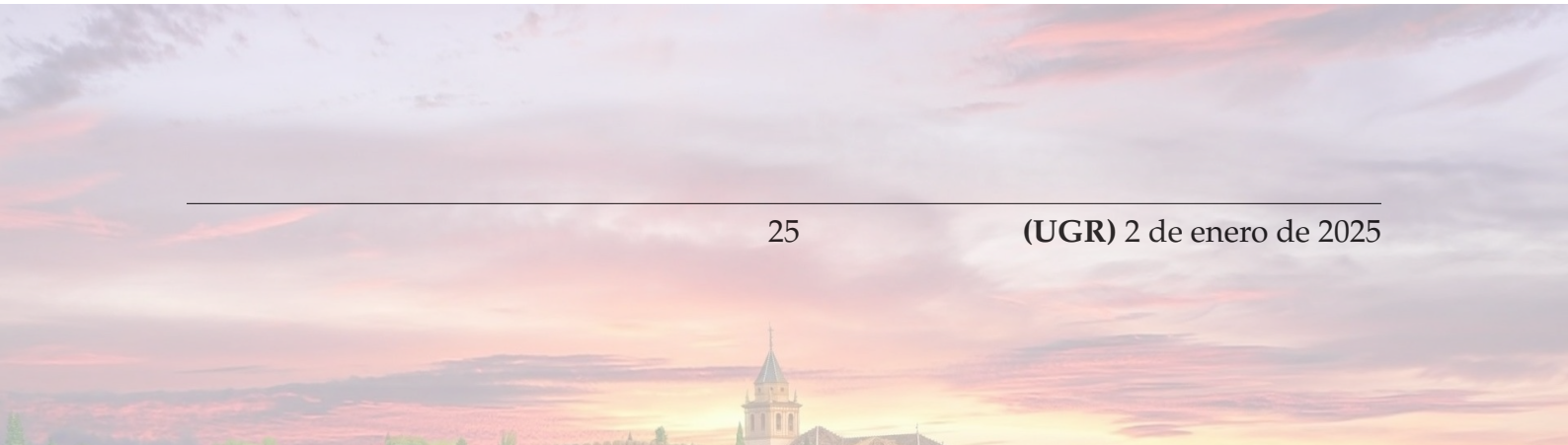
- Dificultad para incorporar tareas con períodos largos.
- Las tareas esporádicas son difíciles de tratar, aunque se podría utilizar un servidor de consulta para gestionar su activación.
- El plan cíclico es difícil de construir cuando los períodos son de diferentes órdenes de magnitud, lo que hace que el número de ciclos secundarios sea muy grande.
- Puede ser necesario dividir una tarea en varios procedimientos.
- La planificación es poco flexible y difícil de mantener: cualquier cambio en las tareas implica rehacer toda la planificación.

2.9. Esquema de Planificación de Tareas

2.9.1. Determinación de la Planificabilidad de un Conjunto de Tareas

La planificación de tareas se refiere al proceso de asignar el acceso de las tareas a los procesadores. Para determinar si un conjunto de tareas puede ejecutarse correctamente, se utilizan dos enfoques principales:

- **Algoritmo de planificación**: Determina el orden en que las tareas acceden a los procesadores.
- **Método de análisis de planificabilidad**: Se basa en un test que predice si las tareas son planificables conjuntamente, teniendo en cuenta las condiciones y restricciones especificadas a priori. Este análisis se realiza durante:
  - Todas las activaciones posibles de las tareas en su ejecución.
  - La ejecución de estas tareas, suponiendo la situación más desfavorable, es decir, el *WCET* (Worst Case Execution Time) para cada tarea.



2.9.2. Modelo Simple de Tareas

El modelo simple de tareas se basa en las siguientes suposiciones:

- **Programa**: Conjunto fijo de tareas que comparten el tiempo de un único procesador.
- **Tareas periódicas**: Con períodos conocidos.
- **Independencia**: Las tareas no se bloquean entre sí al acceder a recursos compartidos.
- **Plazo de respuesta máximo**: Todas las tareas tienen un plazo de respuesta máximo (D) que coincide con su período (T).
- **Ignorancia de retrasos del sistema**: Los retrasos debidos al sistema, como los cambios de contexto, son ignorados.
- **Tiempo máximo de cómputo**: El tiempo máximo de ejecución de las tareas (C_i) es conocido y fijo.
- No se contemplan tareas esporádicas ni aperiódicas.

2.9.3. Modelo de Tareas-II: Notación

La notación utilizada para describir las tareas es la siguiente:

- P: Prioridad de la tarea (si fuera aplicable).
- τ: Tarea (nombre de la tarea).
- t_a: Tiempo de activación (instante en que la tarea está lista para su ejecución).
- t_s: Tiempo de comienzo (instante en que la tarea comienza realmente su ejecución).
- t_f: Tiempo de finalización (instante en que la tarea finaliza su ejecución).
- t_l, d: Tiempo límite (instante límite para la ejecución de la tarea, $t_l(k) = d = t_a + D$).
- T: Periodo de ejecución (intervalo entre dos activaciones sucesivas de una tarea periódica).
- J: Latencia (tiempo entre la activación y el inicio de la ejecución de la tarea, $J(k) = t_s(k) - t_a(k)$).
- C: Tiempo de cómputo (tiempo de ejecución de la tarea).
- $C_{\max}$: Cómputo máximo (tiempo de ejecución en el peor caso de la tarea).
- e: Tiempo transcurrido (tiempo entre el comienzo y la finalización de la tarea, $e(k) = t_f(k) - t_s(k)$).
- R: Tiempo de respuesta (tiempo total para completar la ejecución de la tarea, $R(k) = J(k) + e(k)$).

2.9.4. Propósito de la Planificación de Tareas

La planificación de tareas, o *task-scheduling*, estudia técnicas de Programación Entera para obtener una asignación eficiente de recursos y tiempo a las actividades. El objetivo es cumplir ciertos requisitos de eficiencia, utilizando heurísticas que maximicen la función objetivo $U(N)$, que representa la utilización del procesador para N tareas de tiempo real.

2.9.5. Planificación con Prioridades

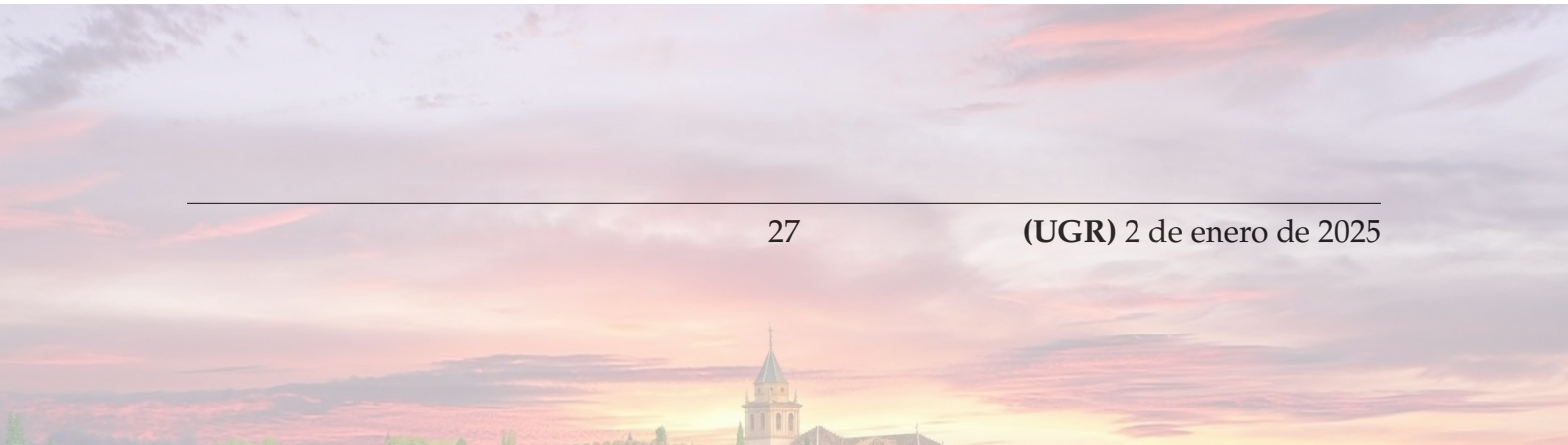
La planificación con prioridades permite solucionar los problemas mencionados previamente. Cada tarea tiene asociada un valor entero positivo, conocido como la prioridad de la tarea, que refleja su importancia relativa en el conjunto de tareas. Las principales características de la planificación con prioridades incluyen:

- La prioridad se asigna en función de las necesidades temporales de la tarea, no de su rendimiento o comportamiento.
- Las tareas ejecutables se despachan para su ejecución en orden de prioridad.
- La prioridad de una tarea puede depender de su *urgencia*, pero no de su *criticidad* en términos de tiempo real.

2.9.6. Esquema de Planificación de Tareas

El esquema de planificación de tareas generalmente consta de los siguientes elementos:

1. Un algoritmo para ordenar el acceso de las tareas al procesador.
2. Un test para predecir el comportamiento del sistema en el peor caso de planificación, es decir, cuando existe mayor interferencia entre las tareas.
3. Características principales de un esquema de planificación de tareas de tiempo real:
   - Dinámico vs. estático.
   - Desplazante (preemptive) de tareas menos prioritarias.
   - Análisis de tareas dentro de una "ventana temporal".
   - Concepto de instante crítico.
4. Una posible solución al problema de análisis temporal de las tareas esporádicas y aperiódicas.



2.9.7. Planificación con Prioridades-II: Tipos de Esquemas

Existen dos tipos de esquemas de planificación con prioridades:

- **Prioridades asignadas estáticamente a las tareas**: Cada tarea tiene una prioridad fija durante toda la ejecución del sistema.
  - *Cadencia monótona o RMS (Rate Monotonic Scheduling)*: La prioridad se asigna a la tarea con el período más corto (mayor frecuencia de activación).
  - *Plazo de respuesta monótono o DMS (Deadline Monotonic Scheduling)*: Se asigna mayor prioridad a la tarea con el plazo de respuesta más corto.
- **Prioridades asignadas dinámicamente a las tareas**: Las prioridades de las tareas cambian durante la ejecución del sistema.
  - *EDF (Earliest Deadline First)*: Se asigna prioridad a la tarea cuyo plazo de respuesta absoluto es más cercano.
  - *LLF (Least Laxity First)*: Se asigna prioridad a la tarea con menor holgura temporal.

2.9.8. Modelo de Tareas-IV: Notación Adicional

Se introducen más parámetros de interés en la planificación de tareas:

- D: Plazo de respuesta máximo.
- φ: Desplazamiento o fase, el tiempo para activarse por primera vez.
- RJ: Fluctuación relativa o jitter, la máxima desviación en el tiempo de comienzo entre dos activaciones sucesivas de una tarea, definida como:

$$RJ = \text{máx}((t_s(k+1) - t_a(k+1)) - (t_s(k) - t_a(k)))$$

- H: Holgura o slack time, el tiempo disponible para permanecer activa dentro del plazo de respuesta máximo:

$$H(k) = t_l - t_a(k) - e(k) = D - e(k)$$

2.10. Esquema de Planificación de Tareas de Tiempo Real Basado en el Algoritmo de Cadencia Monótona (RMS)

2.10.1. 1. Algoritmo de Cadencia Monótona (Rate Monotonic Scheduling)

El algoritmo de cadencia monótona (RMS)[1] se basa en los siguientes principios:

- Las prioridades se asignan de manera estática a cada tarea (τ_i).

[1] Para un ejemplo de ello accede a la diapositiva 4.39 del tema 4

- La prioridad es mayor para las tareas con menor período (T_i):

$$\forall i, j : T_i < T_j \implies P_i > P_j$$

- Este algoritmo no tiene en cuenta la criticidad de las tareas, sino su urgencia, definida como $\frac{1}{T_i}$.
- El RMS es óptimo entre los algoritmos de asignación estática de prioridades bajo las siguientes condiciones:
  - Las tareas son periódicas.
  - El plazo de respuesta de cada tarea coincide con su período ($D_i = T_i$).

2.10.2. 2. Test de Planificabilidad Basado en el Factor de Utilización del Procesador

El test de planificabilidad para un conjunto de tareas periódicas $\{\tau_1, \tau_2, \ldots, \tau_N\}$ verifica la suma del factor de utilización del procesador:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i}$$

- En un sistema de N tareas periódicas, independientes y con prioridades asignadas según su frecuencia (inverso al período), el sistema es planificable si el factor de utilización del procesador ($U(N)$) cumple la siguiente condición:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} < U_0 = N \cdot (2^{\frac{1}{N}} - 1)$$

- Si $U \leq U_0(N)$, el sistema es planificable. En caso contrario, el test no puede determinar si el sistema es planificable (es una condición suficiente, pero no necesaria).

2.10.3. 3. Inexactitud del Test de Planificabilidad RM

El límite máximo de utilización $U_0(N)$ depende del número de tareas N y decrece conforme aumenta N. A continuación, se muestran algunos valores:

N	$U_0(N)$ **(%)**
1	100
2	82.85
3	78.0
4	75.7
5	74.3
10	71.8
∞	69.3

Cuadro 3: Límite máximo de utilización del procesador $U_0(N)$ en función del número de tareas N.

2.10.4. 4. Características del Test RMS

- El test basado en el factor de utilización es solo una condición suficiente para determinar si un conjunto de tareas es planificable.
- En caso de que un conjunto de tareas no pase el test, es posible realizar un análisis más detallado utilizando diagramas de Gantt en una ventana temporal definida como:
$$M_i = \text{m.c.m.}\{T_1, T_2, \ldots, T_i\}$$
- Los tests de planificabilidad basados en la utilización del procesador no proporcionan información sobre los tiempos de respuesta de las tareas.

2.11. Segundo Teorema de Planificabilidad de un Conjunto de Tareas Periódicas

El segundo teorema de planificabilidad establece las siguientes condiciones para un sistema de N tareas periódicas independientes, con prioridades asignadas en orden de su frecuencia (menor período, mayor prioridad):

- Todas las tareas cumplen sus tiempos límite si, cuando se activan simultáneamente, cada tarea finaliza su ejecución antes de que expire el tiempo límite asignado en su primera activación.
- Con este criterio, se puede alcanzar una utilización máxima del procesador:
$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

2.11.1. Características del Segundo Teorema

- La cota máxima que puede alcanzar el factor de utilización (U) es ahora mayor que la permitida por el test RMS.
- Este test es más flexible y puede aplicarse a un mayor número de conjuntos de tareas, ya que permite un mayor factor de utilización del procesador.
- A diferencia del test de RMS, este test es exacto:
  - Proporciona una condición necesaria y suficiente para determinar si un conjunto de tareas es planificable.
  - Si las tareas pasan este test, el sistema es planificable en todos los casos.

2.12. Ventajas de los Esquemas de Planificación Estáticos

Los esquemas de planificación estáticos presentan varias ventajas importantes:

- **Simplicidad y eficiencia**:

  - Son más sencillos y eficientes de implementar en comparación con los esquemas dinámicos.
  - Requieren menos recursos computacionales para su ejecución.

- **Facilidad de diseño**:
  - Es más sencillo diseñar un sistema con tiempos límite (plazos de respuesta absolutos) calculados exactamente.

- **Flexibilidad**:
  - Permiten la incorporación de otros factores que influyen en la planificación de tareas, especialmente cuando las prioridades no están directamente asociadas a los tiempos límite.

- **Predictibilidad bajo sobrecarga transitoria**:
  - Durante períodos de sobrecarga transitoria, los esquemas estáticos suelen ser más predecibles.
  - Esto no siempre es cierto en los esquemas dinámicos, como el algoritmo *Earliest Deadline First* (EDF[2]).

2.13. Modelo General de Tareas de Tiempo Real (TR)

La estructura general asumida para resolver el problema de planificabilidad incluye las siguientes características:

- **Programa:** Conjunto fijo de tareas que comparten el tiempo de un procesador. Pueden incluirse tareas aperiódicas o esporádicas.
- **Bloqueo:** Las tareas pueden bloquearse al acceder a recursos compartidos, como semáforos utilizados en su código.
- **Plazo de respuesta máximo (D):**
  - Generalmente, no coincide con su período (T).
  - En el caso de las tareas esporádicas, el plazo de respuesta máximo suele ser muy corto.
- **Simplificaciones:**
  - Se ignoran los retrasos debidos al sistema (como cambios de contexto).
  - No se guardan eventos de tiempo real.
  - El tiempo máximo de cómputo de las tareas (C) se considera conocido y fijo.

[2]Ejemplo en la diapositiva 4.44 del tema 4

2.14. Inversión de Prioridad

2.14.1. Descripción del Problema

El problema de la inversión de prioridad ocurre en los siguientes casos:

- Una tarea más prioritaria queda bloqueada y debe esperar arbitrariamente largo tiempo debido a la ejecución continua de tareas menos prioritarias.
- Esto invalida cualquier previsión sobre la planificación del conjunto de tareas, incluso si han pasado el test de planificabilidad como el RMS.
- Surge como consecuencia del esquema estático de asignación de prioridades a las tareas.

2.14.2. Características de la Inversión de Prioridad

- La inversión de prioridad no puede eliminarse completamente, pero sus efectos adversos sobre la planificación pueden minimizarse.

2.15. Protocolo de Sección Crítica No Expulsable

2.15.1. Descripción del Protocolo

- Durante la ejecución de las secciones críticas, las tareas tienen una prioridad estática igual a la prioridad máxima del sistema.

2.15.2. Características del Protocolo

- Es el protocolo más simple para evitar la inversión de prioridad.
- Puede inducir bloqueos excesivamente largos en las tareas más prioritarias.
- En ciertos casos, puede interferir con la ejecución de todas las tareas, incluso si estas no hacen uso de recursos compartidos.

2.16. Tiempo de Bloqueo en Sección Crítica No Expulsable

- Para una tarea τ_i, el tiempo de ejecución en el peor caso se incrementará por un factor constante:
$$C_i^* = C_i + B_i$$
- La tarea más prioritaria τ_i puede ser bloqueada, como máximo, durante la ejecución de una única sección crítica:
$$B_i = \max_{j,j>i} \left(\max_k \text{Dur}(s_{jk}) \right)$$
- Aquí:
  - s_{jk} representa la sección crítica k ejecutada por la tarea τ_j, que tiene menor prioridad que τ_i.
  - $\text{Dur}(s_{jk})$ es la duración de dicha sección crítica.

2.17. Protocolo de Herencia de Prioridad

2.17.1. Descripción del Protocolo

El protocolo de herencia de prioridad se basa en el siguiente principio[3]:

- La **prioridad efectiva** de una tarea en ejecución será el máximo entre:
  - Su **prioridad por defecto**, asignada al inicio del programa.
  - Las **prioridades** de otras tareas con las que comparte recursos y que mantiene bloqueadas en ese momento.

2.17.2. Características del Protocolo

- Las prioridades de las tareas no son estáticas; pueden variar dinámicamente durante la ejecución del programa.
- Este enfoque permite:
  - **Minimizar el efecto de la inversión de prioridad**: Al elevar temporalmente la prioridad de las tareas bloqueadas, se evita que estas queden en espera indefinidamente.
  - **Optimizar el aprovechamiento del procesador**: Se prioriza la resolución de bloqueos en recursos compartidos, reduciendo el tiempo total de ejecución.

2.17.3. Ventajas del Protocolo

- Reducción de los bloqueos prolongados.
- Mayor predictibilidad en la planificación de tareas.
- Mejor utilización de los recursos del sistema.

