Doble Grado Ingeniería Informática y Administración y Dirección de Empresas

Sistemas Concurrentes y Distribuidos

Relacion de Ejercicios Tema 1

Ismael Sallami Moreno

27 de noviembre de 2024

Índice

1. Ejercicio 1

Descripción

Considerar el siguiente fragmento de programa para 2 procesos P_1 y P_2. 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;                         Process P2;
    var i: integer;                         var j: integer;
begin                               begin
    for i:= 1 to 2 do begin                 for i:= 1 to 2 do begin
        x:= x + 1;                                  x:= x + 1;
    end                                     end
end
```

Posibles valores de x

Los posibles valores de la variable son: $x = 2, 3, 4$.

- Cada uno de los dos procesos P_1, P_2 hace 2 lecturas $(L_{11}, L_{12}, L_{21}, L_{22})$ y 2 escrituras.
- Cada proceso incrementa x en $+1$, dos veces partiendo de 0, por lo que $x \neq 2$.
- Se realizan 4 incrementos totales, de modo que $x \neq 4$ tampoco puede ser evitado.

Secuencia de operaciones

x	P_1	P_2	x	P_1	P_2	x	P_1	P_2
0	L_{11}	-	0	L_{11}	-	0	L_{11}	-
0	-	L_{21}	0	-	L_{21}	1	E_{11}	-
1	E_{11}	-	1	-	L_{21}	1	-	E_{21}
1	-	E_{21}	1	-	E_{21}	2	-	E_{21}
1	L_{12}	-	1	L_{12}	-	2	L_{12}	-
1	-	L_{22}	2	E_{12}	-	3	E_{12}	-
2	E_{12}	-	2	-	L_{22}	3	-	L_{22}
2	-	E_{22}	4	-	E_{22}	4	-	E_{22}

Cuadro 1: Secuencia de operaciones realizadas por P_1 y P_2 y el valor de x en cada paso.

2. Ejercicio 2

Descripción

¿Cómo se podría hacer la copia del fichero f en otro g, de forma concurrente, utilizando la instrucción concurrente `cobegin-coend`?

- Los archivos son una secuencia de ítems de un tipo arbitrario T, ya abiertos para lectura (f) y escritura (g).
- Para leer un ítem de f, se usa la función `leer(f)`.
- Para verificar si se han leído todos los ítems, se utiliza la función `fin(f)`.
- Para escribir un dato x en g, se utiliza `escribir(g, x)`.
- El orden de los ítems en g debe coincidir con f.

Código Concurrente

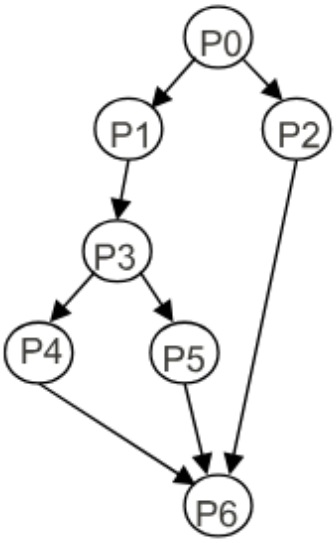
```
process Correcto ;
var v_ant, v_sig : T ;
begin
    v_sig := leer(f) ;
    while not fin(f) do begin
        v_ant := v_sig ;
        cobegin
            escribir(g, v_ant);
            v_sig := leer(f) ;
        coend
    end
end
```

Ejercicio 3

Construir programas concurrentes utilizando las instrucciones `cobegin-coend` y `fork-join`, que correspondan a los siguientes grafos de precedencia.

Grafo (a)

Grafo de sincronización



Bloque de Pseudocódigo 1

```
begin
    P0 ; fork P2 ;
    P1 ; P3 ; fork P5 ; P4 ;
    join P2 ; join P5 ;
    P6 ;
end
```

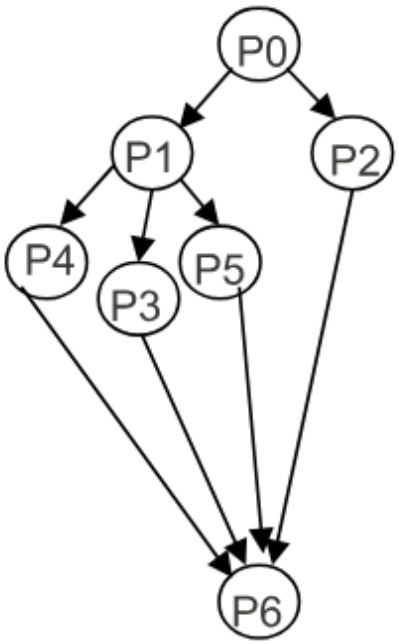
Bloque de Pseudocódigo 2

```
begin
    P0 ;
    cobegin
```

```
        begin
            P1 ; P3 ;
            cobegin P4 ; P5 ; coend
        end
        P2 ;
    coend
    P6 ;
end
```

Grafo (b)

Grafo de sincronización



Bloque de Pseudocódigo 1

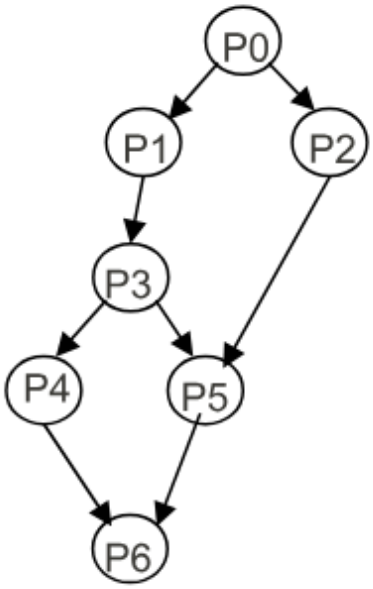
```
begin
    P0 ; fork P2 ;
    P1 ; fork P3 ; fork P5 ;
    P4 ;
    join P2 ; join P3 ; join P5 ;
    P6 ;
end
```

Bloque de Pseudocódigo 2

```
begin
    P0 ;
    cobegin
        begin
            P1 ;
            cobegin P3 ; P4 ; P5 ; coend
        end
        P2 ;
    coend
    P6 ;
end
```

Grafo (c)

Grafo de sincronización



Bloque de Pseudocódigo 1

```
begin
    P0 ; fork P2 ;
    P1 ;
    P3 ; fork P4 ;
    join P2 ;
    P5 ;
    join P4 ;
    P6 ;
end
```

Bloque de Pseudocódigo 2

