Sistemas Concurrentes y Distribuidos

Problemas Imprescindibles para el Primer Parcial

Sprint Intensivo de Emergencia (4 Días) — Convocatoria Oficial 29 Octubre 2026

Grado en Ingeniería Informática + Doble Grado DGIIM / ADE

Universidad de Granada (UGR)

Selección Estratégica de 19 Ejercicios Clave:

Bloque I (7 ejercicios): Exclusión Mutua en Memoria Compartida (Tema 2-2)

Bloque II (6 ejercicios): Monitores y Sincronización Condicional (Tema 2-1)

Bloque III (6 ejercicios): Concurrencia Básica, Bernstein y Lógica de Hoare (Tema 1)

Resolución exhaustiva paso a paso con código C++11, demostraciones de invariantes, tablas de intercalación y contraejemplos oficiales de examen (Prof. Manuel I. Capel Tuñón).

Plataforma **repasaYA**

Documento Especial de Preparación Rápida

Estrategia de Examen: La Táctica de los 19 Problemas

Clave de Examen: Estructura de la Parte Práctica en el Parcial de SCD

El examen parcial consta de dos bloques a realizar en **100 minutos**:

1. **Parte 1: Test con Penalización (35 minutos):** Preguntas teóricas de 5 alternativas (cada 2 fallos restan 1 acierto).
2. **Parte 2: Resolución de Problemas (55–60 minutos):** Se presentan **3 problemas** y el estudiante debe **elegir y resolver exactamente 2**.

La terna de problemas propuesta por el profesor suele responder al siguiente patrón:

- **Problema A:** Algoritmo Software o Hardware de Exclusión Mutua (Tema 2-2).
- **Problema B:** Monitor de Sincronización Condicional en C++11 (Tema 2-1).
- **Problema C:** Lógica de Hoare, Bernstein o No Interferencia de Owicki-Gries (Tema 1).

Recomendación de Combate: La combinación más rápida, segura y con menor propensión a errores algebraicos es **elegir el Problema B (Monitor) y el Problema A (Exclusión Mutua)**. Este documento reúne exactamente los 19 ejercicios imprescindibles de las hojas oficiales que cubren todos los patrones de examen posibles.

Índice general

Estrategia de Examen: La Táctica de los 19 Problemas **3**

1. Exclusión Mutua en Memoria Compartida (Tema 2-2) **7**

2. Monitores y Sincronización Condicional (Tema 2-1) **19**

3. Concurrencia Básica, Bernstein y Lógica de Hoare (Tema 1) **27**

Bloque Temático 1

Exclusión Mutua en Memoria Compartida (Tema 2-2)

Metadatos de la Relación y Fuentes:

Relevancia en el Examen: Representa el 66 % del peso práctico del Tema 2.
• **Conceptos obligatorios:** Intercalación paso a paso, contraejemplo de Hyman, prueba por reducción al absurdo de Peterson para 2 procesos, lemas de etapas del Filtro de Peterson ($N(N-1)/2$), livelock en paso síncrono del 4.º intento, sincronización hardware con instrucción atómica `swap` y análisis de guardas modificadas en Dijkstra.

Ejercicio 55: Dos Indicadores sin Variable de Turno (Cuarto Intento de Dijkstra)

Sean dos procesos con las variables compartidas `b[0]` y `b[1]`, inicialmente falsas, y el siguiente protocolo, ejecutado por $P(i)$ ($j = 1 - i$):

```
1 b[i] := true;
2 while b[j] do
3     b[i] := false;
4     while b[j] do skip;
5     b[i] := true;
6 endwhile;
7 <seccion critica>;
8 b[i] := false;
```

Analizar por separado: (a) exclusión mutua; (b) ausencia de interbloqueo; (c) ausencia de inanición. Si alguna propiedad no se satisface, construir un escenario concreto. Comparar el papel de la variable de turno en este algoritmo y en el de Peterson para dos procesos.

Solución Detallada

Apartado (a): Exclusión Mutua (SÍ se garantiza):
Demostraremos por contradicción que nunca pueden coexistir ambos procesos en la SC:

- Para que P_0 se encuentre en la SC, ha debido salir del bucle `while b[1]` evaluando $b[1] ==$ false. Previamente fijó $b[0] :=$ true, y no ejecuta $b[0] :=$ false hasta abandonar la SC. Por tanto, mientras $P_0 \in$ SC, $b[0] =$ true.
- Para que P_1 esté simultáneamente en la SC, debió evaluar su guarda `while b[0]` como `false`.
- Si $P_0 \in$ SC, $b[0] =$ true. La única forma en que P_1 pudo ver $b[0] ==$ false es que P_0 se encontrara temporalmente en su fase de cortesía ($b[0] :=$ false, línea 3).

- No obstante, si P_0 estaba en la línea 3, para que P_1 continuase hacia la SC debió tener $b[1] = \text{true}$. Esto hace que P_0 quede atrapado en su bucle interno `while b[1] do skip;` (línea 4) mientras P_1 permanezca activo, impidiendo que P_0 reasigne $b[0] := \text{true}$ y alcance la SC.
- Es lógicamente imposible que ambos procesos superen a la vez sus respectivos bucles externos. Por tanto:

$$N_{SC} \leq 1 \quad \text{(Exclusión mutua demostrada)}$$

Apartado (b): Ausencia de Interbloqueo (NO se garantiza: Livelock Dinámico):
El algoritmo no sufre interbloqueo estático (las instrucciones no se bloquean permanentemente en cerrojos mudos), pero adolece de un **bloqueo activo o livelock** simétrico en paso síncrono (*lock-step*):

Paso	**Acción P_0**	**Acción P_1**	$b[0]$	$b[1]$	**Estado del Sistema**
1	$b[0] := \text{true}$	$b[1] := \text{true}$	true	true	Ambos manifiestan intención.
2	Evalúa: `while b[1]` (V)	Evalúa: `while b[0]` (V)	true	true	Ambos entran al bucle externo.
3	$b[0] := \text{false}$ (cede)	$b[1] := \text{false}$ (cede)	false	false	Cortesía simultánea.
4	Evalúa interno: `b[1]` (F)	Evalúa interno: `b[0]` (F)	false	false	Ambos salen del bucle interno.
5	$b[0] := \text{true}$	$b[1] := \text{true}$	true	true	Ambos reafirman intención.
6	*El sistema regresa exactamente al Paso 2. Ciclo infinito sin entrada a SC.*				

Si el planificador ejecuta ambos procesos a ritmos idénticos, ninguno accede jamás a la SC. Se viola la propiedad de progreso.
Apartado (c): Ausencia de Inanición (NO se garantiza):
Como consecuencia directa del livelock anterior, ambos procesos sufren inanición simultánea. Asimismo, en una ejecución asimétrica donde P_0 sea sistemáticamente más lento que P_1, P_0 puede quedar bloqueado en su bucle interno mientras P_1 aprovecha para entrar y salir repetidas veces.
Comparación con el papel de la variable de turno en Peterson:

- **En este algoritmo:** Las decisiones se sustentan exclusivamente en indicadores simétricos $b[0]$ y $b[1]$. Ante una contienda concurrente, ambos observan un estado idéntico y adoptan la misma conducta de cortesía ("pase usted primero"), generando la oscilación infinita.
- **En el Algoritmo de Peterson:** Se incorpora la variable compartida `turno`. Cuando ambos procesos compiten y asignan `quiere[i] := true`, ambos ejecutan inmediatamente `turno := 1-i`. La escritura atómica de `turno` en el hardware resuelve el conflicto de forma asimétrica e instantánea: el último proceso en escribir sobrescribe la variable y asume el rol de víctima. La condición de espera `quiere[j] and turno == j` resulta cierta para uno solo de los procesos, forzando a que el otro entre de inmediato a la SC.

