Ingeniería Informática + ADE

Universidad de Granada (UGR)

Autor: Ismael Sallami Moreno

Asignatura: Apuntes Sistemas Concurrentes y Distribuidos (SCD)

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

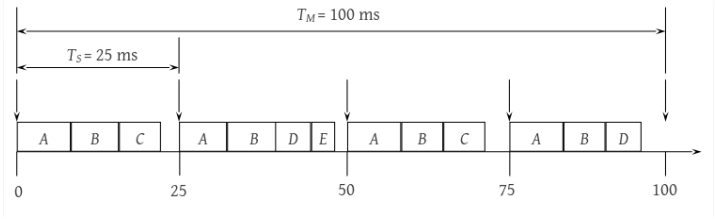

Figura 1: Planificación cíclica

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:

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

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