2.18. Tipos de Bloqueos en el Protocolo de Herencia de Prioridad

2.18.1. Tipos de Bloqueos

Con el protocolo de herencia de prioridad, se identifican los siguientes tipos de bloqueos:

a) **Bloqueos directos:** Ocurren cuando una tarea de mayor prioridad se encuentra bloqueada esperando un recurso compartido que está siendo utilizado por una tarea de menor prioridad.

b) **Bloqueos indirectos:** Ocurren cuando una tarea de mayor prioridad es bloqueada debido a que otra tarea de menor prioridad está esperando un recurso bloqueado por una tercera tarea.

[3]Ejemplo de esto se encuentra en la diapositiva 4.52 del tema 4

2.18.2. Características del Protocolo

- Las tareas sólo pueden ser bloqueadas un número limitado de veces por otras tareas de menor prioridad.
- Consecuencias:
  1. Si una tarea tiene definidas M secciones críticas, entonces el número máximo de veces que puede ser bloqueada es M.
  2. Si hay $N < M$ tareas de menor prioridad, el número máximo de bloqueos se reduce a N.

2.19. Cálculo del Factor de Bloqueo

El factor de bloqueo (B_i) para una tarea τ_i está dado por el mínimo entre dos términos:

1. Bl_i : Bloqueo debido a tareas τ_j menos prioritarias que acceden a secciones críticas k compartidas con tareas de mayor prioridad:

$$Bl_i = \sum_{j=i+1}^{n} \max_{k}[Dur_{j,k} : Limite(S_k) \geq P_i]$$

2. Bs_i : Bloqueo debido a todas las secciones críticas a las que accede la tarea τ_i:

$$Bs_i = \sum_{k=1}^{m} \max_{j>i}[Dur_{j,k} : Limite(S_k) \geq P_i]$$

El factor de bloqueo total para la tarea τ_i es:

$$B_i = \min(Bl_i, Bs_i)$$

2.20. Protocolos de Techo de Prioridad

2.20.1. Techo de Prioridad de un Recurso

El **techo de prioridad** de un recurso es la prioridad de la tarea más prioritaria que puede bloquear el acceso al recurso.

2.20.2. Características de los Protocolos de Techo de Prioridad

- Garantizan que una tarea prioritaria sólo puede ser bloqueada como máximo una vez por otras de menor prioridad.
- Previenen los interbloqueos.
- Eliminan los bloqueos transitivos.
- Aseguran el acceso en exclusión mutua a los recursos compartidos.

2.21. Protocolo de Techo de Prioridad Inmediato (PPP)

1. Cada tarea tiene una **prioridad estática** asignada por defecto.
2. Cada recurso tiene un valor de **techo de prioridad** definido.
3. Una tarea tiene una **prioridad dinámica**, que es el máximo entre:
   - Su prioridad estática inicial.
   - Los valores de los techos de prioridad de los recursos que mantiene bloqueados[4].

2.22. Tiempo de Bloqueo

2.22.1. Cálculo del Tiempo de Bloqueo

El tiempo máximo de bloqueo de una tarea τ_i está dado por la duración de la sección crítica más larga a la que acceden las tareas de prioridad inferior, siempre que el techo de prioridad del recurso sea igual o superior a la prioridad de τ_i:

$$B_i = \max_{\{j,k\}}\{Dur_{j,k} \,|\, \text{prio}(\tau_j) < \text{prio}(\tau_i),\ \text{techo\_prioridad}(S_k) \geq \text{prio}(\tau_i)\}$$

2.22.2. Comportamiento con el Protocolo de Techo de Prioridad Inmediato (PPP)

Con el protocolo PPP:

- Una tarea puede ser bloqueada por otra de menor prioridad aunque no accedan a recursos comunes.
- Esto ocurre debido a la dinámica del techo de prioridad que se asigna a los recursos bloqueados.

2.23. Asignación de Prioridades a las Tareas Aperiódicas

- En aplicaciones de tiempo real, las tareas aperiódicas no deben tener una prioridad inferior a la de las tareas de misión crítica.
- Para gestionar las tareas aperiódicas, se utiliza un **servidor aperiódico**, que puede ser:
  - Una tarea real o conceptual.
  - Diseñado para ejecutar procesos aperiódicos tan pronto como sea posible, sin afectar las garantías de las tareas periódicas.

[4] Ejemplo del PPP en la diapositiva 4.57 del tema 4

2.24. Servicio de Tareas Aperiódicas

2.24.1. Servicio con Tiempo en Segundo Plano

- El tiempo aperiódico se incrementa a intervalos regulares.
- Estas tareas tienen la prioridad más baja del sistema.
- Si no hay peticiones aperiódicas, el tiempo asignado a estas tareas se pierde[5].

2.24.2. Servicio de Aperiódicas con Sondeos

- Se incluye una tarea adicional, denominada **tarea sondeante**, dentro del conjunto de tareas periódicas.
- Características de la tarea sondeante:
  - Tiene un periodo T_s y un tiempo de ejecución en el peor caso C_s.
  - Se le asigna una prioridad adecuada dentro del sistema.
- El test de planificabilidad de cadencia monótona se aplica considerando la tarea sondeante[6]:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq (N+1)\left(2^{\frac{1}{N+1}} - 1\right)$$

2.25. Servidor Diferido

2.25.1. Características del Servidor Diferido

- Preserva el tiempo dedicado a tareas aperiódicas, incluso si temporalmente no hay peticiones de este tipo.
- Se asigna un tamaño de servidor C_s que se utiliza únicamente para atender peticiones aperiódicas.
- El tamaño del servidor C_s se recarga hasta su valor máximo al inicio de cada periodo del servidor T_s.
- El valor inicial de C_s se establece tras un análisis previo de planificabilidad del conjunto de tareas.
- Las peticiones aperiódicas se atienden con una prioridad alta, siempre que el tiempo del servidor no se haya agotado.

[5] Ejemplo en la diapositiva 4.60 del tema 4
[6] Ejemplo en la diapositiva 4.62 del tema 4

2.25.2. Análisis de Planificabilidad con el Servidor Diferido

2.25.2.1. Configuración del Sistema

- Conjunto de N tareas periódicas: $\tau_1, \tau_2, \ldots, \tau_N$.
- Un servidor diferido con prioridad más alta, y un tamaño C_s asociado a un periodo T_s.

2.25.2.2. Cálculo del Límite Superior de Utilización

El límite superior de utilización se calcula como:

$$U_{\text{mls}} = U_s + N \left[\left(\frac{U_s + 2}{2U_s + 1} \right)^{\frac{1}{N}} - 1 \right]$$

Límite cuando $N \to \infty$:

$$\lim_{N \to \infty} U_{\text{mls}} = U_s + \ln \left(\frac{U_s + 2}{2U_s + 1} \right)$$

2.25.2.3. Condición de Planificabilidad

Para un conjunto de N tareas periódicas y un servidor diferido con límites de utilización U_p y U_s, respectivamente, se garantiza la planificabilidad con el algoritmo de cadencia monótona si:

$$U_p + U_s \leq U_{\text{mls}}$$

Forma equivalente:

$$U_p \leq \ln \left(\frac{U_s + 2}{2U_s + 1} \right)$$

2.25.3. Ventajas del Servidor Diferido

- Maximiza el tiempo disponible para las tareas aperiódicas.
- Permite una respuesta eficiente a las tareas aperiódicas con prioridad alta.
- Reduce el impacto de las tareas aperiódicas en la planificación de las tareas periódicas[7].

[7] Ejemplo del servidor diferido en la diapositiva 4.65 del tema 4

3 Relaciones de Ejercicios

3.1. Relación 1

GII-ADE-M. Relación de problemas. Tema 1. 20/09/2024

1. Considerar el siguiente fragmento de programa para 2 procesos P1 y P2: Los dos procesos pueden ejecutarse a cualquier velocidad. ¿Cuáles son los posibles valores resultantes para la variable x? Suponer que x debe ser cargada en un registro para incrementarse y que cada proceso usa un registro diferente para realizar el incremento.

```
{ variables compartidas }
var x : integer := 0 ;
Process P1;
  var i: integer;
begin
  for i:= 1 to 2 do begin
    x:= x + 1;
  end
end
```

```
Process P2;
   var j: integer;
begin
  for i:= 1 to 2 do begin
     x:= x + 1;
   end
end
```

2. ¿Cómo se podría hacer la copia del fichero f en otro g, de forma concurrente, utilizando la instrucción concurrente *cobegin-coend*? Para ello, suponer que:

   - Los archivos son una secuencia de items de un tipo arbitrario T, y se encuentran ya abiertos para lectura (f) y escritura (g). Para leer un ítem de *f* se usa la llamada a función leer(f) y para saber si se han leído todos los ítems de f, se puede usar la llamada fin(f) que devuelve verdadero si ha habido al menos un intento de leer cuando ya no quedan datos. Para escribir un dato x en *g* se puede usar la llamada a procedimiento escribir(g,x).
   - El orden de los items escritos en *g* debe coincidir con el de *f*
   - Dos accesos a dos archivos distintos pueden solaparse en el tiempo

3. Construir, utilizando las instrucciones concurrentes *cobegin-coend* y *fork-join*, programas concurrentes que se correspondan con los grafos de precedencia que se muestran a continuación:

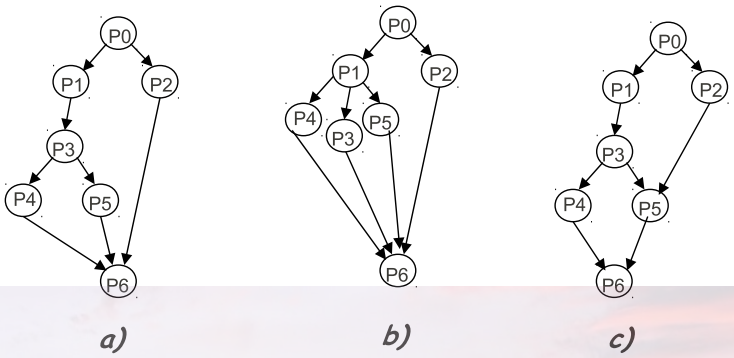


Figura 1: Grafos de sincronización con forks y joins.

4. Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

```
begin
  P0 ;
   cobegin
     P1 ;
     P2 ;
     cobegin
       P3 ; P4 ; P5 ; P6 ;
     coend
     P7 ;
  coend
P8;
end
```

```
begin
  P0 ;
   cobegin
      begin
        cobegin
          P1; P2;
        coend
        P5;
      end;
      begin
        cobegin
          P3; P4;
        coend;
        P6;
      end;
  coend;
  P7;
end;
```

5. Suponer un sistema de tiempo real que dispone de un captador de impulsos conectado a un contador de energía eléctrica. La función del sistema consiste en contar el número de impulsos producidos en 1 hora (cada Kwh consumido se cuenta como un impulso) e imprimir este número en un dispositivo de salida. Para ello se dispone de un programa concurrente con 2 procesos: un proceso acumulador (lleva la cuenta de los impulsos recibidos) y un proceso escritor (escribe en la impresora). En la variable común a los 2 procesos n se lleva la cuenta de los impulsos. El proceso acumulador puede invocar un procedimiento Espera_impulso para esperar a que llegue un impulso, y el proceso escritor puede llamar a Espera_fin_hora para esperar a que termine una hora. El código de los procesos de este programa podría ser el siguiente:

```
{ variable compartida: }
var n : integer; { contabiliza impulsos }
begin
while true do begin
  Espera_impulso();
  < n := n+1 > ; { (1) }
  end
end
process Escritor ;
begin
while true do begin
  Espera_fin_hora();
  write( n ) ; { (2) }
  < n := 0 > ; { (3) }
  end
end
```

En el programa se usan sentencias de acceso a la variable n encerradas entre los símbolos $<$ y $>$. Esto significa que cada una de esas sentencias se ejecuta en exclusión mutua entre los dos

procesos, es decir, esas sentencias se ejecutan de principio a fin sin entremezclarse entre ellas. Supongamos que en un instante dado el acumulador está esperando un impulso, el escritor está esperando el fin de una hora, y la variable n vale k. Después se produce de forma simultánea un nuevo impulso y el fin del periodo de una hora.

Obtener las posibles secuencias de entrelazamiento de las instrucciones (1),(2), y (3) a partir de dicho instante, e indicar cuales de ellas son correctas y cuales incorrectas (las incorrectas son aquellas en las cuales el impulso no se contabiliza).

6. Supongamos un programa concurrente en el cual hay, en memoria compartida dos vectores a y b de enteros y con tamaño par, declarados como sigue:

```
var a,b : array[1..2*n] of integer ; { n es una constante predefinida }
```

Queremos escribir un programa para obtener en b una copia ordenada del contenido de a (nos da igual el estado en que queda a después de obtener b). Para ello disponemos de la función Sort que ordena un tramo de a (entre las entradas s y t, ambas incluidas). También disponemos la función Copiar, que copia un tramo de a (desde s hasta t) en b (a partir de o).

```
procedure Sort( s,t : integer );
  var i, j : integer ;
  begin
   for i := s to t do
   for j:= s+1 to t do
      if a[i] < a[j] then
       swap( a[i], b[j] ) ;
  end
procedure Copiar( o,s,t : integer );
  var d : integer ;
  begin
   for d := 0 to t-s do
   b[o+d] := a[s+d] ;
  end
```

El programa para ordenar se puede implementar de dos formas:

- Ordenar todo el vector a, de forma secuencial con la función Sort, y después copiar cada entrada de a en b, con la función Copiar
- Ordenar las dos mitades de a de forma concurrente, y después mezclar dichas dos mitades en un segundo vector b (para mezclar usamos un procedimiento Merge).

A continuación vemos el código de ambas versiones:

```
procedure Secuencial() ;
var i : integer ;
begin
Sort( 1, 2*n ); { ordena a }
Copiar( 1, 2*n ); { copia a en b }
end
procedure Concurrente() ;
begin
cobegin
Sort( 1, n );
Sort( n+1, 2*n );
coend
Merge( 1, n+1, 2*n );
end
```

El código de Merge se encarga de ir leyendo las dos mitades de a, en cada paso, seleccionar el menor elemento de los dos siguientes por leer (uno en cada mitad), y escribir dicho menor elemento en la siguiente mitad del vector mezclado b. El código es el siguiente:

```
procedure Merge( inferior, medio, superior: integer ) ;
  { siguiente posicion a escribir en b }
  var escribir : integer := 1 ;
  { siguiente pos. a leer en primera mitad de a }
  var leer1 : integer := inferior ;
  { siguiente pos. a leer en segunda mitad de a }
  var leer2 : integer := medio ;
  begin
   { mientras no haya terminado con alguna mitad }
   while leer1 < medio and leer2 <= superior do begin
    if a[leer1] < a[leer2] then begin { minimo en la primera mitad }
      b[escribir] := a[leer1] ;
      leer1 := leer1 + 1 ;
      end else begin { minimo en la segunda mitad }
        b[escribir] := a[leer2] ;
        leer2 := leer2 + 1 ;
   end
   escribir := escribir+1 ;
  end
{ se ha terminado de copiar una de las mitades,
   copiar lo que quede de la otra }
   if leer2 > superior then
     { copiar primera } Copiar( escribir, leer1, medio-1 );
    else Copiar( escribir, leer2, superior ); { copiar segunda }
  end
```

Llamaremos $T_s(k)$ al tiempo que tarda el procedimiento Sort cuando actúa sobre un segmento del vector con k entradas. Suponemos que el tiempo que (en media) tarda cada iteración del bucle interno que hay en Sort es la unidad (por definición). Es evidente que ese bucle tiene $\frac{k(k-1)}{2}$ iteraciones, luego:

$$T_s(k) = \frac{k(k-1)}{2} = 1/2k^2 - 1/2k$$

El tiempo que tarda la versión secuencial sobre 2n elementos (llamaremos S a dicho tiempo) será evidentemente $T_s(2n)$, luego:

$$S = T_s(n) = 1/2(2n)^2 - 1/2(2n) = 2n^2 - n$$

con estas definiciones, calcula el tiempo que tardará la versión paralela, en dos casos:

(a) Las dos instancias concurrentes de Sort se ejecutan en el mismo procesador (llamamos P1 al tiempo que tarda).

(b) Cada instancia de Sort se ejecuta en un procesador distinto (lo llamamos P2)

Escribe una comparación cualitativa de los tres tiempos (S,P1 y P2). Para esto, hay que suponer que cuando el procedimiento Merge actúa sobre un vector con p entradas, tarda p unidades de tiempo en ello, lo cual es razonable teniendo en cuenta que en esas circunstancias Merge copia p valores desde a hacia b. Si llamamos a este tiempo $T_m(p)$, podemos escribir $T_m(p) = p$

7. Supongamos que tenemos un programa con tres matrices (a,b y c) de valores flotantes declaradas como variables globales. La multiplicación secuencial de a y b (almacenando el resultado en c) se puede hacer mediante un procedimiento MultiplicacionSec declarado como aparece aquí:

```
var a, b, c : array[1..3,1..3] of real ;
procedure MultiplicacionSec()
  var i,j,k : integer ;
  begin
    for i := 1 to 3 do
     for j := 1 to 3 do begin
          c[i,j] := 0 ;
          for k := 1 to 3 do
             c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
  end
end
```

   Escribir un programa con el mismo fin, pero que use 3 procesos concurrentes. Suponer que los elementos de las matrices a y b se pueden leer simultáneamente, así como que elementos distintos de c pueden escribirse simultáneamente.

8. Un trozo de programa ejecuta nueve rutinas o actividades (P1, P2, . . . , P9), repetidas veces, de forma concurrente con `cobegin coend` (ver trozo de código), pero que requieren sincronizarse según determinado grafo (ver la figura):

```
while true do
cobegin
P1 ; P2 ; P3 ;
P4 ; P5 ; P6 ;
P7 ; P8 ; P9 ;
coend
```

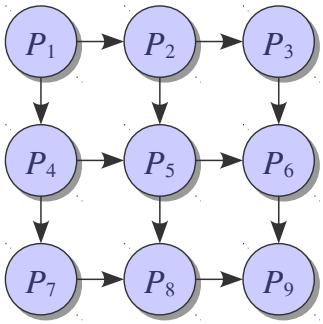


Figura 2: Grafo de sincronización de actividades.

   Supón que queremos realizar la sincronización indicada en el grafo, usando para ello llamadas desde cada rutina a dos procedimientos (`EsperarPor` y `Acabar`). Se dan los siguientes hechos:

   - El procedimiento `EsperarPor(i)` es llamado por una rutina cualquiera (la número k) para esperar a que termine la rutina número i, usando espera ocupada. Por tanto, se usa por la rutina k al inicio para esperar la terminación de las otras rutinas que corresponda según el grafo.
   - El procedimiento `Acabar(i)` es llamado por la rutina número i, al final de la misma, para indicar que dicha rutina ya ha finalizado.

- Ambos procedimientos pueden acceder a variables globales en memoria compartida.
- Las rutinas se sincronizan única y exclusivamente mediante llamadas a estos procedimientos, siendo la implementación de los mismos completamente transparente para las rutinas.

Escribe una implementación de `EsperarPor` y `Acabar` (junto con la declaración e inicialización de las variables compartidas necesarias) que cumpla con los requisitos dados.