Ejercicio 56: El Algoritmo de Hyman y la Violación de Exclusión Mutua

Se comparte `c[0]=c[1]=1` y `turno=1`. Cada proceso $P(i)$, con $j = 1 - i$, ejecuta:

```
c[i] := 0;
while turno != i do
    while c[j] = 0 do skip;
    turno := i;
endwhile;
<seccion critica>;
c[i] := 1;
```

Determinar si el algoritmo garantiza exclusión mutua. Si no la garantiza, proporcionar una intercalación de instrucciones que lleve simultáneamente a ambos procesos a sus secciones críticas. Identificar exactamente la decisión de diseño que permite el fallo.

Solución Detallada

1. Determinación de la Propiedad:
EL ALGORITMO DE HYMAN NO GARANTIZA LA EXCLUSIÓN MUTUA. A pesar de haber sido publicado en 1966 en *Communications of the ACM* como una simplificación del algoritmo de Dekker, contiene una sutil condición de carrera que invalida la seguridad.

2. Intercalación de Instrucciones (Contraejemplo Detallado):
Estado inicial: `c[0] = 1`, `c[1] = 1`, `turno = 1`.

Paso	Proceso	Instrucción Ejecutada	$c[0]$	$c[1]$	turno	Efecto y Comentario
1	P_0	`c[0] := 0;`	0	1	1	P_0 manifiesta interés.
2	P_0	Evalúa `while turno != 0`	0	1	1	Condición **True** ($1 \neq 0$). Entra.
3	P_0	Evalúa `while c[1] == 0`	0	1	1	Condición **False** ($c[1] = 1$). No espera.
4	—	*Planificador suspende a P_0 antes de asignar* `turno := 0`				Interrupción de contexto crítica.
5	P_1	`c[1] := 0;`	0	0	1	P_1 manifiesta interés.
6	P_1	Evalúa `while turno != 1`	0	0	1	Condición **False** (turno = 1).
7	P_1	**Entra a Sección Crítica**	0	0	1	P_1 omite el bucle y accede.
8	P_0	`turno := 0;`	0	0	0	P_0 se reanuda y cambia turno.
9	P_0	Reevalúa `while turno != 0`	0	0	0	Condición **False** (turno = 0).
10	P_0	**Entra a Sección Crítica**	0	0	0	P_0 sale del bucle y accede.

ESTADO FINAL: $P_0 \in$ **SC** $\wedge$ $P_1 \in$ **SC** $\implies N_{SC} = 2$. **VIOLACIÓN DE EXCLUSIÓN MUTUA.**

3. Decisión de Diseño Errónea (Causa Raíz del Fallo):
El defecto radica en una vulnerabilidad de tipo **TOCTOU (Time-of-Check to Time-of-Use)** unida a una apropiación no sincronizada del turno:

1. **Desacoplamiento entre comprobación y apropiación:** P_0 verifica que P_1 no está en el bucle (`c[1] == 1`), pero antes de ejecutar la asignación `turno := 0`, P_1 puede activarse.

2. **Confianza ciega en el turno previo:** P_1 ingresa a la SC sin comprobar el estado de $c[0]$, amparándose exclusivamente en que turno == 1.

3. **Apropiación incondicional del turno:** Cuando P_0 despierta, ejecuta turno := 0 sin revalidar si P_1 ha iniciado la SC mientras estaba suspendido, saliendo del bucle sin

encontrar ningún obstáculo.

Ejercicio 57: Algoritmo de Peterson para Dos Procesos: Demostración y Memoria Débil

Sean `quiere[0]=quiere[1]=false` y `turno` compartido. Cada $P(i)$ ejecuta:

```
1 quiere[i] := true;
2 turno := 1-i;
3 while quiere[1-i] and turno = 1-i do skip;
4 <seccion critica>;
5 quiere[i] := false;
```

(a) Demostrar la exclusión mutua. (b) Argumentar el progreso y la ausencia de inanición, indicando las hipótesis de planificación. (c) Investigar qué sucede si se intercambia el orden de las dos primeras asignaciones. (d) Explicar por qué la corrección en un modelo secuencialmente consistente no autoriza a usar sin más variables ordinarias en un programa real con memoria débil.

Solución Detallada

Apartado (a): Demostración Rigurosa de Exclusión Mutua:
Demostraremos por contradicción lógica:

- Supongamos que en el instante t, ambos procesos P_0 y P_1 se encuentran simultáneamente en la SC.
- Para que P_0 esté en SC, debió evaluar su guarda de espera como falsa:

$$\neg(\texttt{quiere[1]} \wedge \texttt{turno} = 1) \iff (\neg\texttt{quiere[1]} \vee \texttt{turno} \neq 1)$$

- Para que P_1 esté en SC, análogamente debió cumplirse:

$$\neg(\texttt{quiere[0]} \wedge \texttt{turno} = 0) \iff (\neg\texttt{quiere[0]} \vee \texttt{turno} \neq 0)$$

- Puesto que ambos están en SC y no han ejecutado el protocolo de salida, necesariamente `quiere[0]` = true y `quiere[1]` = true.
- En consecuencia, para que ambas expresiones sean verdaderas debe cumplirse forzosamente:

$$(\texttt{turno} \neq 1 \implies \texttt{turno} = 0) \quad \wedge \quad (\texttt{turno} \neq 0 \implies \texttt{turno} = 1)$$

- La variable `turno` es una celda atómica en memoria que sólo puede almacenar un valor escalar en cada instante. Como la asignación a `turno` es la última instrucción antes del bucle en ambos procesos, una de las dos asignaciones debió completarse en último lugar en el bus de memoria:
  - Si P_0 escribió en último lugar, `turno` $= 1$. Entonces P_0 no pudo observar `turno` $= 0$, quedando bloqueado en el bucle.
  - Si P_1 escribió en último lugar, `turno` $= 0$. Entonces P_1 no pudo observar `turno` $= 1$, quedando bloqueado en el bucle.