```
begin
    P0 ;
    cobegin
        begin P1 ; P3 ; end
        P2 ;
    coend
    cobegin P4 ; P5 ; coend
    P6 ;
end
```

Bloque de Pseudocódigo 3

```
begin
    P0 ;
    cobegin
        cobegin
            begin
                P1 ; P3 ; P4 ;
            end
            P2 ;
```

```
        coend
        P5 ;
    coend
    P5 ; P6 ;
end
```

3. Ejercicio 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 la hora (cada Kwh consumido se cuenta como un impulso) e imprimir este número en un dispositivo al final de la hora. Para ello se dispone de un programa concurrente con 2 procesos: un proceso acumulador (que cuenta el número de impulsos recibidos) y un proceso escritor (que los imprime en la impresora). En la variable común a los 2 procesos se lleva la cuenta de los impulsos. El proceso acumulador, después de ejecutar la función `Espera_impulso` para esperar a que se produzca un impulso, incrementa la variable. El proceso escritor, después de llamar a `Espera_fin_hora`, hace esperar a que termine una hora. El código de los procesos de este programa podría ser el siguiente:

```
var contador: compartida;
var n: integer; { contabliliza 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 la hora y la variable `n` vale `k`. Después se produce un impulso y el escritor se despierta al fin del período de una hora.

Describir lo que puede ocurrir con la intercalación de las instrucciones (1), (2), y (3) a partir de ese momento, indicando cuáles de ellas son correctas y cuáles incorrectas (las incorrectas son aquellas en las que el valor de `n` no se contabiliza).

3.1. Resolución

- Suponemos una variable ficticia `OUT` que se crea como resultado de la instrucción `write(n)` (2) que contiene el valor impreso (éste pasa así a formar parte del estado).
- En el estado inicial se cumple `n == k`.
- Solo serán correctos los entrelazamientos de instrucciones atómicas del programa que sean compatibles con el estado final: `OUT + n == k + 1`.
- Los posibles entrelazamientos son: (a) 1,2,3, (b) 2,1,3 y (c) 2,3,1.

	(a)		(b)		(c)		(d)	
inst.	n	OUT	inst.	n	OUT	inst.	n	OUT
-	k	-	-	k	-	-	k	-
n:=n+1	k+1	-	write(n)	k	k	write(n)	k	k
write(n)	k+1	k+1	n:=n+1	k+1	k	n:=0	0	k
n:=0	0	k+1	n:=0	0	k+1	n:=n+1	1	k

4. Ejercicio 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
end
```

4.1. Resolución

- Se podría paralelizar calculando de forma independiente las filas, columnas, ..., de la matriz resultado
- Utilizamos 3 procesos concurrentes `CalcularFila (i:1..3)`:

```
var a, b, c : array [1..3,1..3] of real ;
process CalcularFila[ i : 1..3 ] ;
    var j, k : integer ;
    begin
        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
end
```

5. Ejercicio 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
```

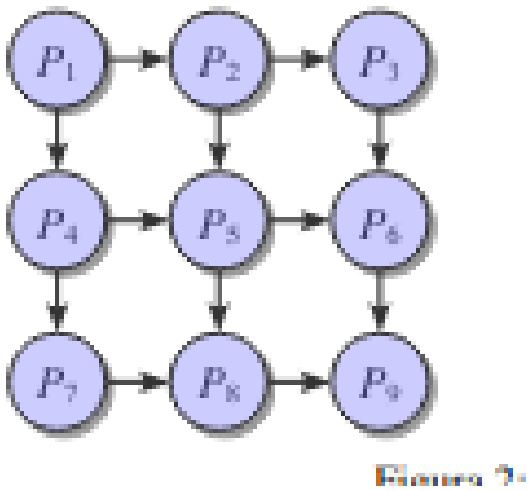


Figura 1: Grafo de sincronización de actividades

5.1. Grafo de sincronización de actividades

5.2. Suposiciones

- 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.

5.3. Resolución

- Se utilizará un vector de valores lógicos
- Dicho vector ha de inicializarse de una sola vez antes de la siguiente iteración de los bucles
- Solución: al terminar el proceso 9 se inicializará el vector

```
{ compartido entre todas las tareas }
var finalizado : array [1..9] of boolean := (false, ..., false) ;

procedure EsperarPor( i : integer )
begin
  while not finalizado[i] do begin; end
end

procedure Acabar( i : integer )
  var j : integer ;
begin
  if i < 9 then
    finalizado[i] := true ;
  else for j := 1 to 9 do
    finalizado[j] := false ;
end
```

6. Ejercicio 10

Obtener la poscondición adecuada para convertir los siguientes fragmentos de código 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 ; {}

6.1. Resolución

Se resuelve aplicando directamente el axioma de asignación basado en la sustitución textual de {P} por $\{P\}_e^x$ en la precondición de los triples:

1. $\{i < 10\}; i = 2 * i + 1; \{i < 21\}$ puesto que: $\{i < 21\}_{2*i+1}^{i} \equiv \{2 * i + 1 < 21\} \equiv \{i < 10\}$
2. $\{i > 0\}; i = i - 1; ; \{i > -1\}$
3. $\{i > j\}; i = i + 1; ; \{i > j + 1\}; j = j + 1; \{i > j\}$
4. $\{F\}; a = a + 7; ; \{V\}$
5. $\{V\}; i = 3; ; \{i = 3\}; j = 2 * i; \{j = 6\}$
6. $\{V\}; c = a + b; ; \{c = a + b\}; c = c/2; \{c = (a + b)/2\}$

7. Ejercicio 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) $\{a > 0\}; a = a - 7; ; \{a > -6\}$

7.1. Resolución

$i, x, a \in \mathbb{Z}$

1. $\{i > 0\}; i = i - 1; ; \{i + 1 > 0\} \Rightarrow \{i \geq 0\}$
2. $\{x \geq 7\}; x = x + 3; ; \{x \geq 10\} \Rightarrow \{x \geq 9\}$
3. $\{i < 9\}; i = 2 * i + 1; ; \{i < 19\} \Rightarrow \{i \leq 20\}$
4. $\{a > 0\}; a = a - 7; ; \{a > -7\}$ NOT $\Rightarrow \{a > -6\}$

7.2. Anotaciones de la resolución

Por ejemplo, en el caso del apartado 1, calculamos la antigua $\{i = i + 1\}$ y la sustituimos en la precondición, asegurándonos de que se cumple.