9. En el problema 8 los procesos P1, P2, . . ., P9 se ponen en marcha usando `cobegin/coend`. Escribe un programa equivalente, que ponga en marcha todos los procesos, pero que use declaración estática de procesos, usando un vector de procesos P, con índices desde 1 hasta 9, ambos incluidos. El proceso P[n] contiene una secuencia de instrucciones desconocida, que llamamos Sn, y además debe incluir las llamadas necesarias a `Acabar` y `EsperarPor` (con la misma implementación que antes) para lograr la sincronización adecuada. Se incluye aquí una plantilla:

```
Process P[ n : 1..9 ]
  begin
   ..... { esperar (si es necesario) a los procesos que corresponda }
   Sn ; { sentencias especificas de este proceso (desconocidas) }
   ..... { senalar que hemos terminado }
  end
```

10. Para los siguientes fragmentos de código, obtener la *poscondición* adecuada para convertirlo en un *triple* demostrable con la Lógica de Programas:

    (a) $\{i < 10\}\ i = 2 * i + 1\ \{\ \}$

    (b) $\{i > 0\}\ i = i - 1; \{\ \ \}$

    (c) $\{i > j\}\ i = i + 1; j = j + 1\ \{\ \ \}$

    (d) $\{falso\}\ a = a + 7;\ \{\ \ \}$

    (e) $\{verdad\}\ i = 3;\ j = 2 * i\ \{\ \ \}$

    (f) $\{verdad\}\ c = a + b; c = c/2\ \{\ \ \}$

11. ¿Cuáles de los siguientes triples no son demostrables con la Lógica de Programas?

    (a) $\{i > 0\} i = i - 1;\ \{i \geq 0\}$

    (b) $\{x \geq 7\}\ x = x + 3;\ \{x \geq 9\}$

    (c) $\{i < 9\}\ i = 2 * i + 1; \{\ i \leq 20\}$

    (d) $\{a > 0\}\ a = a - 7;\ \{a > -6\}$

12. Si el triple $\{P\}\ C\ \{Q\}$ es demostrable, indicar por qué los siguientes triples también lo son (o no se pueden demostrar y por qué):

    (a) $\{P\}\ C\ \{Q \vee P\}$

    (b) $\{P \wedge D\}\ C\ \{Q\}$

    (c) $\{P \vee D\}\ C\ \{Q\}$

    (d) $\{P\}\ C\ \{Q \vee D\}$

    (e) $\{P\}\ C\ \{Q \wedge P\}$

13. Si el triple $\{P\}\ C\ \{Q\}$ es demostrable, ¿cuál de los siguientes triples no se puede demostrar?

    (a) $\{P\ \wedge\ D\}\ C\ \{Q\}$
    (b) $\{P\ \vee\ D\}\ C\ \{Q\}$
    (c) $\{P\}\ C\ \{Q\ \vee\ D\}$
    (d) $\{P\}\ C\ \{Q\ \vee\ P\}$

14. Dado el programa `int x = 5, y = 2; cobegin < x = x + y >;< y = x * y > coend;`, obtener:

    (a) Valores finales de `x` e `y`
    (b) Valores finales de `x` e `y` si quitamos los símbolos $< >$ de instrucción atómica.

15. Comprobar si la demostración del triple $\{x\ \geq\ 2\}\ <x = x\ -\ 2>;\ \{x\ \geq\ 0\}$ interfiere con los teoremas siguientes:

    (a) $\{x\ \geq\ 0\}\ <x = x\ +\ 3>\ \{x\ \geq\ 3\}$
    (b) $\{x\ \geq\ 0\}\ <x = x\ +\ 3>\ \{x\ \geq\ 0\}$
    (c) $\{x\ \geq\ 7\}\ <x = x\ +\ 3>\ \{x\ \geq\ 10\}$
    (d) $\{y\ \geq\ 0\}\ <y = y\ +\ 3>\ \{y\ \geq\ 3\}$
    (e) $\{x\ es\ impar\}\ <y = x\ +\ 1>\ \{y\ es\ par\}$

16. Dado el siguiente triple:

```
                {x==0}
                cobegin
<x=x+a> || <x=x+b> || <x=x+c>
                coend
                {x==a+b+c}
```

    Demostrarlo utilizando la lógica de asertos para cada una de las tres instruccciones atómicas y después que se llega a la poscondición final `x==a+b+c` utilizando para ello la regla de la *composición concurrente* de instrucciones atómicas.

17. El siguiente triple:

$$\{x == 0\ \wedge\ y == 0\ \wedge\ z == 0\}$$
$$< x = z + a > \parallel < y = x + b >$$
$$\{x == a\}\ \wedge\ \{y == b \vee y == a + b\}\ \wedge\ \{z == 0\}$$

    (a) Es indemostrable salvo que se cumpla siempre que `a==0`
    (b) El triple anterior es demostrable para cualquier valor de las variables `a` o `b`
    (c) Es indemostrable salvo que se cumpla siempre que $b == 0\ \ \vee\ \ a == 0$
    (d) Es indemostrable salvo que se cumpla siempre que $a == 0\ \ \wedge\ \ b == 0$

18. Suponer que $\{\texttt{suma} > 1\}\ \texttt{suma} = \texttt{suma} + 4\{\texttt{suma} > 5\}$ es demostrable, entonces: ¿cuál de los siguientes triples es también demostrable? (indicar por qué)

$$(a)\{suma > 2\}suma = suma + 4\{suma > 5\}$$
$$(b)\{suma \geq 1\}suma = suma + 4\{suma > 5\}$$
$$(c)\{suma > 0\}suma = suma + 4\{suma > 5\}$$
$$(d)\{suma > 1\}suma = suma + 4\{suma > 6\}$$

19. Suponer que $\{\texttt{x} < \texttt{y}\}\ \texttt{C}_1\{\texttt{u} < \texttt{v}\}$ es demostrable, entonces: ¿cuál/cuáles de los siguientes triples es/son también demostrable? (indicar por qué)

$$(a)\{x \leq y\}C_1\{u < v\}$$
$$(b)\{x \leq y - 2\}C_1\{u < v\}$$
$$(c)\{x \leq y\}C_1\{u \leq v\}$$
$$(d)\{x < y\}C_1\{u < v - 2\}$$

20. Seleccionar el valor correcto de las 2 variables (x e y) después de ejecutarse el siguiente programa concurrente:

```
int x=5, y =2;
cobegin <x= x+y>; <y= x*y>; <x= x-y>; coend;
```

(a) `x==7` e `y==14`
(b) `x==5` e `x==10`
(c) `x==-7` e `y==14`
(d) `x==-3` e `y==10`

21. El siguiente código concurrente no puede ser demostrado directamente con la LP. Elegir la respuesta que explica correctamente la razón de que ocurra esto.

```
{x==0} cobegin <x=x+a>; <x=x+a> coend; {x==2*a}
(a es un valor entero positivo)
```

(a) Porque la poscondición que se propone `{x==2*a}` es falsa
(b) Porque falta incluir la posibilidad de que el valor final de x sea también `{x==a}`
(c) Porque al aplicar directamente la regla de inferencia de la *composición concurrente* utilizo unas condiciones (pre y post- condiciones) demasiado débiles
(d) Porque tengo que incluir en los asertos el valor del contador de programa de cada procesador

22. Estudiar cuáles son los valores finales de las variables x e y en el siguiente programa secuencial. Insertar los asertos adecuados entre llaves, antes y después de cada sentencia, para poder obtener una traza de demostración del programa, que incluya en su último aserto los valores finales de las variables.

$$(a)int\ x = C1$$
$$(b)int\ y = C2$$
$$(c)x = x + y;$$
$$(d)y = x * y;$$
$$(e)x = x - y;$$

23. Demostrar que el siguiente triple es cierto:

$$\{x == 0\}$$
$$cobegin$$
$$< x = x + 1 > || < x = x + 2 > || < x = x + 4 >$$
$$coend$$
$$\{x == 7\}$$

24. Dada la siguiente construcción de composición concurrente P:

$$cobegin$$
$$< x = x - 1 >; < x = x + 1 >; || < y = y + 1 >; < y = y - 1 >;$$
$$coend;$$

demostrar que se cumple la invariancia de {x==y}, es decir, que {x==y} P {x==y}; es un triple cierto.

25. Usando la regla de la conjunción, demostrar que $\{i > 2\} i := 2 * i \{i > 4\}$

26. Se dan los siguientes triples de Hoare:

$$\{j > 1\}\, i = i + 2;\ j = j + 3;\ \{j > 4\}$$
$$\{i > 2\}\, i = i + 2;\ j = j + 3;\ \{i > 4\}$$

Demostrar que estos triples implican que $\{j > 1,\ i > 2\}\, i = i + 2;\ j = j + 3;$ $\{j > 4,\ i > 4\}$. ¿Qué regla se debe utilizar para la demostración?

27. Sean `A` y `B` los valores iniciales de `a` y `b` respectivamente. Escribir un fragmento de código que tenga `{a=A+B, b=A-B}` como poscondición y demostrar que el código es correcto.

28. Demostrar que la siguiente sentencia tiene la poscondición $\{x \geq 0,\ x^2 = a^2\}$.
$if\ a > 0\ then\ x := a\ else\ x := -a$

$$\{V\}\ if\ a > 0\ \ then\ \ x := a\ else\ x := -a\ \{x \geq 0,\ x^2 = a^2\}$$

29. El siguiente fragmento de código tiene $\{P\} \equiv \{sum = j*(j-1)/2\}$ como precondición y poscondición. Demostrar que es verdadero: $\{P\}\, sum := sum + j; j := j + 1;\ \{P\}$

30. Demostrar que $\{i * j + 2 * j + 3 * i = 0\}\, j = j + 3;\ i = i + 2 \{i * j = 6\}$

31. ¿Por qué en la regla del **while** B **do**, la condición B debe ser verdadera al comienzo del bucle?

32. Considerar una función con dos argumentos que se usa en un programa. Explicar por qué el uso de alias puede ser un problema en este caso.

33. Demostrar la corrección parcial del siguiente fragmento de programa:

$$\begin{array}{rcl} sum := 0; & & j := 1; \\ while(j & \neq & c)do \\ begin & & \\ sum := sum + j & ; & j := j + 1; \\ end & & \\ \{sum & = & c * (c-1)/2\} \end{array}$$

34. Demostrar la corrección del siguiente triple: $\{a[i] \geq 0\}\ a[i] = a[i] + a[j]\{a[i] \geq a[j]\}$

35. Verificar el siguiente segmento de programa:
$$\begin{array}{rcl} \{n & \geq & 0\} \\ i & := & 1; \\ while\ i & \leq & n\ do \\ begin & & \\ a[i] & := & b[i]; \\ i & := & i+1; \\ end & & \\ \{\bigwedge_{i=1}^{n} (a[i] & = & b[i])\} \end{array}$$
Con el invariante: $\{\bigwedge_{j=1}^{i-1} a[j] = b[j]\}$

36. El siguiente fragmento de programa calcula $\sum_{i=1}^{n} i!$. Demostrar que es correcto con el invariante : $sum = \sum_{j=1}^{i-1} j! \wedge f = i!$
$$\begin{array}{l} i := 1; sum := 0; f := 1; \\ while\ i \neq n+1\ do \\ begin \\ sum := sum + f; \\ i := i+1; \\ f := f * i; \\ end \end{array}$$

37. Hallar la precondición $\{P\}$ que hace que el siguiente triple sea correcto: $\{P\}\ a[i] := 2*b\ \{j \leq i,\ k < i,\ a[i] + a[j-1] + a[k] > b\}$

38. Demostrar que para $n > 0$ el siguiente fragmento de programa termina.
$$\begin{array}{l} i := 1; f := 1; \\ \quad while\ i \neq n\ do \\ \quad begin \\ \quad\quad i := i+1; \\ \quad\quad f := f * i; \\ \quad end \end{array}$$

39. Hallar la precondición de la terna: $\{P\}\ a[i] := b\ \{a[j] = 2 * a[i]\}$

40. Para cada uno de los siguientes fragmentos de código, obtener la poscondición apropiada:

    (a) $\{i < 10\}\ i := 2 * i + 1$

    (b) $\{i > 0\}\ i := i - 1$

    (c) $\{i > j\}\ i := i + 1;\ j := j + 1$

    (d) $\{V\}\ i := 3;\ j := 2 * i$

41. Para cada uno de los siguientes fragmentos de código, obtener las precondiciones apropiadas.

    (a) $i := 3 * k \, \{i > 6\}$

    (b) $a := b * c \, \{a = 1\}$

    (c) b:= c – 2; a:= a/b;

42. Verificar el siguiente código. Indicar todas las reglas usadas.
    $\{y > 0\} \; xa := x + y;$ xb := x-y

43. Verificar el siguiente código, indicando todas las reglas usadas.
    $\{V\} \; if \; x < 0 \; then \; x := -x \{x \geq 0\}$

44. Verificar el siguiente segmento de programa, suponiendo el invariante: $\bigwedge_{k=1}^{i} max \geq a[k]$,

$$
\begin{array}{rcl}
max := a[1] & ; & i := 1; \\
while \; (i & \neq & n+1) \; do \\
begin & & \\
if & & (a[i] >= max) \; then \; max = a[i]; \\
i & := & i+1 \\
end & & \\
\{\bigwedge_{i=1}^{n} & & (max \geq a[i])\}
\end{array}
$$

45. Demostrar la corrección parcial del siguiente código:

$$
\begin{array}{l}
max := a[1]; i := 1; \\
\quad while(i < n) \; do \\
\quad\quad begin \\
\quad\quad\quad i :=i+1; \\
\quad\quad\quad\quad if(a[i] >= max) \; then \; max := a[i]; \\
\quad\quad end \\
\{\bigwedge_{i=1}^{n} (max \geq a[i]), \bigvee_{j=1}^{n} max = a[j])\}
\end{array}
$$

46. Demostrar la corrección parcial del siguiente código, suponiendo el invariante:
    $\bigwedge_{k=0,j=n-k}^{i,n-i} a[k] = b[j]$,

$$
\begin{array}{l}
i := 0; j := n; \\
while \; (i < n) \; do \\
\quad begin \\
\quad\quad i := i + 1; \\
\quad\quad j := j - 1; \\
\quad\quad a[i] := b[j] \\
\quad end \\
\{\bigwedge_{i=0}^{n} (a[i] == b[n - i])\}
\end{array}
$$

47. Demostrar la corrección parcial del siguiente código suponiendo el invariante:
$\bigwedge_{k=0}^{i-1} a[k] = s\,,\ s = \sum_{k=0}^{i-1} A[k]$

$$
\begin{array}{l}
i := 0; \\
s := 0; \\
while\ (i \leq n)\ do \\
\quad begin \\
\qquad s := s + a[i]; \\
\qquad a[i] := s; \\
\qquad i := i + 1; \\
\quad end
\end{array}
$$

$$\{\bigwedge_{i=0}^{n} a[i] = \sum_{k=0}^{n} A[k]\}$$

48. Dados $n \geq 0$, $i \leq n$, demostrar que el siguiente segmento de programa evalúa $\frac{n!}{(i!*(n-i)!)}$ dado el invariante $\{i > k \vee afact = i!\} \wedge \{n - i > k \vee bfact = (n-i)!\}$:

$$
\begin{array}{l}
k := 0; fact := 1; \\
while\ (k \neq n)\ do \\
\quad begin \\
k := k + 1; fact := fact * k; \\
\qquad if\ (k \leq i) \\
\qquad\quad then\ afact := fact; \\
\qquad if\ (k \leq n - i) \\
\qquad\quad then\ bfact := fact \\
\quad end \\
\quad bcof = \dfrac{fact}{(afact * bfact)}
\end{array}
$$

49. Demostrar la terminación del fragmento de programa dado en el problema 44 ¿Qué condición se debe imponer para realizar la demostración?

3.1.1. Solución

La Solución la puedes encontrar:

- Relación de Ejercicios Tema 1 en markdown
- Relación de Ejercicios Tema 1 en pdf

3.2. Relación 2

3.2.1. Relaciones 2.1

3.2.2. Solución

La Solución la puedes encontrar:

- Relación de Ejercicios Tema 1 en markdown

GII-ADE-M. Relación de problemas. Tema 2-1. 11/10/2024

Problemas de semáforos generales.

50. Sean los procesos $\mathtt{P}_1$, $\mathtt{P}_2$, y $\mathtt{P}_3$, cuyas secuencias de instrucciones son las que se muestran en el cuadro.

```
{variables globales}
Proceso P1;          Proceso P2;         Proceso P3;
  begin                begin               begin
    while true do       while true do       while true do
     begin                begin               begin
      a;                    d;                    g;
      b;                    e;                    h;
      c;                    f;                    i;
     end                  end                 end
  end                  end                 end
```

Se pide:resolver los siguientes problemas de sincronización, considerando que son independientes unos de otros, con semáforos:

(a)P_2 podrá pasar a ejecutar e sólo si P_1 ha ejecutado a o P_3 ha ejecutado g

(b)P_2 podrá pasar a ejecutar e sólo si P_1 ha ejecutado a y P_3 ha ejecutado g

(c)Sólo cuando P_1 haya ejecutado b, podrá pasar P_2 a ejecutar e y P_3 a ejecutar h

(d)Sincroniza los procesos de forma que las sentencias b en P_1, f en P_2, y h en P_3, sean ejecutadas como mucho por 2 procesos simultáneamente.

51. El cuadro que sigue nos muestra dos procesos concurrentes, P_1 y P_2, que comparten una variable global x y las restantes variables son locales a los procesos.
Se pide:

(a)Sincronizar los procesos para que P_1 use todos los valores x suministrados por P_2

(b)Sincronizar los procesos para que P_1 utilice un valor *sí y otro no* de la variable x, es decir, utilice los valores primero, tercero, quinto, etc. que vaya alcanzando dicha variable.

```
{variables globales}
proceso P1;                          proceso P2;
  var m: integer;                      var d: integer;
  begin                                begin
    while true do                        while true do
      begin                                begin
        m:= 2*x - n;                         d:= leer_teclado();
        print(m);                            x:= d - c*5;
      end                                  end
  end                                  end
```

52. Supongamos que estamos en una discoteca y resulta que está estropeado el servicio de chicas y todos tienen que compartir el de chicos. Se pretende establecer un protocolo de entrada al servicio usando semáforos que asegure siempre el cumplimiento de las siguientes restricciones:
    - Chicas: sólo puede estar 1 dentro del servicio
    - Chicos: pueden entrar más de 1, pero como máximo se admitirán a 5 dentro del servicio

- Versión machista del protocolo: los chicos tienen preferencia sobre las chicas Esto quiere decir que si una chica está esperando entrar al servicio y llega un chico, este puede pasar y ella sigue esperando. Incluso si el chico que ha llegado no pudiera entrar inmediatamente porque ya hay 5 chicos dentro del servicio, sin embargo, pasará antes que la chica cuando salga algún chico del servicio.
- Versión feminista del protocolo: las chicas tienen preferencia sobre los chicos. Esto quiere decir que si un chico está esperando y llega una chica, ésta debe pasar antes. Incluso si la chica que ha llegado no puede entrar inmediatamente al servicio porque ya hay una chica dentro, pasará antes que el chico cuando salga la chica que está dentro.

Se pide: implementar las 2 versiones del protocolo anterior utilizando semáforos POSIX. Ayuda sobre la sintaxis de las operaciones de los semáforos (*no nombrados*) de POSIX 1003:

inicialización : `int sem_init(sem_t* semaforo, int pcompartido, unsigned int contador)`

destrucción : `int sem_destroy(sem_t* semaforo)`

sincronización-espera : `int sem_wait(sem_t* semaforo)`

sincronización-señala : `int sem_post(sem_t* semaforo)`

Notas:

(a) El valor inicial del semáforo se le asigna a *contador*. Si *pcompartido* es distinto de cero, entonces el semáforo puede ser utilizado por hilos que residen en procesos diferentes; si no, sólo puede ser utilizado por hilos dentro del espacio de direcciones de un único proceso.