- Se llega a una contradicción formal insalvable. Por tanto, $N_{SC} \leq 1$.

Apartado (b): Progreso y Ausencia de Inanición:

- **Progreso:** Si ningún proceso está en la SC y ambos compiten a la vez, `turno` adopta el valor 0 o 1. El proceso para el cual `turno` $\neq 1-i$ (aquel que escribió primero) supera el bucle y accede. Si solo un proceso desea entrar, `quiere[1-i]` = false, accediendo sin retardo alguno.
- **Ausencia de Inanición:** Asumiendo **planificación débilmente justa** (los procesos activos reciben ciclos de CPU y terminan su SC en tiempo finito), cuando el proceso rival P_j abandona la SC ejecuta `quiere[j]` := false. En ese instante, la guarda de P_i se vuelve inmediatamente falsa y P_i entra. Si P_j intentara reingresar con rapidez, ejecutará `turno` := i, cediéndole la prioridad a P_i. La cota de espera es estricta: **a lo sumo 1 entrada del proceso rival**.

Apartado (c): Intercambio de las dos primeras asignaciones:
Si el protocolo se altera a:

```
turno := 1-i;
quiere[i] := true;
while quiere[1-i] and turno = 1-i do skip;
```

Se destruye la Exclusión Mutua. Traza de contraejemplo:

1. P_0 ejecuta `turno` := 1.
2. Planificador conmuta a P_1.
3. P_1 ejecuta `turno` := 0.
4. P_1 ejecuta `quiere[1]` := true.
5. P_1 evalúa el bucle: `quiere[0]` sigue en `false`, luego P_1 entra a la SC.
6. Planificador conmuta a P_0.
7. P_0 ejecuta `quiere[0]` := true.
8. P_0 evalúa el bucle: `turno` $= 0$. Como `turno` $== 0 \neq 1$, la guarda es falsa.
9. P_0 accede a la SC **simultáneamente con** P_1. Violación de exclusión mutua.

Apartado (d): Consistencia Secuencial frente a Memoria Débil (*Weak Memory*):
La demostración matemática asume el modelo de **Consistencia Secuencial de Lamport (1979)**, donde toda lectura y escritura se ejecuta en estricto orden de código y es visible de inmediato para todos los hilos.

- En procesadores modernos (x86, ARM, RISC-V) y compiladores optimizadores (GCC/Clang), las escrituras se colocan en *Store Buffers* locales para evitar latencias de memoria. Esto permite la reordenación **Store-Load**: la lectura de `quiere[1-i]` puede adelantarse antes de que la escritura de `quiere[i]` se haya hecho visible en la caché compartida del otro núcleo.
- Ambos hilos pueden leer simultáneamente `quiere[1-i]` == false, accediendo a la vez a la SC en hardware real.
- Por ello, en programas reales es obligatorio declarar las variables con tipos atómicos (`std::atomic<bool>`) bajo ordenación secuencialmente consistente (`memory_order_seq_cst`) o insertar barreras completas de memoria (`mfence` en x86, `dmb` en ARM).

Ejercicio 58: Sincronización Hardware para N Procesos con Instrucción Atómica Swap

Se dispone de `swap(x,y)`, que intercambia atómicamente los valores booleanos de x e y. Diseñar protocolos de entrada y salida para $N \geq 2$ procesos mediante una variable compartida `ocupado`, inicialmente falsa, y una variable booleana local por proceso. Demostrar la exclusión mutua y discutir el progreso, la inanición y el coste de espera ocupada. ¿Basta la atomicidad de `swap` para asegurar equidad? Justificar.

Solución Detallada

1. Diseño de los Protocolos de Entrada y Salida:
Variables compartidas y privadas:

```
// Variable compartida global:
bool ocupado = false;

// Protocolo de Entrada para el proceso Pi:
bool mi_llave = true; // Variable local privada
do {
    swap(ocupado, mi_llave);
} while (mi_llave == true);

// < Seccion Critica >

// Protocolo de Salida para el proceso Pi:
ocupado = false;
```

2. Demostración de la Exclusión Mutua:

- **Invariante de conservación del testigo:** En todo instante se cumple:

$$\text{Valor}(\texttt{ocupado}) + \sum_{i=1}^{N} [\texttt{mi_llave}_i == \text{false}] = 1$$

  donde el corchete denota la función indicatriz de valor 1 si es cierta y 0 si es falsa.

- Inicialmente `ocupado` = false y todas las variables locales tienen $\texttt{mi_llave}_i$ = true. La suma es $0 + 0 = 0$. Existe una única llave con valor `false` disponible en el sistema.

- La primitiva `swap` es atómica e indivisible a nivel de bus de hardware. El primer proceso que ejecuta `swap` extrae el valor `false` a su variable privada `mi_llave`, mientras que la variable global `ocupado` pasa a valer `true`. Dicho proceso supera el bucle y entra a la SC.

- Mientras el recurso está ocupado (`ocupado` = true), cualquier otro proceso que ejecute `swap` intercambia su `mi_llave` (`true`) por `ocupado` (`true`), reteniendo el valor `true` en su variable privada y continuando en espera activa.

- Como en ningún momento puede haber más de una variable local con valor `false`, a lo sumo un solo proceso puede encontrarse ejecutando la SC:

$$N_{SC} \leq 1 \quad \text{(Exclusión mutua garantizada)}$$

3. Discusión de Progreso, Inanición y Equidad:

- **Progreso:** Si ningún proceso está en la SC y `ocupado` = false, el primer proceso que ejecute `swap` adquiere la llave y accede. No puede producirse interbloqueo colectivo.
- **¿Basta la atomicidad de swap para asegurar equidad? NO.** La atomicidad de la instrucción sólo garantiza la indivisibilidad de un intercambio individual; **no impone ninguna disciplina temporal ni cola de espera FIFO** sobre los procesos contendientes.
- Cuando el proceso en SC sale y ejecuta `ocupado` = false, todos los procesos que esperan en su bucle compiten en una carrera asíncrona. La resolución depende del arbitraje del controlador de memoria y de la topología física (sistemas NUMA). Un proceso desfavorecido puede perder consecutivamente todas las contiendas y sufrir **inanición indefinida**. Para asegurar equidad se precisa un cerrojo por tickets (*ticket lock*).

4. Coste de Espera Ocupada y Optimización:
La ejecución continua de `swap` genera continuas transacciones de invalidación en el protocolo de coherencia de caché (*cache line bouncing*), saturando el bus de interconexión. Para mitigarlo se aplica el patrón **Test and Test-and-Set**: el proceso espera en un bucle pasivo leyendo en caché `while (ocupado) skip;` y sólo ejecuta `swap` cuando detecta que la variable ha pasado a valer `false`.

