Sistemas Concurrentes y Distribuidos

Exámenes Oficiales Resueltos y Corregidos

Convocatorias Reales: Parcial 2021 y Parcial 2025 (Repetido)

Grado en Ingeniería Informática + Doble Grado DGIIM / ADE

Universidad de Granada (UGR)

Auditoría y Resolución Exhaustiva de Problemas:

- **Examen 2025 (Repetido):** 4 problemas completos (Semáforos POSIX con sincronización asimétrica, Lógica de Hoare y Owicki-Gries, Exclusión mutua de aforo n con cola FIFO, y Monitor de dos tipos con prioridad acumulada).
- **Examen 2021:** 2 problemas fundamentales (Monitor de Cuenta Bancaria con retiradas en orden estricto FIFO, y Demostración concurrente formal de 3 procesos con Owicki-Gries).

Incluye análisis crítico de los intentos de alumnos, detección y enmienda de errores conceptuales y sintácticos, invariantes formales y código C++11 / POSIX verificado.

Plataforma **repasaYA**

Material de Preparación para Examen Oficial — Prof. Manuel I. Capel Tuñón

Presentación y Auditoría de los Exámenes

Clave de Examen: ¿Por qué es crucial estudiar estos dos exámenes?

Las pruebas propuestas por el Prof. Manuel I. Capel Tuñón en la Universidad de Granada siguen patrones muy concretos que se repiten históricamente. De hecho, el examen titulado **Examen 2025 (Repetido)** reutilizó íntegramente la estructura y ejercicios de convocatorias anteriores (incluyendo problemas clásicos de exámenes de 2018–2021).
En los manuscritos y borradores habituales que circulan entre alumnos se detectan **errores conceptuales muy graves**:

1. **Semáforos mal sincronizados (Ej. 1 de 2025):** La solución del alumno omitía por completo la espera de P_2 sobre P_3, provocando que P_2 ejecutara su instrucción sin que P_3 hubiera intervenido.
2. **Problemas en blanco (Ej. 3 de 2025):** El problema del cerrojo de aforo n con cola FIFO suele dejarse sin responder por falta de conocimiento sobre algoritmos basados en tickets.
3. **Monitores con errores de estado y concurrencia (Ej. 4 de 2025):** La gestión de prioridades dinámicas acumuladas y la reactivación de tipos incompatibles contenía condiciones contradictorias que generaban interbloqueo y violación de exclusión mutua.
4. **Monitores con sintaxis inventada (Ej. 1 de 2021):** El uso de métodos inexistentes como `cola.queue(pos)` dentro de un bucle `while` bloqueaba permanentemente el monitor bancario.
5. **Pruebas de Owicki-Gries incompletas (Ej. 2 de 2021 y 2025):** Aserciones sin justificar mediante el axioma de asignación y omisión de las tablas cruzadas de no interferencia.

Este cuaderno ofrece los enunciados transcritos tal cual, el diagnóstico pedagógico de cada error típico y la **resolución formal rigurosa** para obtener la máxima calificación en el examen.

Índice general

Presentación y Auditoría de los Exámenes **3**

1. Examen Parcial 2025 (Convocatoria Repetida) **7**
1.1. Problema 1: Sincronización de Tres Procesos con Semáforos POSIX 7
1.2. Problema 2: Verificación Formal Axiomática en Lógica de Hoare 11
1.3. Problema 3: Exclusión Mutua con Aforo n y Cola FIFO 13
1.4. Problema 4: Monitor CentroRecursos con Prioridad Acumulada 16

2. Examen Parcial 2021 **19**
2.1. Problema 1: Monitor Cuenta Bancaria con Retiradas FIFO Estrictas 19
2.2. Problema 2: Demostración Axiomática Concurrente de Tres Procesos 22

1

Examen Parcial 2025 (Convocatoria Repetida)

Metadatos de la Relación y Fuentes:

Ficha de la Convocatoria: Examen Parcial de SCD (Temas 1 y 2), reutilizado en 2025.
• **Contenido evaluado:** Sincronización asimétrica con semáforos POSIX, demostración de triples con bucles concurrentes en lógica de Hoare, exclusión mutua de aforo n con cola FIFO en memoria compartida, y monitor de gestión de recursos con dos tipos y prioridad acumulada.