(b) Para que se pueda destruir, el semáforo ha debido ser explícitamente inicializado mediante la operación `sem_init(...)`. La operación anterior no debe ser utilizada con semáforos *nombrados*.

(c) Los hilos llamarán a la función int `sem_wait(sem_t* semaforo)`, pasándole un identificador de semáforo inicializado con el valor '0', para sincronizarse con una condición. Si el valor del semáforo fuera distinto de '0', entonces el valor de s se decrementa en una unidad y no bloquea.

(d) La operación `int sem_post(sem_t* semaforo)` sirve para señalar a los hilos bloqueadas en un semáforo y hacer que uno pase a estar preparado para ejecutarse. Si no hay hilos bloqueados en este semáforo, entonces la ejecución de esta operación simplemente incrementa el valor de la variable protegida (s) del semáforo. Hay que tener en cuenta que no existe ningún orden de desbloqueo definido si hay varios hilos esperando en la cola asociada a un semáforo, ya que la implementación a nivel de sistema de la operación anterior supone que el planificador puede escoger para desbloquear a cualquiera de los hilos que esperan. En particular, podría darse el siguiente escenario, otro hilo ejecutándose puede decrementar el valor del semáforo antes que cualquier hilo que vaya a ser desbloqueado como resultado de `sem_post(...)` lo pueda hacer y, posteriormente, se volvería a bloquear el hilo despertado.

Problemas de monitores.

53. Aunque un monitor garantiza la exclusión mutua, los procedimientos tienen que ser reentrantes. Explicar por qué.

54. Se consideran dos tipos de recursos accesibles por varios procesos concurrentes (denominamos a los recursos como recursos de tipo 1 y de tipo 2). Existen N_1 ejemplares de recursos de tipo 1 y N_2 ejemplares de recursos de tipo 2. Para la gestión de estos ejemplares, queremos diseñar

un monitor (con semántica SU) que exporta un procedimiento (`pedir_recurso`), para pedir un ejemplar de uno de los dos tipos de recursos. Este procedimiento incluye un parámetro entero (tipo), que valdrá 1 ó 2 indicando el tipo del ejemplar que se desea usar, así mismo, el monitor incorpora otro procedimiento (`liberar_recurso`) para indicar que se deja de usar un ejemplar de un recurso previamente solicitado (este procedimiento también admite un entero que puede valer 1 ó 2, según el tipo de ejemplar que se quiera liberar). En ningún momento puede haber un ejemplar de un tipo de recurso en uso por más de un proceso.

En este contexto, responde a estas cuestiones:

(a) Implementa el monitor con los dos procedimientos citados, suponiendo que N_1 y N_2 son dos constantes arbitrarias, mayores que cero.

(b) El uso de este monitor puede dar lugar a interbloqueo. Esto ocurre cuando más de un proceso, en algún punto en su código, tiene la necesidad de usar dos ejemplares de recursos de distinto tipo a la vez. Describe la secuencia de peticiones (llamadas al procedimiento correspondiente del monitor) que da lugar a interbloqueo.

(c) Una posible solución al problema anterior es obligar a que si un proceso necesita dos recursos de distinto tipo a la vez, deba de llamar a `pedir_recurso`, dando un parámetro con valor 0, para indicar que necesita los dos ejemplares. En esta solución, cuando un ejemplar quede libre, se dará prioridad a los posibles procesos esperando usar dos ejemplares, frente a los que esperan usar solo uno de ellos.

55. Escribir una solución al problema de lectores-escritores con monitores:

(a) Con *prioridad a los lectores*: quiere decir que, si en un momento puede acceder al recurso, tanto un lector como un escritor, se da paso preferentemente al lector.

(b) Con *prioridad a los escritores*: quiere decir que, si en un momento puede acceder tanto un lector como un escritor, se da paso preferentemente al escritor.

(c) Con *prioridades iguales*: en este caso, los procesos acceden al recurso estrictamente en orden de llegada, lo cual implica, en particular, que si hay lectores leyendo y un escritor esperando, los lectores que intenten acceder después del escritor no podrán hacerlo hasta que no lo haga dicho escritor.

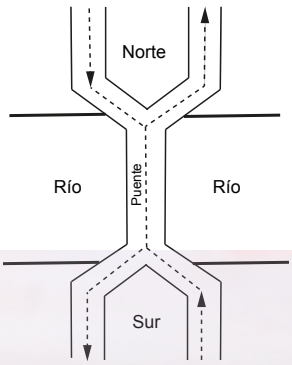


Figura 1: Problema de exclusión mutua en el acceso de coches desde 2 sentidos opuestos a un puente de un solo carril.

56. Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río (ver Fig.1). Sólo existe un carril sobre dicho puente. Por lo tanto, en un momento dado, el puente solo puede ser cruzado por uno o más coches en la misma dirección (pero no en direcciones opuestas).

    (a) Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando el puente en ese momento.

```
Monitor Puente
var ... ;
procedure EntrarCocheDelNorte()
begin
...
end
procedure SalirCocheDelNorte()
begin
....
end
procedure EntrarCocheDelSur()
begin
....
end
procedure SalirCocheDelSur()
begin
...
end
{ Inicializacion }
begin
....
end
```

    (b) Mejorar el monitor anterior, de forma que la dirección del trafico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.

57. Una tribu de antropófagos comparte una olla en la que caben `M` misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros M misioneros. Para solucionar la sincronización usamos un monitor llamado `Olla`, que se puede usar así:

```
monitor Olla ;
....
begin
....
end;
proceso ProcSalvaje[ i:1..N ] ;
begin
while true do begin
   Olla.Servirse_1_misionero();
   Comer(); { es un retraso aleatorio }
  end
end;
proceso ProcCocinero ;
begin
```

```
while true do begin
   Olla.Dormir();
   Olla.Rellenar_Olla();
  end
end;
```

Se pide: diseñar el código del monitor `Olla` para que se satisfaga la sincronización requerida en el enunciado del problema, teniendo en cuenta que:

- La solución propuesta no debe producir interbloqueos.
- Los salvajes podrán comer siempre que haya comida en la olla.
- Sólo se ha de *despertar* al proceso cocinero cuando la olla esté vacía.

58. Una cuenta de ahorros es compartida por varias personas (procesos). Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo. Queremos usar un monitor para resolver el problema.

    El monitor debe tener 2 procedimientos: `depositar(c)` y `retirar(c)`. Suponer que los argumentos de las 2 operaciones son siempre positivos, e indican las cantidades a depositar o retirar. El monitor usará la semántica señalar y espera urgente (SU). Se deben de escribir varias versiones de la solución, según las variaciones de los requerimientos que se describen a continuación:

    (a)Todo proceso puede retirar fondos mientras la cantidad solicitada `c` sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad `c` mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente (como consecuencia de que otros procesos depositen fondos en la cuenta) para que se pueda atender la petición.

    Hacer dos versiones del monitor: (a.1) colas normales FIFO sin prioridad y (a.2) con colas de prioridad.

    (b)El reintegro de fondos a los clientes se hace únicamente según el orden de llegada, si hay más de un cliente esperando, sólo el primero que llegó puede optar a retirar la cantidad que desea, mientras esto no sea posible, esperarán todos los clientes, independientemente de cuanto quieran retirar los demás. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades, entonces si llega otro cliente debe esperarse, incluso si quiere retirar 200 unidades. De nuevo, resolverlo utilizando dos versiones: (b.1) colas normales (FIFO) sin prioridad y (b.2) con colas de prioridad.

59. Los procesos $P_1, P_2, \ldots, P_n$ comparten un único recurso R, pero sólo un proceso puede utilizarlo cada vez. Un proceso P_i puede comenzar a utilizar R si está libre; en caso contrario, el proceso debe esperar a que el recurso sea liberado por otro proceso. Si hay varios procesos esperando a que quede libre R, se concederá al proceso que tenga mayor prioridad. La regla de prioridad de los procesos es la siguiente: el proceso P_i tiene prioridad i, (con $1 \leq i \leq n$), donde los números menores implican mayor prioridad (es decir, si $i < j$, entonces P_i pasa por delante de P_j). Implementar un monitor que implemente los procedimientos `Pedir(...)` y `Liberar()` con un monitor que garantice la exclusión mutua y el acceso prioritario del procesos al recurso R.

60. El siguiente monitor (`Barrera2`) proporciona un único procedimiento de nombre `entrada()`, que provoca que el primer proceso que lo llama sea suspendido y el segundo que lo llama

despierte al primero que lo llamó (a continuación ambos continúan), y así actúa cíclicamente. Obtener una implementación de este monitor usando semáforos.

```
Monitor Barrera2 ;
  var n : integer; { num. de proc. que han llegado desde el signal }
  s : condicion ; { cola donde espera el segundo }
  procedure entrada() ;
    begin
     n := n+1 ; { ha llegado un proceso mas }
     if n < 2 then { si es el primero: }
       s.wait() { esperar al segundo }
       else begin { si es el segundo: }
         n := 0; { inicializa el contador }
         s.signal() { despertar al primero }
       end
    end
  { Inicializacion }
  begin
   n := 0 ;
  end
```

61. Este es un ejemplo clásico que ilustra el problema del interbloqueo, y aparece en la literatura informática con el nombre de el problema de los *filósofos-comensales*. Se puede enunciar como se indica a continuación: sentados a una mesa están cinco filósofos, la actividad de cada filósofo es un ciclo sin fin de las operaciones de pensar y comer; entre cada dos filósofos hay un tenedor y para poder comer, un filósofo necesita obligatoriamente dos tenedores: el de su derecha y el de su izquierda. Se han definido cinco procesos concurrentes, cada uno de ellos describe la actividad de un filósofo. Los procesos usan un monitor, llamado `MonFilo`. Antes de comer cada filósofo debe disponer de su tenedor de la derecha y el de la izquierda, y cuando termina la actividad de comer, libera ambos tenedores. El filósofo i alude al tenedor de su derecha como el número `i`, y al de su izquierda como el número `i + 1 mod 5`. El monitor `MonFilo` exportará dos procedimentos: `coge_tenedor(num_tenedor,num_proceso)` y `libera_tenedor(num_tenedor)` para indicar que un proceso filósofo desea coger un tenedor determinado. El código del programa (sin incluir la implementación del monitor) es el siguiente:

```
monitor MonFilo ;
  ....
  procedure coge_tenedor( num_ten, num_proc : integer );
  ....
  procedure libera_tenedor( num_ten : integer );
  ....
  begin
    ....
  end
  proceso Filosofo[ i: 0..4 ] ;
   begin
    while true do begin
      MonFilo.coge_tenedor(i,i); { argumento 1=codigo tenedor }
      MonFilo.coge_tenedor(i+1 mod 5,i); { argumento 2=numero de proceso }
      comer();
      MonFilo.libera_tenedor(i);
      MonFilo.libera_tenedor(i+1 mod 5);
      pensar();
```

```
    end
  end
```

Con este interfaz para el monitor, responde a las siguientes cuestiones:

(a)Diseña una solución para el monitor `MonFilo`

(b)Describe la situación de interbloqueo que puede ocurrir con la solución que has escrito antes.

(c)Diseña una nueva solución, en la cual se evite el interbloqueo descrito, para ello, esta solución no debe permitir que haya más de cuatro filósofos simultáneamente intentado coger su primer tenedor.

3.2.3. Relaciones 2.2

GII-ADE-M. Relación de Problemas. Tema 2-2. 25/10/24

62. Un algoritmo para el cual sólo pudiésemos demostrar que cumple las 4 condiciones de Dijkstra, ¿qué tipo de propiedades concurrentes satisfacería: (a) seguridad, (b) vivacidad, (c) equidad? Justificar las respuestas.

63. En algunas aplicaciones es necesario tener exclusión mutua entre procesos con la particularidad de que puede haber como mucho `n` procesos en una sección crítica, con `n` arbitrario y fijo, pero no necesariamente igual a la unidad, sino posiblemente mayor. Diseña una solución para este problema basada en el uso de espera ocupada y cerrojos. Estructura dicha solución como un par de subrutinas (usando una misma estructura de datos en memoria compartida), una para el protocolo de entrada y otro el de salida, e incluye el pseudocódigo de las mismas.

64. ¿Podría pensarse que una posible solución al problema de la exclusión mutua, sería el siguiente algoritmo que no necesita compartir una variable `turno` entre los 2 procesos? Demostrar (sí o no) se satisfacen las siguientes propiedades:

    (a) ¿la exclusión mutua? (propiedad de seguridad)

    (b) ¿la ausencia de interbloqueo? (propiedad de alcanzabilidad)

```
//variables compartidas y valores iniciales
var b0 : boolean := false; //true si P0 quiere acceder o esta en SC
    b1 : boolean := false ; //true si P1 quiere acceder o esta en SC
```

```
Process P0 ;
  begin
        while true do begin
        //protocolo de entrada:
        b0 := true ; //indica quiere
           entrar
          while b1 do begin //si el
             otro tambien:
                b0 := false ;//cede
                   temporalmente }
                while b1 do begin end
                     //espera
                 b0 := true ; //
                     vuelve a cerrar
                     paso
                end
          //seccion critica ....
          //protocolo de salida
                b0 := false ;
          //resto sentencias ....
          end
        end
```

```
Process P1 ;
   begin
        while true do begin 3
        //protocolo de entrada:
        b1 := true ; //indica quiere
           entrar
        while b0 do begin //si el
           otro tambien:
          b1 := false ; //cede
             temporalmente
          while b0 do begin end //
             espera
                b1 := true ;//vuelve
                   a cerrar paso
          end
        //seccion critica ....
        //protocolo de salida
        b1 := false ;
        //resto sentencias ....
        end
  end
```

65. Al siguiente algoritmo se le conoce como solución de Hyman al problema de la exclusión mutua (fue publicado en una revista de impacto en 1966[1]). ¿Es correcta dicha solución?

[1] Harris Hyman, "Comments on a problem in concurrent programming control", Communications of the ACM, v.9 n.1, p.45, 1966

```
//variables compartidas y valores iniciales
var c0 : integer := 1 ; c1 : integer := 1 ; turno : integer := 1 ;
```

```
process P0 ;
  begin
        while true do begin
         c0 := 0 ;
         while turno != 0 do begin
          while c1 = 0 do begin end
         turno := 0 ;
         end
         //seccion critica
         c0 := 1 ;
         //resto sentencias }
  end
end
```

```
process P1 ;
  begin
        while true do begin
        c1 := 0 ;
        while turno != 1 do begin
         while c0 = 0 do begin end
        turno := 1 ;
        end
        //seccion critica
        c1 := 1 ;
        //resto sentencias
 end
end
```

66. Supongamos el algoritmo de exclusión mutua que expresamos a continuación. Tenemos los procesos: $0, 1, \ldots$ n $- 1$. Cada proceso i tiene una variable $s[i]$, inicializada a 0, que puede tomar los valores 0/1. El proceso i puede entrar en la sección crítica si:

$$s[i] \; != \; s[i-1] \;\; para\, i > 0;$$
$$s[0] \;\; = \;\; s[n-1] \;\; para\, i = 0;$$

Tras ejecutar su sección crítica, el proceso i deberá hacer:

$$s[i] \;\; = \;\; s[i-1] \;\; para\, i > 0;$$
$$s[0] \;\; = \;\; (s[0]+1)\, mod\, 2 \;\; para\, i == 0;$$

67. Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno en que ha de ser atendido el cliente), los números de los tickets se representan por dos variables n1 y n2 que valen inicialmente 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener 2 números iguales se procesa primero el proceso número 1.

    (a) Demostrar que se verifica la ausencia de interbloqueo (propiedad de *alcanzabilidad* de la sección crítica), la ausencia de inanición (propiedad de *vivacidad*) y la exclusión mutua (una propiedad de *seguridad*).

    (b) Demostrar que las asignaciones `n1:=1` y `n2:=1` son ambas necesarias.

```
//variables compartidas y valores iniciales
var n1 : integer := 0 ; n2 : integer := 0 ;
```

```
process P1 ;
  begin
        while true do begin
        n1 := 1 ; // E1.1
        n1 := n2+1 ; // L1.1 ; E2.1
        while n2 != 0 and // L2.1
          n2 < n1 do begin end;//L3.1
        // seccion critica } { SC.1 }
          n1 := 0 ; // E3.1
        // resto sentencias  { RS.1 }
        end
  end
```

```
process P2 ;
  begin
        while true do begin
        n2 := 1 ; // E1.2
        n2 := n1+1 ; // L1.2 ; E2.2
        while n1 != 0 and //L2.2
          n1<= n2 do begin end;//L3.2
        // seccion critica  { SC.2 }
          n2 := 0 ; // E3.2
        // resto sentencias  { RS.2 }
  end
end
```

68. El siguiente programa es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución: si es correcta, entonces probarlo. Si no fuese correcta, escribir escenarios que demuestren que la solución es incorrecta.

```
//variables compartidas y valores iniciales
var c0 : integer := 1; c1 : integer := 1 ;
```

```
process P0 ;
      begin
        while true do begin
              repeat
                c0 := 1-c1 ;
              until c1 != 0 ;
              // seccion critica
              c0 := 1 ;
              //resto sentencias }
              end
      end
```

```
process P1 ;
      begin
        while true do begin
              repeat
                c1 := 1-c0 ;
              until c0 != 0 ;
              // seccion critica
              c1 := 1 ;
              // resto sentencias
        end
      end
```

69. Con respecto al algoritmo de Peterson para N–procesos: ¿sería posible que llegaran 2 procesos a la etapa N-2, 0 procesos a la etapa N-3 y en todas las etapas anteriores existiera al menos 1 proceso? Justificar la respuesta.

70. En el *algoritmo de Peterson* para N procesos y considerando cualquier escenario de ejecución de dicho algoritmo, el número máximo de turnos que tiene que esperar cualquier proceso para entrar en sección crítica es N-1 turnos.

71. Con respecto al algoritmo de la siguiente figura (algoritmo de Dijkstra para N procesos), demostrar la falsedad de la siguiente proposición: *si un conjunto de procesos está intentando pasar simultáneamente el primer bucle* (5)*, y el proceso que tiene el turno está* pasivo*, entonces siempre conseguirá entrar primero en sección crítica el proceso de dicho grupo que consiga asignar la variable* turno *en último lugar*.

```
var turn :0..N-1;
flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
Process P(i);
begin
(1)   <resto instrucciones>
(2)   repeat
(3)      flag[i] := solicitando;
(4)      j := turn;
(5)      while (turno !=i) do
(6)         if (flag[turno] =  pasivo) then
(7)            turn:=i;
(8)         endif;
(9)    enddo;
(10)    flag[i] := enSC;
(11)    j := 0;
(12)    while ( j < N )and
        ( (j = i)or flag[j]!= enSC )do
              j := j + 1;
(14)    enddo;
```

```
(15) until (j >= N);
<<seccion critica>>
(16) flag[i] := pasivo;
end
```

72. El algoritmo de la figura siguiente (algoritmo de Knuth para N-procesos) resuelve el problema de la exclusión mutua para N-procesos, para lo cual utiliza N variables booleanas, `flag: array[ 0..N - 1 ] of ( solicitando, enSC, pasivo );` una variable `turn: 0..n - 1` y la variable local `j`.