Ejercicio 59: Exclusión de Capacidad k mediante Cerrojo Atómico

Un recurso admite simultáneamente a lo sumo k procesos, con $1 \leq k < N$. Diseñar las subrutinas `entrar()` y `salir()` para N procesos utilizando únicamente una variable entera compartida `dentro`, inicialmente cero, y un cerrojo atómico `lock()` / `unlock()`. Se permite espera ocupada fuera del cerrojo. (a) Escribir pseudocódigo. (b) Demostrar el invariante $0 \leq \text{dentro} \leq k$. (c) Explicar por qué no debe mantenerse el cerrojo mientras se espera a que haya capacidad. (d) Discutir si la solución asegura ausencia de inanición.

Solución Detallada

Apartado (a): Pseudocódigo de las Subrutinas:

```
// Variables compartidas globales:
int dentro = 0;
Mutex cerrojo; // Primitivas atómicas lock() y unlock()

void entrar() {
    bool acceso_concedido = false;
    while (!acceso_concedido) {
        cerrojo.lock();
        if (dentro < k) {
            dentro = dentro + 1;
            acceso_concedido = true;
        }
        cerrojo.unlock();
        // Si no se concedio, el reintento se efectua fuera del cerrojo
    }
}

```

```
18 void salir() {
19     cerrojo.lock();
20     dentro = dentro - 1;
21     cerrojo.unlock();
22 }
```

Apartado (b): Demostración del Invariante $0 \leq$ dentro $\leq k$:
Demostraremos por inducción estructural sobre los estados del sistema:

- **Base inductiva:** Inicialmente `dentro` $= 0$. Como $1 \leq k < N$, se cumple trivialmente $0 \leq 0 \leq k$.
- **Paso inductivo:** La variable `dentro` únicamente es modificada dentro de las secciones críticas protegidas por `cerrojo`:
  - En `entrar()`: Se evalúa `dentro` $< k$. Como `dentro` es entero, esto equivale a `dentro` $\leq k - 1$. Tras la asignación `dentro`$' =$ `dentro` $+ 1$, se cumple `dentro`$' \leq k$. Además, como por hipótesis inductiva `dentro` ≥ 0, tenemos `dentro`$' \geq 1 \geq 0$. Si la condición no se cumple, la variable no varía.
  - En `salir()`: Por especificación, `salir()` sólo es invocada por procesos que previamente completaron `entrar()`. El número total de ejecuciones de `salir()` no puede superar al de `entrar()`, lo que garantiza `dentro` ≥ 1 antes del decremento. Por ende, tras `dentro`$' =$ `dentro` $- 1$, se cumple `dentro`$' \geq 0$. Puesto que `dentro` $\leq k$, tenemos `dentro`$' \leq k - 1 < k$.
- Queda formalmente demostrado el invariante de seguridad: $\forall t,\ 0 \leq \text{dentro}(t) \leq k$.

Apartado (c): Por qué NO debe retenerse el cerrojo durante la espera:
Si un proceso adquiriese el cerrojo y ejecutase el bucle de espera sin liberarlo:

```
1 cerrojo.lock();
2 while (dentro >= k) { /* espera activa con cerrojo adquirido */ }
```

se produciría un **Interbloqueo Fatal (Deadlock)**:

- Ningún otro proceso podría adquirir `cerrojo`.
- Los procesos que se encuentran dentro del recurso no podrían ejecutar la subrutina `salir()`, puesto que `salir()` necesita adquirir `cerrojo` para decrementar `dentro`.
- Al ser imposible decrementar `dentro`, la condición `dentro` $\geq k$ permanecería eternamente cierta, congelando a todo el sistema.
- Por tanto, la liberación de `cerrojo` antes de reiterar la espera activa es estrictamente indispensable para el progreso del sistema.

Apartado (d): Discusión sobre la Ausencia de Inanición:
La solución NO asegura la ausencia de inanición. Cuando la capacidad está colmada (`dentro` $= k$), múltiples procesos iteran concurrentemente en sus respectivos bucles `while (!acceso_concedido)`. Cuando un proceso sale y decrementa la variable a $k - 1$, el proceso que logra capturar el cerrojo y ocupar la plaza no sigue un orden de llegada FIFO, sino que depende del azar y del planificador. Procesos con menor prioridad de bus pueden ser adelantados arbitrariamente muchas veces, sufriendo inanición.

Ejercicio 62: Peterson para N Procesos: Filtro por Niveles y Cotas de Espera

Considerar la generalización de Peterson con `nivel[i]` inicialmente cero y `victima[L]` compartidos. Cada proceso P_i recorre los niveles $L = 1, \ldots, N-1$:

```
for L := 1 to N-1 do
    nivel[i] := L;
    victima[L] := i;
    while exists j != i such that
        (nivel[j] >= L and victima[L] = i) do skip;
endfor;
<seccion critica>;
nivel[i] := 0;
```

(a) Justificar por qué no pueden coexistir dos procesos en sección crítica. (b) Analizar si pueden encontrarse dos procesos en el nivel $N-2$ mientras ninguno está en $N-3$; razonar según la definición precisa de «estar en un nivel». (c) Examinar qué hipótesis adicionales serían necesarias para establecer cotas de espera. Evitar identificar automáticamente ausencia de inanición con espera acotada.

Solución Detallada

Apartado (a): Demostración de Exclusión Mutua (Lema del Filtro):
El algoritmo organiza la contienda en $N-1$ etapas o filtros sucesivos.

- **Lema Fundamental del Filtro de Peterson:**
  En cualquier estado alcanzable, a lo sumo $N-L$ procesos pueden encontrarse en el nivel L o en niveles superiores.

- **Demostración por inducción sobre L:**

  - *Base ($L = 1$):* En el nivel 1, la variable `victima[1]` almacena el identificador de un único proceso (el último en llegar). Este proceso no puede superar el bucle `while` mientras exista otro proceso con `nivel`$[j] \geq 1$. Por ende, al menos un proceso queda retenido en el nivel 1, permitiendo avanzar como máximo a $N-1$ procesos.
  - *Paso Inductivo:* Asumamos que a lo sumo $N-L$ procesos pueden alcanzar el nivel L. En dicho nivel, la variable compartida `victima[L]` retiene obligatoriamente al último proceso en registrarse. En consecuencia, de los $N-L$ candidatos, al menos uno no puede franquear el bucle, permitiendo avanzar a lo sumo a $(N-L)-1 = N-(L+1)$ procesos al nivel $L+1$.

- **Sección Crítica:** Para acceder a la SC, un proceso debe completar el nivel $L = N-1$. Aplicando el lema:
  $$N_{\text{procesos en SC}} \leq N - (N-1) = 1$$
  Por tanto, es imposible que dos procesos coexistan en la SC: $N_{SC} \leq 1$.