8. Ejercicio 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) $\{P\}\ C\ \{Q \wedge P\}$

8.1. Resolución

El triple $\{P\}\ C\ \{Q\}$ es demostrable,

1. $\{P\}\ C\ \{Q \vee P\}$ también lo es por debilitamiento de la poscondición.
2. $\{P \wedge D\}\ C\ \{Q\}$ también lo es por fortalecimiento de la precondición.
3. $\{P \vee D\}\ C\ \{Q\}$ no lo es porque se debilita la precondición.
4. $\{P\}\ C\ \{Q \vee D\}$ lo mismo que (1).
5. $\{P\}\ C\ \{Q \wedge P\}$ no lo es porque se fortalece la poscondición.

9. Ejercicio 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) $\{P\}\ C\ \{Q \vee P\}$

9.1. Resolución

1. $\{P \wedge D\}\ C\ \{Q\}$
2. $\{P \vee D\}\ C\ \{Q\}$ no se puede demostrar porque se debilita la precondición.
3. $\{P\}\ C\ \{Q \vee D\}$
4. $\{P\}\ C\ \{Q \vee P\}$

10. Ejercicio 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.

10.1. Resolución

```
int x = 5, y = 2; cobegin <x = x + y >;<y = x * y >coend;
```

(a) Considerando operaciones atómicas (con los símbolos <>)

a) $\{x == 5 \land y == 2\} < x = x + y >; < y = x * y > \{x == 7 \land y == 14\}$

b) $\{x == 5 \land y == 2\} < y = x * y >; < x = x + y > \{x == 15 \land y == 10\}$

(b) Sin considerarlas operaciones atómicas (quitando los símbolos <>)

a) Los valores de (a) y además $\{x == 7 \land y == 10\}$

11. Ejercicio 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) $\{x \geq 0\} < y = y + 3 > \{y \geq 3\}$

(e) $\{x \text{ es impar}\} < y = x + 1 > \{y \text{ es par}\}$

11.1. Resolución

Regla de no interferencia de predicado

$$\{P\} \text{ con acción atómica } \{P \land \text{pre}(a)\} < a > \{P\}$$

11.1.1. Tabla de Interferencias

Condición inicial	Acción	Condición final	Interfiere	Justificación
$\{x \geq 0\}$	$< x = x + 3 >$	$\{x \geq 3\}$	Sí	$x - 2$ puede violar $x \geq 3$.
$\{x \geq 0\}$	$< x = x + 3 >$	$\{x \geq 0\}$	No	$x - 2$ mantiene $x \geq 0$.
$\{x \geq 7\}$	$< x = x + 3 >$	$\{x \geq 10\}$	Sí	$x - 2$ puede violar $x \geq 10$.
$\{y \geq 0\}$	$< y = y + 3 >$	$\{y \geq 3\}$	No	Las variables x y y son disjuntas y no interfieren entre sí.
$\{x \text{ es impar}\}$	$< y = x + 1 >$	$\{y \text{ es par}\}$	No	$x - 2$ no afecta la paridad de x, y por ende tampoco la de y.

Cuadro 2: Evaluación de interferencia entre condiciones iniciales, acciones y condiciones finales.

11.1.2. Explicación Detallada del Ejercicio 15

El objetivo del ejercicio es verificar si la demostración del triple de Hoare $\{x \geq 2\} < x = x-2 > \{x \geq 0\}$ interfiere con los triples dados. Esto se realiza aplicando la regla de no interferencia de predicado: si una acción modifica el estado del programa de forma que el predicado de otro triple no se cumple, entonces existe interferencia.

Triple Original

$$\{x \geq 2\} < x = x - 2 > \{x \geq 0\}$$

- **Condición inicial**: $x \geq 2$
- **Acción**: $x := x - 2$
- **Condición final**: $x \geq 0$

Análisis Fila por Fila

Primera fila Triple: $\{x \geq 0\} < x = x + 3 > \{x \geq 3\}$

- **Condición inicial**: $x \geq 0$
- **Acción**: $x := x + 3$
- **Condición final**: $x \geq 3$
- **Interferencia**: Sí
- **Justificación**:
  - Si aplicamos $x := x + 3$, la condición final $x \geq 3$ se cumple.
  - Pero si después aplicamos $x := x - 2$, el nuevo valor de x será $x - 2$, lo que puede dar como resultado $x < 3$. Por ejemplo:
    - Si $x = 2$, después de $x := x + 3$, tenemos $x = 5$.
    - Después de $x := x - 2$, tenemos $x = 3$, lo cual aún cumple $x \geq 3$.
    - Pero para valores iniciales más bajos ($x = 0$), no se garantiza $x \geq 3$. Por esto, hay interferencia.

Segunda fila Triple: $\{x \geq 0\} < x = x + 3 > \{x \geq 0\}$

- **Condición inicial**: $x \geq 0$
- **Acción**: $x := x + 3$
- **Condición final**: $x \geq 0$
- **Interferencia**: No
- **Justificación**:
  - La condición inicial ($x \geq 0$) no cambia su validez después de $x := x - 2$.
  - Por ejemplo:
    - Si $x = 2$, después de $x := x + 3$, $x = 5$.
    - Aplicando $x := x - 2$, $x = 3$, y aún $x \geq 0$. La condición final sigue cumpliéndose.

Tercera fila Triple: $\{x \geq 7\} < x = x + 3 > \{x \geq 10\}$

- **Condición inicial**: $x \geq 7$
- **Acción**: $x := x + 3$
- **Condición final**: $x \geq 10$
- **Interferencia**: Sí
- **Justificación**:
  - Si $x := x + 3$, la condición final $x \geq 10$ se cumple.
  - Pero al aplicar $x := x - 2$, el valor de x disminuye y puede no cumplir $x \geq 10$.
  - Por ejemplo:
    - Si $x = 7$, después de $x := x + 3$, tenemos $x = 10$.
    - Después de $x := x - 2$, $x = 8$, lo cual viola $x \geq 10$.

Cuarta fila Triple: $\{y \geq 0\} < y = y + 3 > \{y \geq 3\}$

- **Condición inicial**: $y \geq 0$
- **Acción**: $y := y + 3$
- **Condición final**: $y \geq 3$
- **Interferencia**: No
- **Justificación**:
  - La acción modifica la variable y, pero el triple original afecta solo a x.
  - Dado que las variables x e y son disjuntas, no hay interferencia.

Quinta fila Triple: $\{x \text{ es impar}\} < y = x + 1 > \{y \text{ es par}\}$

- **Condición inicial**: x es impar
- **Acción**: $y := x + 1$
- **Condición final**: y es par
- **Interferencia**: No
- **Justificación**:
  - Aunque y depende de x, la acción $x := x - 2$ no afecta la paridad de x.
  - Ejemplo:
    - Si $x = 3$ (x es impar), entonces $y := x + 1 = 4$ (y es par).
    - Si aplicamos $x := x - 2$, $x = 1$ sigue siendo impar, y $y = x + 1 = 2$ sigue siendo par.