    (a) Demostrar que el algoritmo de Knuth verifica todas las propiedades exigibles a un programa concurrente, incluyendo la de equidad.

    (b) Escribir un escenario en el que 2 procesos consiguen pasar el bucle de la instrucción (5), suponiendo que el turno lo tiene inicialmente el proceso `p(0)`.

```
var flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
turn := 0;
Process P(i);
begin
(1)   <resto instrucciones>
(2)   repeat
(3)      flag[i] := solicitando;
(4)      j := turn;
(5)      while (j !=i) do
(6)         if flag[j] !=  pasivo then
(7)            j := turn
(8)         else j := ( j - 1 ) mod N;
(9)         endif;
(10)    enddo;
(11)    flag[i] := enSC;
(12)    j := 0;
(13)    while ( j < N )and
        ( (j = i)or flag[j]!= enSC )do
              j := j + 1;
(14)    enddo;
(15) until (j >= N);
(16) turn := i;
<<seccion critica>>
(17) j := ( turn + 1 ) mod N;
(18) turn := j;
(19) flag[i] := pasivo;
end
```

73. Si en el algoritmo de Dijkstra se cambia la instrucción (6) por esta otra: `if flag[turno] !=SC`, entonces el algoritmo dejaría de ser correcto. Indicar qué propiedad(es) de corrección fallaría(n) y justificar por qué.

74. Si en el algoritmo de Knuth se hacen las siguientes sustituciones:

    - La condición de la instrucción `until` de (15) por la condición: `(j >= N)and( turno=i or flag[turno]= pasivo)`

- Se inserta el siguiente bucle después de la instrucción (17):

```
while (j !=turn )and( flag[j]=pasivo) do
(19)     j := j + 1;
(20) enddo;
```

(a) Verificar las propiedades de exclusión mutua, alcanzabilidad de la sección crítica, vivacidad y equidad del algoritmo.

(b) Calcular el número de turnos máximo que puede llegar a tener que esperar un proceso que quiera entrar en su sección crítica con el algoritmo anterior.

75. Demostrar que las instrucciones (13)-(16) del algoritmo de exclusión mutua distribuido de Ricart-Agrawala no necesitan ser protegidas dentro de la sección crítica definida por las operaciones wait(), signal() del semáforo "s".

```
ns: 0..+INF;
mns: 0..INF;
numrepesperadas:0..n-1;
intentaSC: boolean;
prioridad: boolean;
repretrasadas: array[1..N] of boolean;
```

```
Process Pi;
begin
(1) wait(s);
(2) intentaSC:= TRUE;
(3) ns:= mns +1;
(4) numrepesperadas:=n-1;
(5) signal(s);
(6) for j:= 1 to n do
(7) if j <> i then
(8) send(j, pet, ns, i);
(9) wait(sinc);
 <<seccion critica>>
(10) wait(s);
(11) intentaSC:= FALSE;
(12) signal(s);
(13) for j:=1 to n do
(14) if repretrasadas[j] then begin
(15)    repretrasadas[j]:= FALSE;
(16)    send(j, rep);
(17)    end;
end
```

```
Process Pet(i);
begin
(1)     receive(pet, k, j);
(2)     wait(s);
(3)     mns:= max(mns, k);
(4)     prioridad:= (intentaSC) and (
   k>ns or (k=ns and i<j));
(5)     if prioridad then
   repretrasadas[j]:= TRUE
        else send(j, rep);
(6)     signal(s);
end

Process Rep(i);
begin
(1)     receive(rep);
(2) numrepesperadas:= numrepesperadas
    - 1;
(3) if numrepesperadas= 0 then signal
   (sinc);
end
```

76. Suponer que el algoritmo de Suzuki-Kasami para resolver el problema de la exclusión mutua distribuida para n-procesos se modifica como aparece en la siguiente figura. Explicar por qué dejaría de ser correcto el algoritmo, relacionándolo con cada una de las propiedades de corrección que se demuestran para el algoritmo original.

```
token_presente:boolean:=FALSE;
enSC:boolean:= FALSE;
peticion:array[1..n] of boolean:= FALSE;
//En el algoritmo original ->peticion: array[1..N] of 0..+INF
//ademas se declara otro array-> token: array[1..N] of 0..+INF
```

```
Process P(i);
begin
(0) wait(s);
(1) if NOT token_presente then begin
(2) broadcast(pet, i);
(3) receive(acceso);
(4) token_presente:= TRUE;
(5) end;
(6) enSC:= TRUE;
(7) signal(s);
    <<seccion critica>>
(8) enSC:=FALSE;
(9) wait(s);
(10) for j:= i+1 to n, 1 to i-1 do
(11) if peticion[j] and
   token_presente       then begin
(12) token_presente:= FALSE;
(13) send(j, acceso);
(14) peticion[j]:= FALSE;
(15)end;
(16) signal(s);
end
```

```
Process Pet(i);
begin
(1)      receive(pet, j);
(2)      wait(s);
(3)      peticion[j]:= TRUE;
(4)      if token_presente and NOT
   enSC then
<<< repetir (10)- (16) >>>
end
```

3.3. Relacion 3

GII-ADE-M. Relación de problemas. Tema 3. 15/11/2024

77. En un sistema distribuido, 6 procesos clientes necesitan sincronizarse de forma específica para realizar cierta tarea, de forma que dicha tarea sólo podrá ser realizada cuando tres procesos estén preparados para realizarla. Para ello, envían peticiones a un proceso controlador del recurso y esperan respuesta para poder realizar la tarea específica. El proceso controlador se encarga de asegurar la sincronización adecuada. Para ello, recibe y cuenta las peticiones que le llegan de los procesos, las dos primeras no son respondidas y producen la suspensión del proceso que envía la petición (debido a que se bloquea esperando respuesta) pero la tercera petición produce el desbloqueo de los tres procesos pendientes de respuesta. A continuación, una vez desbloqueados los tres procesos que han pedido (al recibir respuesta), inicializa la cuenta y procede cíclicamente de la misma forma sobre otras peticiones. El código de los procesos clientes aparece aquí abajo. Los clientes usan envío asíncrono seguro para realizar su petición, y esperan con una recepción síncrona antes de realizar la tarea:

```
process Cliente[ i : 0..5 ] ;
  begin
    while true do begin
      send( peticion, Controlador );
      receive( permiso, Controlador )
          ;
      Realiza_tarea_grupal( );
    end
  end
```

```
process Controlador ;
  begin
    while true do begin
      ...
    end
  end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo `select for i=... to ...` para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice `i`.

78. En un sistema distribuido, 3 procesos productores producen continuamente valores enteros y los envían a un proceso buffer que los almacena temporalmente en un array local de 4 celdas enteras para ir enviándoselos a un proceso consumidor. A su vez, el proceso buffer realiza lo siguiente, sirviendo de forma equitativa al resto de procesos:

    (a) Envía enteros al proceso consumidor siempre que su array local tenga al menos dos elementos disponibles

    (b) Acepta envíos de los productores mientras el array no esté lleno, pero no acepta que cualquier productor pueda escribir dos veces consecutivas en el búfer

    El código de los procesos productor y consumidor es el siguiente, asumiendo que se usan operaciones síncronas:

```
process Productor[ i : 0..2 ] ;
  var dato : integer ;
  begin
    while true do begin
      dato := Producir();
      send( dato, Buffer );
    end
end
```

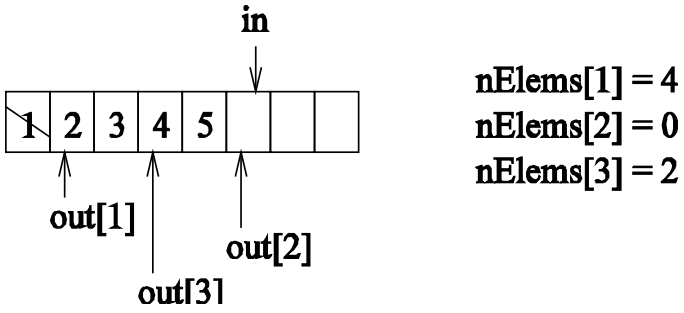
```
process Consumidor ;
  begin
   while true do begin
     receive ( dato, Buffer );
     Consumir( dato );
   end
end
```

Describir en pseudocódigo el comportamiento del proceso `Buffer`, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos:

```
process Buffer ;
  begin
    while true do begin
      ...
    end
  end
```

79. Suponer un proceso productor y 3 procesos consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el proceso productor debe ser retirado por todos los 3 procesos consumidores para ser eliminado del buffer. Cada consumidor retirará los datos del buffer en el mismo orden en el que son depositados, aunque los diferentes consumidores pueden ir retirando los elementos a ritmo diferente unos de otros. Por ejemplo, mientras un consumidor ha retirado los elementos 1, 2 y 3, otro consumidor puede haber retirado solamente el elemento 1. De esta forma, el consumidor más rápido podría retirar hasta B elementos más que el consumidor más lento. Describir en pseudocódigo el comportamiento de un proceso que implemente el buffer de acuerdo con el esquema de interacción descrito usando una construcción de espera selectiva, así como el del proceso productor y de los procesos consumidores. Comenzar identificando qué información es necesario representar, para después resolver las cuestiones de sincronización.

    Una posible implementación del buffer mantendría, para cada proceso consumidor, el puntero de salida y el número de elementos que quedan en el buffer por consumir:



80. Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros `M` misioneros.

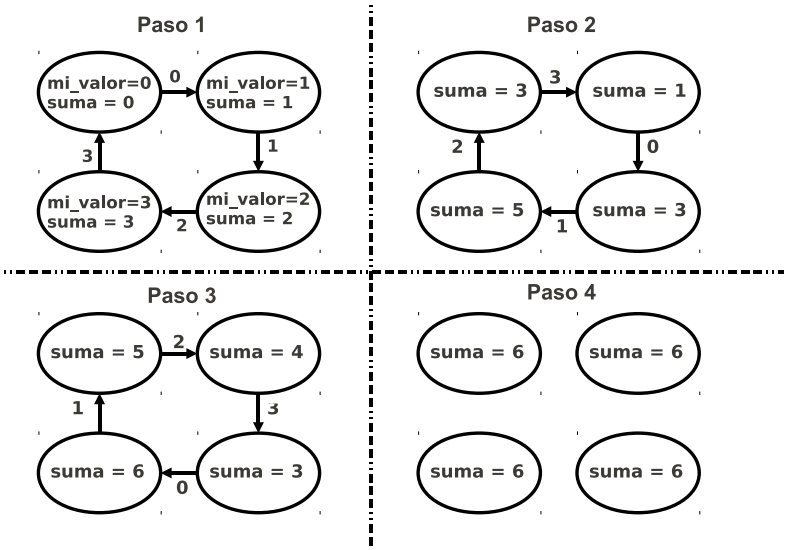
```
process Salvaje[ i : 0..2 ] ;
 begin
  var peticion : integer := ... ;
  begin
   while true do begin
   // esperar a servirse un misionero
       :
      s_send( peticion, Olla );
   // comer:
      Comer();
end
end
```

```
process Cocinero ;
 begin
   while true do begin
   // dormir esperando solicitud para
        llenar: }
           ......
   // confirmar que se ha rellenado
      la olla }
            ......
   end
 end
```

Implementar los procesos salvajes y cocinero usando paso de mensajes, usando un proceso olla que incluye una construcción de espera selectiva que sirve peticiones de los salvajes y el cocinero para mantener la sincronización requerida, teniendo en cuenta que:

- La solución no debe producir interbloqueo
- Los salvajes podrán comer siempre que haya comida en la olla,
- Solamente se despertará al cocinero cuando la olla esté vacía.

81. Considerar un conjunto de `N` procesos, `P[i], (i = 0, ..., N - 1)` que se pasan mensajes cada uno al siguiente (y el primero al último), en forma de anillo. Cada proceso tiene un valor local almacenado en su variable local `mi_valor`. Deseamos calcular la suma de los valores locales almacenados por los procesos de acuerdo con el algoritmo que se expone a continuación.



Los procesos realizan una serie de iteraciones para hacer circular sus valores locales por el anillo. En la primera iteración, cada proceso envía su valor local al siguiente proceso del anillo, al mismo tiempo que recibe del proceso anterior el valor local de éste. A continuación acumula la suma de su valor local y el recibido desde el proceso anterior. En las siguientes iteraciones, cada proceso envía al siguiente proceso siguiente el valor recibido en la anterior iteración, al mismo tiempo que recibe del proceso anterior un nuevo valor. Después acumula la suma. Tras un total de `N - 1` iteraciones, cada proceso conocerá la suma de todos los valores locales de los procesos. Dar una descripción en pseudocódigo de los procesos siguiendo un estilo SPMD y usando operaciones de envío y recepción síncronas:

```
process P[ i : 0..N-1 ] ;
  var mi_valor : integer := ... ; //valor arbitrario (== i en la figura,
     por ejemplo)
          suma : integer := mi_valor ; // suma inicializada a mi_valor
  begin
    for j := 0 to N-1 do begin
       ...
    end
 end
```

82. Considerar un estanco en el que hay tres fumadores y un estanquero. Cada fumador continuamente lía un cigarro y se lo fuma. Para liar un cigarro, el fumador necesita tres ingredientes: tabaco, papel y cerillas. Uno de los fumadores tiene solamente papel, otro tiene solamente tabaco, y el otro tiene solamente cerillas. El estanquero tiene una cantidad infinita de los tres ingredientes.

    - El estanquero coloca aleatoriamente dos ingredientes diferentes de los tres que se necesitan para hacer un cigarro, desbloquea al fumador que tiene el tercer ingrediente y después se bloquea. El fumador seleccionado, se puede obtener fácilmente mediante una función `genera_ingredientes` que devuelve el índice (0,1, ó 2) del fumador escogido.
    - El fumador desbloqueado toma los dos ingredientes del mostrador, desbloqueando al estanquero, lía un cigarro y fuma durante un tiempo.
    - El estanquero, una vez desbloqueado, vuelve a poner dos ingredientes aleatorios en el mostrador, y se repite el ciclo.

    Describir una solución distribuida que use envío asíncrono seguro y recepción síncrona, para este problema usando un proceso Estanquero y tres procesos fumadores `Fumador(i)` (con `i=0,1` y `2`).

```
process Estanquero ;
 begin
  while true do begin
   ....
  end
 end
```

```
process Fumador[ i : 0..2 ] ;
 begin
  while true do begin
   ....
  end
 end
```

83. En un sistema distribuido, un gran número de procesos clientes usa frecuentemente un determinado recurso y se desea que puedan usarlo simultáneamente el máximo número de procesos. Para ello, los clientes envían peticiones a un proceso controlador para usar el recurso y esperan respuesta para poder usarlo (véase el código de los procesos clientes). Cuando un cliente termina de usar el recurso, envía una solicitud para dejar de usarlo y espera respuesta del Controlador. El proceso controlador se encarga de asegurar la sincronización adecuada imponiendo una única restricción por razones supersticiosas: nunca habrá 13 procesos exactamente usando

el recurso al mismo tiempo.

```
process Cli[ i : 0....n ] ;
 var pet_usar : integer := +1 ;
  pet_liberar : integer := -1 ;
      permiso : integer := ... ;
 begin
   while true do begin
     send( pet_usar, Controlador );
     receive( permiso, Controlador
        );
     Usar_recurso( );
     send( pet_liberar, Controlador
         );
     receive( permiso, Controlador
        );
   end
 end
```

```
process Controlador ;
 begin
   while true do begin
    select
        ...
    end
 end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo `select for i=... to ...` para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice `i`.

84. En un sistema distribuido, tres procesos `Productor` se comunican con un proceso `Impresor` que se encarga de ir imprimiendo en pantalla una cadena con los datos generados por los procesos productores. Cada proceso productor (`Productor[i]` con `i = 0, 1, 2`) genera continuamente el correspondiente entero `i`, y lo envía al proceso Impresor.

    El proceso Impresor se encarga de ir recibiendo los datos generados por los productores y los imprime por pantalla (usando el procedimiento `imprime(entero)`) generando una cadena dígitos en la salida. No obstante, los procesos se han de sincronizar adecuadamente para que la impresión por pantalla cumpla las siguientes restricciones:

    - Los dígitos 0 y 1 deben aceptarse por el impresor de forma alterna. Es decir, si se acepta un 0 no podrá volver a aceptarse un 0 hasta que se haya aceptado un 1, y viceversa, si se acepta un 1 no podrá volver a aceptarse un 1 hasta que se haya aceptado un 0.
    - El número total de dígitos 0 o 1 aceptados en un instante no puede superar el doble de número de digitos 2 ya aceptados en dicho instante.

    Cuando un productor envía un digito que no se puede aceptar por el imprersor, el productor quedará bloqueado esperando completar el `s_send`. El pseudocódigo de los procesos productores (`Productor`) se muestra a continuación , asumiendo que se usan operaciones bloqueantes no buferizadas (síncronas):

```
process Productor[ i : 0,1,2 ]
  while true do begin
    s_send( i, Impresor ) ;
  end
```

```
process Impresor
  var
       .....
    begin
      while true do begin
       select
        .....
      end
   end
end
```

Escribir en pseudocódigo el código del proceso `Impresor`, utilizando para ello un bucle infinito con una orden de espera selectiva `select` que permita implementar la sincronización requerida entre los procesos, según el esquema anterior.

85. En un sistema distribuido hay un vector de n procesos iguales que envían con `send` (en un bucle infinito) valores enteros a un proceso receptor, que los imprime. Si en algún momento no hay ningún mensaje pendiente de recibir en el receptor, este proceso debe de imprimir "no hay mensajes, duermo"; después de bloquearse durante 10 segundos (con `sleep_for(10)`), antes de volver a comprobar si hay mensajes (esto podría hacerse para ahorrar energía, ya que el procesamiento de mensajes se hace en ráfagas separadas por 10 segundos). Este problema no se puede solucionar usando `receive` o `i_receive`. Indica a que se debe esto. Sin embargo, sí se puede hacer con select. Diseña una solución a este problema con select:

```
process Emisor[ i : 1..n ]
  var dato : integer ;
  begin
   while true do begin
     dato := Producir() ;
     send( dato, Receptor );
   end
end
```

```
process Receptor()
  var dato : integer ;
  begin
   while true do
       ......
 end
```

86. En un sistema tenemos N procesos emisores que envían de forma segura un único mensaje cada uno de ellos a un proceso receptor, mensaje que contiene un entero con el número de proceso emisor. El proceso receptor debe de imprimir el número del proceso emisor que inició el envío en primer lugar. Dicho emisor debe terminar, y el resto quedarse bloqueados:

```
process Emisor[ i : 1.. N ]
  begin
   s_send(i,Receptor);
  end
process Receptor ;
  var ganador : integer ;
  begin
        // calcular ganador
                ....
                ....
    print "El primer envio lo ha realizado: ....", ganador ;
  end
```

Para cada uno de los siguientes casos, describir razonadamente si es posible diseñar una solución a este problema o no lo es. En caso afirmativo, escribe una posible solución:

(a) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a receive

(b) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a i_receive

(c) el proceso receptor usa exclusivamente recepcíon mediante una o varias instrucciones select

87. Supongamos que tenemos `N` procesos concurrentes semejantes:

```
process P[ i : 1..N ] ;
   ....
  begin
    ....
  end
```

Cada proceso produce `N-1` caracteres (con `N-1` llamadas a la función `ProduceCaracter`) y envía cada carácter a los otros `N-1` procesos. Además, cada proceso debe imprimir todos los caracteres recibidos de los otros procesos (el orden en el que se escriben es indiferente).

