Sistemas Concurrentes y Distribuidos

Relaciones de Problemas y Ejercicios Resueltos

Grado en Ingeniería Informática + Doble Grado DGIIM / ADE

Universidad de Granada (UGR)

Resolución Exhaustiva y Explicada:

Relaciones Oficiales de Problemas del Profesor (Manuel I. Capel Tuñón)
Soluciones Formadas y Verificadas de Infomates (LosDelDGIIM)
Aportes Didácticos de Ismael Sallami Moreno

Plataforma **repasaYA**

Curso Académico 2024–2025 / 2025–2026

Índice general

Relación de Problemas 1

Relación 1: Concurrencia, Intercalación y Lógica de Hoare

Metadatos de la Relación y Fuentes:

Fuentes de referencia cotejadas:
• **Relación Oficial de Problemas de Teoría del Profesor** (Manuel I. Capel Tuñón, Dpto. de Lenguajes y Sistemas Informáticos, Universidad de Granada, Curso 2026/2027).
• **Temario y Relaciones Resueltas de Infomates** (LosDelDGIIM, Sección 5.1: 49 ejercicios de concurrencia y verificación formal).
• **Relación de Ejercicios Resueltos de Ismael Sallami Moreno** (Doble Grado Informática + ADE, UGR).
• **Materiales de InfoAdeTech Hub** y colecciones de exámenes de la UGR.

Clave de Examen: Estructura Integral de la Relación 1

Esta relación contiene los **36 problemas oficiales** del Tema 1 de Sistemas Concurrentes y Distribuidos de la UGR completamente resueltos, demostrados formalmente y explicados paso a paso desde los fundamentos.
Están organizados en **7 bloques temáticos** respetando el orden secuencial estricto ($1 \rightarrow 36$) para permitir una localización inmediata de cualquier ejercicio de la hoja del profesor:

- **Bloque I (Ej. 1–3):** Concurrencia básica, intercalación, condiciones de carrera y búferes.
- **Bloque II (Ej. 4–10):** Grafos de precedencia, Bernstein, cobegin/coend, fork/join y sincronización con banderas/mallas.
- **Bloque III (Ej. 11–13):** Semántica axiomática de Hoare, axioma de asignación y reglas de consecuencia.
- **Bloque IV (Ej. 14–21):** Concurrencia en lógica de Hoare, teoremas de no interferencia de Owicki-Gries e invariantes.
- **Bloque V (Ej. 22–27):** Reglas de derivación (conjunción, intercambio sin aux, condicional, bucles y aliasing).
- **Bloque VI (Ej. 28–29):** Asignación a arrays/vectores y cálculo de la precondición más débil (wp).
- **Bloque VII (Ej. 30–36):** Bucles, invariantes inductivos, algoritmos de búsqueda, combinatorios y función de cota de terminación.

1.1 Bloque I: Concurrencia Básica, Intercalación y Condiciones de Carrera

Ejercicio 1: Posibles valores de variable compartida tras bucles concurrentes

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 j:= 1 to 2 do begin
    x:= x + 1;                     x:= x + 1;
  end;                           end;
end;                           end;
```

Solución Detallada

Cada proceso ejecuta un bucle de 2 iteraciones. En cada iteración, la sentencia $x := x+1$ no es atómica y se descompone a nivel de arquitectura en tres acciones atómicas elementales:

1. **Lectura (l_{ij}):** Carga del valor actual de la memoria compartida x en el registro privado R_i.
2. **Incremento:** Operación aritmética interna en la ALU: $R_i \leftarrow R_i + 1$.
3. **Escritura (e_{ij}):** Almacenamiento del valor de R_i de vuelta en la variable compartida x.

En total, P_1 ejecuta la secuencia $\{l_{11}, e_{11}, l_{12}, e_{12}\}$ y P_2 ejecuta $\{l_{21}, e_{21}, l_{22}, e_{22}\}$.
1. Valor Máximo Posible ($x = 4$): Ocurre cuando no hay solapamiento destructivo entre lecturas y escrituras (ejecución puramente secuencial o intercalada sin solapamiento de lecturas):

$$\text{Traza: } l_{11}, e_{11}(x = 1) \rightarrow l_{12}, e_{12}(x = 2) \rightarrow l_{21}, e_{21}(x = 3) \rightarrow l_{22}, e_{22}(x = 4).$$

2. Valor Mínimo Posible ($x = 2$): Ocurre cuando las lecturas de ambos procesos se solapan en ambas iteraciones, anulando mutuamente sus incrementos:

Acción P_1	**Acción P_2**	**Registros (R_1, R_2)**	**Valor de x**
l_{11} (lee 0)	l_{21} (lee 0)	$R_1 = 0, R_2 = 0$	0
e_{11} (escribe 1)	—	$R_1 = 1$	1
—	e_{21} (escribe 1)	$R_2 = 1$	1
l_{12} (lee 1)	l_{22} (lee 1)	$R_1 = 1, R_2 = 1$	1
e_{12} (escribe 2)	—	$R_1 = 2$	2
—	e_{22} (escribe 2)	$R_2 = 2$	**2**

3. Valor Intermedio ($x = 3$): Ocurre si el solapamiento destructivo solo tiene lugar en una de las dos iteraciones mientras que la otra se ejecuta limpiamente de forma secuencial. Por ejemplo: P_1 y P_2 solapan la primera iteración alcanzando $x = 1$, y luego P_1 y P_2 ejecutan secuencialmente sus segundas iteraciones incrementando sucesivamente a 2 y 3.

¿Pueden obtenerse valores menores que 2 (por ejemplo, $x = 0$ o $x = 1$)? No. Para que el programa finalice, ambos procesos deben terminar obligatoriamente sus dos iteraciones. Como $x_0 = 0$, cada proceso lee un valor no negativo, suma 1 y escribe; y ningún proceso decrementa x. Además, al menos el último proceso en escribir en su segunda iteración habrá leído un valor ≥ 1 (obtenido tras el primer incremento), por lo que forzosamente escribirá un valor final ≥ 2.
Conclusión: El conjunto exacto de valores posibles finales es $\{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$.

Ejercicio 2: Condiciones de carrera, determinismo y atomicidad

Para cada uno de los siguientes programas concurrentes:

- Indicar si existe una condición de carrera (*race condition*).
- Determinar si el resultado final es determinista. En caso contrario, obtener los posibles valores finales de las variables.
- Identificar qué operaciones de los procesos pueden interferir entre sí.
- Indicar, cuando sea necesario, qué instrucciones deberían ejecutarse de forma atómica para eliminar la condición de carrera. Razonar si ello es suficiente para obtener un resultado determinista.

Programa (a):

```
var x : integer := 0;
cobegin
    x := x + 1;
 ||
    x := x + 1;
coend;
```

Programa (b):

```
var x : integer := 0;
var y : integer := 0;
cobegin
    x := x + 1;
 ||
    y := y + 1;
coend;
```

Programa (c):

```
var x : integer := 1;
var y : integer := 0;
cobegin
    x := y + 1;
 ||
    y := x + 1;
coend;
```

Programa (d):

```
var x : integer := 2;
var y : integer := 3;
cobegin
    x := x * y;
 ||
    y := x * y;
coend;
```

Solución Detallada

Analizamos cada programa sistemáticamente aplicando las condiciones de Bernstein y el modelo de intercalación de instrucciones a nivel de lectura/escritura en registros:

Programa (a)

- **Condición de carrera: SÍ**. Ambos procesos acceden concurrentemente a la variable compartida x y ambos realizan operaciones de escritura ($E(P_1) \cap E(P_2) = \{x\} \neq \emptyset$).
- **Determinismo: NO es determinista**. Si se solapan las lecturas (l_1, l_2, e_1, e_2), ambos leen 0 y escriben 1, resultando en $x = 1$. Si se ejecutan secuencialmente, el resultado es $x = 2$. Valores posibles: $\mathbf{x} \in \{\mathbf{1}, \mathbf{2}\}$.

- **Operaciones que interfieren:** La lectura de un proceso con la escritura del otro (l_1 con e_2 y l_2 con e_1), así como las dos escrituras concurrentes.
- **Atomicidad:** Hacer atómicas ambas asignaciones ($\langle x := x + 1\rangle$) es **suficiente para garantizar el determinismo**, ya que la suma es asociativa y conmutativa. Las dos únicas trazas atómicas posibles dan idéntico resultado: $x = 2$.

Programa (b)

- **Condición de carrera: NO**. P_1 únicamente lee y escribe en x ($L_1 = E_1 = \{x\}$), mientras que P_2 únicamente lee y escribe en y ($L_2 = E_2 = \{y\}$). Los conjuntos de variables son completamente disjuntos:

$$L_1 \cap E_2 = \emptyset, \quad E_1 \cap L_2 = \emptyset, \quad E_1 \cap E_2 = \emptyset.$$

  Se satisfacen estrictamente las tres condiciones de Bernstein.
- **Determinismo: SÍ es determinista**. El resultado final es siempre único e invariable: $\mathbf{x = 1, y = 1}$.
- **Interferencias:** Ninguna.
- **Atomicidad:** No es necesaria ninguna acción atómica adicional.

Programa (c)

- **Condición de carrera: SÍ**. P_1 lee y y escribe en x ($L_1 = \{y\}, E_1 = \{x\}$). P_2 lee x y escribe en y ($L_2 = \{x\}, E_2 = \{y\}$). Se violan dos condiciones de Bernstein: $E_1 \cap L_2 = \{x\} \neq \emptyset$ y $L_1 \cap E_2 = \{y\} \neq \emptyset$.
- **Determinismo: NO es determinista**. Posibles valores finales:
  1. Si P_1 termina antes de que P_2 lea: $x = 0 + 1 = 1$, luego $y = 1 + 1 = 2 \implies (\mathbf{x = 1, y = 2})$.
  2. Si P_2 termina antes de que P_1 lea: $y = 1 + 1 = 2$, luego $x = 2 + 1 = 3 \implies (\mathbf{x = 3, y = 2})$.
  3. Si ambos leen antes de que ninguno escriba: P_1 lee $y = 0 \Rightarrow x = 1$; P_2 lee $x = 1 \Rightarrow y = 2 \implies (\mathbf{x = 1, y = 2})$.

  Conjunto de estados finales posibles: $\{(\mathbf{1}, \mathbf{2}), (\mathbf{3}, \mathbf{2})\}$.
- **Operaciones que interfieren:** La lectura de x por P_2 interfiere con la escritura de x por P_1; la lectura de y por P_1 interfiere con la escritura de y por P_2.
- **Atomicidad:** Hacer atómicas las instrucciones ($\langle x := y + 1\rangle$ e $\langle y := x + 1\rangle$) **NO es suficiente para lograr determinismo**, ya que subsiste una carrera de grano grueso entre qué instrucción atómica se ejecuta primero: si se ejecuta P_1 primero da $(1, 2)$, y si se ejecuta P_2 primero da $(3, 2)$. Para asegurar determinismo se requeriría imponer un orden de sincronización estricto.

Programa (d)

- **Condición de carrera: SÍ**. Ambos procesos leen y escriben sobre las mismas variables compartidas ($L_1 = \{x, y\}, E_1 = \{x\}$; $L_2 = \{x, y\}, E_2 = \{y\}$).
- **Determinismo: NO es determinista**.
  1. Si P_1 precede totalmente a P_2: $x = 2 \times 3 = 6$, luego $y = 6 \times 3 = 18 \implies (\mathbf{x = 6, y = 18})$.
  2. Si P_2 precede totalmente a P_1: $y = 2 \times 3 = 6$, luego $x = 2 \times 6 = 12 \implies (\mathbf{x = 12, y = 6})$.
  3. Si ambos leen inicialmente $x = 2, y = 3$ antes de escribir: P_1 calcula $2 \times 3 = 6$ y escribe $x = 6$; P_2 calcula $2 \times 3 = 6$ y escribe $y = 6 \implies (\mathbf{x = 6, y = 6})$.

  Conjunto de estados finales posibles: $\{(\mathbf{6}, \mathbf{18}), (\mathbf{12}, \mathbf{6}), (\mathbf{6}, \mathbf{6})\}$.
- **Atomicidad:** Hacer atómicas las operaciones elimina el estado $(6, 6)$, pero **NO elimina el no-determinismo**, pues persisten los resultados $(6, 18)$ y $(12, 6)$ según el orden de intercalación.

Ejercicio 3: Copia concurrente de un fichero f en g con doble búfer

¿Cómo se podría hacer la copia del fichero f en otro g, de forma concurrente, usando dos búferes intermedios B_1 y B_2, de tal manera que el proceso que lee de f y el proceso que escribe en g puedan trabajar en paralelo?

Solución Detallada

La técnica de **doble búfer** (*double buffering*) permite solapar la operación de lectura de entrada/salida (I/O) desde el disco con la operación de procesamiento/escritura en destino:

- Mientras el proceso **Lector** carga el siguiente bloque de datos desde el fichero f en el búfer B_1, el proceso **Escritor** vuelca el contenido del bloque previo desde el búfer B_2 hacia el fichero g.
- En la siguiente iteración, ambos procesos intercambian los roles de los búferes de manera alternada y sincronizada.

Implementación formal con banderas de sincronización / semáforos binarios:

```
// Recursos compartidos
var B1, B2 : TipoBloque;
var lleno1, lleno2 : boolean := false, false;
var fin_fichero : boolean := false;

Process Lector;
begin
  repeat
    // Fase 1: Leer sobre B1 mientras Escritor procesa B2
    if not fin_fichero then begin
      leer_bloque(f, B1, fin_fichero);
      lleno1 := true;
      esperar_a_que(lleno2 == false); // Esperar que B2 haya sido vaciado
    end;
    // Fase 2: Leer sobre B2 mientras Escritor procesa B1
```

```
    if not fin_fichero then begin
      leer_bloque(f, B2, fin_fichero);
      lleno2 := true;
      esperar_a_que(lleno1 == false); // Esperar que B1 haya sido vaciado
    end;
  until fin_fichero;
end;

Process Escritor;
begin
  repeat
    // Esperar y vaciar B1
    esperar_a_que(lleno1 == true or fin_fichero);
    if lleno1 then begin
      escribir_bloque(g, B1);
      lleno1 := false;
    end;
    // Esperar y vaciar B2
    esperar_a_que(lleno2 == true or fin_fichero);
    if lleno2 then begin
      escribir_bloque(g, B2);
      lleno2 := false;
    end;
  until fin_fichero and not lleno1 and not lleno2;
end;
```

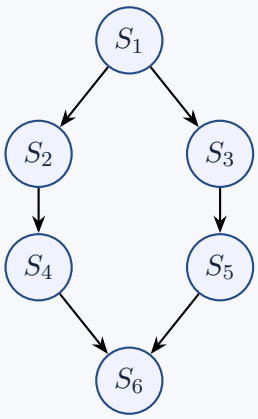
Ventaja clave: El tiempo total de copia se reduce drásticamente de $T_{\text{sec}} = N \cdot (T_{\text{leer}} + T_{\text{escribir}})$ a $T_{\text{conc}} \approx N \cdot \max(T_{\text{leer}}, T_{\text{escribir}})$, maximizando el uso del bus y los canales DMA de entrada/salida.

1.2 Bloque II: Grafos de Precedencia, Sincronización y Paralelismo

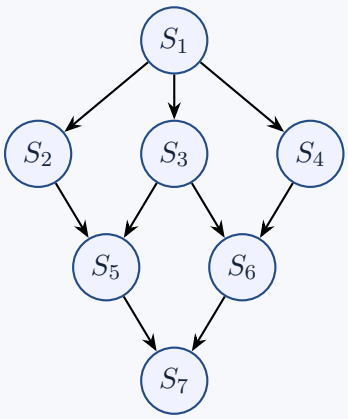
Ejercicio 4: Construcción de programas con cobegin-coend y fork-join para grafos

Construir, utilizando las instrucciones concurrentes `cobegin-coend` y `fork-join`, programas concurrentes que se correspondan con los dos grafos de precedencia siguientes:

Grafo G_1



Grafo G_2



Solución Detallada

Análisis del Grafo G_1

G_1 es un grafo bien anidado (serie-paralelo): S_1 precede a dos ramas paralelas disjuntas ($S_2 \to S_4$ y $S_3 \to S_5$), las cuales deben converger antes de que inicie S_6.

- **Con `cobegin-coend`:**

```
S1;
cobegin
    begin S2; S4; end;
 ||
    begin S3; S5; end;
coend;
S6;
```

- **Con `fork-join`:**

```
    var c : integer := 2;
    S1;
    fork L_rama2;
    // Rama 1
    S2;
    S4;
    goto L_convergencia;
L_rama2:
    S3;
    S5;
L_convergencia:
    join c;
    S6;
```

Análisis del Grafo G_2

En G_2, S_3 bifurca su flujo hacia S_5 y S_6, pero S_5 depende de $\{S_2, S_3\}$ mientras que S_6 depende de $\{S_3, S_4\}$. Esto genera un cruce de dependencias interno no estrictamente anidado.

- **Con `fork-join` (expresión directa de grafos arbitrarios):**

```
    var c5 : integer := 2; // S5 espera a S2 y S3
    var c6 : integer := 2; // S6 espera a S3 y S4
    var c7 : integer := 2; // S7 espera a S5 y S6

    S1;
    fork L_S2;
    fork L_S3;
    // Tarea S4 ejecutada por el hilo principal
    S4;
    join c6; // Si c6==0, pasa a ejecutar S6
    goto L_S6;

L_S2:
    S2;
    join c5;
    goto L_S5;

L_S3:
    S3;
    // S3 senala a ambos contadores
    // Si al decrementar c5 llega a 0, activa S5
```

```
    if (decrementar(c5) == 0) fork L_S5;
    if (decrementar(c6) == 0) fork L_S6;
    quit;

L_S5:
    S5;
    join c7;
    goto L_S7;

L_S6:
    S6;
    join c7;
    goto L_S7;

L_S7:
    S7;
```

- **Con `cobegin-coend` (usando variables de sincronización / semáforos):** Como no es estrictamente serie-paralelo, se requiere sincronización auxiliar (o duplicación estructurada):

```
var finS3 : boolean := false;
S1;
cobegin
    begin
        S2;
        esperar_a_que(finS3);
        S5;
    end
 ||
    begin
        S3;
        finS3 := true;
    end
 ||
    begin
        S4;
        esperar_a_que(finS3);
        S6;
    end
coend;
S7;
```

Ejercicio 5: Obtención de grafos de precedencia a partir de código

Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

Fragmento (a):

```
S1;
cobegin
    S2;
 ||
    begin
        S3;
        cobegin S4; || S5; coend;
        S6;
    end
 ||
    S7;
coend;
S8;
```

Fragmento (b):

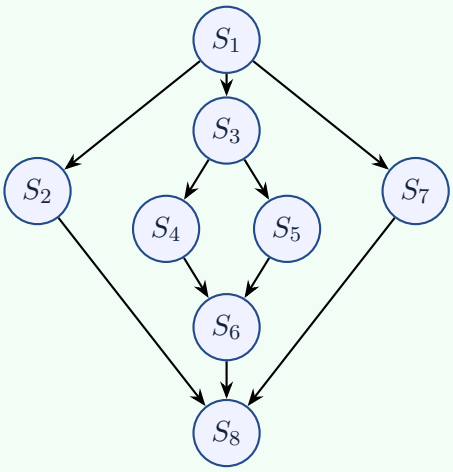
```
    var c1, c2 : integer := 2;
    S1;
    fork L1;
    S2;
    goto L2;
L1: S3;
L2: join c1;
    fork L3;
    S4;
    goto L4;
L3: S5;
L4: join c2;
    S6;
```

Solución Detallada

Analizamos el orden de ejecución y sincronizaciones de cada fragmento:

Grafo del Fragmento (a)

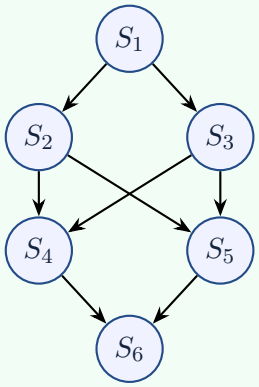
1. S_1 se ejecuta secuencialmente al principio y precede a las tres ramas del `cobegin`.
2. El `cobegin` exterior lanza 3 ramas concurrentes:
   - Rama 1: S_2.
   - Rama 2: $S_3 \rightarrow$ (`cobegin` interno de $S_4 \parallel S_5$) $\rightarrow S_6$.
   - Rama 3: S_7.
3. El `coend` exterior obliga a que finalicen S_2, S_6 y S_7 antes de ejecutar S_8.



Grafo del Fragmento (b)

1. S_1 inicia y realiza `fork L1`, creando dos hilos que ejecutan S_2 y S_3 en paralelo.
2. En `L2`, la instrucción `join c1` (con contador inicial 2) sincroniza la finalización de ambos (S_2 y S_3).
3. Una vez superado el primer join, se lanza `fork L3`, ejecutando en paralelo S_4 y S_5.

4. En `L4`, la instrucción `join c2` sincroniza a S_4 y S_5.
5. Finalmente se ejecuta secuencialmente S_6.



Ejercicio 6: Sistema de tiempo real con captador de impulsos

Suponer un sistema de tiempo real que dispone de un captador de impulsos conectado a un contador de energía eléctrica. La función del sistema consiste en contar el número de impulsos producidos en 1 hora (cada KWh consumido se cuenta como un impulso) e imprimir este número en un dispositivo de salida. Para ello se dispone de un programa concurrente con 2 procesos: un proceso acumulador (lleva la cuenta de los impulsos recibidos) y un proceso escritor (escribe en la impresora). En la variable común a los 2 procesos n se lleva la cuenta de los impulsos. El proceso acumulador puede invocar un procedimiento `Espera_impulso` para esperar a que llegue un impulso, y el proceso escritor puede llamar a `Espera_fin_hora` para esperar a que termine una hora. El código de los procesos de este programa podría ser el siguiente:

```
1  var n : integer := 0;
2
3  Process Acumulador;              Process Escritor;
4  begin                            begin
5    while true do begin              while true do begin
6      Espera_impulso;                  Espera_fin_hora;
7      n := n + 1;                      imprimir(n);
8    end;                               n := 0;
9  end;                               end;
10                                  end;
```

¿Existe algún problema de condición de carrera en este programa? Justificar la respuesta y proponer una solución correcta.

Solución Detallada

1. Identificación de Condiciones de Carrera: SÍ, existen graves condiciones de carrera debido al acceso no sincronizado a la variable compartida n:

- **Pérdida de impulsos durante el reseteo:** Cuando el proceso `Escritor` despierta tras una hora, lee n para imprimirlo y a continuación ejecuta $n := 0$. Si un impulso físico llega justo entre la lectura para imprimir y la asignación $n := 0$, el proceso `Acumulador` ejecutará $n := n + 1$. Inmediatamente después, el `Escritor` sobreescribirá $n := 0$, **perdiendo para siempre el impulso recibido**.
- **No atomicidad en $n := n + 1$ frente a $n := 0$:** A nivel de registros de máquina,

si la lectura de n por el acumulador se solapa con la escritura de 0 por el escritor, el acumulador puede restaurar un valor obsoleto incrementado, falseando por completo la cuenta de la siguiente hora.

2. Solución Correcta: Para resolverlo, la lectura del valor de la hora anterior y la puesta a cero de n deben realizarse de forma **atómica** respecto al incremento, utilizando una variable local auxiliar y exclusión mutua:

```
var n : integer := 0;
var cerrojo : Mutex;

Process Acumulador;
begin
  while true do begin
    Espera_impulso;
    cerrojo.lock();
    n := n + 1;
    cerrojo.unlock();
  end;
end;