1.1 Problema 1: Sincronización de Tres Procesos con Semáforos POSIX

Ejercicio 1: Coordinación de P1, P2 y P3 bajo Condiciones Asimétricas

Se tienen los procesos P_1, P_2 y P_3 con sus instrucciones originales en bucles infinitos:

```
Proceso P1;
begin
  while true do begin
    a;
    b;
    c;
  end;
end;

Proceso P2;
begin
  while true do begin
    d;
    e;
    f;
  end;
end;

Proceso P3;
begin
  while true do begin
    g;
    h;
    i;
  end;
end;
```

Se desea modificar el comportamiento de forma que:

1. P_2 solo pueda ejecutar '`e`' después de que P_1 haya ejecutado '`b`' al menos una vez, y P_3 haya ejecutado '`h`' exactamente una vez (por cada ejecución de '`e`').
2. P_2 no debe bloquearse indefinidamente si P_3 ejecuta '`h`' antes que P_1.
3. Solo pueden usarse semáforos POSIX no nombrados.

Tareas:

(a) Diseñar la solución sin interbloqueo.

(b) Demostrar que no hay inanición para P_2 aunque P_1 sea más lento.

Solución Detallada

1. Análisis de Requisitos y Diagnóstico del Error del Alumno:
El borrador original del estudiante proponía:

```
// ERROR TIPICO:
// En P2: sem_wait(b); e; sem_signal(h);
// En P3: sem_wait(h); h;
```

Esta propuesta es **completamente incorrecta** por tres razones críticas:

- P_2 **nunca esperaba a P_3**: P_2 solo hacía `wait(b)`. Podía ejecutar `e` sin que P_3 hubiera ejecutado jamás `h`.
- P_3 se bloqueaba tras la primera iteración porque nadie le devolvía el permiso si P_2 no pasaba por `sem_signal(h)`.
- Se confundía la condición de *."al menos una vez"* con consumir un semáforo contador en cada iteración.

2. Especificación Formal de Sincronización:

- **Condición sobre P_1 (."al menos una vez"):** Basta con que P_1 ejecute '`b`' una sola vez en la vida del sistema para desbloquear permanentemente la precondición de P_1. Podemos modelarlo mediante un semáforo `b_listo` que, una vez señalizado, permanece accesible o mediante un cerrojo con indicador booleano.
- **Condición sobre P_3 (."exactamente una vez por cada 'e'"):** Existe una relación 1 a 1 entre la ejecución de '`h`' y '`e`'. P_3 debe avisar a P_2 de que '`h`' se ha completado (`sem_post(&h_hecho)`), y no debe ejecutar una segunda '`h`' hasta que P_2 haya consumido la anterior ejecutando '`e`' (`sem_wait(&h_ack)`).
- **Independencia de velocidad (P_3 antes que P_1):** Si P_3 ejecuta '`h`' primero, deposita su permiso en `h_hecho`. Cuando más tarde P_1 ejecute '`b`', P_2 encontrará ambos requisitos satisfechos y avanzará sin bloqueo.

3. Código C++ con Semáforos POSIX:

```
#include <semaphore.h>

// Semaforos para coordinar P3 y P2 (handshake 1 a 1)
sem_t sem_h_hecho; // Inic 0: P3 senala a P2 que 'h' ha concluido
sem_t sem_h_ack;   // Inic 1: P2 autoriza a P3 a ejecutar su siguiente 'h'