Condición inicial	Acción	Condición final	Interfiere	Justificación
$\{x \geq 0\}$	$< x = x + 3 >$	$\{x \geq 3\}$	Sí	Reducir $x-2$ después de la acción puede dar como resultado $x < 3$, lo que viola la condición final.
$\{x \geq 0\}$	$< x = x + 3 >$	$\{x \geq 0\}$	No	Reducir $x-2$ después de la acción no afecta la condición inicial, que se sigue cumpliendo.
$\{x \geq 7\}$	$< x = x + 3 >$	$\{x \geq 10\}$	Sí	Reducir $x - 2$ después de la acción puede dar como resultado $x < 10$, lo que viola la condición final.
$\{y \geq 0\}$	$< y = y + 3 >$	$\{y \geq 3\}$	No	Las variables x y y son independientes y no interfieren entre sí.
$\{x \text{ es impar}\}$	$< y = x + 1 >$	$\{y \text{ es par}\}$	No	La operación no altera la paridad de x ni de y, que sigue siendo par.

Cuadro 3: Análisis de interferencia entre condiciones iniciales, acciones y condiciones finales.

Resumen Final

12. Ejercicio 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 instrucciones 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.

12.1. Resolución

Resolución Ejercicio 16

$\{x == 0\}$

– Inicio cobegin

$\{x == 0 \lor x == b \lor x == c \lor x == b + c\}$

$\langle x = x + a\rangle \parallel$

$\{x == a \lor x == a + b \lor x == a + c \lor x == a + b + c\}$

$\{x == 0 \lor x == a \lor x == c \lor x == a + c\}$

$\langle x = x + b\rangle \parallel$

$\{x == b \lor x == b + a \lor x == b + c \lor x == a + b + c\}$

$\{x == 0 \lor x == b \lor x == a \lor x == a + b\}$

$\langle x = x + c\rangle \parallel$

$\{x == c \lor x == c + b \lor x == c + a \lor x == a + b + c\}$

– Fin coend

– Aplicando regla de la concurrencia

$\{x == a + b + c\}$

12.2. Resolución Detallada

Se analiza la ejecución paso a paso considerando la regla de concurrencia para instrucciones atómicas. El análisis utiliza asertos intermedios para demostrar que la poscondición final se cumple.

$\{x == 0\}$

Inicio del bloque `cobegin`

Primera instrucción atómica: $\langle x = x + a\rangle$

$\{x == 0 \lor x == b \lor x == c \lor x == b + c\} \quad \Rightarrow \quad \{x == a \lor x == a + b \lor x == a + c \lor x == a + b + c\}$

Segunda instrucción atómica: $\langle x = x + b\rangle$

$\{x == 0 \lor x == a \lor x == c \lor x == a + c\} \quad \Rightarrow \quad \{x == b \lor x == b + a \lor x == b + c \lor x == a + b + c\}$

Tercera instrucción atómica: $\langle x = x + c\rangle$

$\{x == 0 \lor x == b \lor x == a \lor x == a + b\} \quad \Rightarrow \quad \{x == c \lor x == c + b \lor x == c + a \lor x == a + b + c\}$

Fin del bloque `coend`

Aplicando la regla de concurrencia para instrucciones atómicas:

Cada una de las tres instrucciones contribuye de forma independiente al incremento de x.

Por lo tanto, al terminar las tres, el valor de x será: $x = a + b + c$.

$\{x == a + b + c\}$

12.3. Conclusión

Se ha demostrado que el triple $\{x == 0\}$ `cobegin` $\langle x = x+a\rangle \parallel \langle x = x+b\rangle \parallel \langle x = x+c\rangle$ `coend` $\{x == a + b + c\}$ es válido, utilizando la lógica de asertos y la regla de composición concurrente.

13. Ejercicio 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;`

13.1. Estudio de Valores Finales

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;`

13.2. Solución y Explicación Detallada

```
// {x = C1, y = C2}
int x = C1;
// {x = C1, y = C2}
int y = C2;
// {x = C1, y = C2}
x = x + y;
// {x = C1 + C2, y = C2}
y = x * y;
// {x = C1 + C2, y = (C1 + C2) * C2}
x = x - y;
// {x = C1 + C2 - (C1 + C2) * C2, y = (C1 + C2) * C2}
```

- **Inicialización de x:**
  - **Código:** `int x = C1;`
  - **Aserto:** // {x = C1, y = C2}
  - **Explicación:** Inicializamos x con $C1$. En este momento, x es igual a $C1$ y y no está inicializado.
- **Inicialización de y:**
  - **Código:** `int y = C2;`
  - **Aserto:** // {x = C1, y = C2}
  - **Explicación:** Inicializamos y con $C2$. Ahora, x es igual a $C1$ y y es igual a $C2$.
- **Primera Operación (Suma):**
  - **Código:** `x = x + y;`

- **Aserto:** // {x = C1 + C2, y = C2}
- **Explicación:** Sumamos y a x y guardamos el resultado en x. Después de esta operación, x es igual a $C1 + C2$ y y sigue siendo $C2$.

- **Segunda Operación (Multiplicación):**
  - **Código:** `y = x * y;`
  - **Aserto:** // {x = C1 + C2, y = (C1 + C2) * C2}
  - **Explicación:** Multiplicamos x por y y guardamos el resultado en y. Después de esta operación, y es igual a $(C1 + C2) * C2$ y x sigue siendo $C1 + C2$.
- **Tercera Operación (Resta):**
  - **Código:** `x = x - y;`
  - **Aserto:** // {x = C1 + C2 - (C1 + C2) * C2, y = (C1 + C2) * C2}
  - **Explicación:** Restamos y de x y guardamos el resultado en x. Después de esta operación, x es igual a $C1 + C2 - (C1 + C2) * C2$ y y sigue siendo $(C1 + C2) * C2$.