Process Escritor;
var n_hora_anterior : integer;
begin
  while true do begin
    Espera_fin_hora;
    // Seccion critica: capturar y resetear de forma indivisible
    cerrojo.lock();
    n_hora_anterior := n;
    n := 0;
    cerrojo.unlock();

    imprimir(n_hora_anterior);
  end;
end;
```

De este modo, si llega un impulso durante la impresión, el acumulador esperará a que termine la sección crítica y lo sumará sobre el nuevo contador en cero, garantizando conservación estricta de todos los impulsos.

Ejercicio 7: Producto y suma concurrente de vectores

Supongamos un programa concurrente en el cual hay, en memoria compartida dos vectores a y b de enteros y con tamaño par, declarados como sigue:

```
var a, b : array[1..2*n] of integer; { n es una constante predefinida }
```

Queremos escribir un programa para obtener en b una copia ordenada del contenido de a. Para ello disponemos de la función `Sort` que ordena un tramo de a (entre las entradas s y t, ambas incluidas) y deja el resultado en el mismo tramo de a, y del procedimiento `Merge` que mezcla de forma ordenada dos tramos ordenados consecutivos de a (desde s hasta m, y desde $m+1$ hasta t) y escribe el resultado en el tramo correspondiente de b (desde s hasta t). Escribir un programa concurrente que ordene el vector a en paralelo aprovechando al máximo el multiprocesamiento.

Solución Detallada

Dado que el vector tiene tamaño par $2n$, el enfoque óptimo de división y conquista (*Divide and Conquer*) consiste en:

1. Dividir el vector a en dos mitades disjuntas: el tramo inferior $a[1..n]$ y el tramo superior $a[n+1.,2n]$.
2. Lanzar dos procesos concurrentes con `cobegin-coend`, ordenando cada mitad en paralelo mediante la función `Sort`.
3. Una vez completadas ambas ordenaciones (garantizado por el `coend`), mezclar ambas mitades ordenadas en el vector $b[1.,2n]$ usando `Merge`.

Código del programa concurrente:

```
program OrdenacionParalela;
const n = 1000;
var a, b : array[1..2*n] of integer;

begin
  // Fase 1: Ordenacion en paralelo de las dos mitades disjuntas
  cobegin
    Sort(1, n);        // Proceso 1 ordena a[1..n]
 ||
    Sort(n+1, 2*n);    // Proceso 2 ordena a[n+1..2*n]
  coend;

  // Fase 2: Mezcla ordenada secuencial de ambos tramos en b
  Merge(1, n, 2*n);    // Mezcla tramos [1..n] y [n+1..2*n] dejando el
      resultado en b[1..2*n]
end.
```

Comprobación de Condiciones de Bernstein en la Fase Paralela:

- Proceso 1 (P_1): Lee y escribe únicamente en el rango de memoria $a[1..n]$. Por tanto: $L(P_1) = E(P_1) = \{a[1], \ldots, a[n]\}$.
- Proceso 2 (P_2): Lee y escribe únicamente en el rango $a[n+1.,2n]$. Por tanto: $L(P_2) = E(P_2) = \{a[n+1], \ldots, a[2n]\}$.
- Intersecciones:

$$L(P_1) \cap E(P_2) = \emptyset, \quad E(P_1) \cap L(P_2) = \emptyset, \quad E(P_1) \cap E(P_2) = \emptyset.$$

Se satisfacen estrictamente las tres condiciones de Bernstein, garantizando ausencia de interferencias y determinismo absoluto.

Ejercicio 8: Multiplicación de matrices concurrentes y condiciones de Bernstein

Supongamos que tenemos un programa con tres matrices (a, b y c) de valores flotantes declaradas como variables globales. La multiplicación secuencial de a y b (almacenando el resultado en c) se puede hacer mediante un procedimiento `MultiplicacionSec` declarado como aparece aquí:

```
var a, b, c : array[1..3, 1..3] of real;

procedure MultiplicacionSec()
var i, j, k : integer;
```

```
begin
  for i := 1 to 3 do
    for j := 1 to 3 do begin
      c[i,j] := 0;
      for k := 1 to 3 do
        c[i,j] := c[i,j] + a[i,k] * b[k,j];
    end;
end;
```

Escribir un programa con el mismo fin, pero que use 3 procesos concurrentes. Suponer que los elementos de las matrices a y b se pueden leer simultáneamente, así como que elementos distintos de c pueden escribirse simultáneamente.

Solución Detallada

1. Análisis de Bernstein por filas: Para realizar la multiplicación con 3 procesos concurrentes (P_1, P_2, P_3), asignamos a cada proceso el cálculo completo de una fila entera de la matriz resultado c:

- Proceso P_i calcula la fila i $(c[i,1], c[i,2], c[i,3])$.
- **Conjunto de lectura de P_i:** Para calcular la fila i, P_i necesita los elementos de la fila i de la matriz a y todas las columnas de la matriz b:

$$L(P_i) = \{a[i,1], a[i,2], a[i,3]\} \cup \{b[k,j] \mid 1 \leq k, j \leq 3\}.$$

- **Conjunto de escritura de P_i:** P_i escribe única y exclusivamente en los elementos de la fila i de c:

$$E(P_i) = \{c[i,1], c[i,2], c[i,3]\}.$$

2. Comprobación de las Condiciones de Bernstein ($i \neq j$):

1. $E(P_i) \cap E(P_j) = \{c[i,*]\} \cap \{c[j,*]\} = \emptyset$, pues pertenecen a filas distintas de c.
2. $L(P_i) \cap E(P_j) = (a \cup b) \cap \{c[j,*]\} = \emptyset$, pues c no es leída por ningún proceso.
3. $E(P_i) \cap L(P_j) = \{c[i,*]\} \cap (a \cup b) = \emptyset$.

Dado que las lecturas simultáneas sobre a y b están permitidas por el hardware y no generan interferencia ($L(P_i) \cap L(P_j) \neq \emptyset$ es inocuo), los 3 procesos son totalmente independientes.

3. Código Concurrente con 3 Procesos:

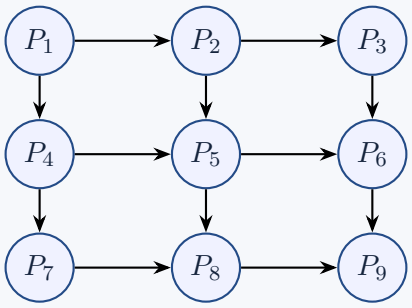
```
procedure MultiplicarFila(i : integer);
var j, k : integer;
begin
  for j := 1 to 3 do begin
    c[i,j] := 0;
    for k := 1 to 3 do
      c[i,j] := c[i,j] + a[i,k] * b[k,j];
  end;
end;

procedure MultiplicacionConc();
begin
  cobegin
    MultiplicarFila(1); // Proceso P1 calcula fila 1
  ||
```

```
    MultiplicarFila(2); // Proceso P2 calcula fila 2
 ||
    MultiplicarFila(3); // Proceso P3 calcula fila 3
  coend;
end;
```

Nota de paralelismo máximo: Se podrían utilizar hasta 9 procesos concurrentes asignando un proceso independiente a cada celda individual $c[i, j]$, ya que todos los $E(P_{ij}) = \{c[i, j]\}$ son disjuntos dos a dos.

Ejercicio 9: Grafo de dependencias de 9 actividades con EsperarPor y Acabar

Un trozo de programa ejecuta nueve procesos o actividades $(P_1, P_2, \ldots, P_9)$, repetidas veces, de forma concurrente con `cobegin-coend`, pero que requieren sincronizarse según un grafo de dependencias en malla 3×3:




```
while true do
  cobegin
    P1; P2; P3; P4; P5; P6; P7; P8; P9;
  coend;
```

Supón que queremos realizar la sincronización indicada en el grafo usando llamadas desde cada rutina a dos procedimientos:

- `EsperarPor(i)`: llamado por una rutina cualquiera (la número k) para esperar a que termine la rutina número i, usando espera ocupada.
- `Acabar(i)`: llamado por la rutina número i, al final de la misma, para indicar que dicha rutina ya ha finalizado.

Escribe una implementación de `EsperarPor` y `Acabar` (junto con la declaración e inicialización de las variables compartidas necesarias) que cumpla con los requisitos dados.

Solución Detallada

1. Diseño del vector de estados compartidos: Para señalizar la terminación de cada proceso se utiliza un vector global booleano `finalizado[1..9]`.

- `finalizado[i] == true` indica que el proceso P_i ha completado su ejecución en la iteración actual.
- `finalizado[i] == false` indica que el proceso P_i aún no ha terminado.

2. Reto clave: Reinicialización cíclica segura: Dado que el programa está envuelto en un bucle infinito `while true do cobegin ... coend`, el vector `finalizado` debe

restablecerse a `false` para la siguiente iteración. Si no se reinicia, en la iteración 2 todos los `EsperarPor` evaluarían a verdadero inmediatamente, perdiéndose la sincronización. Dado que P_9 es el último nodo del grafo (sumidero hacia el cual convergen directa o indirectamente todos los flujos), cuando P_9 finaliza se garantiza que **todos los procesos $P_1, \ldots, P_8$ ya han finalizado**. Por tanto, P_9 es el único responsable de reiniciar el vector a `false`.

3. Implementación de los Procedimientos:

```
// Variable compartida en memoria comun
var finalizado : array[1..9] of boolean := (false, false, false, false,
    false, false, false, false, false);

procedure EsperarPor(i : integer);
begin
  // Espera ocupada (polling) hasta que el proceso predecesor senale
      finalizacion
  while not finalizado[i] do begin
    // No-op (espera activa)
  end;
end;

procedure Acabar(i : integer);
var k : integer;
begin
  if i < 9 then begin
    finalizado[i] := true; // Senala que Pi ha concluido
  end else begin
    // El proceso P9 es el sumidero final de la malla.
    // Antes de terminar el cobegin, reinicia el vector para la siguiente
        vuelta del while.
    for k := 1 to 9 do
      finalizado[k] := false;
  end;
end;
```

Ejercicio 10: Vector de procesos estáticos P[n: 1..9

para la malla de actividades] En el problema anterior los procesos $P_1, P_2, \ldots, P_9$ se ponían en marcha usando `cobegin/coend`. Escribe un programa equivalente que use declaración estática de procesos, mediante un vector de procesos P, con índices desde 1 hasta 9, ambos incluidos. El proceso $P[n]$ contiene una secuencia de instrucciones desconocida, que llamamos S_n, y además debe incluir las llamadas necesarias a `Acabar` y `EsperarPor` (con la misma implementación que antes) para lograr la sincronización adecuada. Se incluye aquí una plantilla:

```
Process P[ n : 1..9 ]
begin
  ..... { esperar (si es necesario) a los procesos que corresponda }
  Sn ;  { sentencias especificas de este proceso (desconocidas) }
  ..... { senalar que hemos terminado }
end;
```

Solución Detallada

En la topología en malla 3×3, cada nodo P_n tiene como máximo dos predecesores directos de los que depende:

- **Predecesor vertical (superior):** P_{n-3}. Solo existe si el nodo no está en la primera fila ($n > 3$).
- **Predecesor horizontal (izquierdo):** P_{n-1}. Solo existe si el nodo no está en la primera columna ($(n \pmod 3) \neq 1$).

Alternativa 1: Determinación de dependencias por Aritmética Modular (Óptima)

```
Process P[ n : 1..9 ]
begin
  while true do begin
    // 1. Espera al predecesor de arriba (si no esta en la primera fila)
    if (n > 3) then
      EsperarPor(n - 3);

    // 2. Espera al predecesor de la izquierda (si no esta en la primera
       columna)
    if (n mod 3) <> 1 then
      EsperarPor(n - 1);

    // 3. Ejecucion del trabajo especifico del proceso n
    Sn;