```

```
// Sincronizacion para "b ejecutado al menos una vez"
sem_t sem_b_primera; // Inic 0: P1 avisa de su primer 'b'
bool b_ya_ejecutado = false;
sem_t mutex_b;       // Inic 1: protege la comprobacion de b_ya_ejecutado

void inicializar() {
    sem_init(&sem_h_hecho, 0, 0);
    sem_init(&sem_h_ack, 0, 1);     // Permite a P3 hacer su primer 'h'
    sem_init(&sem_b_primera, 0, 0);
    sem_init(&mutex_b, 0, 1);
}

void P1() {
    while (true) {
        a();
        b();
        // Notificar que b se ha ejecutado al menos una vez
        sem_wait(&mutex_b);
        if (!b_ya_ejecutado) {
            b_ya_ejecutado = true;
            sem_post(&sem_b_primera);
        }
        sem_post(&mutex_b);
        c();
    }
}

void P2() {
    while (true) {
        d();

        // 1. Verificar requisito de P1 (al menos una vez):
        sem_wait(&mutex_b);
        if (!b_ya_ejecutado) {
            sem_post(&mutex_b);
            sem_wait(&sem_b_primera); // Espera al primer 'b'
            sem_post(&sem_b_primera); // Deja el semaforo abierto para
                futuras vueltas
        } else {
            sem_post(&mutex_b);
        }

        // 2. Verificar requisito de P3 (exactamente un 'h'):
        sem_wait(&sem_h_hecho); // Espera a que P3 haya ejecutado 'h'

        e(); // Seccion protegida

        sem_post(&sem_h_ack);   // Autoriza a P3 a producir su siguiente
            'h'

        f();
    }
}

void P3() {
    while (true) {
        g();
        sem_wait(&sem_h_ack);   // Espera autorizacion de P2
        h();
        sem_post(&sem_h_hecho); // Notifica a P2 que 'h' termino
        i();
```

```
66      }
67  }
```

4. Demostración de Ausencia de Interbloqueo e Inanición (Apartado b):

- **Ausencia de interbloqueo:** No existe ciclo de espera circular entre procesos. P_1 nunca se bloquea esperando a P_2 ni a P_3 (solo usa una sección crítica corta con `mutex_b`). P_3 solo espera a `sem_h_ack`, el cual está inicializado a 1, permitiendo que su primera iteración avance sin trabas.
- **Inanición de P_2 ante lentitud de P_1:** Supongamos que P_1 es arbitrariamente lento. P_3 ejecutará '`g`' y '`h`', haciendo `sem_post(&sem_h_hecho)`. A continuación, P_3 se detiene en su siguiente vuelta en `sem_wait(&sem_h_ack)`. El proceso P_2 se bloqueará en `sem_wait(&sem_b_primera)`. En cuanto P_1 complete '`b`' por primera vez en tiempo finito, realiza `sem_post(&sem_b_primera)`, liberando a P_2. Al despertar, como el permiso de P_3 en `sem_h_hecho` ya fue depositado previamente, P_2 avanza inmediatamente hacia '`e`' sin retraso. Por tanto, no existe inanición.

1.2 Problema 2: Verificación Formal Axiomática en Lógica de Hoare

Ejercicio 2: Demostración Concurrente de Bucle y Asignación Atómica

Considérese el siguiente programa concurrente, donde cada instrucción entre ⟨ y ⟩ se ejecuta de forma atómica:

```
{ x == 0 /\ y == 0 /\ n > 0 }
cobegin
    < x = x + 1 >;
 ||
    while (y < n) do begin
        < y = y + 1 >;
    end;
coend
{ x == 1 /\ y == n }
```

Se pide:

(a) Determinar si el triple de Hoare concurrente es demostrable usando la Lógica de Programas (incluyendo las reglas del bucle `while` y la regla de composición concurrente).

(b) En caso afirmativo, construir una demostración completa indicando: el invariante del bucle `while`, las triples atómicas utilizadas y las triples resultantes de combinar cada proceso mediante composición secuencial, iterativa y concurrente.

(c) Discutir formalmente la no interferencia de Owicki-Gries entre ambos procesos.