13.3. Resultado Final

Después de ejecutar todas las operaciones, los valores finales de las variables son:

- $x = C1 + C2 - (C1 + C2) * C2$
- $y = (C1 + C2) * C2$

14. Pregunta 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.

14.1. Resolución 24

Sea x = y es invariante

- $\langle x = x - 1 \rangle \mid \langle x = x + 1 \rangle \parallel \langle y = y + 1 \rangle \mid \langle y = y - 1 \rangle$
- $\{x = y\}$
- $\langle x = x - 1 \rangle$
- $\{x + 1 = y\} \rightarrow \{x = y - 1\}$
- $\langle x = x + 1 \rangle$
- $\{x - 1 = y - 1\} \rightarrow \{x = y\}$
- $\{x = y\}$
- $\langle y = y + 1 \rangle$
- $\{x = y - 1\}$
- $\langle y = y + 1 \rangle$
- $\{x = (y + 1) - 1\} \rightarrow \{x = y\}$

15. Ejercicio 25

Usando la regla de la conjunción, demostrar que:

$$\{i > 2\}; i := 2 * i; \{i > 4\}$$

Regla de la conjunción:

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

15.1. Resolución 25 Nº1

En este caso:

$$P_1 : i > 2 Q_1 : i > 4 S : i := 2 * i$$

Primero verificamos la corrección de P_1 con respecto a Q_1:

$$\{i > 2\}\ i := 2 * i\ \{i > 4\}$$

Para demostrar esto, consideramos el estado después de la asignación:

$$i := 2 * i \implies i = 2 * i$$

Dado que $i > 2$, multiplicando por 2 obtenemos:

$$2 * i > 4$$

Por lo tanto, $i > 4$.

Verificamos también P_2 y Q_2 en este caso no necesitamos otro, ya que:

$$P_2 : i > 2 Q_2 : i > 4$$

Es evidente que

$$i > 2 \Rightarrow i > 4$$

Por lo tanto, con la regla de la conjunción, demostramos:

$$\{i > 2\}\ i := 2 * i\ \{i > 4\}$$

15.2. Resolución Detallada Nº2

Usando la regla de la conjunción, demostrar que

$$\{i > 2\}\ i = 2 * i\ \{i > 4\}$$

Aunque se podría demostrar de forma directa mediante el axioma de asignación, vamos a demostrarlo mediante la regla de la conjunción. Para ello, consideramos los siguientes triples:

$$\{V\}\ i = 2 * i\ \{i = 2 * i\}$$

$$\{i > 2\}\ i = 2 * i\ \{i > 2\}$$

Estos son directamente ciertos por el axioma de asignación. Por tanto, podemos aplicar la regla de la conjunción, llegando a que el siguiente triple es cierto:

$$\{i > 2\} \equiv \{V \wedge i > 2\}\ i = 2 * i\ \{i > 2 \wedge i = 2 * i\} \equiv \{i > 4\}$$

16. Explicación de los Axiomas Ejercicio 25

Axioma de Asignación

El axioma de asignación se utiliza para validar los triples de Hoare de la forma $\{P\}\ x := E\ \{Q\}$, donde:

- P es la precondición.
- $x := E$ es la asignación.
- Q es la postcondición.

Para que este triple sea válido, la postcondición Q debe ser verdadera cuando x toma el valor de la expresión E. Matemáticamente, esto se expresa como:

$$\{P\}\ x := E\ \{Q\} \text{ es válido si y solo si } P \implies Q[E/x]$$

Regla de la Conjunción

La regla de la conjunción permite combinar dos triples de Hoare para llegar a una nueva conclusión. Si tienes dos triples $\{P_1\}\ S\ \{Q_1\}$ y $\{P_2\}\ S\ \{Q_2\}$, puedes concluir que $\{P_1 \wedge P_2\}\ S\ \{Q_1 \wedge Q_2\}$.

En otras palabras, si puedes demostrar que S satisface tanto Q_1 partiendo de P_1 como Q_2 partiendo de P_2, entonces S también satisface $Q_1 \wedge Q_2$ partiendo de $P_1 \wedge P_2$.

17. Ejercicio 26

26. Se dan los siguientes triples de Hoare:
$\{\mathrm{j} > 1\}\ i = i + 2;\ j = j + 3;\ \{\mathrm{j} > 4\}$
$\{\mathrm{i} > 2\}\ i = i + 2;\ j = j + 3;\ \{\mathrm{i} > 4\}$
Demostrar que estos triples implican que $\{\mathrm{j} > 1,\ \mathrm{i} > 2\}\ i = i + 2;\ j = j + 3;\ \{\mathrm{j} > 4,\ \mathrm{i} > 4\}$. ¿Qué regla se debe utilizar para la demostración?

17.1. Resolución 26

Utilizaremos la **Regla de la Conjunción** para demostrar que

$$\{j > 1, i > 2\}\mathrm{i} = \mathrm{i} + 2;\mathrm{j} = \mathrm{j} + 3;\{j > 4, i > 4\}.$$

La regla de la conjunción se enuncia como:

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

Aquí:

$$P_1 : j > 1 Q_1 : j > 4 P_2 : i > 2 Q_2 : i > 4 S : i := i + 2; j := j + 3$$

Primero verificamos la corrección de P_1 con respecto a Q_1:

$$\{j > 1\}\ i := i + 2; j := j + 3\ \{j > 4\}$$

Para demostrar esto, consideramos el estado después de la asignación:

$$j := j + 3 \implies j = j + 3$$

Dado que $j > 1$, sumando 3 obtenemos:

$$j + 3 > 4$$

Por lo tanto, $j > 4$.

Verificamos también P_2 con respecto a Q_2:

$$\{i > 2\}\ i := i + 2; j := j + 3\ \{i > 4\}$$

Consideramos el estado después de la asignación:

$$i := i + 2 \implies i = i + 2$$

Dado que $i > 2$, sumando 2 obtenemos:

$$i + 2 > 4$$

Por lo tanto, $i > 4$.

Con la regla de la conjunción, combinamos ambos resultados:

$$\frac{\{j > 1\}S\{j > 4\}, \{i > 2\}S\{i > 4\}}{\{j > 1 \wedge i > 2\}S\{j > 4 \wedge i > 4\}}$$

Finalmente, demostramos que:

$$\{j > 1, i > 2\}\ i := i + 2; j := j + 3\ \{j > 4, i > 4\}$$

18. Ejercicio 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` $x := -a\{x \geq 0, x^2 = a^2\}$

18.1. Resolucion Nº1 del Ejercicio 28

$$\{x^2 \leq B\}\ S_1\ \{x^2 \leq B\}, \quad \{x^2 \leq B\}\ S_2\ \{x^2 \leq B\}$$

$$\{x^2 \leq B\} \quad \text{if } (B) \text{ then } S_1 \text{ else } S_2 \text{ endif } \{x^2\}$$

Directa:

$$\{\forall x\ \{x^2 \leq B\}\ x := a\ \{x = a,\ a > 0\} \rightarrow \{x^2 = a^2\}$$