    // 4. Senalizacion de finalizacion
    Acabar(n);
  end;
end;
```

Alternativa 2: Matriz Genérica de Dependencias

Para topologías arbitrarias que no sigan un patrón geométrico regular, se precalcula una matriz estática `espera[1..9, 1..2]` con los IDs de los predecesores (usando -1 para indicar ausencia de predecesor):

```
var espera : array[1..9, 1..2] of integer := (
  (-1, -1),   // P1: no espera a nadie (fuente)
  (1, -1),    // P2: espera a P1
  (2, -1),    // P3: espera a P2
  (1, -1),    // P4: espera a P1
  (2, 4),     // P5: espera a P2 y P4
  (3, 5),     // P6: espera a P3 y P5
  (4, -1),    // P7: espera a P4
  (5, 7),     // P8: espera a P5 y P7
  (6, 8)      // P9: espera a P6 y P8 (sumidero)
);

Process P[ n : 1..9 ]
var i : integer;
begin
  while true do begin
    for i := 1 to 2 do
      if espera[n, i] <> -1 then
        EsperarPor(espera[n, i]);

    Sn;
    Acabar(n);
  end;
end;
```

1.3 Bloque III: Semántica Axiomática Secuencial y Reglas de Hoare

Ejercicio 11: Obtención de poscondiciones adecuadas en Hoare

Para los siguientes fragmentos de código, obtener la poscondición adecuada para convertirlo en un triple demostrable con la Lógica de Programas de Hoare:

(a) $\{i < 10\}\ i := 2 * i + 1;\ \{\ \}$

(b) $\{i > 0\}\ i := i - 1;\ \{\ \}$

(c) $\{i > j\}\ i := i + 1;\ j := j + 1;\ \{\ \}$

(d) $\{\text{falso}\}\ a := a + 7;\ \{\ \}$

(e) $\{\text{verdad}\}\ i := 3;\ j := 2 * i;\ \{\ \}$

(f) $\{\text{verdad}\}\ c := a + b;\ c := c/2;\ \{\ \}$

Solución Detallada

Para deducir la poscondición más fuerte y natural en cada caso, aplicamos el **Axioma de Asignación**:

$$\{P[e/x]\}\ x := e\ \{P\}$$

junto con la regla de composición secuencial para programas de varias instrucciones:

(a) **Fragmento (a):** Si $i_{\text{ant}} < 10$, tras la asignación $i_{\text{nuevo}} = 2 \cdot i_{\text{ant}} + 1$, despejando tenemos $i_{\text{ant}} = (i_{\text{nuevo}} - 1)/2$. Sustituyendo en la precondición:

$$\frac{i-1}{2} < 10 \iff i - 1 < 20 \iff i < 21.$$

Por tanto, el triple demostrable es:

$$\{\mathbf{i < 10}\}\ \mathbf{i := 2 * i + 1}\ \{\mathbf{i < 21}\}.$$

(b) **Fragmento (b):** Si $i_{\text{ant}} > 0$ (con $i \in \mathbb{Z}$ significa $i_{\text{ant}} \geq 1$), tras $i := i - 1$:

$$i_{\text{ant}} = i_{\text{nuevo}} + 1 > 0 \iff i_{\text{nuevo}} > -1 \iff i_{\text{nuevo}} \geq 0.$$

Por tanto:

$$\{\mathbf{i > 0}\}\ \mathbf{i := i - 1}\ \{\mathbf{i \geq 0}\}.$$

(c) **Fragmento (c):** Aplicamos composición secuencial con aserto intermedio:

$$\{i > j\}\ i := i + 1\ \{i > j + 1\}\ j := j + 1\ \{i > j\}.$$

Dado que ambos se incrementan en 1, la relación de orden estricto se preserva:

$$\{\mathbf{i > j}\}\ \mathbf{i := i + 1;\ j := j + 1}\ \{\mathbf{i > j}\}.$$

(d) **Fragmento (d):** La precondición es {falso}. Por el principio lógico *ex falso quodlibet* (de una contradicción se sigue cualquier cosa), el triple es demostrable para **cualquier poscondición** Q, en particular para {verdad} o {falso}:

$$\{\text{falso}\}\ \mathbf{a := a + 7}\ \{\text{verdad}\}.$$

(e) **Fragmento (e):**

$$\{\text{verdad}\}\ i := 3\ \{i = 3\}\ j := 2 * i\ \{i = 3 \wedge j = 6\}.$$

Poscondición: $\{\mathbf{i = 3 \wedge j = 6}\}$.

(f) **Fragmento (f):**

$$\{\text{verdad}\}\ c := a + b\ \{c = a + b\}\ c := c/2\ \left\{c = \frac{a+b}{2}\right\}.$$

Poscondición: $\{\mathbf{c = (a + b)/2}\}$.

Ejercicio 12: Triples no demostrables con la Lógica de Programas

¿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) $\{a > 0\}\ a := a - 7;\ \{a > -6\}$

Solución Detallada

Para cada triple $\{P\}\ x := e\ \{Q\}$, calculamos la precondición más débil requerida por el axioma de asignación $Q[e/x]$, y comprobamos si la precondición dada P implica lógicamente $Q[e/x]$ ($P \implies Q[e/x]$):

- **Triple (a):** $Q \equiv \{i \geq 0\}$.

$$Q[i - 1/i] \equiv \{i - 1 \geq 0\} \equiv \{i \geq 1\} \equiv \{i > 0\} \quad (\text{para } i \in \mathbb{Z}).$$

Como $\{i > 0\} \implies \{i \geq 1\}$, el triple **SÍ es demostrable**.

- **Triple (b):** $Q \equiv \{x \geq 9\}$.

$$Q[x + 3/x] \equiv \{x + 3 \geq 9\} \equiv \{x \geq 6\}.$$

Como la precondición dada es $\{x \geq 7\}$ y trivialmente $\{x \geq 7\} \implies \{x \geq 6\}$, por la regla de fortalecimiento de precondición el triple **SÍ es demostrable**.

- **Triple (c):** $Q \equiv \{i \leq 20\}$.

$$Q[2i + 1/i] \equiv \{2i + 1 \leq 20\} \iff \{2i \leq 19\} \iff \{i \leq 9{,}5\} \iff \{i \leq 9\}.$$

Como la precondición dada es $\{i < 9\} \implies \{i \leq 8\} \implies \{i \leq 9\}$, por la regla de fortalecimiento el triple **SÍ es demostrable**.

- **Triple (d):** $Q \equiv \{a > -6\}$.

$$Q[a - 7/a] \equiv \{a - 7 > -6\} \iff \{a > 1\}.$$

La precondición dada es $\{a > 0\}$. Para que fuese demostrable, se requeriría $\{a > 0\} \implies \{a > 1\}$, lo cual es **FALSO**:

$$\textbf{Contraejemplo:} \quad \text{Si } a = 1, \text{ se cumple la precondición } 1 > 0.$$

Tras ejecutar $a := a - 7$, el valor final es $a = 1 - 7 = -6$. Al evaluar la poscondición: $-6 > -6$ es **falso**.

Conclusión: El único triple que **NO es demostrable** es el **(d)**.

Ejercicio 13: Reglas de consecuencia y deducibilidad de triples

Si el triple $\{P\}\ C\ \{Q\}$ es demostrable, ¿cuál de los siguientes triples **no se puede demostrar** en general?

(a) $\{P \wedge D\}\ C\ \{Q\}$

(b) $\{P \vee D\}\ C\ \{Q\}$

(c) $\{P\}\ C\ \{Q \vee D\}$

(d) $\{P\}\ C\ \{Q \vee P\}$

Solución Detallada

Las dos **Reglas de Consecuencia** de la lógica de Hoare establecen:

1. **Fortalecimiento de la Precondición:**

$$\frac{P' \implies P, \quad \{P\}\ C\ \{Q\}}{\{P'\}\ C\ \{Q\}}$$

2. **Debilitamiento de la Poscondición:**

$$\frac{\{P\}\ C\ \{Q\}, \quad Q \implies Q'}{\{P\}\ C\ \{Q'\}}$$

Analizamos cada opción:

- **Opción (a):** $\{P \wedge D\} \implies P$ es una tautología lógica. Por la regla de fortalecimiento de precondición, el triple $\{P \wedge D\}\ C\ \{Q\}$ **siempre se puede demostrar**.
- **Opción (b):** Requiere $(P \vee D) \implies P$, lo cual exige $D \implies P$. Si D no implica P (por ejemplo, estados donde D es verdadero pero P es falso), no tenemos ninguna garantía sobre el comportamiento de C. Por tanto, $\{P \vee D\}\ C\ \{Q\}$ **NO se puede demostrar en general**.
- **Opción (c):** $Q \implies (Q \vee D)$ es una tautología lógica. Por la regla de debilitamiento de poscondición, el triple **siempre se puede demostrar**.
- **Opción (d):** Análogamente, $Q \implies (Q \vee P)$ es una tautología lógica. Por debilitamiento de poscondición, el triple **siempre se puede demostrar**.

Conclusión: El triple que **no se puede demostrar** es el **(b)**.

1.4 Bloque IV: Concurrencia en Lógica de Hoare y Teorema de Owicki-Gries

Ejercicio 14: Valores finales con acciones atómicas cobegin

Dado el programa concurrente:

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

obtener:

(a) Los valores finales de x e y.

(b) Los valores finales de x e y si quitamos los símbolos de instrucción atómica $\langle\ \rangle$.

Solución Detallada

Apartado (a): Con instrucciones atómicas $\langle\ \rangle$

Como las instrucciones son atómicas, solo existen $2! = 2$ intercalaciones posibles a nivel de sentencias:

1. **Orden 1 ($P_1 \to P_2$):**
   - Se ejecuta P_1: $x = 5 + 2 = 7$.
   - Se ejecuta P_2: $y = 7 \times 2 = 14$.
   - Resultado final: $(\mathbf{x = 7, y = 14})$.
2. **Orden 2 ($P_2 \to P_1$):**
   - Se ejecuta P_2: $y = 5 \times 2 = 10$.
   - Se ejecuta P_1: $x = 5 + 10 = 15$.
   - Resultado final: $(\mathbf{x = 15, y = 10})$.

Conjunto de estados finales con atomicidad: $\{(\mathbf{7}, \mathbf{14}), (\mathbf{15}, \mathbf{10})\}$.

Apartado (b): Sin instrucciones atómicas (descomposición en registros)

Cada asignación se descompone en lectura de operandos en registros locales, cálculo y escritura en memoria:

$$\begin{aligned} P_1: &\quad l_1^x : R_1 \leftarrow x; \quad l_1^y : R_2 \leftarrow y; \quad e_1^x : x \leftarrow R_1 + R_2. \\ P_2: &\quad l_2^x : R_3 \leftarrow x; \quad l_2^y : R_4 \leftarrow y; \quad e_2^y : y \leftarrow R_3 \times R_4. \end{aligned}$$

Analizamos las posibles intercalaciones no atómicas:

- Además de los dos órdenes puramente secuenciales anteriores que dan $(7, 14)$ y $(15, 10)$:
- **Lecturas cruzadas simultáneas antes de cualquier escritura:** Ambos procesos leen los valores iniciales $x = 5, y = 2$:
  - P_1 lee $x = 5, y = 2 \implies$ calcula $R_1 + R_2 = 7$.

- P_2 lee $x = 5, y = 2 \implies$ calcula $R_3 \times R_4 = 10$.
- Sin importar en qué orden escriban (e_1^x y luego e_2^y, o viceversa): x recibe 7 e y recibe 10.
- Resultado final: $(\mathbf{x = 7, y = 10})$.

Conjunto completo de estados finales posibles sin atomicidad: $\{(\mathbf{7}, \mathbf{14}), (\mathbf{15}, \mathbf{10}), (\mathbf{7}, \mathbf{10})\}$.

Ejercicio 15: Comprobación formal de No Interferencia de Owicki-Gries

Comprobar si la demostración del triple:

$$\{x \geq 2\}\ \langle x := x - 2\rangle\ \{x \geq 0\}$$

interfiere con los siguientes teoremas auxiliares:

(a) $\{x \geq 0\}\ \langle x := x + 3\rangle\ \{x \geq 3\}$

(b) $\{x \geq 0\}\ \langle x := x + 3\rangle\ \{x \geq 0\}$

(c) $\{x \geq 7\}\ \langle x := x + 3\rangle\ \{x \geq 10\}$

(d) $\{y \geq 0\}\ \langle y := y + 3\rangle\ \{y \geq 3\}$

Solución Detallada

Definición del Criterio de No Interferencia de Owicki-Gries: Dado un triple $\{P\}\ S\ \{Q\}$ y otra instrucción concurrente S' con asertos asociados $\text{pre}(S')$ y $\text{post}(S')$, decimos que S' **no interfiere** con el triple $\{P\}\ S\ \{Q\}$ si las aserciones P y Q se preservan invariantes bajo la ejecución de S', es decir, si son demostrables los triples de no interferencia:

$$\text{NI}_1: \quad \{P \wedge \text{pre}(S')\}\ S'\ \{P\} \qquad \text{y} \qquad \text{NI}_2: \quad \{Q \wedge \text{pre}(S')\}\ S'\ \{Q\}.$$

En nuestro problema: $P \equiv \{x \geq 2\}$ y $Q \equiv \{x \geq 0\}$. Debemos comprobar si cada acción candidata S' preserva P y Q:

- **Candidato (a):** $S' \equiv \langle x := x + 3\rangle$, con precondición $\text{pre}(S') \equiv \{x \geq 0\}$.

  1. Comprobamos preservación de P:

     $$\{x \geq 2 \wedge x \geq 0\}\ x := x + 3\ \{x \geq 2\}.$$

     Por el axioma de asignación: $(x + 3 \geq 2) \iff x \geq -1$. Como $(x \geq 2 \wedge x \geq 0) \equiv (x \geq 2) \implies (x \geq -1)$, se cumple.
  2. Comprobamos preservación de Q:

     $$\{x \geq 0 \wedge x \geq 0\}\ x := x + 3\ \{x \geq 0\}.$$

     Por el axioma: $(x + 3 \geq 0) \iff x \geq -3$. Como $(x \geq 0) \implies (x \geq -3)$, se cumple.

  Por tanto, el candidato (a) **NO INTERFIERE**.

- **Candidato (b):** Idéntico a (a) porque la instrucción es la misma y la precondición es también $\{x \geq 0\}$. **NO INTERFIERE**.

- **Candidato (c):** $S' \equiv \langle x := x + 3 \rangle$, con precondición $\{x \geq 7\}$. Como $\{x \geq 7\} \implies \{x \geq 0\}$, las precondiciones conjuntas son aún más fuertes, por lo que también **NO INTERFIERE**.
- **Candidato (d):** $S' \equiv \langle y := y + 3 \rangle$ opera sobre una variable totalmente disjunta y. Como las aserciones P y Q dependen solo de x y S' solo escribe en y, por el axioma de asignación con variables disjuntas la no interferencia es inmediata. **NO INTERFIERE**.

Conclusión: Ninguno de los cuatro teoremas interfiere con el triple dado, pues todos incrementan x (o modifican otra variable y), favoreciendo el mantenimiento de las cotas inferiores $x \geq 2$ y $x \geq 0$.

Ejercicio 16: Triple concurrente con tres sumandos atómicos

Dado el siguiente triple:

```
{ x == 0 }
cobegin
    < x = x + a >;
 ||
    < x = x + b >;
 ||
    < x = x + c >;
coend
{ x == a + b + c }
```

demostrar que es correcto utilizando la Lógica de Hoare Concurrente.

Solución Detallada

Para demostrar la corrección del programa concurrente mediante el Teorema de Owicki-Gries, debemos asociar asertos a cada proceso y demostrar:

1. La corrección local de cada proceso por separado.
2. La no interferencia mutua entre las aserciones de cada proceso y las acciones atómicas de los demás.

1. Definición de Variables Auxiliares de Estado: Introducimos variables booleanas auxiliares conceptuales $done_a, done_b, done_c$, inicializadas a `false`, que indican qué sumandos se han incorporado ya a x. El **Invariante Global** del sistema es:

$$I \equiv (x = (done_a ? a : 0) + (done_b ? b : 0) + (done_c ? c : 0)) .$$

2. Demostración Local de cada Proceso: Para el proceso P_a (y análogamente para P_b y P_c):

$$\text{pre}(P_a) \equiv I \wedge \neg done_a, \qquad \text{post}(P_a) \equiv I \wedge done_a.$$

La instrucción atómica es $\langle x := x + a;\ done_a := true \rangle$. Aplicando el axioma de asignación compuesto:

$$(I \wedge done_a)[x + a/x, true/done_a] \equiv I \wedge \neg done_a.$$

Por tanto, el triple local $\{I \wedge \neg done_a\}\ P_a\ \{I \wedge done_a\}$ es estrictamente demostrable.

3. Demostración de No Interferencia de Owicki-Gries: Debemos comprobar que la ejecución de P_b no destruye las aserciones de P_a:

$$\{(I \wedge \neg done_a) \wedge \text{pre}(P_b)\}\ \langle x := x + b;\ done_b := true \rangle\ \{I \wedge \neg done_a\}.$$

Dado que P_b únicamente modifica x sumándole b y conmuta $done_b$ a `true`, el estado resultante mantiene $\neg done_a$ inalterado y actualiza I de forma consistente. La misma invariancia se cumple para la poscondición $\{I \wedge done_a\}$.
4. Conclusión de la Poscondición Final: Al alcanzar el `coend`, se cumple la conjunción de las poscondiciones locales:

$$Q_{\text{final}} \equiv (I \wedge done_a \wedge done_b \wedge done_c) \implies (x = a + b + c).$$

Queda demostrada la corrección total del triple concurrente.

Ejercicio 17: Aplicación de las reglas de consecuencia de Hoare

Suponer que el triple $\{suma > 1\}\ suma = suma + 4\ \{suma > 5\}$ es demostrable. ¿Cuál de los siguientes triples es también demostrable? Indicar por qué.

(a) $\{suma > 2\}\ suma = suma + 4\ \{suma > 5\}$

(b) $\{suma \geq 1\}\ suma = suma + 4\ \{suma > 5\}$

(c) $\{suma > 0\}\ suma = suma + 4\ \{suma > 5\}$

(d) $\{suma > 1\}\ suma = suma + 4\ \{suma > 6\}$

Solución Detallada

Nos basamos en las dos reglas de consecuencia de Hoare:

$$\text{Regla 1 (Fortalecer Precondición):} \quad \frac{P' \implies P, \quad \{P\}S\{Q\}}{\{P'\}S\{Q\}}$$

$$\text{Regla 2 (Debilitar Poscondición):} \quad \frac{\{P\}S\{Q\}, \quad Q \implies Q'}{\{P\}S\{Q'\}}$$

Analizamos las cuatro opciones:

- **Opción (a):** La nueva precondición es $\{suma > 2\}$. Es un hecho matemático que $(suma > 2) \implies (suma > 1)$. Por la regla de fortalecimiento de precondición, el triple $\{suma > 2\}\ suma = suma + 4\ \{suma > 5\}$ **SÍ es demostrable**.
- **Opción (b):** La nueva precondición es $\{suma \geq 1\}$. Sin embargo, $(suma \geq 1) \not\implies (suma > 1)$ (contraejemplo: $suma = 1 \implies 1 + 4 = 5 \not> 5$). Falso.
- **Opción (c):** La nueva precondición es $\{suma > 0\}$, que tampoco implica $suma > 1$ (si $suma = 1$, daría $5 \not> 5$). Falso.
- **Opción (d):** La nueva poscondición es $\{suma > 6\}$. Para aplicar debilitamiento de poscondición se necesitaría $Q \implies Q'$, es decir, $(suma > 5) \implies (suma > 6)$, lo cual es falso (por ejemplo, $suma = 5{,}5$ o $suma = 6$). Falso.

Conclusión: El único triple demostrable es el **(a)**.

Ejercicio 18: Selección de valores finales tras tres asignaciones atómicas

Seleccionar el valor correcto de las 2 variables (x e y) después de ejecutarse el siguiente programa concurrente:

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

Opciones disponibles: (a) $x = 7, y = 14$ (b) $x = 5, y = 10$ (c) $x = -7, y = 14$ (d) $x = -3, y = 10$.

Solución Detallada

Denotamos las 3 instrucciones atómicas como:

$$S_1 \equiv \langle x := x + y \rangle, \qquad S_2 \equiv \langle y := x * y \rangle, \qquad S_3 \equiv \langle x := x - y \rangle.$$

Partiendo de $x_0 = 5, y_0 = 2$, evaluamos las $3! = 6$ permutaciones atómicas posibles:

Permutación	**Paso 1**	**Paso 2**	**Paso 3**	**Resultado** (x, y)	
$S_1 \to S_2 \to S_3$	$x = 5 + 2 = 7$	$y = 7 \times 2 = 14$	$x = 7 - 14 = -7$	$(\mathbf{-7}, \mathbf{14})$	[Opción c]
$S_1 \to S_3 \to S_2$	$x = 5 + 2 = 7$	$x = 7 - 2 = 5$	$y = 5 \times 2 = 10$	$(\mathbf{5}, \mathbf{10})$	[Opción b]
$S_3 \to S_1 \to S_2$	$x = 5 - 2 = 3$	$x = 3 + 2 = 5$	$y = 5 \times 2 = 10$	$(\mathbf{5}, \mathbf{10})$	[Opción b]
$S_2 \to S_1 \to S_3$	$y = 5 \times 2 = 10$	$x = 5 + 10 = 15$	$x = 15 - 10 = 5$	$(\mathbf{5}, \mathbf{10})$	[Opción b]
$S_2 \to S_3 \to S_1$	$y = 5 \times 2 = 10$	$x = 5 - 10 = -5$	$x = -5 + 10 = 5$	$(\mathbf{5}, \mathbf{10})$	[Opción b]
$S_3 \to S_2 \to S_1$	$x = 5 - 2 = 3$	$y = 3 \times 2 = 6$	$x = 3 + 6 = 9$	$(\mathbf{9}, \mathbf{6})$	

Conclusión: De las opciones presentadas en el enunciado oficial de tipo test, las respuestas correctas son **(b)** $(x = 5, y = 10)$ y **(c)** $(x = -7, y = 14)$.

Ejercicio 19: Traza de asertos y valores finales en programa secuencial

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 formal del programa, que incluya en su último aserto los valores finales de las variables.