Apartado (b): ¿Pueden encontrarse dos procesos en $N-2$ y ninguno en $N-3$?:
Definición de «estar en el nivel k»: Un proceso P_i se encuentra en el nivel k si ha ejecutado `nivel`$[i] := k$ y aún no ha ejecutado `nivel`$[i] := k+1$ (es decir, se encuentra ejecutando o esperando en la k-ésima iteración del bucle `for`).
Respuesta: NO ES POSIBLE.
Justificación Formal (Lema 3 de Peterson / Infomates Ejercicio 5.2.8):

- Para que dos procesos P_a y P_b hayan superado el filtro del nivel $N-3$ y se encuentren simultáneamente en el nivel $N-2$, debió ocurrir que uno de ellos asumió la condición de víctima en $N-3$ mientras el otro avanzaba, o bien que un tercer proceso P_c asumió el rol de víctima en dicho nivel.

- La invariante de distribución de procesos en los filtros de Peterson establece una propiedad monótona: para que existan $M \geq 2$ procesos en la etapa k, en cada una de las etapas previas $0, 1, \ldots, k-1$ debe existir al menos un proceso actuando como víctima de retención.
- Si el nivel $N-3$ estuviera completamente vacío (0 procesos), significaría que no existe ningún proceso que retenga la variable `victima[N-3]`, lo que contradice el principio de avance del filtro. Por consiguiente, dicha configuración es inalcanzable.

Apartado (c): Hipótesis Adicionales para Cotas de Espera frente a Ausencia de Inanición:

- **Distinción conceptual:** La *ausencia de inanición* es una propiedad cualitativa de vivacidad ($\forall i, \Diamond P_i^{SC}$). La *espera acotada* es una propiedad cuantitativa de seguridad que acota uniformemente el número de adelantamientos por una función matemática determinista $f(N)$.
- **Hipótesis necesarias para derivar la cota de espera formal:**
  1. **Tiempo Finito en SC:** Todo proceso que entra en la SC sale de ella en un tiempo acotado por una constante finita ($\Delta_{SC} \leq C_{SC} < \infty$).
  2. **Planificación Débilmente Justa en CPU:** Cada hilo activo en el protocolo de entrada progresa de forma continua en sus instrucciones.
  3. **Reingreso Cíclico sin Privilegios:** Todo proceso que abandone la SC y vuelva a solicitarla debe reiniciar su tránsito obligatoriamente desde el nivel $L = 1$, garantizando que se colocará detrás (como nueva víctima) de los procesos rezagados en niveles inferiores.
- Bajo estas condiciones, se demuestra que un proceso compitiendo en el nivel L espera a lo sumo a que cada uno de los demás procesos entre una vez por cada nivel superior, estableciendo una cota matemática de espera de $O(N^2)$ adelantamientos (exactamente $\frac{N(N-1)}{2}$ turnos).

Ejercicio 63: Algoritmo de Dijkstra para N Procesos: Análisis de Modificación de Guarda

Se comparte `turno` y el vector `estado[i]` con valores `pasivo`, `solicitando` y `enSC`. En el algoritmo de Dijkstra estudiado en clase, durante el protocolo de entrada, un proceso que espera su turno ejecuta:

```
1 while turno != i do
2     if estado[turno] = pasivo then turno := i;
3 endwhile;
```

Se propone sustituir la guarda por `estado[turno] != enSC`, sin alterar el resto del algoritmo. (a) Comparar las dos condiciones. (b) Construir, si es posible, una ejecución que invalide alguna propiedad de corrección. (c) Precisar si el defecto afecta a la seguridad, al progreso o a la equidad. Para el análisis, utilizar el protocolo completo de Dijkstra presentado en clase, no únicamente el fragmento anterior.

Solución Detallada

1. Protocolo Canónico Completo de Dijkstra (1965) para N Procesos:

```
// Variables compartidas:
enum Estado { pasivo, solicitando, enSC };
Estado estado[N] = { pasivo, ..., pasivo };
int turno = 0;

// Protocolo para el proceso Pi:
void proceso(int i) {
    while (true) {
        // Protocolo de entrada:
        estado[i] = solicitando;
        L1: while (turno != i) {
            // Guarda original: if (estado[turno] == pasivo) turno = i;
            // Guarda propuesta:
            if (estado[turno] != enSC) {
                turno = i;
            }
        }
        estado[i] = enSC;
        for (int j = 0; j < N; ++j) {
            if (j != i && estado[j] == enSC) {
                goto L1; // Retroceso si detecta colision en SC
            }
        }

        // Seccion Critica:
        seccion_critica();

        // Protocolo de salida:
        estado[i] = pasivo;

        seccion_no_critica();
    }
}
```

Apartado (a): Comparación Operativa de las Dos Condiciones:

- **Condición Original (`estado[turno] == pasivo`):** Un proceso P_i únicamente se apropia del turno (`turno := i`) si el titular actual no tiene interés en la sección crítica. Si el titular está en estado `solicitando`, los demás procesos respetan pacíficamente su turno y aguardan a que él concluya su entrada.
- **Condición Propuesta (`estado[turno] != enSC`):** El proceso P_i puede arrebatar el turno no sólo si el titular está pasivo, sino **también cuando el titular está activamente compitiendo en el protocolo de entrada** (`estado[turno] == solicitando`), con tal de que no haya alcanzado todavía la SC.

Apartado (b): Construcción del Contraejemplo (Interbloqueo Dinámico / Livelock):
Consideremos $N = 2$ procesos (P_0 y P_1) con valor inicial `turno` $= 0$. Ambos solicitan entrar:

Paso	Proceso	Línea Ejecutada	est[0]	est[1]	turno	Efecto en el Sistema
1	P_0	`estado`[0] := solicitando	sol	pas	0	P_0 inicia solicitud.
2	P_1	`estado`[1] := solicitando	sol	sol	0	P_1 inicia solicitud.
3	P_1	Evalúa `while turno != 1`	sol	sol	0	Condición cierta ($0 \neq 1$).
4	P_1	Evalúa `estado[0] != enSC`	sol	sol	0	Cierta (sol $\neq$ enSC).
5	P_1	`turno` := 1	sol	sol	1	P_1 arrebata el turno a P_0.
6	P_0	Evalúa `while turno != 0`	sol	sol	1	Condición cierta ($1 \neq 0$).
7	P_0	Evalúa `estado[1] != enSC`	sol	sol	1	Cierta (sol $\neq$ enSC).
8	P_0	`turno` := 0	sol	sol	0	P_0 arrebata el turno a P_1.
9	P_1	Reevalúa `while turno != 1`	sol	sol	0	Cierta ($0 \neq 1$). Arrebata: `turno` := 1.
10	P_0	Reevalúa `while turno != 0`	sol	sol	1	Cierta ($1 \neq 0$). Arrebata: `turno` := 0.
CICLO INFINITO: Ningún proceso logra validar `turno` $== i$ para pasar a enSC. LIVELOCK TOTAL.						

Ambos procesos se disputan el turno de forma destructiva y oscilante, impidiendo que ninguno complete la verificación de entrada.