Inversa (Else):

$$\{\forall x\ \{x^2 \leq a\}\ x := -a\ \{x = -a,\ a \leq 0\} \rightarrow \{x^2 = a^2\}$$

$$\{V\}$$

if (a > 0) then x := a else x := -a ;

$$\{x^2 = a^2\},\ x \geq 0$$

18.2. Resolucion Nº2 del Ejercicio 28

Demostración

Queremos demostrar que la siguiente sentencia tiene la poscondición $\{x \geq 0, x^2 = a^2\}$:

```
if a > 0 then x := a else x := -a
```

Resolución del Ejercicio 28

$$\{V\} \text{ if } a > 0 \text{ then } x := a \text{ else } x := -a\{x \geq 0, x^2 = a^2\}$$

Análisis de la Sentencia

Para demostrar que esta sentencia cumple la poscondición dada, utilizamos las reglas de Hoare y la estructura condicional ïf". Vamos a analizar ambas ramas de la condición:

18.2.1. Caso 1: $a > 0$

Si $a > 0$, entonces la sentencia `x := a` se ejecuta. Debemos demostrar que después de esta asignación, la poscondición $\{x \geq 0, x^2 = a^2\}$ se cumple.

$$\{a > 0\}\ x := a\ \{x \geq 0 \land x^2 = a^2\}$$

- Antes de la asignación, sabemos que $a > 0$. - Después de la asignación, $x = a$, por lo que $x \geq 0$ y $x^2 = a^2$ se cumplen.

Caso 2: $a \leq 0$

Si $a \leq 0$, entonces la sentencia `x := -a` se ejecuta. Debemos demostrar que después de esta asignación, la poscondición $\{x \geq 0, x^2 = a^2\}$ se cumple.

$$\{a \leq 0\}\ x := -a\ \{x \geq 0 \land x^2 = a^2\}$$

- Antes de la asignación, sabemos que $a \leq 0$. - Después de la asignación, $x = -a$, por lo que $x \geq 0$ (ya que $-a \geq 0$ si $a \leq 0$) y $x^2 = a^2$ se cumplen.

Conclusión

Por la regla del ïf", podemos concluir que:

$$\{V\}\ \text{if } a > 0 \text{ then } x := a \text{ else } x := -a\{x \geq 0, x^2 = a^2\}$$

Esto demuestra que la sentencia dada cumple la poscondición $\{x \geq 0, x^2 = a^2\}$.

19. Ejercicio 33

Demostrar la corrección parcial del siguiente fragmento de programa:

```
sum := 0; j := 1;
while(j != c) do
begin
    sum := sum + j; j := j + 1;
end
{sum = c * (c - 1)/2}
```

19.1. Resolución Nº1 Ejercicio 33

Para demostrar la corrección parcial del programa, necesitamos probar que, al final del bucle, la suma es igual a $\frac{c \times (c-1)}{2}$.

19.1.1. Invariante del bucle

Elegimos el invariante del bucle como:

$$I : sum = \frac{j \times (j-1)}{2}$$

Este invariante se elige porque representa la suma de los primeros $j - 1$ números naturales. Es relevante aquí porque, al incrementar j en cada iteración del bucle, queremos acumular la suma de todos los números hasta $j - 1$.

19.1.2. Inicialización

Antes de que el bucle comience, tenemos:

$$sum := 0; \quad j := 1$$

Sustituyendo estos valores en el invariante, obtenemos:

$$I : 0 = \frac{1 \times (1-1)}{2} = 0$$

Esto es cierto, por lo que el invariante se cumple inicialmente.

19.1.3. Mantenimiento

Si el invariante es verdadero antes de una iteración del bucle, debe seguir siendo verdadero después de la iteración. Supongamos que el invariante es verdadero antes de una iteración:

$$sum = \frac{j \times (j-1)}{2}$$

Durante la iteración, se ejecutan las siguientes instrucciones:

$$sum := sum + j; \quad j := j + 1$$

Después de la ejecución de estas instrucciones, los valores se actualizan a:

$$sum = \frac{j \times (j-1)}{2} + j$$
$$j = j + 1$$

Sustituyendo el nuevo valor de j en el invariante, tenemos:

$$sum = \frac{(j-1) \times (j-2)}{2} + j$$
$$= \frac{(j^2 - 3j + 2) + 2j}{2}$$
$$= \frac{j^2 - j + 2}{2}$$

Ajustando correctamente:

$$= \frac{(j^2 - j) + 2}{2}$$
$$= \frac{j(j-1)}{2}$$

Esto demuestra que el invariante se mantiene.

19.1.4. Terminación

El bucle termina cuando $j = c$. En este punto, el invariante se convierte en:

$$sum = \frac{c \times (c-1)}{2}$$

19.2. Resolución Nº2 Ejercicio 33

Demostración con la Regla de Iteración

Para demostrar que la siguiente sentencia tiene la poscondición $\{x \geq 0, x^2 = a^2\}$:
`if` $a > 0$ `then` $x := a$ `else` $x := -a$

19.2.1. Regla de la Iteración

Usamos la regla de la iteración:

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

19.2.2. Identificación de Términos

Sean:

$$I \equiv \sum = \frac{j(j-1)}{2}, \quad j < c$$

$$B \equiv j \neq c$$

$$S \equiv \sum = \sum + j; \quad j = j + 1$$

19.2.3. Prueba del Triple de Hoare

Queremos probar que se cumple el triple:

$$\left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j \neq c \end{array} \right\} \sum = \sum + j; \quad j = j + 1; \left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j < c \end{array} \right\}$$

Para ello, será suficiente con demostrar los siguientes sub-triples y aplicar la regla de composición.

Primer Sub-Triple

$$\left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j \neq c \end{array} \right\} \sum = \sum + j; \left\{ \begin{array}{c} \sum = \frac{(j+1)j}{2} \\ \hline j \neq c \end{array} \right\}$$

Usamos el axioma de asignación:

$$\left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j \neq c \end{array} \right\} \sum = \sum + j; \left\{ \begin{array}{c} \sum + j = \frac{(j+1)j}{2} \\ \hline j \neq c \end{array} \right\} \equiv$$

$$\left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j \neq c \end{array} \right\} \sum = \sum + j; \left\{ \begin{array}{c} \sum = \frac{(j+1)j}{2} - j \\ \hline j \neq c \end{array} \right\} \equiv$$

$$\left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j \neq c \end{array} \right\} \sum = \sum + j; \left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j \neq c \end{array} \right\}$$

Segundo Sub-Triple

$$\left\{ \begin{array}{c} \sum = \frac{(j+1)j}{2} \\ \hline j \neq c \end{array} \right\} j = j + 1; \left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j < c \end{array} \right\}$$

También usamos el axioma de asignación:

$$\left\{ \begin{array}{c} \sum = \frac{(j+1)j}{2} \\ \hline j \neq c \end{array} \right\} j = j + 1; \left\{ \begin{array}{c} \sum = \frac{j(j-1)}{2} \\ \hline j < c \end{array} \right\}$$

19.2.4. Conclusión

Por la regla del ïf", podemos concluir que:

$$\{V\} \text{ if } a > 0 \text{ then } x := a \text{ else } x := -a \ \{x \geq 0, x^2 = a^2\}$$

Esto demuestra que la sentencia dada cumple la poscondición $\{x \geq 0, x^2 = a^2\}$.