```
int x = c1;
int y = c2;
x = x + y;
y = x * y;
x = x - y;
```

Solución Detallada

Insertamos los asertos de Hoare aplicando sucesivamente el axioma de asignación y la regla de composición secuencial:

```
// Estado inicial parametrizado por las constantes c1 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 }
```

Factorizando algebraicamente el aserto final:

$$x_{\text{final}} = (c_1 + c_2) - (c_1 + c_2) \cdot c_2 = (\mathbf{c_1 + c_2})(\mathbf{1 - c_2}),$$
$$y_{\text{final}} = (\mathbf{c_1 + c_2}) \cdot \mathbf{c_2}.$$

Por tanto, el aserto final que demuestra la corrección total del programa es:

$$\{\mathbf{x} = (\mathbf{c_1 + c_2})(\mathbf{1 - c_2}) \ \wedge \ \mathbf{y} = (\mathbf{c_1 + c_2}) \cdot \mathbf{c_2}\}.$$

Ejercicio 20: Demostración de suma concurrente de potencias de 2

Demostrar que el siguiente triple es cierto:

```
{ x == 0 }
cobegin
    < x = x + 1 >;
 ||
    < x = x + 2 >;
 ||
    < x = x + 4 >;
coend
{ x == 7 }
```

Solución Detallada

Denotamos los procesos atómicos como:

$$S_1 \equiv \langle x := x + 1 \rangle, \qquad S_2 \equiv \langle x := x + 2 \rangle, \qquad S_3 \equiv \langle x := x + 4 \rangle.$$

Notemos que cada proceso suma una potencia de 2 ($2^0, 2^1, 2^2$). El valor de x codifica directamente en binario qué procesos han finalizado:

- P_1 conmuta el bit 0 ($+1$).
- P_2 conmuta el bit 1 ($+2$).
- P_3 conmuta el bit 2 ($+4$).

Precondiciones Locales (condición de que su bit respectivo aún vale 0):

$$\text{pre}(S_1) \equiv \{x \in \{0, 2, 4, 6\}\},$$
$$\text{pre}(S_2) \equiv \{x \in \{0, 1, 4, 5\}\},$$
$$\text{pre}(S_3) \equiv \{x \in \{0, 1, 2, 3\}\}.$$

Poscondiciones Locales (su bit respectivo ha pasado a valer 1):

$$\begin{aligned}\text{post}(S_1) &\equiv \{x \in \{1,3,5,7\}\},\\ \text{post}(S_2) &\equiv \{x \in \{2,3,6,7\}\},\\ \text{post}(S_3) &\equiv \{x \in \{4,5,6,7\}\}.\end{aligned}$$

Comprobación de No Interferencia de Owicki-Gries: Debemos comprobar que la ejecución de S_2 o S_3 no invalida ni la precondición ni la poscondición de S_1. Tomemos por ejemplo $\text{pre}(S_1) \equiv x \in \{0,2,4,6\}$. Si se ejecuta S_2 $(x := x+2)$:

- Si $x = 0 \to x = 2 \in \{0,2,4,6\}$.
- Si $x = 4 \to x = 6 \in \{0,2,4,6\}$.

El aserto $x \in \{0,2,4,6\}$ se preserva estrictamente invariantemente bajo S_2. El mismo razonamiento se aplica simétricamente a todas las demás combinaciones de procesos y asertos.
Poscondición Global Concurrente: Al concluir el `coend`, se verifica la intersección de todas las poscondiciones locales:

$$x \in \text{post}(S_1) \cap \text{post}(S_2) \cap \text{post}(S_3) = \{1,3,5,7\} \cap \{2,3,6,7\} \cap \{4,5,6,7\} = \{\mathbf{7}\}.$$

Queda demostrado formalmente que el triple concurrente concluye con $\{x = 7\}$.

Ejercicio 21: Demostración de invariancia con cobegin

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.

Solución Detallada

1. Corrección Secuencial de cada Rama: Para cada proceso por separado, verificamos la preservación de la igualdad $x = y$:

- **Rama 1:**

  $$\{x = y\}\ \langle x := x-1\rangle\ \{x+1 = y\}\ \langle x := x+1\rangle\ \{x = y\}.$$

  Por el axioma de asignación: $(x-1)+1 = y \iff x = y$. Demostrado.

- **Rama 2:**

  $$\{x = y\}\ \langle y := y+1\rangle\ \{x = y-1\}\ \langle y := y-1\rangle\ \{x = y\}.$$

  Por el axioma de asignación: $x = (y+1)-1 \iff x = y$. Demostrado.

2. Demostración de No Interferencia de Owicki-Gries: El conjunto de variables escritas por el proceso 1 es $E(P_1) = \{x\}$, mientras que el proceso 2 escribe en $E(P_2) = \{y\}$. Para verificar que los pasos intermedios no interfieren destructivamente:

- Si se intercala un paso de P_2 cuando P_1 está en su estado intermedio $\{x+1=y\}$, P_2 ejecuta $\langle y := y+1\rangle$, llevando el estado a $\{x+2=y\}$.
- Como ambos procesos realizan operaciones complementarias ($+1$ y -1) en sus variables privadas respectivas, la diferencia neta $\Delta x - \Delta y$ al finalizar ambos es estrictamente $(0)-(0)=0$.

Formalmente, aplicando variables auxiliares o el invariante global $I \equiv (x_{\text{inicial}} - y_{\text{inicial}} = x - y + \delta_1 - \delta_2)$, al finalizar ambos procesos se tiene $\delta_1 = 0$ y $\delta_2 = 0$, concluyendo:

$$x - y = x_{\text{inicial}} - y_{\text{inicial}} = 0 \implies \{\mathbf{x = y}\}.$$

1.5 Bloque V: Reglas de Derivación de Hoare, Condicionales y Alias

Ejercicio 22: Regla de la conjunción

Usando la regla de la conjunción de la lógica de Hoare, demostrar que:

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

Solución Detallada

La **Regla de la Conjunción** establece que si un programa S satisface dos especificaciones con la misma precondición, satisface simultáneamente su conjunción:

$$\frac{\{P\}\ S\ \{Q\}, \qquad \{P\}\ S\ \{R\}}{\{P\}\ S\ \{Q \wedge R\}}$$

Para demostrar el triple requerido, consideramos los dos triples siguientes:

1. **Triple 1:** $\{i > 2\}\ i := 2 * i\ \{i > 2\}$ Aplicando el axioma de asignación:

   $$\{i > 2\}^{i}_{2i} \equiv \{2i > 2\} \equiv \{i > 1\}.$$

   Como $\{i > 2\} \implies \{i > 1\}$, por fortalecimiento de precondición el triple es cierto.

2. **Triple 2:** $\{V\}\ i := 2 * i\ \{i \text{ es par}\}$. O bien, con la variable inicial i_0:

   $$\{i = i_0 \wedge i_0 > 2\}\ i := 2 * i\ \{i = 2 \cdot i_0\}.$$

   Directamente por el axioma de asignación: $\{2i = 2i_0\} \equiv \{i = i_0\}$.

Aplicando la regla de la conjunción entre $\{i > 2\}\ i := 2 * i\ \{i = 2 \cdot i_0\}$ y la condición $i_0 > 2$:

$$\{i = i_0 \wedge i_0 > 2\}\ i := 2 * i\ \{i = 2 \cdot i_0 \wedge i_0 > 2\}.$$

Dado que $(i = 2 \cdot i_0 \wedge i_0 > 2) \implies (i > 4)$, por la regla de debilitamiento de poscondición concluimos:

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

Ejercicio 23: Intercambio y combinación de variables sin variable auxiliar

Sean A y B los valores iniciales de a y b respectivamente. Escribir un fragmento de código que tenga $\{a = A + B \land b = A - B\}$ como poscondición y demostrar formalmente que el código es correcto.

Solución Detallada

Queremos diseñar un bloque de sentencias C tal que:

$$\{a = A \land b = B\}\ C\ \{a = A + B \land b = A - B\}$$

sin recurrir a ninguna variable temporal auxiliar.

1. Propuesta de Código C:

```
a = a + b;
b = a - 2*b;
```

2. Demostración Formal Paso a Paso (Sustitución hacia atrás): Aplicamos la regla de composición secuencial:

- **Paso 2 (Segunda instrucción $b := a - 2 * b$):** Queremos que la poscondición sea $Q \equiv \{a = A + B \land b = A - B\}$. Por el axioma de asignación, sustituimos b por $(a - 2b)$ en Q:

$$\begin{aligned} Q[a-2b/b] &\equiv \{a = A + B \ \land\ a - 2b = A - B\} \\ &\equiv \{a = A + B \ \land\ (A + B) - 2b = A - B\} \\ &\equiv \{a = A + B \ \land\ 2B = 2b\} \\ &\equiv \{a = A + B \ \land\ b = B\}. \end{aligned}$$

  Por tanto, el triple para la segunda instrucción es:

$$\{a = A + B \land b = B\}\ b := a - 2 * b\ \{a = A + B \land b = A - B\}.$$

- **Paso 1 (Primera instrucción $a := a + b$):** Tomamos como poscondición intermedia $R \equiv \{a = A + B \land b = B\}$. Por el axioma de asignación, sustituimos a por $(a + b)$ en R:

$$\begin{aligned} R[a+b/a] &\equiv \{a + b = A + B \ \land\ b = B\} \\ &\equiv \{a + B = A + B \ \land\ b = B\} \\ &\equiv \{a = A \ \land\ b = B\}. \end{aligned}$$

  Por tanto, el triple para la primera instrucción es:

$$\{a = A \land b = B\}\ a := a + b\ \{a = A + B \land b = B\}.$$

Por la regla de composición secuencial, uniendo ambos pasos queda demostrado que el código es estrictamente correcto.

Ejercicio 24: Demostración formal de sentencia condicional

Demostrar que el siguiente triple de Hoare es correcto:

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

Solución Detallada

La **Regla del Condicional** (sentencia `if-then-else`) establece:

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

En este problema: $P \equiv V$, $B \equiv (a > 0)$, $S_1 \equiv (x := a)$, $S_2 \equiv (x := -a)$, y la poscondición objetivo es $Q \equiv \{x \geq 0 \wedge x^2 = a^2\}$.

1. **Demostración de la rama THEN (S_1):** Debemos probar $\{V \wedge a > 0\}\ x := a\ \{x \geq 0 \wedge x^2 = a^2\}$. Aplicamos el axioma de asignación sobre Q:

   $$Q[a/x] \equiv \{a \geq 0 \wedge a^2 = a^2\} \equiv \{a \geq 0\}.$$

   Como la precondición de la rama es $\{a > 0\}$, y trivialmente $(a > 0) \implies (a \geq 0)$, por la regla de fortalecimiento de precondición queda probado el primer triple:

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

2. **Demostración de la rama ELSE (S_2):** Debemos probar $\{V \wedge \neg(a > 0)\} \equiv \{a \leq 0\}\ x := -a\ \{x \geq 0 \wedge x^2 = a^2\}$. Aplicamos el axioma de asignación sobre Q:

   $$Q[-a/x] \equiv \{-a \geq 0 \wedge (-a)^2 = a^2\} \equiv \{a \leq 0 \wedge a^2 = a^2\} \equiv \{a \leq 0\}.$$

   La precondición generada por el axioma coincide exactamente con $\{a \leq 0\}$, por lo que el triple es un axioma directo.

Habiéndose probado ambas premisas, la regla del condicional concluye directamente la corrección del programa.

Ejercicio 25: Demostración de secuencia de asignaciones

Demostrar formalmente utilizando los axiomas de la lógica de Hoare que:

$$\{i \cdot j + 2j + 3i = 0\} \quad j := j + 3;\ i := i + 2 \quad \{i \cdot j = 6\}$$

Solución Detallada

Sea el programa compuesto $S \equiv S_1; S_2$ con $S_1 \equiv (j := j + 3)$ y $S_2 \equiv (i := i + 2)$, y la poscondición final $Q \equiv \{i \cdot j = 6\}$.
Paso 1: Cálculo del aserto intermedio para la segunda instrucción (S_2): Por el axioma de asignación:

$$Q[i + 2/i] \equiv \{(i + 2) \cdot j = 6\} \equiv \{i \cdot j + 2j = 6\}.$$

Por tanto, tenemos el triple demostrado:

$$\{i \cdot j + 2j = 6\}\ i := i + 2\ \{i \cdot j = 6\}.$$

Paso 2: Cálculo de la precondición inicial para la primera instrucción (S_1): Tomamos como poscondición requerida el aserto intermedio $R \equiv \{i \cdot j + 2j = 6\}$. Aplicamos

el axioma de asignación sustituyendo j por $(j+3)$ en R:

$$\begin{aligned} R[j+3/j] &\equiv \{i \cdot (j+3) + 2 \cdot (j+3) = 6\} \\ &\equiv \{i \cdot j + 3i + 2j + 6 = 6\} \\ &\equiv \{i \cdot j + 2j + 3i = 0\}. \end{aligned}$$

Por tanto, tenemos el triple demostrado:

$$\{i \cdot j + 2j + 3i = 0\}\ j := j + 3\ \{i \cdot j + 2j = 6\}.$$

Paso 3: Aplicación de la regla de composición: Uniendo ambos triples mediante la regla de composición secuencial:

$$\frac{\{i \cdot j + 2j + 3i = 0\}\ j := j + 3\ \{i \cdot j + 2j = 6\}, \quad \{i \cdot j + 2j = 6\}\ i := i + 2\ \{i \cdot j = 6\}}{\{i \cdot j + 2j + 3i = 0\} \quad j := j + 3;\ i := i + 2 \quad \{i \cdot j = 6\}}$$

El resultado coincide exactamente con la precondición dada en el enunciado. Q.E.D.

Ejercicio 26: Guarda booleana en la regla del bucle while

¿Por qué en la regla del bucle `while B do`, la condición B debe ser verdadera al comienzo de cada ejecución del cuerpo del bucle?

Solución Detallada

La regla de inferencia de Hoare para el bucle `while` (*regla de iteración*) se formula como:

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

donde I representa el invariante de bucle, B es la guarda o condición booleana y S es el cuerpo del bucle.
Razones fundamentales:

1. **Semántica operacional del bucle:** Por definición, el cuerpo S solo llega a ejecutarse si y solo si la evaluación de la guarda booleana B resulta `true`. Si B fuese falsa, el flujo de ejecución omite inmediatamente el cuerpo S y transfiere el control a la instrucción posterior.

2. **Precisión del invariante inductivo:** Al comenzar cualquier iteración de S, sabemos con total certeza que el estado actual satisface no solo el invariante general I, sino también la condición de entrada B. Esta premisa adicional $\{I \wedge B\}$ es estrictamente necesaria para poder demostrar que las transformaciones realizadas por S logran restablecer el invariante I al final de la iteración.

3. **Garantía de salida:** Al terminar el bucle, la condición de parada garantiza que la última evaluación de la guarda fue falsa ($\neg B$). Puesto que cada iteración preservó I, el aserto resultante final es forzosamente la conjunción $\{I \wedge \neg B\}$.

Ejercicio 27: El problema de los alias en la lógica de programas

Considerar una función con dos argumentos que se usa en un programa. Explicar por qué el uso de **alias** (*aliasing*) puede ser un problema grave para la verificación formal con la Lógica de Hoare.

Solución Detallada

El fenómeno de **aliasing** (alias) ocurre cuando dos o más identificadores, nombres de variables o punteros/referencias diferentes hacen referencia exactamente a la misma posición física de memoria en tiempo de ejecución.
Por qué rompe la Lógica de Hoare: El axioma fundamental de asignación de Hoare:

$$\{P[e/x]\}\ x := e\ \{P\}$$

se basa en la hipótesis implícita de **independencia sintáctica**: asume que modificar la variable x **no afecta a ninguna otra variable** presente en la aserción P.
Ejemplo Demostrativo: Supongamos un procedimiento con dos argumentos pasados por referencia:

```
procedure duplicar(var x : integer; var y : integer)
begin
  x := 10;
  y := 20;
end;
```

Si suponemos variables distintas, la lógica estándar deduciría:

$$\{V\} \quad x := 10;\ y := 20 \quad \{x = 10 \land y = 20\}.$$

Sin embargo, si el invocador llama a `duplicar(a, a)`, los parámetros x e y son **alias** de la misma variable a.

- La sentencia $y := 20$ sobreescribe silenciosamente la posición referenciada por x.
- El resultado real en memoria es $a = 20$, lo que hace que la poscondición formal $\{x = 10 \land y = 20\}$ sea **FALSA** ($20 = 10 \land 20 = 20 \implies$ Falso).

Por esta razón, la semántica axiomática clásica prohíbe el aliasing o exige modelar la memoria de forma explícita mediante funciones de estado de montón (*heap/store semantics*) o Lógica de Separación (*Separation Logic*).

1.6 Bloque VI: Asignación a Arrays y Precondición Más Débil

Ejercicio 28: Asignación a elementos de un vector

Demostrar la corrección del siguiente triple:

$$\{a[i] \geq 0\} \quad a[i] := a[i] + a[j] \quad \{a[i] \geq a[j]\}$$

Solución Detallada

Para razonar sobre elementos de un array, la lógica de Hoare modela la asignación a un elemento de un vector $a[i] := e$ como una actualización de la función que representa el

array completo:

$$a := (a;\ i : e)$$

donde el array modificado $(a;\ i : e)$ tiene la propiedad:

$$(a;\ i : e)[k] = \begin{cases} e & \text{si } k = i \\ a[k] & \text{si } k \neq i \end{cases}$$

Queremos demostrar $\{a[i] \geq 0\}\ a[i] := a[i] + a[j]\ \{a[i] \geq a[j]\}$. Aplicando la sustitución formal de arrays en la poscondición $Q \equiv \{a[i] \geq a[j]\}$:

$$Q[(a;\ i : a[i] + a[j])/a] \equiv \{(a;\ i : a[i] + a[j])[i]\ \geq\ (a;\ i : a[i] + a[j])[j]\}\,.$$

Por definición, el término de la izquierda siempre es $(a;\ i : a[i] + a[j])[i] = a[i] + a[j]$. Para el término de la derecha, distinguimos dos casos según si los índices coinciden o no:

- **Caso 1 ($i = j$, mismo elemento):** En este caso, $(a;\ i : a[i] + a[j])[j] = (a;\ i : a[i] + a[i])[i] = a[i] + a[i]$. La condición requerida resulta:

  $$a[i] + a[i] \geq a[i] + a[i] \iff 0 \geq 0 \quad \text{(Tautología, siempre verdadero } V\text{)}.$$

  Como $\{a[i] \geq 0\} \implies V$, se cumple trivialmente.

- **Caso 2 ($i \neq j$, elementos distintos):** Dado que $j \neq i$, el elemento en la posición j no sufre modificación alguna:

  $$(a;\ i : a[i] + a[j])[j] = a[j].$$

  La condición requerida resulta:

  $$a[i] + a[j] \geq a[j] \iff a[i] \geq 0.$$

  Esta expresión coincide exactamente con la precondición dada en el enunciado $\{a[i] \geq 0\}$.

En ambos casos, la precondición $\{a[i] \geq 0\}$ garantiza que tras la asignación se verificará $a[i] \geq a[j]$. Queda demostrado.

Ejercicio 29: Cálculo de Precondición más débil con arrays

Hallar la precondición $\{P\}$ que hace que el siguiente triple sea correcto:

$$\{P\} \quad a[i] := 2 * b \quad \{j \leq i\ \wedge\ k < i\ \wedge\ a[i] + a[j-1] + a[k] > b\}$$

Solución Detallada

La instrucción asigna al elemento $a[i]$ el valor $2b$, transformando el array a en $(a;\ i : 2b)$. Por el axioma de asignación para vectores, la precondición más débil se obtiene sustituyendo el array a por $(a;\ i : 2b)$ en la poscondición:

$$P \equiv (j \leq i\ \wedge\ k < i\ \wedge\ (a;\ i : 2b)[i] + (a;\ i : 2b)[j-1] + (a;\ i : 2b)[k] > b)\,.$$

Evaluamos cada acceso al array en el nuevo estado:

1. $(a;\ i : 2b)[i] = 2b$ (acceso directo a la posición recién escrita).

2. Para el índice k: la poscondición impone $k < i$, por lo que $k \neq i$. Por la propiedad del array actualizado:
$$(a;\ i : 2b)[k] = a[k].$$

3. Para el índice $j-1$: la poscondición impone $j \leq i$, de lo que se deduce que $j-1 < i$, por lo que $(j-1) \neq i$. Por consiguiente:
$$(a;\ i : 2b)[j-1] = a[j-1].$$

Sustituyendo estos valores simplificados en la desigualdad:

$$2b + a[j-1] + a[k] > b \iff a[j-1] + a[k] > b - 2b \iff a[j-1] + a[k] > -b.$$

Por tanto, la precondición más débil exacta $\{P\}$ es:

$$\{\mathbf{P}\} \equiv \{\mathbf{j} \leq \mathbf{i} \ \wedge\ \mathbf{k} < \mathbf{i} \ \wedge\ \mathbf{a[j-1]} + \mathbf{a[k]} > -\mathbf{b}\}.$$

1.7 Bloque VII: Bucles, Invariantes y Terminación

Ejercicio 30: Demostración de terminación de bucle para n >0

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