Apartado (c): Clasificación del Defecto (Seguridad vs. Progreso vs. Equidad):

- **Seguridad (Exclusión Mutua): NO SE VE AFECTADA.** La exclusión mutua estricta en el algoritmo de Dijkstra está custodiada por la segunda barrera (el bucle `for` que verifica si algún otro proceso tiene `estado`[j] == enSC, retrocediendo con `goto L1` si detecta concurrencia). El cambio en la primera guarda no permite que dos procesos superen juntos este segundo filtro.
- **Progreso (Ausencia de Interbloqueo): QUEDA COMPLETAMENTE DESTRUIDO.** El defecto provoca un **fallo catastrófico de progreso** (vivacidad global del sistema). La sección crítica se encuentra desocupada y dos procesos compiten activamente por acceder, pero el sabotaje mutuo sobre la variable `turno` les impide entrar indefinidamente.
- **Equidad:** Se degrada como consecuencia del fallo de progreso.

Bloque Temático 2

Monitores y Sincronización Condicional (Tema 2-1)

Metadatos de la Relación y Fuentes:

Relevancia en el Examen: El problema práctico indispensable para asegurar la máxima puntuación.
• **Conceptos obligatorios:** Semántica Señalar y Espera Urgente (Hoare/SU, usa `if`) frente a Señalar y Continuar (Mesa/SC/POSIX, requiere obligatoriamente `while`); patrón de paso de testigo; prevención de inanición mediante cupos máximos de vehículos; sincronización servidor pasivo (olla de salvajes); lectores-escritores con prioridad a escritores y filósofos comensales.

Ejercicio 39: El Aseo Compartido con Semáforos POSIX

Un aseo admite simultáneamente o bien una mujer o bien hasta cinco hombres; nunca ambos a la vez. Resolver con semáforos POSIX no nombrados:

(a) Prioridad estricta de hombres: si hay hombres esperando, ninguna mujer nueva puede entrar, incluso si el aforo masculino está completo.

(b) Prioridad estricta de mujeres: si hay mujeres esperando, ningún hombre nuevo puede entrar.

Solución Detallada

Se modela la variante (a) (Prioridad a Hombres). Se mantienen contadores protegidos para hombres dentro (H), hombres esperando (H_{esp}), mujeres dentro (M) y mujeres esperando (M_{esp}):

```
sem_t mutex = 1;
sem_t sem_hombres = 0;
sem_t sem_mujeres = 0;
int H = 0, H_esp = 0;
int M = 0, M_esp = 0;

void entrar_hombre() {
    sem_wait(&mutex);
    H_esp++;
    // Puede entrar si no hay mujeres dentro y hay menos de 5 hombres
    if (M == 0 && H < 5) {
        H_esp--;
        H++;
        sem_post(&sem_hombres);
```

```
    }
    sem_post(&mutex);
    sem_wait(&sem_hombres);
}

void salir_hombre() {
    sem_wait(&mutex);
    H--;
    // Si quedan hombres esperando, entra el siguiente hombre
    if (H_esp > 0 && H < 5) {
        H_esp--;
        H++;
        sem_post(&sem_hombres);
    } else if (H == 0 && H_esp == 0 && M_esp > 0) {
        // Si no quedan hombres, pasa una mujer
        M_esp--;
        M = 1;
        sem_post(&sem_mujeres);
    }
    sem_post(&mutex);
}

void entrar_mujer() {
    sem_wait(&mutex);
    // Prioridad a hombres: solo entra si H == 0, M == 0 y NO hay hombres
        esperando
    if (H == 0 && M == 0 && H_esp == 0) {
        M = 1;
        sem_post(&sem_mujeres);
    } else {
        M_esp++;
    }
    sem_post(&mutex);
    sem_wait(&sem_mujeres);
}

void salir_mujer() {
    sem_wait(&mutex);
    M = 0;
    // Tienen prioridad los hombres que esperan
    if (H_esp > 0) {
        int a_despertar = std::min(H_esp, 5);
        H_esp -= a_despertar;
        H += a_despertar;
        for (int k = 0; k < a_despertar; k++) sem_post(&sem_hombres);
    } else if (M_esp > 0) {
        M_esp--;
        M = 1;
        sem_post(&sem_mujeres);
    }
    sem_post(&mutex);
}
```

Discusión sobre Inanición: La prioridad estricta causa **inanición de mujeres** si el flujo de llegada de hombres es continuo ($H_{esp} > 0$). Los semáforos POSIX no garantizan política FIFO en el desbloqueo, por lo que dentro de la misma clase el planificador del kernel puede elegir arbitrariamente.

Ejercicio 40: Reentrancia frente a Exclusión Mutua en Monitores

Un monitor garantiza exclusión mutua al ejecutar sus procedimientos. Explicar qué significa que un procedimiento sea reentrante, por qué exclusión mutua y reentrancia no son equivalentes y en qué situaciones es necesario exigir reentrancia. Distinguir la semántica de señales Hoare (SU) y Mesa (SC).

Solución Detallada

- **Definición de Reentrancia:** Un código es reentrante si puede ser interrumpido a mitad de su ejecución por otro hilo (o por una llamada recursiva/señal) y reanudado posteriormente sin que su estado lógico resulte alterado. Para ello, **no debe modificar variables estáticas o globales no protegidas** y debe almacenar su estado local exclusivamente en la pila (*stack*) del hilo invocador.
- **No equivalencia con Exclusión Mutua:** La exclusión mutua restringe el acceso a un solo hilo a la vez, pero no previene que si un procedimiento realiza una llamada recursiva sobre sí mismo o cede el monitor en un `wait`, el estado local sea sobreescrito si reside en memoria estática.
- **Situaciones donde se exige reentrancia:**
  1. Cuando un hilo en un monitor invoca `c.wait()` o `c.signal()`: el hilo cede el monitor y otro hilo puede entrar al mismo procedimiento. Si las variables locales estuvieran en variables globales del monitor, el nuevo hilo las destruiría.
  2. Invocaciones cruzadas o recursivas entre procedimientos del monitor.
- **Diferencia entre SU y SC:**
  - **SU (Hoare):** El señalador pasa a la cola urgente y el despertado entra de inmediato. La condición lógica se garantiza intacta tras `wait` (basta `if (!cond) wait()`).
  - **SC (Mesa):** El señalador retiene el monitor y el despertado va a la cola de entrada. La condición puede ser invalidada por otro hilo antes de que el despertado tome el control, haciendo **obligatorio el bucle** `while (!cond) wait()`.

Ejercicio 42: Lectores-Escritores con Monitores

Resolver el problema de lectores y escritores mediante monitores con prioridad a escritores y con política equitativa sin inanición.