20. Ejercicio 36

El siguiente fragmento de programa calcula $\sum_{i=1}^{n} i!$. Demostrar que es correcto con el invariante:

$$sum = \sum_{j=1}^{i-1} j! \wedge f = i!$$

```
i := 1; sum := 0; f := 1;
while i \neq n + 1 do
begin
    sum := sum + f;
    i := i + 1;
    f := f * i;
end
```

20.1. Resolución Nº1 Ejercicio 36

Proceso similar al del ejercicio anterior

20.1.1. Inicialización

Antes de que el bucle comience, tenemos:

$$i := 1; \quad \text{sum} := 0; \quad f := 1$$

Sustituyendo estos valores en el invariante, obtenemos:

$$\text{sum} = \sum_{j=1}^{1-1} j! \wedge f = 1!$$

Lo que se simplifica a:

$$\text{sum} = 0 \wedge f = 1$$

Esto es cierto, por lo que el invariante se cumple inicialmente.

20.1.2. Mantenimiento

Si el invariante es verdadero antes de una iteración del bucle, debe seguir siendo verdadero después de la iteración. Supongamos que el invariante es verdadero antes de una iteración:

$$\text{sum} = \sum_{j=1}^{i-1} j! \wedge f = i!$$

Durante la iteración, se ejecutan las siguientes instrucciones:

$$\text{sum} := \text{sum} + f$$
$$i := i + 1$$
$$f := f * i$$

Después de la ejecución de estas instrucciones, los valores se actualizan a:

$$\text{sum} = \sum_{j=1}^{i-1} j! + i!$$
$$i = i + 1$$
$$f = i * i! = (i + 1)!$$

Sustituyendo estos valores en el invariante, obtenemos:

$$\text{sum} = \sum_{j=1}^{i} j!$$
$$f = (i + 1)!$$

Esto demuestra que el invariante se mantiene.

20.1.3. Terminación

El bucle termina cuando $i = n + 1$. En este punto, el invariante se convierte en:

$$\text{sum} = \sum_{j=1}^{n} j!$$
$$f = (n + 1)!$$

Esto es precisamente lo que queríamos demostrar. Por lo tanto, el programa es correcto con respecto a la especificación dada.

20.2. Resolución Nº2 Ejercicio 36

20.2.1. Demostración del Invariante usando la Regla de Iteración

Para ello, usaremos la regla de iteración:

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

Buscamos un invariante global I que nos permita concluir al final que el programa calcula $\sum_{j=1}^{n} j!$.

Observando el código, podemos ver que en una variable sum se almacena dicho número, mientras otra variable i se incrementa en cada iteración y va calculando en f el factorial de i. Planteamos por tanto el siguiente invariante I:

$$I \equiv \left\{ \text{sum} = \sum_{j=1}^{i-1} j! \wedge f = i! \right\}$$

En primer lugar, demostraremos el triple para comprobar que el invariante es cierto al inicio del programa:

$$\{V\} \equiv \{i = 1; \text{sum} = 0; f = 1;\}$$

Esto es directamente cierto usando el axioma de asignación:

$$\{V\} \equiv \{i = 1; \text{sum} = 0; f = 1;\} \equiv \left\{ i = 1 \wedge \text{sum} = 0 \wedge f = 1 \equiv \left\{ i = 1 \wedge \text{sum} = \sum_{j=1}^{0} j! = 0 \right\} \right\}$$

A continuación, trataremos de probar el triple $\{I \wedge B\}\ S\ \{I\}$, para $B \equiv \{i \neq n+1\}$ y S el cuerpo del bucle:

$$\{I \wedge B\} \equiv \left\{ \text{sum} = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i \neq n+1 \right\}$$

$$\text{sum} = \text{sum} + f; \quad i = i + 1; \quad f = f * i;$$

$$\left\{ \text{sum} = \sum_{j=1}^{i-1} j! + i! \wedge f = (i+1)! \wedge i \neq n+1 \right\} \equiv \left\{ \text{sum} = \sum_{j=1}^{i} j! \wedge f = (i+1)! \wedge i \neq n+1 \right\}$$

$$\left\{ \text{sum} = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i \neq n+1 \right\} \equiv \left\{ \text{sum} = \sum_{j=1}^{i-1} j! \wedge f = i! \wedge i \neq n+1 \right\}$$

Luego podemos aplicar la regla de iteración, para obtener finalmente que:

$$\{I\} \equiv \left\{ \text{sum} = \sum_{j=1}^{i-1} j! \wedge f = i! \right\}$$

$$\text{while } i \neq n+1 \text{ do begin}$$

$$\text{sum} = \text{sum} + f;$$

$$i = i + 1;$$

$$f = f * i;$$

end

$$\{I \wedge \neg B\} \equiv \left\{ \text{sum} = \sum_{j=1}^{n} j! \wedge f = (n+1)! \right\}$$

De esta manera, hemos demostrado que el programa es correcto con respecto a la especificación dada.

21. Ejercicio 38

Demostrar que para $n > 0$ el siguiente fragmento de programa termina.

```
i := 1; f := 1;
while i != n do
begin
    i := i + 1;
    f := f * i;
end
```

21.1. Resolución 38

Para demostrar que el programa termina, debemos mostrar que el bucle `while` eventualmente se detendrá. Utilizaremos la técnica de la **Función de Decremento**, que es una función que decrece con cada iteración del bucle y está acotada inferiormente. La idea es encontrar una medida que decrezca en cada iteración y eventualmente llegue a un valor límite donde el bucle se detendrá.

21.1.1. Inicialización

Antes de que el bucle comience, los valores iniciales son:

$$i := 1; \quad f := 1$$

21.1.2. Invariante del bucle

Elegimos el siguiente invariante del bucle:

$$1 \leq i \leq n$$

Este invariante se elige porque i comienza en 1 y se incrementa en cada iteración hasta alcanzar n.

21.1.3. Condición de finalización

La condición de finalización del bucle es $i = n$. El bucle se ejecuta mientras $i \neq n$, por lo que una vez que $i = n$, el bucle termina.