```
1 i := 1;
2 f := 1;
3 while (i <> n) do begin
4   i := i + 1;
5   f := f * i;
6 end;
```

Solución Detallada

Para demostrar formalmente la **terminación** de un bucle se utiliza el Teorema de Terminación basado en una **función variante** o **función de cota** V : Estado $\to \mathbb{Z}$ sobre un conjunto bien fundado (generalmente $\mathbb{N}$). Debemos probar:

1. V está acotada inferiormente mientras el bucle continúe iterando: $\{I \wedge B\} \implies V \geq 0$.

2. Cada ejecución del cuerpo del bucle decrementa estrictamente el valor de V: si el valor antes de iterar es V_0, al final de la iteración se cumple $V < V_0$.

1. Definición de la Función de Cota V: La guarda del bucle es $B \equiv (i \neq n)$. Como i parte de 1 y se incrementa de 1 en 1 hacia n, definimos:

$$V(i) = n - i.$$

2. Demostración de Cota Inferior: El invariante del bucle es $I \equiv (1 \leq i \leq n \wedge f = i!)$. Mientras la condición del bucle $B \equiv (i \neq n)$ sea verdadera:

$$(1 \leq i \leq n \ \wedge\ i \neq n) \implies 1 \leq i \leq n-1 \implies n - i \geq 1 > 0.$$

Por tanto, $V(i) = n - i \geq 1 \geq 0$.

3. Decrecimiento Estricto en cada Iteración: Sea $V_0 = n - i$ el valor de la función de cota al inicio de la iteración. El cuerpo del bucle ejecuta $i := i + 1$. El nuevo valor de la cota V' es:

$$V' = n - (i + 1) = (n - i) - 1 = V_0 - 1.$$

Como $V_0 - 1 < V_0$, la función de cota disminuye estrictamente en 1 en cada iteración.
Conclusión: Como $V(i)$ toma valores en los números enteros, está acotada inferiormente por 0 y decrece estrictamente en cada ciclo, el bucle no puede iterar indefinidamente y termina en exactamente $n - 1$ iteraciones.

Ejercicio 31: Obtención de poscondiciones apropiadas

Para cada uno de los siguientes fragmentos de código, obtener la poscondición más apropiada:

(a) $\{i < 10\}\ i := 2 * i + 1;$

(b) $\{i > 0\}\ i := i - 1;$

(c) $\{i > j\}\ i := i + 1;\ j := j + 1;$

(d) $\{V\}\ i := 3;\ j := 2 * i;$

Solución Detallada

Aplicando directamente el axioma de asignación y composición secuencial:

(a) **Fragmento (a):** Si $i_0 < 10$, y la nueva variable es $i = 2i_0 + 1$, despejando $i_0 = (i - 1)/2 < 10 \iff i < 21$. Poscondición: $\{\mathbf{i < 21}\}$.

(b) **Fragmento (b):** Si $i_0 > 0$, con $i_0 \in \mathbb{Z} \implies i_0 \geq 1$. Tras $i = i_0 - 1 \implies i_0 = i + 1 \geq 1 \iff i \geq 0$. Poscondición: $\{\mathbf{i \geq 0}\}$ (o equivalentemente $\{i > -1\}$).

(c) **Fragmento (c):** Ambos operandos se incrementan simultáneamente en 1 unidad:

$$\{i > j\}\ i := i + 1\ \{i > j + 1\}\ j := j + 1\ \{i > j\}.$$

Poscondición: $\{\mathbf{i > j}\}$.

(d) **Fragmento (d):** Con precondición trivial $\{V\}$, tras $i := 3$ se tiene $i = 3$. Luego $j := 2 * 3 = 6$. Poscondición: $\{\mathbf{i = 3 \wedge j = 6}\}$.

Ejercicio 32: Verificación de código secuencial con reglas de Hoare

Verificar formalmente el siguiente código indicando todas las reglas y axiomas utilizados:

$$\{y > 0\} \quad x := x + y;\ x := x - y \quad \{x = x_0 \wedge y > 0\}$$

Solución Detallada

Sea x_0, y_0 los valores iniciales de las variables antes de ejecutar el bloque, con precondición $P \equiv \{y = y_0 \wedge y_0 > 0 \wedge x = x_0\}$. La poscondición buscada es $Q \equiv \{x = x_0 \wedge y = y_0 \wedge y > 0\}$.
1. Segunda Asignación ($x := x - y$): Por el **Axioma de Asignación**:

$$Q[x - y/x] \equiv \{x - y = x_0 \wedge y = y_0 \wedge y > 0\} \equiv \{x = x_0 + y \wedge y = y_0 \wedge y > 0\}.$$

Denotamos este aserto intermedio como R. El triple:

$$\{x = x_0 + y \land y = y_0 \land y > 0\}\ x := x - y\ \{x = x_0 \land y = y_0 \land y > 0\}$$

es un axioma.
2. Primera Asignación ($x := x + y$): Aplicamos el **Axioma de Asignación** sobre el aserto intermedio R:

$$R[x + y/x] \equiv \{(x + y) = x_0 + y \land y = y_0 \land y > 0\} \equiv \{x = x_0 \land y = y_0 \land y > 0\}.$$

El triple:

$$\{x = x_0 \land y = y_0 \land y > 0\}\ x := x + y\ \{x = x_0 + y \land y = y_0 \land y > 0\}$$

es un axioma.
3. Regla de Composición Secuencial: Por la regla de composición entre (2) y (1):

$$\frac{\{P\}\ x := x + y\ \{R\}, \qquad \{R\}\ x := x - y\ \{Q\}}{\{P\}\quad x := x + y;\ x := x - y\quad \{Q\}}$$

Queda verificado formalmente.

Ejercicio 33: Verificación de estructura condicional if-then

Verificar el siguiente código indicando todas las reglas usadas:

$$\{V\}\quad \text{if } x < 0 \text{ then } x := -x\quad \{x \geq 0\}$$

Solución Detallada

La sentencia `if B then S` equivale sintácticamente a `if B then S else null`. La **Regla del Condicional** requiere probar dos triples:

$$\frac{\{V \land x < 0\}\ x := -x\ \{x \geq 0\}, \qquad \{V \land \neg(x < 0)\}\ \text{null}\ \{x \geq 0\}}{\{V\}\ \text{if } x < 0 \text{ then } x := -x\ \{x \geq 0\}}$$

1. Rama THEN ($x < 0$): Aplicamos el axioma de asignación sobre la poscondición $Q \equiv \{x \geq 0\}$:

$$Q[-x/x] \equiv \{-x \geq 0\} \equiv \{x \leq 0\}.$$

Por el axioma de asignación: $\{x \leq 0\}\ x := -x\ \{x \geq 0\}$. Como la precondición de la rama es $\{x < 0\}$ y $(x < 0) \implies (x \leq 0)$, por la **Regla de Fortalecimiento de Precondición**:

$$\{x < 0\}\ x := -x\ \{x \geq 0\}.$$

2. Rama ELSE ($\neg(x < 0) \equiv x \geq 0$): Por el **Axioma de la Sentencia Vacía** (`null`):

$$\{x \geq 0\}\ \text{null}\ \{x \geq 0\}.$$

Aplicando la regla del condicional a ambas ramas, queda demostrado el triple con poscondición $\{x \geq 0\}$.

Ejercicio 34: Verificación de un algoritmo de búsqueda con invariante

Verificar el siguiente segmento de programa para encontrar el máximo de un array $a[1..n]$, suponiendo el invariante dado:

$$I \equiv \left(\bigwedge_{k=1}^{i} (\text{máx} \geq a[k]) \ \wedge \ 1 \leq i \leq n+1 \right)$$

```
1 max := a[1];
2 i := 1;
3 while (i <> n + 1) do begin
4   if (a[i] >= max) then
5     max := a[i];
6   i := i + 1;
7 end;
```

poscondición final a demostrar: $\left\{ \bigwedge_{k=1}^{n} (\text{máx} \geq a[k]) \right\}$.

Solución Detallada

Para demostrar la corrección parcial del bucle mediante la **Regla de Iteración de Hoare**:

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

debemos demostrar tres etapas formales:

Etapa 1: Inicialización (Establecimiento del invariante antes del bucle): Debemos probar $\{n \geq 1\}$ máx $:= a[1];\ i := 1\ \{I\}$. Por sustitución hacia atrás:

$$I[1/i, a[1]/\text{máx}] \equiv \left(\bigwedge_{k=1}^{1} (a[1] \geq a[k]) \ \wedge \ 1 \leq 1 \leq n+1 \right) \equiv (a[1] \geq a[1] \wedge 1 \leq n+1) \equiv V.$$

Por tanto, la inicialización establece válidamente el invariante I.

Etapa 2: Mantenimiento del Invariante por el cuerpo del bucle: La guarda es $B \equiv (i \neq n+1)$. El estado antes de iterar cumple:

$$\{I \wedge B\} \equiv \left(\bigwedge_{k=1}^{i-1} (\text{máx} \geq a[k]) \ \wedge \ \text{máx} \geq a[i] \text{ o no } \wedge \ 1 \leq i \leq n \right).$$

Analizamos el condicional:

- Si $a[i] \geq$ máx, se ejecuta máx $:= a[i]$. En el nuevo estado, máx es mayor o igual que el nuevo elemento y, por transitividad, mayor o igual que todos los anteriores.
- Si $a[i] <$ máx, se omite la asignación. El máximo previo ya era estrictamente mayor que $a[i]$ y mayor o igual que todos los anteriores.

En ambos casos, tras la sentencia condicional se satisface:

$$\bigwedge_{k=1}^{i} (\text{máx} \geq a[k]).$$

A continuación, la instrucción $i := i+1$ actualiza el contador. Al sustituir i por $i+1$, el rango cubierto es exactamente $\{1, \ldots, (i'-1)\}$, preservando la forma del invariante I para la siguiente iteración.

Etapa 3: Poscondición al finalizar el bucle: Al terminar el bucle se cumple $\{I \wedge \neg B\}$:

$$\neg(i \neq n+1) \iff i = n+1.$$

Sustituyendo $i = n+1$ en el invariante I:

$$I \wedge (i = n+1) \implies \bigwedge_{k=1}^{n+1-1} (\text{máx} \geq a[k]) \equiv \bigwedge_{\mathbf{k=1}}^{\mathbf{n}} (\text{máx} \geq \mathbf{a[k]}).$$

La poscondición coincide exactamente con la especificación del máximo de los n elementos del array.

Ejercicio 35: Evaluación del número combinatorio con bucle único

Dados $n \geq 0, i \leq n$, demostrar que el siguiente segmento de programa evalúa el coeficiente binomial $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ dado el invariante:

$$I \equiv \big((i > k \ \vee \ afact = i!) \ \wedge \ (n - i > k \ \vee \ bfact = (n-i)!) \ \wedge \\ fact = k! \ \wedge \ 0 \leq k \leq n\big)$$

```
1 k := 0; fact := 1;
2 while (k <> n) do begin
3   k := k + 1;
4   fact := fact * k;
5   if (k <= i) then afact := fact;
6   if (k <= n - i) then bfact := fact;
7 end;
8 bcof := fact / (afact * bfact);
```

Solución Detallada

1. Verificación de la Inicialización: Antes de entrar al bucle: $k = 0, fact = 1 = 0!$. Como $i \geq 0$ y $n - i \geq 0$:

- Si $i > 0$, la disyunción $(i > 0 \vee afact = i!)$ es inmediatamente verdadera por el primer término $(i > 0)$. Si $i = 0$, $afact$ se asignará adecuadamente o se evalúa con el factorial vacío $0! = 1$.
- Análogamente para $n - i \geq 0$.

El invariante I se satisface formalmente antes de comenzar.
2. Mantenimiento del Invariante en el Bucle: En cada iteración, k se incrementa en 1 $(k' = k + 1)$ y $fact' = fact \times k' = k! \times k' = k'!$.

- Mientras $k' \leq i$, el condicional actualiza $afact := fact' = k'!$. Cuando k' alcanza exactamente el valor i, se ejecuta la asignación que deja fijado $afact = i!$. Para las iteraciones posteriores $(k' > i)$, la condición $k' \leq i$ es falsa, por lo que $afact$ nunca vuelve a modificarse, preservando eternamente $afact = i!$.
- El mismo razonamiento se aplica idénticamente a $bfact$: cuando k' alcanza $n - i$, se asigna $bfact = (n-i)!$, valor que permanece congelado hasta el final.

Por tanto, el invariante I se mantiene inductivamente en cada iteración.

3. Terminación y Cálculo del Resultado: El bucle finaliza cuando $\neg(k \neq n) \iff k = n$. Sustituyendo $k = n$ en el invariante I:

- Como $i \leq n$, la condición $i > n$ es falsa, obligando a que se cumpla el segundo término de la disyunción: $\mathbf{afact = i!}$.
- Como $n - i \leq n$, análogamente se cumple: $\mathbf{bfact = (n - i)!}$.
- El valor acumulado del factorial completo es: $\mathbf{fact = n!}$.

Por último, la asignación final calcula:

$$bcof := \frac{fact}{afact \times bfact} = \frac{\mathbf{n!}}{\mathbf{i!(n-i)!}} = \binom{\mathbf{n}}{\mathbf{i}}.$$

Queda demostrada la corrección total del programa.

Ejercicio 36: Demostración de terminación del algoritmo de búsqueda de máximo

Demostrar la terminación del fragmento de programa dado en el problema 34 anterior. ¿Qué condición se debe imponer sobre el parámetro n para poder realizar la demostración formal?

Solución Detallada

1. Condición necesaria sobre el parámetro n: Para que el bucle termine de forma determinista, se debe imponer la condición:

$$\mathbf{n \geq 0} \quad \text{(y en particular } \mathbf{n \geq 1} \text{ para que el acceso inicial } a[1] \text{ esté bien definido).}$$

Justificación del contraejemplo si $n < 0$: El bucle inicializa $i := 1$ y su guarda de continuación es $(i \neq n + 1)$. Como el cuerpo incrementa i estrictamente de uno en uno $(i \leftarrow i + 1)$:

- La secuencia de valores generada por i es $\{1, 2, 3, 4, \dots\}$.
- Si $n < 0$, entonces la meta $n+1 \leq 0$. Como i parte de $1 > n+1$ y crece positivamente, i **jamás igualará a** $n + 1$, produciéndose un **bucle infinito** (divergencia).
- Por consiguiente, $n \geq 0$ es la precondición mínima indispensable para garantizar convergencia.

2. Demostración Formal con Función Variante (Cota): Bajo la precondición $n \geq 0$, definimos la función variante:

$$V(i) = (n + 1) - i.$$

Comprobación de las dos condiciones de terminación:

1. **Cota inferior positiva mientras el bucle ejecuta:** Mientras la guarda $(i \neq n+1)$ sea verdadera, y dado que por el invariante $1 \leq i \leq n + 1$, se tiene $i \leq n$, luego:

   $$V(i) = n + 1 - i \geq n + 1 - n = 1 > 0.$$

   La función de cota está estrictamente acotada inferiormente por 0 en todos los estados admisibles.

2. **Decrecimiento monótono estricto:** Al inicio de la iteración, el valor de la función de cota es $V_0 = n + 1 - i$. La sentencia $i := i + 1$ modifica el valor a:

$$V' = n + 1 - (i + 1) = (n + 1 - i) - 1 = V_0 - 1.$$

Puesto que $V' < V_0$, la función variante decrementa exactamente en 1 en cada paso.

Por el principio de inducción bien fundada, el bucle concluye obligatoriamente tras exactamente n iteraciones.