Solución Detallada

Implementación Oficial UGR (`HoareMonitor` y `CondVar` de `scd.h`):
Bajo semántica SU (Hoare), la exclusión mutua es automática y las variables de condición `CondVar` garantizan orden FIFO. En la variante con prioridad a escritores, si hay algún escritor escribiendo o esperando turno, los nuevos lectores deben suspenderse:

```
#include "scd.h"
using namespace std;
using namespace scd;
```

```
4
5  class LectoresEscritoresPrioridadEscritores : public HoareMonitor {
6  private:
7      int      n_lec;              // Numero de lectores leyendo actualmente
8      bool     escrib;             // true si hay un escritor escribiendo
9      int      escrib_esperando;   // Numero de escritores esperando turno
10     CondVar  cola_lectores;
11     CondVar  cola_escritores;
12
13 public:
14     LectoresEscritoresPrioridadEscritores() : n_lec(0), escrib(false),
           escrib_esperando(0) {
15         cola_lectores   = newCondVar();
16         cola_escritores = newCondVar();
17     }
18
19     void iniciar_lectura() {
20         // En semantica SU basta 'if': si hay escritor activo o esperando
               , esperar
21         if (escrib || escrib_esperando > 0) {
22             cola_lectores.wait();
23         }
24         n_lec++;
25         // Desbloqueo en cascada para dar paso a otros lectores en espera
26         cola_lectores.signal();
27     }
28
29     void fin_lectura() {
30         n_lec--;
31         // El ultimo lector en salir despierta al escritor prioritario
32         if (n_lec == 0) {
33             cola_escritores.signal();
34         }
35     }
36
37     void iniciar_escritura() {
38         escrib_esperando++;
39         if (escrib || n_lec > 0) {
40             cola_escritores.wait();
41         }
42         escrib_esperando--;
43         escrib = true;
44     }
45
46     void fin_escritura() {
47         escrib = false;
48         // Prioridad estricta a escritores: si hay alguno esperando, se
               despierta
49         if (escrib_esperando > 0) {
50             cola_escritores.signal();
51         } else {
52             cola_lectores.signal(); // Si no hay escritores, pasan los
                   lectores
53         }
54     }
55 };
```

Ejercicio 43: Puente de un Solo Carril con Cupo Máximo

Vehículos del norte y del sur cruzan un puente de un solo carril.

(a) Monitor básico y discusión de inanición.

(b) Modificación para conceder turnos de hasta diez entradas consecutivas cuando hay espera opuesta.

Solución Detallada

Apartado (a): En la política básica (entran del norte si no hay coches del sur cruzando), si llega un flujo ininterrumpido de coches del norte, los coches del sur sufrirán **inanición indefinida**.

Apartado (b): Monitor Canónico en C++11 con `HoareMonitor` (UGR):

```
#include "scd.h"
using namespace std;
using namespace scd;

class MonitorPuenteCupo : public HoareMonitor {
private:
    int      coches_cruzando;  // Numero de vehiculos en el puente
    int      sentido_actual;   // 0: vacio, 1: Norte, 2: Sur
    int      esperando_norte;  // Vehiculos esperando en el Norte
    int      esperando_sur;    // Vehiculos esperando en el Sur
    int      cupo_actual;      // Vehiculos que han entrado en la tanda
        actual (max 10)
    CondVar cola_norte;
    CondVar cola_sur;

public:
    MonitorPuenteCupo() : coches_cruzando(0), sentido_actual(0),
                          esperando_norte(0), esperando_sur(0),
                              cupo_actual(0) {
        cola_norte = newCondVar();
        cola_sur   = newCondVar();
    }

    void entrarNorte() {
        esperando_norte++;
        // Esperar si el puente esta ocupado por el Sur, o si siendo
            Norte se
        // agoto el cupo de 10 y hay vehiculos esperando en el sentido
            opuesto:
        if ((sentido_actual == 2 && coches_cruzando > 0) ||
            (sentido_actual == 1 && cupo_actual >= 10 && esperando_sur >
                0)) {
            cola_norte.wait();
        }
        esperando_norte--;
        sentido_actual = 1;
        coches_cruzando++;
        cupo_actual++;

        // Cascada SU: permitir la entrada de otros coches del Norte si
            queda cupo
        if (cupo_actual < 10 && esperando_norte > 0) {
            cola_norte.signal();
        }
    }
```

```
40
41      void salirNorte() {
42          coches_cruzando--;
43          if (coches_cruzando == 0) {
44              if (cupo_actual >= 10 || esperando_norte == 0) {
45                  // Cambio de sentido hacia el Sur
46                  cupo_actual = 0;
47                  sentido_actual = (esperando_sur > 0) ? 2 : 0;
48                  cola_sur.signal(); // Despierta al primer coche del Sur
49              } else {
50                  cola_norte.signal();
51              }
52          }
53      }
54
55      void entrarSur() {
56          esperando_sur++;
57          if ((sentido_actual == 1 && coches_cruzando > 0) ||
58              (sentido_actual == 2 && cupo_actual >= 10 && esperando_norte
                    > 0)) {
59              cola_sur.wait();
60          }
61          esperando_sur--;
62          sentido_actual = 2;
63          coches_cruzando++;
64          cupo_actual++;
65
66          if (cupo_actual < 10 && esperando_sur > 0) {
67              cola_sur.signal();
68          }
69      }
70
71      void salirSur() {
72          coches_cruzando--;
73          if (coches_cruzando == 0) {
74              if (cupo_actual >= 10 || esperando_sur == 0) {
75                  cupo_actual = 0;
76                  sentido_actual = (esperando_norte > 0) ? 1 : 0;
77                  cola_norte.signal();
78              } else {
79                  cola_sur.signal();
80              }
81          }
82      }
83  };
```

Ejercicio 44: La Olla de los Salvajes y el Cocinero

Olla de capacidad $M > 0$ raciones y $N > 0$ comensales. El primer salvaje que encuentra la olla vacía despierta al cocinero; el resto espera sin generar nuevas llamadas. Diseñar el monitor con semántica SU.

Solución Detallada

Implementación en C++11 con `HoareMonitor` (UGR):

```
#include "scd.h"
using namespace std;
using namespace scd;

class MonitorOlla : public HoareMonitor {
private:
    const int capacidad;
    int       raciones;
    bool      aviso_enviado;
    CondVar   cola_salvajes;
    CondVar   cola_cocinero;

public:
    MonitorOlla(int M) : capacidad(M), raciones(M), aviso_enviado(false)
        {
        cola_salvajes = newCondVar();
        cola_cocinero = newCondVar();
    }

    void servir() {
        // En semantica SU basta 'if' porque la senal reactiva de
            inmediato con raciones > 0
        if (raciones == 0) {
            if (!aviso_enviado) {
                aviso_enviado = true;
                cola_cocinero.signal(); // Despierta al cocinero (pasa a
                    cola urgente)
            }
            cola_salvajes.wait();
        }
        raciones--;
        if (raciones > 0) {
            cola_salvajes.signal(); // Cascada SU: avisa al siguiente
                salvaje en espera
        }
    }

    void esperarPeticion() {
        if (!aviso_enviado) {
            cola_cocinero.wait();
        }
    }

    void reponer() {
        raciones = capacidad;
        aviso_enviado = false;
        cola_salvajes.signal(); // Despierta al primer salvaje en espera
    }
};
```

Invariante: $0 \leq raciones \leq M$. Si $raciones == 0$, ningún salvaje puede completar `servir()` antes de que el cocinero reponga.