21.1.4. Función de Decremento

La función de decremento que utilizaremos es:

$$T(i) = n - i$$

Esta función mide la distancia entre i y n. Con cada iteración del bucle, i se incrementa en 1, por lo que $T(i)$ decrece en 1.

Inicialización: Al inicio, $i = 1$, por lo tanto:

$$T(i) = n - 1$$

Esto es mayor o igual a 0 para $n > 0$.

Mantenimiento: Durante cada iteración del bucle, i se incrementa en 1:

$$i := i + 1$$

$$T(i) = n - (i + 1) = n - i - 1$$

Como $T(i)$ decrece en cada iteración y está acotada inferiormente por 0, eventualmente $T(i) = 0$.

Terminación: El bucle termina cuando $i = n$, en cuyo punto:

$$T(i) = n - n = 0$$

Así, hemos demostrado que el bucle se detiene para $n > 0$.

21.1.5. Notas de clase

- La única variable `variante` en la condición del bucle `while B do` es `i`.
- `i` solo toma valores en la sucesión {1, 2, 3 ..., n^4}
- Nada impide que la variable `i = n` (condición de parada)

22. Problema 48

Dados $n \geq 0$, $i \leq n$, demostrar que el siguiente segmento de programa evalúa $\frac{n!}{(i!*(n-i)!)}$ dado el invariante $\{i > k \vee afact = i!\} \wedge \{n - i > k \vee bfact = (n - i)!\}$:

```
k := 0; fact := 1;
while (k != n) do
begin
    k := k + 1; fact := fact * k;
    if (k <= i)
        then afact := fact;
    if (k <= n - i)
        then bfact := fact
end
bcof = fact / (afact * bfact)
```

22.1. Resolución Nº1 Problema 48

Para resolver este problema, debemos demostrar que el segmento de programa dado evalúa $\frac{n!}{(i!*(n-i)!)}$ utilizando el invariante proporcionado. Aquí está el paso a paso:

- Inicialización:
  - Se inicializa k a 0 y $fact$ a 1.
  - El invariante inicial es $\{i > k \vee afact = i!\} \wedge \{n - i > k \vee bfact = (n - i)!\}$.
- Bucle While:
  - El bucle se ejecuta mientras $k \neq n$.
  - Dentro del bucle, se incrementa k en 1 y se actualiza $fact$ multiplicándolo por k.
- Condiciones If:
  - Si $k \leq i$, se asigna $fact$ a $afact$. Esto asegura que $afact$ sea igual a $i!$ cuando $k = i$.
  - Si $k \leq n - i$, se asigna $fact$ a $bfact$. Esto asegura que $bfact$ sea igual a $(n - i)!$ cuando $k = n - i$.
- Finalización del Bucle:
  - Cuando el bucle termina, $fact$ será igual a $n!$ porque se ha multiplicado por todos los números de 1 a n.
- Cálculo de $bcof$:
  - Finalmente, se calcula $bcof$ como $\frac{fact}{(afact*bfact)}$.

- Dado que $fact = n!$, $afact = i!$, y $bfact = (n-i)!$, el resultado es $\frac{n!}{(i!*(n-i)!)}$.

```
k := 0; fact := 1;
while (k \neq n) do
begin
    k := k + 1; fact := fact * k;
    if (k \leq i)
        then afact := fact;
    if (k \leq n - i)
        then bfact := fact
end
bcof = \frac{fact}{(afact * bfact)}
```

Este paso a paso demuestra que el segmento de programa dado evalúa correctamente $\frac{n!}{(i!*(n-i)!)}$ utilizando el invariante proporcionado.

22.2. Resolución Nº2 Problema 48

22.2.1. Demostración del Invariante usando la Regla de Iteración

Para demostrar que el código evalúa $\frac{n!}{i!(n-i)!}$, hemos de buscar un invariante global que nos permita llegar a la poscondición. Observando el código, vemos que calcula el factorial de n y almacena en `afact` el factorial de i y en `bfact` el factorial de $n-i$. Por tanto, un invariante que nos puede servir es:

$$\{I\} \equiv \{fact = k! \land afact = (\text{mín}\{i,k\})! \land bfact = (\text{mín}\{n-i,k\})!\}$$

En primer lugar, hemos de ver que el invariante es cierto al inicio del programa:

$$\{0 \leq i \leq n\} \quad k = 0;\ fact = 1;\ \{I\}$$

Esto es directamente cierto. Posteriormente, hemos de demostrar el triple dado por $\{I \land B\}\ S\ \{I\}$ con $B \equiv \{k \neq n\}$ y S el cuerpo del bucle para poder aplicar la regla de iteración:

$$\{I \land B\} \equiv \{fact = k! \land afact = (\text{mín}\{i,k\})! \land bfact = (\text{mín}\{n-i,k\})! \land k \neq n\}$$

```
k = k + 1;
fact = fact * k;
if k \leq i then afact = fact;
if k \leq n - i then bfact = fact;
```

$$\{fact = (k-1)! \land afact = (\text{mín}\{i,k-1\})! \land bfact = (\text{mín}\{n-i,k-1\})! \land k \neq n+1\}$$

$$\{fact = k! \land afact = (\text{mín}\{i,k-1\})! \land bfact = (\text{mín}\{n-i,k-1\})! \land k \neq n+1\}$$

$$\begin{cases} \text{Si } k \leq i: & \{fact = k! \land afact = fact \land bfact = (\text{mín}\{n-i,k-1\})! \land k \neq n+1\} \\ \text{Si } k > i: & \{fact = k! \land afact = (\text{mín}\{i,k\})! \land bfact = (\text{mín}\{n-i,k-1\})! \land k \neq n+1\} \end{cases}$$

$$\begin{cases} \text{Si } k \leq n-i: & \{fact = k! \land afact = (\text{mín}\{i,k\})! \land bfact = fact \land k \neq n+1\} \\ \text{Si } k > n-i: & \{fact = k! \land afact = (\text{mín}\{i,k\})! \land bfact = (\text{mín}\{n-i,k\})! \land k \neq n+1\} \end{cases}$$

Después de comprobar ambas ramas del condicional, podemos concluir que $\{I\} \equiv \{fact = k! \land afact = (\text{mín}\{i,k\})! \land bfact = (\text{mín}\{n-i,k\})!\}$ se mantiene después del cuerpo del bucle.

Finalmente, usando la regla de iteración y el hecho de que el invariante I es cierto inicialmente y se mantiene en cada iteración, concluimos que al finalizar el bucle con $k = n$, la postcondición $fact = n! \land afact = i! \land bfact = (n-i)!$ es cierta.