- Describe razonadamente si es o no posible hacer esto usando exclusivamente `s_send` para los envíos. En caso afirmativo, escribe una solución.
- Escribe una solución usando `send` y `receive`

88. Escribe una nueva solución al problema anterior en la cual se garantize que el orden en el que se imprimen los caracteres es el mismo orden en el que se inician los envíos de dichos caracteres (pista: usa `select` para recibir).

89. Supongamos de nuevo el problema anterior en el cual todos los procesos envían a todos. Ahora cada item de datos a producir y transmitir es un bloque de bytes con muchos valores (por ejemplo, es una imagen que puede tener varios megabytes de tamaño). Se dispone del tipo de datos `TipoBloque` para ello, y el procedimiento `ProducirBloque`, de forma que si `b` es una variable de tipo `TipoBloque`, entonces la llamada a `ProducirBloque(b)` produce y escribe una secuencia de bytes en b. En lugar de imprimir los datos, se deben consumir con una llamada a `ConsumirBloque(b)`.

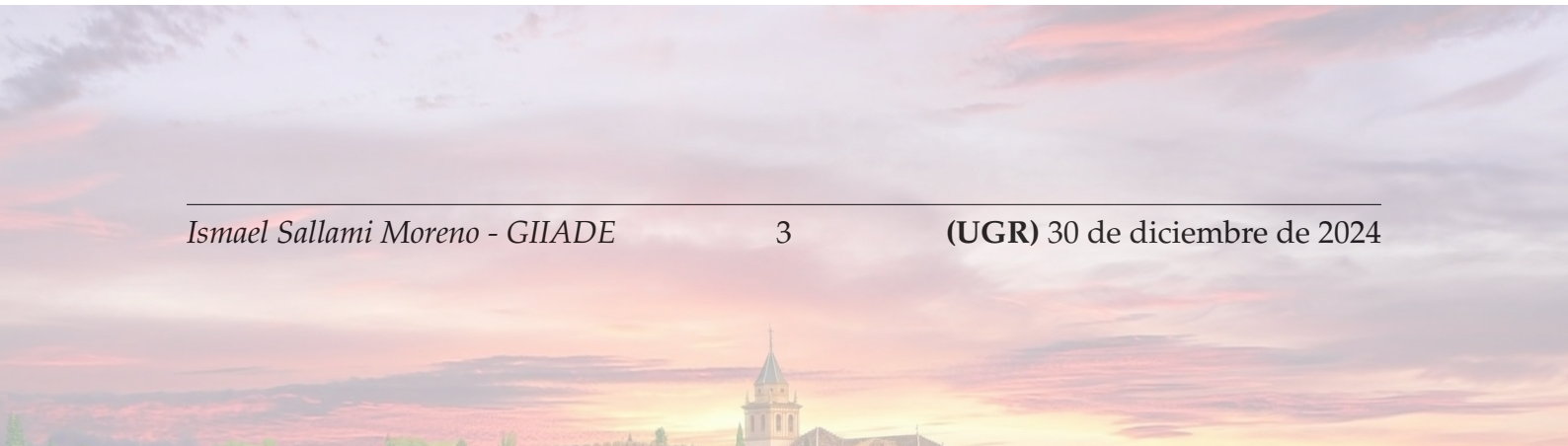
Cada proceso se ejecuta en un ordenador, y se garantiza que hay la suficiente memoria en ese ordenador como para contener simultáneamente, al menos, hasta `N` bloques. Sin embargo, el sistema de paso de mensajes (SPM) podría no tener memoria suficiente como para contener los $(\mathtt{N}-1)^2$ mensajes en tránsito simultáneos que podría llegar a haber en un momento dado con la solución anterior.

En estas condiciones, si el SPM agota la memoria, debe retrasar los `send` dejando bloqueados los procesos y, en esas circunstancias, se podría producir interbloqueo. Para evitarlo, se pueden usar operaciones inseguras de envío, `i_send`. Escribe dicha solución, usando como orden de recepción el mismo que en el problema anterior.

90. En los tres problemas anteriores, cada proceso va esperando a recibir un item de datos de cada uno de los otros procesos, consume dicho item, y después pasa a recibir del siguiente emisor (en distintos órdenes). Esto implica que un envío ya iniciado, pero pendiente, no puede completarse hasta que el receptor no haya consumido los anteriores bloques, es decir, se podría estar consumiendo mucha memoria en el SPM por mensajes en tránsito pendientes cuya recepción se ve retrasada. Escribe una solución en la cual cada proceso inicia sus envíos y recepciones y después espera a que se completen todas las recepciones antes de iniciar el primer consumo de un bloque recibido. De esta forma todos los mensajes pueden transferirse potencialmente de

forma simultánea. Se debe intentar que la transimisión y las producción de bloques sean lo más simultáneas posible. Suponer que cada proceso puede almacenar como mínimo `2*N` bloques en su memoria local, y que el orden de recepción o de consumo de los bloques es indiferente.

3.3.1. Solución



1 Ejercicio 77

1.1. Enunciado

En un sistema distribuido, 6 procesos clientes necesitan sincronizarse de forma específica para realizar cierta tarea, de forma que dicha tarea sólo podrá ser realizada cuando tres procesos estén preparados para realizarla. Para ello, envían peticiones a un proceso controlador del recurso y esperan respuesta para poder realizar la tarea específica. El proceso controlador se encarga de asegurar la sincronización adecuada. Para ello, recibe y cuenta las peticiones que le llegan de los procesos, las dos primeras no son respondidas y producen la suspensión del proceso que envía la petición (debido a que se bloquea esperando respuesta) pero la tercera petición produce el desbloqueo de los tres procesos pendientes de respuesta. A continuación, una vez desbloqueados los tres procesos que han pedido (al recibir respuesta), inicializa la cuenta y procede cíclicamente de la misma forma sobre otras peticiones. El código de los procesos clientes aparece aquí abajo. Los clientes usan envío asíncrono seguro para realizar su petición, y esperan con una recepción síncrona antes de realizar la tarea:

```
process Cliente[ i : 0..5 ] ;
begin
  while true do begin
    send( peticion, Controlador );
    receive( permiso, Controlador );
    Realiza_tarea_grupal();
  end
end

process Controlador ;
begin
  while true do begin
    ...
  end
end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo select for i=... to ... para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice i.

1.2. Solucion

En un sistema distribuido, 6 procesos clientes necesitan sincronizarse de forma específica para realizar cierta tarea, de forma que dicha tarea sólo podrá ser realizada cuando tres procesos estén preparados para realizarla. Para ello, envían peticiones a un proceso controlador del recurso y esperan respuesta para poder realizar la tarea específica. El proceso controlador se encarga de asegurar la sincronización adecuada.

El controlador cuenta las peticiones que le llegan de los procesos:

- Las dos primeras peticiones no son respondidas, lo que provoca que los procesos que las envían queden bloqueados esperando una respuesta.

- La tercera petición produce el desbloqueo de los tres procesos pendientes, enviándoles una respuesta.

Una vez desbloqueados los tres procesos, el controlador inicializa la cuenta y procede cíclicamente de la misma forma para nuevas peticiones. Los clientes usan envío asíncrono seguro para realizar su petición y esperan con una recepción síncrona antes de realizar la tarea.

El código de los procesos clientes es el siguiente:

```
process Cliente[ i : 0..5 ] ;
begin
  while true do begin
    send( peticion, Controlador );   // Envía una petición al proceso
       controlador.
    receive( permiso, Controlador ); // Espera un permiso del controlador.
    Realiza_tarea_grupal();          // Realiza la tarea grupal una vez
       recibido el permiso.
  end
end
```

Listing 1: Código de los procesos clientes

El comportamiento del proceso controlador, que asegura la sincronización, se describe en pseudocódigo a continuación:

```
process Controlador ;
var
  contador : integer := 0;           // Cuenta las peticiones recibidas.
  buffer : array[0..2] of integer;   // Almacena los índices de los
     procesos en espera.

begin
  while true do begin
    select
      for i := 0 to 5                // Itera sobre las peticiones de los
         procesos clientes.
      when receive( peticion, Cliente[i] ) do
        buffer[contador] := i;       // Almacena el índice del proceso
           cliente en el buffer.
        contador := contador + 1;    // Incrementa el contador de
           peticiones.

        if contador = 3 then begin   // Si se han recibido tres peticiones
           ...
          for j := 0 to 2 do
            send( permiso, Cliente[buffer[j]] ); // Envía permiso a los
               tres procesos.
          contador := 0;             // Reinicia el contador.
        end
    end
  end
end
```

Listing 2: Pseudocódigo del proceso Controlador

Explicación del código del controlador:

- El proceso controlador mantiene un contador de peticiones y un buffer que almacena los índices de los clientes en espera.
- Por cada petición recibida, almacena el índice del cliente en el buffer y aumenta el contador.
- Cuando el contador alcanza tres, el controlador envía permisos a los tres procesos almacenados en el buffer y reinicia el contador.

Este enfoque asegura que los clientes se sincronizan correctamente antes de realizar la tarea grupal, tal como se requiere en el enunciado.

2 Ejercicio 78

2.1. Enunciado

En un sistema distribuido, 3 procesos productores producen continuamente valores enteros y los envían a un proceso buffer que los almacena temporalmente en un array local de 4 celdas enteras para ir enviándoselos a un proceso consumidor. A su vez, el proceso buffer realiza lo siguiente, sirviendo de forma equitativa al resto de procesos:

- Envía enteros al proceso consumidor siempre que su array local tenga al menos dos elementos disponibles.
- Acepta envíos de los productores mientras el array no esté lleno, pero no acepta que cualquier productor pueda escribir dos veces consecutivas en el búfer.

El código del productor y del consumidor es el siguiente:

```
process Productor[ i : 0..2 ] ;
var dato : integer ;
begin
  while true do begin
    dato := Producir();
    send( dato, Buffer );
  end
end

process Consumidor ;
begin
  while true do begin
    receive ( dato, Buffer );
    Consumir( dato );
  end
end

process Buffer ;
begin
  while true do begin
    ...
  end
end
```

Se pide: describir en pseudocódigo el comportamiento del proceso buffer, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos.

2.2. Solución

```
Process Buffer;
var
  buffer : array[0..3] of integer; // Array local de 4 celdas enteras.
  count : integer := 0;             // Contador de elementos en el buffer.
  lastProducer : integer := -1;     // Índice del último productor que escribió en el buffer.

  begin

    while true do begin

      select
        when receive(dato, Productor[0]) do
          if(count <4 and lastProducer != 0) then
            buffer[count] := dato; // Almacena el dato en el buffer.
            count := count + 1;   // Incrementa el contador de elementos.
            lastProducer := 0; // Actualiza el índice del último productor.
          end

        when receive(dato, Productor[1]) do
          if(count <4 and lastProducer != 1) then
            buffer[count] := dato; // Almacena el dato en el buffer.
            count := count + 1;   // Incrementa el contador de elementos.
            lastProducer := 1; // Actualiza el índice del último productor.
          end

        when receive(dato, Productor[2]) do
          if(count <4 and lastProducer != 2) then
            buffer[count] := dato; // Almacena el dato en el buffer.
            count := count + 1;   // Incrementa el contador de elementos.
            lastProducer := 2; // Actualiza el índice del último productor.
          end

        when count >= 2 do // Si hay al menos dos elementos en el buffer...
          send(buffer[0], Consumidor); // Envía el primer elemento al consumidor.
          for i := 0 to 2 do
            buffer[i] := buffer[i + 1]; // Desplaza los elementos restantes.
          end
          count := count - 1; // Decrementa el contador de elementos.


```

```
41      end
42
43
44    end
```

Listing 3: Pseudocódigo del proceso Buffer

3 Ejercicio 79

3.1. Enunciado

Suponer un proceso productor y 3 procesos consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el proceso productor debe ser retirado por todos los 3 procesos consumidores para ser eliminado del buffer. Cada consumidor retirará los datos del buffer en el mismo orden en el que son depositados, aunque los diferentes consumidores pueden ir retirando los elementos a ritmo diferente unos de otros.

Por ejemplo, mientras un consumidor ha retirado los elementos 1, 2 y 3, otro consumidor puede haber retirado solamente el elemento 1. De esta forma, el consumidor más rápido podría retirar hasta B elementos más que el consumidor más lento.

Describir en pseudocódigo el comportamiento de un proceso que implemente el buffer de acuerdo con el esquema de interacción descrito usando una construcción de espera selectiva, así como el del proceso productor y de los procesos consumidores. Comenzar identificando qué información es necesario representar, para después resolver las cuestiones de sincronización.

Una posible implementación del buffer mantendría, para cada proceso consumidor, el puntero de salida y el número de elementos que quedan en el buffer por consumir:

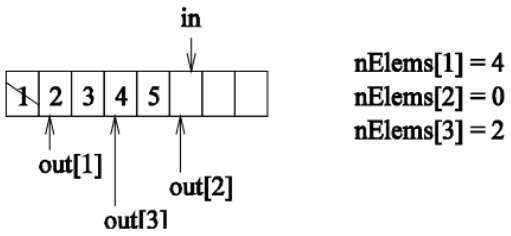


Figura 1: Esquema de interacción entre el productor y los consumidores.

3.2. Solución

```
1   process Buffer;
2
3   var
4     out: array[0..2] of integer; // Punteros de salida para cada consumidor
          .
5     nElems: array[0..2] of integer; // Número de elementos que quedan por
          consumir para cada consumidor.
```

```
6
7     B: integer; // Tamaño del buffer.
8     buf: array[0..B-1]; // Buffer acotado de tamaño B.
9     begin
10    select for i:=0 to 2 do
11
12      when nElemens[i] != 0 do // Si el consumidor i tiene elementos por
            consumir...
13        send (buf[out[i]], consumidor[i]); // Envía el elemento al
            consumidor i.
14        out[i] + 1 mod B; // Actualiza el puntero de salida del consumidor
            i.
15        nElemens[i] = nElemens[i]-1; // Decrementa el número de elementos
            por consumir del consumidor i.
16      end do
17
18      when nElemens[0] != B or nElemens[1] != B or nElemens[2] != B do //
          Si el buffer no está lleno para algún consumidor...
19        receive(dato, productor); // Recibe un dato del productor.
20        bufer[i] = dato; // Almacena el dato en el buffer.
21        for j:=0 to 2 do
22          nElemens[j] = nElemens[j] + 1; // Incrementa el número de
              elementos por consumir para cada consumidor.
23        end do
24      end do
25    end select
26  end
```

Listing 4: Pseudocódigo del proceso Buffer

Explicación del código del proceso Buffer

El proceso Buffer se encarga de gestionar la interacción entre un productor y tres consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el productor debe ser retirado por todos los consumidores para ser eliminado del buffer. La idea general del código es mantener un buffer acotado de tamaño B donde el productor puede depositar elementos y los consumidores pueden retirarlos. Cada consumidor tiene su propio puntero de salida y contador de elementos por consumir. El proceso Buffer asegura que cada elemento depositado por el productor sea retirado por todos los consumidores antes de ser eliminado del buffer. Además, se asegura de que el buffer no se llene completamente para ningún consumidor y que los consumidores retiren los elementos en el mismo orden en el que fueron depositados.

4 Ejercicio 80

4.1. Enunciado

Implementación de los procesos Salvajes y Cocinero

Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté

vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros M misioneros.

```
process Salvaje[ i : 0..2 ] ;
begin
    var peticion : integer := ... ;
    begin
        while true do begin
            // esperar a servirse un misionero
            s_send( peticion, Olla );
            // comer:
            Comer();
        end
    end
end

process Cocinero ;
begin
    while true do begin
        // dormir esperando solicitud para llenar
        // ...
        // confirmar que se ha rellenado la olla
        // ...
    end
end
```

Implementar los procesos salvajes y cocinero usando paso de mensajes, utilizando un proceso Olla que incluye una construcción de espera selectiva que sirve peticiones de los salvajes y el cocinero para mantener la sincronización requerida, teniendo en cuenta que:

- La solución no debe producir interbloqueo.
- Los salvajes podrán comer siempre que haya comida en la olla.
- Solamente se despertará al cocinero cuando la olla esté vacía.

4.2. Solución

```
process Olla;

var
    comida: integer := 0; // Cantidad de comida disponible en la olla.
    capacidad: integer := M; // Capacidad máxima de la olla.
    peticiones: queue of integer; // Cola para gestionar las peticiones de
       los salvajes.

begin
    select
        when not peticiones.empty() and comida > 0 do
            // Atender a un salvaje que quiere comer
            salvaje := peticiones.pop();
            comida := comida - 1;
            send("comer", salvaje);
```

```
        end

        when comida = 0 do
            // Solicitar al cocinero que rellene la olla
            send("rellenar", cocinero);
            receive("rellenado", cocinero);
            comida := capacidad;
        end
    end select
end
```

Listing 5: Pseudocódigo del proceso Olla

```
process Salvaje[i: 0..N-1];

begin
    while true do
        // Solicitar comida a la olla
        send(i, Olla);
        receive("comer", Olla);
        Comer();
    end
end
```

Listing 6: Pseudocódigo del proceso Salvaje

```
process Cocinero;

begin
    while true do
        // Esperar solicitud para rellenar la olla
        receive("rellenar", Olla);
        // Rellenar la olla
        Rellenar();
        send("rellenado", Olla);
    end
end
```

Listing 7: Pseudocódigo del proceso Cocinero

Esta solución asegura:

- Sincronización adecuada entre salvajes y cocinero mediante paso de mensajes.
- Los salvajes pueden comer siempre que haya comida en la olla.
- El cocinero solo se despierta cuando la olla esté vacía.
- Evita el interbloqueo utilizando una cola para gestionar las peticiones de los salvajes.

5 Ejercicio 81

5.1. Enunciado

Considerar un conjunto de N procesos, $P[i]$, $(i = 0, \ldots, N-1)$ que se pasan mensajes cada uno al siguiente (y el primero al último), en forma de anillo. Cada proceso tiene un valor local almacenado en su variable local `mi_valor`. Deseamos calcular la suma de los valores locales almacenados por los procesos de acuerdo con el algoritmo que se expone a continuación.

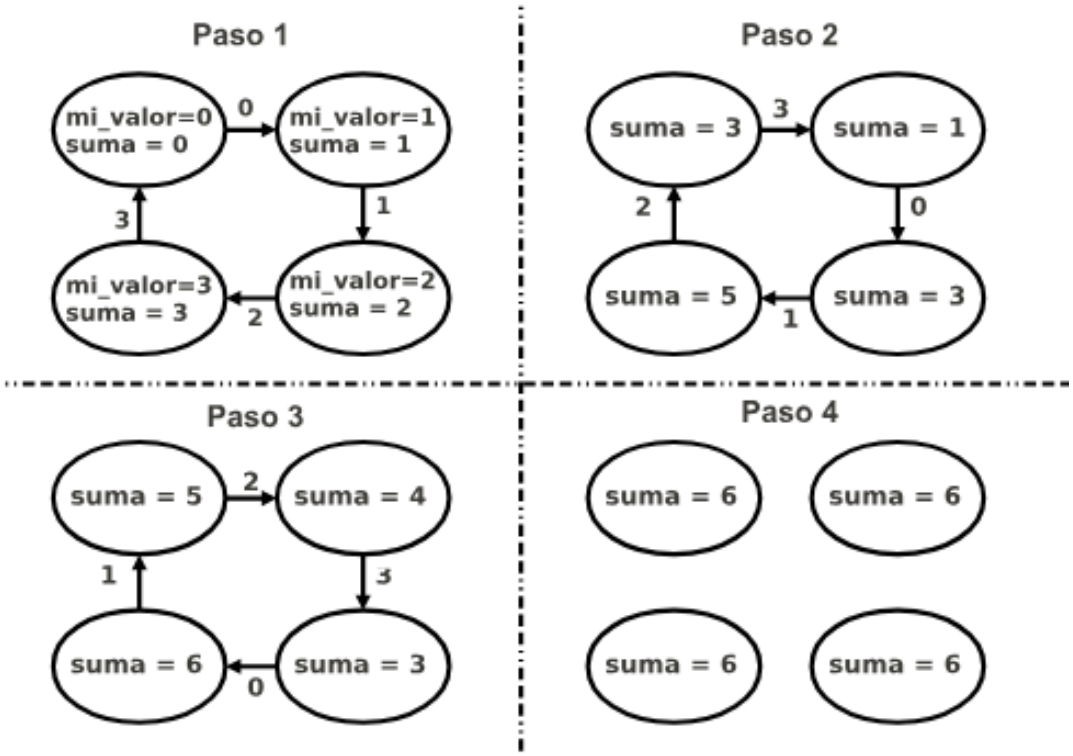


Figura 2: Esquema de interacción entre los procesos.

Los procesos realizan una serie de iteraciones para hacer circular sus valores locales por el anillo. En la primera iteración, cada proceso envía su valor local al siguiente proceso del anillo, al mismo tiempo que recibe del proceso anterior el valor local de éste. A continuación acumula la suma de su valor local y el recibido desde el proceso anterior. En las siguientes iteraciones, cada proceso envía al siguiente proceso siguiente el valor recibido en la anterior iteración, al mismo tiempo que recibe del proceso anterior un nuevo valor. Después acumula la suma. Tras un total de $N-1$ iteraciones, cada proceso conocerá la suma de todos los valores locales de los procesos.

Dar una descripción en pseudocódigo de los procesos siguiendo un estilo SPMD y usando operaciones de envío y recepción síncronas:

```
process P[ i : 0..N-1 ] ;
var mi_valor : integer := ... ; // valor arbitrario (== i en la figura, por
    ejemplo)
    suma : integer := mi_valor ; // suma inicializada a mi_valor
```

```
begin
    for j := 0 to N-1 do begin
        ...
    end
end
```

5.2. Solución

En este caso se afirma que se debe de seguir un estilo SPMD (Single Program Multiple Data), lo que significa que todos los procesos ejecutan el mismo código, pero con datos diferentes. En este caso, cada proceso $P[i]$ tiene un valor local *mi_valor* que se suma a la suma total.