Ejercicio 48: Los Cinco Filósofos Comensales con Monitor

Cinco filósofos comparten 5 tenedores.

(a) Adquisición individual en monitor y corrección de la liberación.

(b) Ejecución con interbloqueo.

(c) Prevención mediante limitación de aforo a 4 filósofos.

Solución Detallada

Apartado (a) y (b): Interbloqueo Circular:
Si cada filósofo i toma primero su tenedor izquierdo T_i, todos los 5 filósofos retienen un tenedor y esperan el derecho $T_{(i+1)\,\%5}$. Hay un ciclo de espera cerrada $\implies$ **Interbloqueo**.
Apartado (c): Solución con Aforo de 4 Filósofos (`HoareMonitor` UGR):

```
#include "scd.h"
using namespace std;
using namespace scd;

class MonitorFilosofosAforo : public HoareMonitor {
private:
    CondVar cola_aforo;
    CondVar cola_tenedores[5];
    bool    tenedor_libre[5] = {true, true, true, true, true};
    int     sentados_o_esperando = 0;

public:
    MonitorFilosofosAforo() {
        cola_aforo = newCondVar();
        for (int i = 0; i < 5; i++) {
            cola_tenedores[i] = newCondVar();
        }
    }

    void entrar_a_comer(int i) {
        // Aforo maximo 4: garantiza que al menos un filosofo podra tomar
            ambos tenedores
        if (sentados_o_esperando >= 4) {
            cola_aforo.wait();
        }
        sentados_o_esperando++;

        if (!tenedor_libre[i]) {
            cola_tenedores[i].wait();
        }
        tenedor_libre[i] = false;

        if (!tenedor_libre[(i + 1) % 5]) {
            cola_tenedores[(i + 1) % 5].wait();
        }
        tenedor_libre[(i + 1) % 5] = false;
    }

    void salir_de_comer(int i) {
        tenedor_libre[i] = true;
        tenedor_libre[(i + 1) % 5] = true;

        cola_tenedores[i].signal();
        cola_tenedores[(i + 1) % 5].signal();
```

```
44
45          sentados_o_esperando--;
46          cola_aforo.signal();
47      }
48  };
```

Por el Principio del Palomar, con 4 filósofos y 5 tenedores, al menos uno tendrá disponibles sus dos tenedores contiguos, garantizando **ausencia absoluta de interbloqueo**.

Bloque Temático 3

Concurrencia Básica, Bernstein y Lógica de Hoare (Tema 1)

Metadatos de la Relación y Fuentes:

Relevancia en el Examen: El problema de reserva y fundamentos teóricos de concurrencia.
• **Conceptos obligatorios:** Bucles de incrementos concurrentes con registros locales (máximo $2k$, mínimo 2); condiciones de Bernstein e intersección de conjuntos de lectura/escritura; paralelización con `cobegin-coend` y grafos de precedencias; cálculo de estados finales con acciones atómicas; teorema de no interferencia de Owicki-Gries mediante verificación de triples intermedios.

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):

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

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

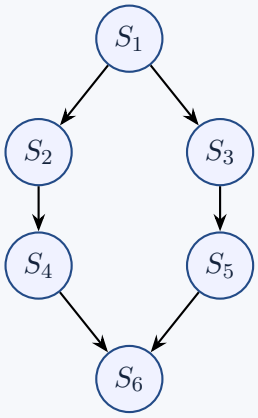
Acción P_1	**Acción P_2**	**Registros (R_1, R_2)**	**Valor de x**
l_{11} (lee 0)	l_{21} (lee 0)	$R_1 = 0, R_2 = 0$	0
e_{11} (escribe 1)	—	$R_1 = 1$	1
—	e_{21} (escribe 1)	$R_2 = 1$	1
l_{12} (lee 1)	l_{22} (lee 1)	$R_1 = 1, R_2 = 1$	1
e_{12} (escribe 2)	—	$R_1 = 2$	2
—	e_{22} (escribe 2)	$R_2 = 2$	**2**

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 $\{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$.

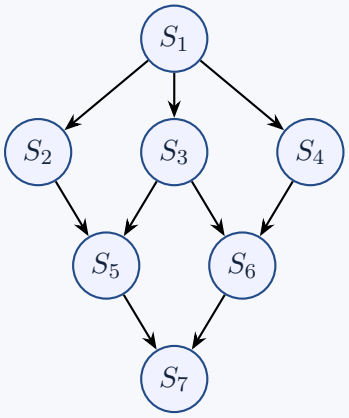
Ejercicio 4: Construcción de programas con cobegin-coend y fork-join para grafos

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):

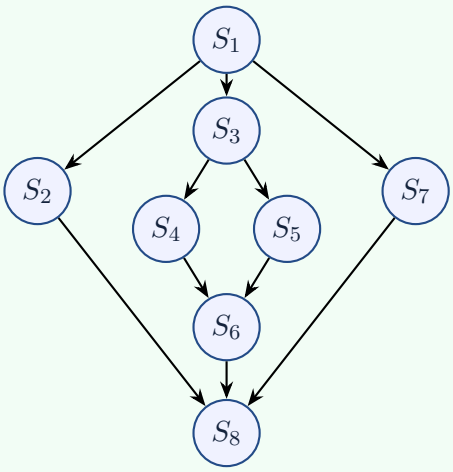
```
    var c1, c2 : integer := 2;
    S1;
    fork L1;
    S2;
    goto L2;
L1: S3;
L2: join c1;
    fork L3;
    S4;
    goto L4;
L3: S5;
L4: join c2;
    S6;
```

Solución Detallada

Analizamos el orden de ejecución y sincronizaciones de cada fragmento:

Grafo del Fragmento (a)

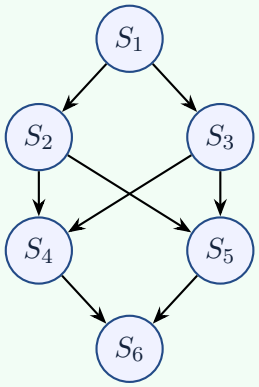
1. S_1 se ejecuta secuencialmente al principio y precede a las tres ramas del `cobegin`.
2. El `cobegin` exterior lanza 3 ramas concurrentes:
   - Rama 1: S_2.
   - Rama 2: $S_3 \rightarrow$ (`cobegin` interno de $S_4 \parallel S_5$) $\rightarrow S_6$.
   - Rama 3: S_7.
3. El `coend` exterior obliga 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 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: $\{(\mathbf{7, 14}), (\mathbf{15, 10})\}$.

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 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):

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

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 $\{x = 7\}$.