```
process P[ i : 0..N-1 ] ;
var mi_valor : integer := i ; // valor arbitrario (== i en la figura, por
    ejemplo)
    suma : integer := mi_valor ; // suma inicializada a mi_valor
begin
    for j := 0 to N-1 do begin
        send( mi_valor, P[ (i + 1) mod N ] );
        receive( valor_recibido, P[ (i - 1 + N) mod N ] );
        suma := suma + valor_recibido;
        mi_valor := valor_recibido;
    end
end
```

6 Ejercicio 82

6.1. Enunciado

Considerar un estanco en el que hay tres fumadores y un estanquero. Cada fumador continuamente lía un cigarro y se lo fuma. Para liar un cigarro, el fumador necesita tres ingredientes: tabaco, papel y cerillas. Uno de los fumadores tiene solamente papel, otro tiene solamente tabaco, y el otro tiene solamente cerillas. El estanquero tiene una cantidad infinita de los tres ingredientes.

El estanquero coloca aleatoriamente dos ingredientes diferentes de los tres que se necesitan para hacer un cigarro, desbloquea al fumador que tiene el tercer ingrediente y después se bloquea. El fumador seleccionado se puede obtener fácilmente mediante una función `genera_ingredientes` que devuelve el índice (0, 1, ó 2) del fumador escogido.

El fumador desbloqueado toma los dos ingredientes del mostrador, desbloqueando al estanquero, lía un cigarro y fuma durante un tiempo. El estanquero, una vez desbloqueado, vuelve a poner dos ingredientes aleatorios en el mostrador, y se repite el ciclo.

Describir una solución distribuida que use envío asíncrono seguro y recepción síncrona para este problema usando un proceso Estanquero y tres procesos fumadores Fumador(i) (con $i = 0, 1$ y 2).

```
process Estanquero ;
begin
    while true do begin
        // El estanquero coloca dos ingredientes aleatorios en el mostrador
        // y desbloquea al fumador que tiene el tercer ingrediente
        ...
    end
end

process Fumador[ i : 0..2 ] ;
begin
    while true do begin
        // El fumador espera hasta que sea desbloqueado
        // Toma los dos ingredientes del mostrador, lía un cigarro y fuma
        ...
    end
end
```

6.2. Solución

Usando envío asíncrono seguro y recepción síncrona. Para ello debemos de usar MPI_Isend y MPI_Recv.

```
  process Estanquero ;
  begin
      while true do begin
          // El estanquero coloca dos ingredientes aleatorios en el
             mostrador
          int ingrediente1, ingrediente2;
          ingrediente1 = generarIngrediente();
          do{
              ingrediente2 = generarIngrediente();
          }
          while(ingrediente1 == ingrediente2);
          // y desbloquea al fumador que tiene el tercer ingrediente
          MPI_Isend(ingrediente1, 1, MPI_INT, Fumador[3 - ingrediente1 -
             ingrediente2], 0, MPI_COMM_WORLD);
          MPI_Isend(ingrediente2, 1, MPI_INT, Fumador[3 - ingrediente1 -
             ingrediente2], 0, MPI_COMM_WORLD);
      end
  end

  process Fumador[ i : 0..2 ] ;
  begin
      while true do begin
          // El fumador espera hasta que sea desbloqueado
          int ingrediente1, ingrediente2;
          MPI_Recv(ingrediente1, 1, MPI_INT, Estanquero, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
          MPI_Recv(ingrediente2, 1, MPI_INT, Estanquero, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
          // Toma los dos ingredientes del mostrador, lía un cigarro y fuma
          LiarCigarro();
          Fumar();
```

```
27        end
28    end
```

7 Ejercicio 83

7.1. Enunciado

En un sistema distribuido, un gran número de procesos clientes usa frecuentemente un determinado recurso y se desea que puedan usarlo simultáneamente el máximo número de procesos. Para ello, los clientes envían peticiones a un proceso controlador para usar el recurso y esperan respuesta para poder usarlo (véase el código de los procesos clientes). Cuando un cliente termina de usar el recurso, envía una solicitud para dejar de usarlo y espera respuesta del Controlador. El proceso controlador se encarga de asegurar la sincronización adecuada imponiendo una única restricción por razones supersticiosas: nunca habrá 13 procesos exactamente usando el recurso al mismo tiempo.

```
process Cli[ i : 0....n ] ;
var pet_usar : integer := +1 ;
    pet_liberar : integer := -1 ;
    permiso : integer := ... ;
begin
    while true do begin
        send( pet_usar, Controlador );
        receive( permiso, Controlador );
        Usar_recurso( );
        send( pet_liberar, Controlador );
        receive( permiso, Controlador );
    end
end
```

```
process Controlador ;
begin
    while true do begin
        select
            ...
        end
end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo select for i=... to ... para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice i.

7.2. Solución

```
1
2   process Controlador ;
3   var contador: integer;
4   begin
5       while true do begin
6           select
7             for i:= 0 to n do
8
9               when receive(pet_usar,Cli[i]) do
10                if contador < 13 then
11                  send(permiso,Cli[i]);
12                  contador := contador + 1;
13                end if
14
15              when receive(pet_liberar,Cli[i]) do
16                send(permiso,Cli[i]);
17                contador := contador - 1;
18              end do
19
20          end select
21      end
22  end
```

Lógica de la solución

En este caso se ha pensado la solución de manera simple. Se trata de un controlador que recibe peticiones de los clientes para usar un recurso. Si el contador de procesos que están usando el recurso es menor que 13, el controlador envía un permiso al cliente para que pueda usar el recurso. Cuando un cliente termina de usar el recurso, envía una petición para liberarlo y el controlador disminuye el contador. De esta forma, se asegura que nunca haya exactamente 13 procesos usando el recurso al mismo tiempo. Es impotante **uso del if** para asegurarnos de que solo se usan los 13 procesos a la vez, y debemos de asegurarnos de que actualizamos de manera correcta la variable contador.

8 Ejercicio 84

8.1. Enunciado

En un sistema distribuido, tres procesos Productor se comunican con un proceso Impresor que se encarga de ir imprimiendo en pantalla una cadena con los datos generados por los procesos productores. Cada proceso productor (`Productor[i]` con $i = 0, 1, 2$) genera continuamente el correspondiente entero i, y lo envía al proceso Impresor.

El proceso Impresor se encarga de ir recibiendo los datos generados por los productores y los imprime por pantalla (usando el procedimiento `imprime(entero)`) generando una cadena de dígitos en la salida. No obstante, los procesos se han de sincronizar adecuadamente para que la impresión por pantalla cumpla las siguientes restricciones:

- Los dígitos 0 y 1 deben aceptarse por el impresor de forma alterna. Es decir, si se acepta un 0 no podrá volver a aceptarse un 0 hasta que se haya aceptado un 1, y viceversa, si se acepta un 1 no podrá volver a aceptarse un 1 hasta que se haya aceptado un 0.
- El número total de dígitos 0 o 1 aceptados en un instante no puede superar el doble de número de dígitos 2 ya aceptados en dicho instante.

Cuando un productor envía un dígito que no se puede aceptar por el impresor, el productor quedará bloqueado esperando completar el s_send. El pseudocódigo de los procesos productores (Productor) se muestra a continuación, asumiendo que se usan operaciones bloqueantes no buferizadas (síncronas):

```
process Productor[ i : 0,1,2 ]
while true do begin
    s_send( i, Impresor ) ;
end
```

```
process Impresor
  var
      .....
    begin
      while true do begin
        select
          .....
        end
      end
    end
```

Listing 8: Pseudocódigo del proceso Impresor

8.2. Solución

```
  process Impresor
  var
      contador_0_1 : integer := 0 ;
      contador_2 : integer := 0 ;
      turno : integer := 0 ; // 0: turno para aceptar 0, 1: turno para
         aceptar 1
  begin
      while true do begin
          select
            for i:= 0 to 2
              when receive(dato, Productor[i]) do
                  if (dato == 0 and turno == 0 and contador_0_1 < 2 *
                     contador_2) then
                      imprime(dato);
                      contador_0_1 := contador_0_1 + 1;
                      turno := 1;
                  else if (dato == 1 and turno == 1 and contador_0_1 < 2 *
                     contador_2) then
                      imprime(dato);
                      contador_0_1 := contador_0_1 + 1;
```

```
                    turno := 0;
                else if (dato == 2) then
                    imprime(dato);
                    contador_2 := contador_2 + 1;
                end if;
            end when
        end select
    end
end
```

Lógica de la solución

En este código del proceso `Impresor`, se maneja la sincronización entre los productores y el impresor usando un `select`, y además diseñandolo la solución usando la opción for para hacer más simple el código. Para cada productor, se controla la aceptación de los dígitos 0, 1 y 2, con las restricciones mencionadas en el enunciado. El proceso acepta un dígito según las condiciones del turno (alternancia entre 0 y 1) y el límite sobre el número de dígitos 0 y 1 que pueden aceptarse en comparación con los dígitos 2.

9 Ejercicio 85

9.1. Enunciado

En un sistema distribuido hay un vector de n procesos iguales que envían con `send` (en un bucle infinito) valores enteros a un proceso receptor, que los imprime. Si en algún momento no hay ningún mensaje pendiente de recibir en el receptor, este proceso debe imprimir "no hay mensajes, duermo"; después de bloquearse durante 10 segundos (con `sleep_for(10)`), antes de volver a comprobar si hay mensajes (esto podría hacerse para ahorrar energía, ya que el procesamiento de mensajes se hace en ráfagas separadas por 10 segundos).

Este problema no se puede solucionar usando `receive` o `i_receive`. Indica a qué se debe esto. Sin embargo, sí se puede hacer con `select`. Diseña una solución a este problema con `select`:

```
process Emisor[ i : 1..n ]
var dato : integer ;
begin
    while true do begin
        dato := Producir() ;
        send( dato, Receptor );
    end
end
```

```
process Receptor()
var dato : integer ;
begin
    while true do
        ......
end
```

9.2. Solución

Para resolver este problema, utilizamos un enfoque basado en `select`, ya que ni `receive` ni `i_receive` permiten detectar si no hay mensajes pendientes de forma directa. Esto se debe a que:

- `receive` es una operación bloqueante que espera hasta que haya un mensaje disponible, impidiendo que podamos comprobar si no hay mensajes sin bloquear el proceso.
- `i_receive` es una operación no bloqueante, pero simplemente devuelve un indicador de éxito o fallo y no permite implementar un mecanismo de espera como el solicitado.

Con `select`, podemos implementar un mecanismo de espera que permita al proceso receptor manejar mensajes cuando estén disponibles, o ejecutar una acción alternativa (como imprimir "no hay mensajes, duermo") si no hay mensajes. La solución se describe a continuación:

```
process Emisor[ i : 1..n ]
var dato : integer ;
begin
    while true do begin
        dato := Producir(); // Generar un nuevo dato
        send(dato, Receptor); // Enviar el dato al Receptor
    end
end
```

```
process Receptor()
var dato : integer ;
begin
    while true do begin
        select
            when receive(dato, Emisor[1..n]) do // Si hay mensajes
                imprime(dato); // Imprimir el mensaje recibido
            else // Si no hay mensajes pendientes
                imprime("no hay mensajes, duermo");
                sleep_for(10); // Bloquear durante 10 segundos
        end select
    end
end
```

Explicación de la solución

1. Emisor: Cada proceso `Emisor[i]` produce un valor entero con la función `Producir()` y lo envía al `Receptor` utilizando la operación `send`. Este proceso se ejecuta en un bucle infinito.

2. Receptor: El proceso `Receptor` implementa un bucle infinito que realiza las siguientes acciones:

- Usa una sentencia `select` para manejar dos escenarios:

- Si hay mensajes disponibles, el receptor los consume con `receive` y los imprime utilizando `imprime(dato)`.
- Si no hay mensajes pendientes (gracias al `else` del `select`), el receptor imprime "no hay mensajes, duermo" y entra en un estado de espera por 10 segundos usando `sleep_for(10)`.

3. Uso de `select`: La operación `select` permite al proceso `Receptor` manejar la recepción de mensajes de múltiples emisores (`Emisor[1..n]`), así como realizar una acción alternativa (`else`) cuando no hay mensajes pendientes.

Respuesta a las preguntas planteadas

1. ¿Por qué no se puede solucionar el problema con `receive` o `i_receive`?

- `receive` es bloqueante, lo que significa que el receptor esperará indefinidamente hasta que llegue un mensaje, impidiendo detectar que no hay mensajes.
- `i_receive` no es bloqueante, pero solo devuelve un indicador de éxito o fallo. No proporciona un mecanismo para realizar acciones alternativas cuando no hay mensajes pendientes, como el que se implementa con `select`.

2. ¿Por qué es útil `select`?

- `select` permite manejar múltiples condiciones de recepción de mensajes y, además, proporciona un mecanismo `else` para definir acciones alternativas cuando ninguna condición se cumple. Esto es crucial para implementar el comportamiento del receptor cuando no hay mensajes.

Con esta implementación, se garantiza que el `Receptor` cumpla con las especificaciones planteadas: procesar mensajes cuando estén disponibles o entrar en un estado de espera eficiente si no hay mensajes.

10 Ejercicio 86

10.1. Enunciado

En un sistema tenemos **N procesos emisores** que envían de forma segura un único mensaje cada uno de ellos a un proceso receptor. El mensaje contiene un entero con el número del proceso emisor. El proceso receptor debe imprimir el número del proceso emisor que inició el envío en primer lugar. Dicho emisor debe terminar, y el resto quedarse bloqueados:

```
process Emisor[ i : 1.. N ]
begin
    s_send(i, Receptor);
end

process Receptor ;
var ganador : integer ;
begin
```

```
    // calcular ganador
    ....
    ....
    print "El primer envío lo ha realizado: ....", ganador;
end
```

Para cada uno de los siguientes casos, describir razonadamente si es posible diseñar una solución a este problema o no lo es. En caso afirmativo, escribe una posible solución:

(a) El proceso receptor usa exclusivamente recepción mediante una o varias llamadas a `receive`.

(b) El proceso receptor usa exclusivamente recepción mediante una o varias llamadas a `i_receive`.

(c) El proceso receptor usa exclusivamente recepción mediante una o varias instrucciones `select`.

10.2. Solución

A continuación, se analiza cada uno de los casos planteados y se proporciona una solución cuando es posible:

a) **Recepción exclusivamente mediante `receive`:**

Usar `receive` garantiza que el proceso receptor obtiene un mensaje completo antes de procesarlo, lo que permite determinar cuál fue el primer emisor. Sin embargo, `receive` no tiene orden garantizado cuando varios emisores envían simultáneamente. Para garantizar que se identifica correctamente al primer emisor, los mensajes deben llegar en el orden en que fueron enviados.

Solución:

```
process Receptor ;
var ganador : integer ;
    recibido : integer ;
    encontrado : boolean := false ;
begin
    while not encontrado do begin
        receive(recibido, *);  // Recibir de cualquier emisor
        if not encontrado then begin
            ganador := recibido;  // El primer mensaje recibido
            encontrado := true;
            print "El primer envío lo ha realizado: ", ganador;
        end
    end
end
```

En esta solución:

- El receptor procesa el primer mensaje recibido y almacena el identificador del emisor como `ganador`.

- Los demás emisores quedan bloqueados porque el receptor no solicita más mensajes.

b) **Recepción exclusivamente mediante `i_receive`:**

La operación `i_receive` permite verificar si un mensaje está disponible sin bloquearse. Sin embargo, `i_receive` no garantiza un orden, por lo que sería necesario iterar constantemente para determinar el primer emisor. Esto puede generar comportamiento indeterminado, ya que `i_receive` solo indica disponibilidad y no un orden temporal.

Conclusión: No es posible garantizar que el primer mensaje sea procesado correctamente usando solo `i_receive`.

c) **Recepción exclusivamente mediante `select`:**

La instrucción `select` permite manejar múltiples canales de recepción simultáneamente. Esto asegura que el receptor atienda el primer mensaje recibido en cualquiera de los canales, determinando así correctamente al primer emisor.

Solución:

```
process Receptor ;
var ganador : integer ;
    recibido : integer ;
    encontrado : boolean := false ;
begin
    while not encontrado do begin
        select
            for i := 1 to N do
                when receive(recibido, Emisor[i]) do begin
                    ganador := recibido;  // El primer mensaje recibido
                    encontrado := true;
                    print "El primer envío lo ha realizado: ", ganador ;
                end
            end for
        end select
    end
end
```

En esta solución:

- El receptor utiliza `select` para atender al primer mensaje recibido desde cualquier emisor.
- El bucle asegura que se identifica al emisor más rápido, y se imprime su número.
- Los emisores restantes quedan bloqueados, ya que no se procesan más mensajes después de encontrar al ganador.

Resumen:

a) Es posible resolver el problema usando `receive`, y la solución es viable.

b) No es posible garantizar una solución con `i_receive`, debido a la falta de orden y bloqueo.

c) Es posible resolver el problema con `select`, y la solución es eficiente y correcta.

11 Ejercicio 87

11.1. Enunciado

Supongamos que tenemos **N** procesos concurrentes semejantes:

```
process P[ i : 1..N ] ;
    ....
  begin
    ....
  end
```

Cada proceso produce **N-1 caracteres** (con `N-1` llamadas a la función `ProduceCaracter`) y envía cada carácter a los otros `N-1` procesos. Además, cada proceso debe imprimir todos los caracteres recibidos de los otros procesos (el orden en el que se escriben es indiferente).

- **Describe razonadamente si es o no posible hacer esto usando exclusivamente `s_send` para los envíos.** En caso afirmativo, escribe una solución.
- **Escribe una solución usando `send` y `receive`.**

11.2. Solución

- **1. Análisis del uso exclusivo de `s_send`:**

  El uso exclusivo de `s_send` (envío sin búfer y bloqueante) puede causar problemas de interbloqueo en este escenario. Dado que `s_send` requiere que el receptor esté listo para recibir el mensaje en el momento del envío, si todos los procesos están intentando enviar al mismo tiempo y ninguno está recibiendo, el sistema se bloqueará.

  Por lo tanto, **no es posible implementar esta solución utilizando exclusivamente `s_send`** debido a la naturaleza de la operación bloqueante.

- **2. Solución utilizando `send` y `receive`:**

  Para resolver este problema, se utiliza una combinación de `send` y `receive`, asegurando que cada proceso envíe sus caracteres a los demás procesos y, al mismo

tiempo, reciba los caracteres enviados por otros procesos.

```
process P[ i : 1..N ] ;
var
    caract : char ;       // Carácter producido
    recibido : char ;     // Carácter recibido
begin
    // Enviar N-1 caracteres a los otros procesos
    for j := 1 to N do
        if j != i then begin
            caract := ProduceCaracter(); // Produce un carácter
            send(caract, P[j]);          // Enviar al proceso P[j]
        end
    end

    // Recibir N-1 caracteres de los otros procesos
    for j := 1 to N do
        if j != i then begin
            receive(recibido, P[j]);      // Recibir de proceso P[j]
            imprime(recibido);           // Imprimir el carácter
                recibido
        end
    end
end
```

Explicación de la solución:

- Cada proceso `P[i]` realiza dos bucles:
  - En el primer bucle, genera `N-1` caracteres con la función `ProduceCaracter` y los envía a los otros `N-1` procesos utilizando `send`.
  - En el segundo bucle, recibe los caracteres enviados por los otros `N-1` procesos utilizando `receive` y los imprime.
- Para evitar enviar mensajes a sí mismo, se incluye la condición `if j != i`.
- El uso de `send` y `receive` permite que los procesos se sincronicen correctamente sin riesgo de interbloqueo, ya que las operaciones de recepción permiten manejar los mensajes enviados.

12 Ejercicio 88

12.1. Enunciado

Escribe una nueva solución al problema anterior en la cual se garantice que el orden en el que se imprimen los caracteres es el mismo orden en el que se inician los envíos de dichos caracteres.

Pista: usa `select` para recibir.

12.2. Solución

Para garantizar que los caracteres se impriman en el mismo orden en el que se inician los envíos, utilizamos la instrucción `select` en el proceso receptor. Esto permite gestionar de manera ordenada la recepción de los caracteres basándose en el orden de llegada de los mensajes.

```
process P[ i : 1..N ] ;
var
    caract : char ;       // Carácter producido
begin
    // Enviar N-1 caracteres a los otros procesos
    for j := 1 to N do
        if j != i then begin
            caract := ProduceCaracter(); // Produce un carácter
            send(caract, Receptor);      // Enviar al proceso Receptor
        end
    end
end

process Receptor ;
var
    caract : char ;       // Carácter recibido
    sender : integer ;    // ID del proceso emisor, se supone que se usa en
        imprime
    recibido[N] : integer := [0, ..., 0]; // Vector de contadores por
       emisor, se usa como extra
begin
    while true do begin
        select
            for i := 1 to N do
                when receive(caract, P[i]) do begin
                    imprime(caract, sender);        // Imprimir el carácter
                        recibido
                    //imprime(caract);
                    recibido[i] := recibido[i] + 1; // Actualizar contador
                end
            end
        end select
    end
end
```

Explicación de la solución:

- **Proceso `P[i]`:**
  - Cada proceso genera `N-1` caracteres usando la función `ProduceCaracter`.
  - Los caracteres se envían al proceso `Receptor` mediante `send`.
- **Proceso `Receptor`:**
  - Utiliza una instrucción `select` para manejar los mensajes recibidos desde los procesos `P[i]` en orden de llegada.
  - Cada mensaje contiene el carácter producido por el proceso emisor y su ID (implícito en la recepción).

- Los caracteres se imprimen en el orden de recepción, asegurando que se respeta el orden en el que se iniciaron los envíos.
- Un vector `recibido[i]` se utiliza para llevar un conteo del número de caracteres recibidos de cada proceso, si fuera necesario para el análisis.

13 Ejercicio 89

13.1. Enunciado

Supongamos de nuevo el problema anterior en el cual todos los procesos envían a todos. Ahora cada **item de datos** a producir y transmitir es un **bloque de bytes** con muchos valores (por ejemplo, es una imagen que puede tener varios megabytes de tamaño). Se dispone del tipo de datos `TipoBloque` para ello, y el procedimiento `ProducirBloque`, de forma que si `b` es una variable de tipo `TipoBloque`, entonces la llamada a `ProducirBloque(b)` produce y escribe una secuencia de bytes en `b`.

En lugar de imprimir los datos, se deben consumir con una llamada a `ConsumirBloque(b)`.

- Cada proceso se ejecuta en un ordenador, y se garantiza que hay la suficiente memoria en ese ordenador como para contener simultáneamente, al menos, hasta `N` bloques.
- Sin embargo, el sistema de paso de mensajes (SPM) podría no tener memoria suficiente como para contener los $(N-1)^2$ mensajes en tránsito simultáneos que podría llegar a haber en un momento dado con la solución anterior.
- En estas condiciones, si el **SPM** agota la memoria, debe retrasar los `send` dejando bloqueados los procesos y, en esas circunstancias, se podría producir **interbloqueo**.
- Para evitarlo, se pueden usar operaciones inseguras de envío, `i_send`.

Escribe dicha solución, usando como orden de recepción el mismo que en el problema anterior.

13.2. Solución

13.3. Solución

Para resolver el problema en el que los procesos envían bloques de datos grandes y garantizar que no haya interbloqueo debido a limitaciones de memoria en el sistema de paso de mensajes, utilizamos operaciones inseguras de envío `i_send`. Además, aseguramos que los bloques se reciben en el mismo orden en que se inician los envíos mediante el uso de `select`.

```
process P[ i : 1..N ] ;
var
    bloque : TipoBloque;  // Bloque de datos producido
begin
```

```
    // Enviar N-1 bloques de datos a los otros procesos
    for j := 1 to N do
        if j != i then begin
            ProducirBloque(bloque);       // Producir un bloque de datos
            i_send(bloque, Receptor);    // Enviar de manera insegura al
               Receptor
        end
    end
end

process Receptor ;
var
    bloque : TipoBloque;   // Bloque de datos recibido
    sender : integer;      // ID del proceso emisor
    recibido[N] : integer := [0, ..., 0]; // Vector de contadores por
       emisor
begin
    while true do begin
        select
            for i := 1 to N do
                when receive(bloque, P[i]) do begin
                    ConsumirBloque(bloque);         // Consumir el bloque
                       recibido
                    recibido[i] := recibido[i] + 1; // Actualizar contador
                end
            end
        end select
    end
end
```

Explicación de la solución:

- **Proceso `P[i]`:**
  - Cada proceso genera `N-1` bloques de datos usando la función `ProducirBloque`.
  - Los bloques se envían al proceso `Receptor` utilizando `i_send`, que es una operación de envío no bloqueante.
  - Esto garantiza que los procesos emisores no se bloqueen entre sí debido a las limitaciones de memoria del sistema de paso de mensajes.
- **Proceso `Receptor`:**
  - Utiliza una instrucción `select` para manejar los mensajes recibidos desde los procesos `P[i]` en orden de llegada.
  - Cada mensaje contiene el bloque de datos producido por el proceso emisor.
  - Los bloques se consumen inmediatamente con `ConsumirBloque`, liberando la memoria del sistema de paso de mensajes.
  - Un vector `recibido[i]` se utiliza para llevar un conteo del número de bloques recibidos de cada proceso, si fuera necesario para análisis o depuración.

Garantías de esta solución:

- **Evita interbloqueos:** El uso de `i_send` permite que los procesos emisores no queden bloqueados si el sistema de paso de mensajes agota la memoria.
- **Orden de recepción:** La instrucción `select` garantiza que los bloques se procesen en el orden en el que llegan al receptor.
- **Consumo eficiente de memoria:** Los bloques se consumen tan pronto como son recibidos, liberando memoria en el sistema de paso de mensajes.

14 Ejercicio 90

14.1. Enunciado

En los tres problemas anteriores, cada proceso va esperando a recibir un ítem de datos de cada uno de los otros procesos, consume dicho ítem, y después pasa a recibir del siguiente emisor (en distintos órdenes). Esto implica que un envío ya iniciado, pero pendiente, no puede completarse hasta que el receptor no haya consumido los anteriores bloques. Es decir, se podría estar consumiendo mucha memoria en el sistema de paso de mensajes (SPM) por mensajes en tránsito pendientes cuya recepción se ve retrasada.

Escribe una solución en la cual cada proceso inicia sus envíos y recepciones y después espera a que se completen todas las recepciones antes de iniciar el primer consumo de un bloque recibido. De esta forma, todos los mensajes pueden transferirse potencialmente de forma simultánea. Se debe intentar que la transmisión y la producción de bloques sean lo más simultáneas posible.

Suponer:

- Cada proceso puede almacenar como mínimo `2*N` bloques en su memoria local.
- El orden de recepción o de consumo de los bloques es indiferente.

14.2. Solución

Para implementar esta solución, cada proceso realiza los siguientes pasos:

1. Produce `N-1` bloques y los envía a los otros procesos utilizando `i_send`.
2. Recibe `N-1` bloques utilizando `receive`, asegurándose de completar todas las recepciones antes de comenzar a consumir los bloques.
3. Consume los bloques recibidos en cualquier orden.

```
process P[ i : 1..N ] ;
var
    bloquesEnviados[N-1] : TipoBloque;  // Bloques producidos para enviar
    bloquesRecibidos[N-1] : TipoBloque; // Bloques recibidos
    j : integer;                        // Iterador
begin
    // Paso 1: Producir y enviar bloques
    for j := 1 to N do
```

```
        if j != i then begin
            ProducirBloque(bloquesEnviados[j]); // Producir un bloque
            i_send(bloquesEnviados[j], P[j]);   // Enviar al proceso P[j]
        end
    end

    // Paso 2: Recibir bloques
    for j := 1 to N do
        if j != i then
            receive(bloquesRecibidos[j], P[j]); // Recibir bloque de P[j]
        end
    end

    // Paso 3: Consumir bloques
    for j := 1 to N do
        if j != i then
            ConsumirBloque(bloquesRecibidos[j]); // Consumir bloque
                recibido
        end
    end
end
```

Explicación de la solución:

- **Producción y envío simultáneos:** Cada proceso genera los bloques para los otros `N-1` procesos de manera concurrente, enviándolos de inmediato mediante `i_send`. Esto permite que la producción y la transmisión sean simultáneas y aprovechen al máximo los recursos del sistema.
- **Recepción antes de consumo:** Cada proceso espera a recibir todos los bloques de los otros procesos antes de comenzar a consumirlos. Esto reduce la memoria ocupada en el sistema de paso de mensajes, ya que los mensajes en tránsito se procesan rápidamente.
- **Orden indiferente:** Dado que no se requiere un orden específico de consumo, los bloques pueden procesarse en cualquier secuencia tras completar las recepciones.

Ventajas de esta solución:

- **Minimización del uso de memoria del SPM:** Al completar las recepciones antes de comenzar el consumo, los mensajes en tránsito pendientes se reducen significativamente.
- **Simultaneidad:** Producción, transmisión y recepción ocurren de forma paralela, aprovechando la capacidad del sistema.
- **Simplicidad:** La solución es sencilla y asegura que no se producen interbloqueos, ya que las recepciones son bloqueantes y se manejan de manera ordenada.

3.4. Relacion 4

GII-ADE-M. Relación de problemas. Tema 4. 13/12/2024

91. Dado el conjunto de tareas periódicas y sus atributos temporales que se indica en la tabla de aquí abajo, determinar si se puede planificar el conjunto de dichas tareas utilizando un esquema de planificación basado en planificación cíclica. Diseña el plan cíclico determinando el marco secundario, y el entrelazamiento de las tareas sobre un cronograma.

Tarea	C_i	T_i	D_i
T1	10	40	40
T2	18	50	50
T3	10	200	200
T4	20	200	200

92. El siguiente conjunto de tareas periódicas se puede planificar con ejecutivos cíclicos. Determina si esto es cierto calculando el marco secundario que debería tener. Dibuja el cronograma que muestre las ocurrencias de cada tarea y su entrelazamiento. ¿Cómo se tendría que implementar? (escribe el pseudo-código de la implementación)

Tarea	C_i	T_i	D_i
T1	2	6	6
T2	2	8	8
T3	3	12	12

93. Comprobar si el conjunto de procesos periódicos que se muestra en la siguiente tabla es planificable con el algoritmo RMS utilizando el test basado en el factor de utilización del tiempo del procesador. Si el test no se cumple, ¿debemos descartar que el sistema sea planificable?

Tarea	C_i	T_i
T1	9	30
T2	10	40
T3	10	50

94. Considérese el siguiente conjunto de tareas compuesto por tres tareas periódicas:

Tarea	C_i	T_i
T1	10	40
T2	20	60
T3	20	80

    Comprueba la planificabilidad del conjunto de tareas con el algoritmo RMS utilizando el test basado en el factor de utilización. Calcular el hiperperiodo y construir el correspondiente cronograma.

95. Comprobar la planificabilidad y construir el cronograma de acuerdo al algoritmo de planificación RMS del siguiente conjunto de tareas periódicas.

Tarea	C_i	T_i
T1	20	60
T2	20	80
T3	20	120

96. Determinar si el siguiente conjunto de tareas puede planificarse con la política de planificación RMS y con la política EDF, utilizando los tests de planificabilidad adecuados para cada uno de los dos casos. Comprobar también la planificabilidad en ambos casos construyendo los dos cronogramas.

Tarea	C_i	T_i
T1	1	5
T2	1	10
T3	2	20
T4	10	20
T5	7	100

97. Describe razonadamente si el siguiente conjunto de tareas puede planificarse o no puede planificarse en un sistema monoprocesador usando un ejecutivo cíclico o usando algún algoritmo basado en prioridades estáticas o dinámicas.

Tarea	C_i	T_i
T1	1	5
T2	1	10
T3	2	10
T4	11	20
T5	5	100

Problemas adicionales:

98. Para el conjunto de tareas cuyos datos se muestran más abajo, se pide:
    - Dibujar el gráfico de ejecución y obtener el tiempo de respuesta de cada tarea
    - Determinar, mediante inspección del gráfico anterior, cuántas veces interfiere la tarea τ_1 a la tarea τ_3 durante un intervalo temporal dado por el tiempo de respuesta de esta última tarea
    - Hacer lo mismo que en (b) pero para las tareas τ_1 y τ_2

Tarea	C_i	T_i	D_i
T1	1	3	2
T2	3	6	5
T3	2	13	13

99. Verificar la planificabilidad del siguiente conjunto de tareas utilizando para ello el algoritmo del "primero el del tiempo límite más cercano" (EDF)

Tarea	C_i	T_i
T1	1	4
T2	2	6
T3	3	8

100. Verificar la planificabilidad utilizando el algoritmo EDF de asignación dinámica de prioridades a las tareas y construir el diagrama de ejecución de tareas del siguiente conjunto:

Tarea	C_i	D_i	T_i
T1	2	5	6
T2	2	4	8
T3	4	8	12

101. Verificar la planificabilidad del conjunto de tareas descrito en el ejercicio 100 utilizando el algoritmo del "plazo de respuesta máximo (D)" (algoritmo *deadline monotonic* o DM)

102. Indicar cuáles de las siguientes afirmaciones son correctas respecto de los algoritmos para resolver el problema de *inversión de prioridad* de las tareas en sistemas de tiempo real de *misión crítica*:

    (a) Suponiendo que las tareas se planifican con el protocolo de *herencia de prioridad*: la prioridad heredada por una tarea sólo se mantiene mientras dicha tarea esté utilizando un recurso compartido con otra tarea más prioritaria.

    (b) Con el protocolo de *techo de prioridad*, cuando una tarea adquiere un recurso no puede verse interrumpida, hasta que termine su ejecución, por otras tareas que se activan después que ésta y que vayan a utilizar en el futuro un recurso con límite de prioridad igual o inferior.

    (c) Con el protocolo de techo de prioridad (PPP), una tarea no puede comenzar a ejecutarse si no están libres todos los recursos que va a utilizar durante su primer ciclo.

    (d) Si consideramos una tarea periódica que utilice el protocolo de techo de prioridad inmediato para cambiar su prioridad dinámica cuando accede a recursos, siempre se cumplirá que dicha tarea no puede ser interrumpida por otra menos prioritaria que ella.

    (e) Con el protocolo de techo de prioridad las tareas más prioritarias del sistema pueden ser interrumpidas durante cada ciclo de su ejecución como máximo 1 vez cuando acceden a recursos que comparten con otras tareas menos prioritarias.

    (f) El protocolo de techo de prioridad original producirá siempre tiempos de respuesta menores para las tareas que el algoritmo de *herencia de prioridad*.

103. Calcular la utilización máxima del procesador que se puede asignar al *Servidor Esporádico* para garantizar la planificabilidad del siguiente conjunto de tareas periódicas utilizando RM

τ_1	1	5
τ_2	2	8

104. Calcular la utilización máxima del procesador que puede ser asignada al *Servidor Diferido* (SD) para garantizar la planificabilidad del conjunto de tareas periódicas dado en el ejercicio 103 anterior

105. Junto con las tareas periódicas que se muestran en el ejercicio 103, definir un plan para planificar las siguientes tareas aperiódicas utilizando un *SD* (tarea sondeante) que posea una utilización máxima del tiempo del procesador y prioridad intermedia:

	t_a	C_i
J_1	2	2
J_2	7	2
J_3	17	1

106. Resolver ahora el mismo problema de planificación descrito en el ejercicio 105 utilizando ahora un *Servidor Esporádico* que tenga una utilización máxima y prioridad intermedia

107. Resolver el mismo problema de planificación descrito en el ejercicio 105 utilizando ahora un *Servidor Diferido* que tenga utilización máxima y prioridad intermedia

108. Utilizar un *Servidor Esporádico* con capacidad $\mathrm{C_s} = 2$ y periodo $\mathrm{T_s} = 6$ para planificar las siguientes tareas:

	C_i	T_i
τ_1	1	4
τ_2	2	6
	a_i	C_i
J_1	2	2
J_2	5	1
J_3	10	2

4 Actividades Extras

4.1. Criba de Eratóstenes

Para ver el archivo pdf pinche aquí.

4.2. Demostración de las Propiedades de un Algoritmo de Exclusión Mutua

Para ver el archivo pdf pinche aquí.

5 Referencias

- Diapositivas de clase.