Solución Detallada

Apartado (a): Determinación de Demostrabilidad:
El triple es plenamente demostrable. Como el primer proceso manipula exclusivamente la variable x y el segundo proceso manipula exclusivamente la variable y (siendo n una constante de lectura), los conjuntos de variables modificadas son estrictamente disjuntos:

$$\text{VarMod}(P_1) = \{x\}, \qquad \text{VarMod}(P_2) = \{y\}.$$

Por consiguiente, la condición de no interferencia de Owicki-Gries se satisface de forma automática e invariante.
Apartado (b): Demostración Formal Paso a Paso:
Paso 1: Demostración del Proceso P_1 ($S_1 \equiv \langle x := x + 1 \rangle$):
Aplicamos el Axioma de Asignación:

$$\{Q[x/x+1]\}\ x := x + 1\ \{Q\}$$

Tomando $Q \equiv \{x = 1\}$, obtenemos:

$$\{x + 1 = 1\}\ \langle x := x + 1 \rangle\ \{x = 1\} \iff \{x = 0\}\ \langle x := x + 1 \rangle\ \{x = 1\}.$$

Por tanto, el triple de P_1 es válido:

$$\boxed{\{x = 0\}\ P_1\ \{x = 1\}}$$

Paso 2: Demostración del Proceso P_2 (Bucle `while`):
Definimos el invariante del bucle I y la condición de guarda B:

$$I \equiv 0 \leq y \leq n, \qquad B \equiv y < n.$$

Demostramos el cuerpo del bucle para la acción atómica $T \equiv \langle y := y + 1 \rangle$:

$$\{I \wedge B\}\ \langle y := y + 1 \rangle\ \{I\}$$

Por el axioma de asignación:

$$\{0 \leq y + 1 \leq n\}\ \langle y := y + 1 \rangle\ \{0 \leq y \leq n\}.$$

Comprobamos la regla de consecuencia previa:

$$(I \wedge B) \equiv (0 \leq y \leq n \wedge y < n) \implies 0 \leq y \leq n - 1 \implies 0 \leq y + 1 \leq n.$$

Como la implicación es tautológicamente cierta, el cuerpo del bucle es correcto. Aplicando la **Regla del Bucle While**:

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

Al terminar el bucle, la poscondición resultante es:

$$I \wedge \neg B \equiv (0 \leq y \leq n \wedge \neg(y < n)) \equiv (0 \leq y \leq n \wedge y \geq n) \iff \mathbf{y = n}.$$

Dado que la precondición inicial es $y = 0 \wedge n > 0$, evaluamos:

$$(y = 0 \wedge n > 0) \implies 0 \leq 0 \leq n \equiv I.$$

Por la Regla de Consecuencia (reforzamiento de precondición):

$$\boxed{\{y = 0 \wedge n > 0\}\ P_2\ \{y = n\}}$$

Apartado (c): Discusión Exhaustiva de No Interferencia (Owicki-Gries):
Para componer concurrentemente $\{P_1\}S_1\{Q_1\}$ y $\{P_2\}S_2\{Q_2\}$, debemos demostrar que ninguna sentencia atómica de un proceso invalida las aserciones intermedias o finales del otro. La condición formal de Owicki-Gries establece que para toda aserción $A \in$ Aserciones(P_1) y toda acción atómica $T \in P_2$ con precondición pre(T):

$$\{A \wedge \text{pre}(T)\}\ T\ \{A\}$$

- **Aserciones de P_1 frente a P_2:** Las aserciones de P_1 son $\{x = 0\}$ y $\{x = 1\}$. La única acción de P_2 es $T \equiv \langle y := y + 1 \rangle$. Dado que T modifica exclusivamente la variable y, el valor de verdad de cualquier aserción que dependa únicamente de x permanece inmutable:

$$\{x = 0 \wedge 0 \leq y < n\}\ \langle y := y + 1 \rangle\ \{x = 0\}, \quad \text{es trivialmente válido.}$$

- **Aserciones de P_2 frente a P_1:** Las aserciones de P_2 son $\{I\}$ e $\{I \wedge \neg B\}$, las cuales dependen únicamente de y y de la constante n. La acción de P_1 es $S_1 \equiv \langle x := x + 1 \rangle$, que no modifica y. Por tanto:

$$\{0 \leq y \leq n \wedge x = 0\}\ \langle x := x + 1 \rangle\ \{0 \leq y \leq n\}, \quad \text{es trivialmente válido.}$$

Demostrada la no interferencia global, aplicamos la **Regla de Composición Concurrente**:

$$\frac{\{x = 0\}P_1\{x = 1\} \quad \{y = 0 \wedge n > 0\}P_2\{y = n\} \quad \text{NoInterferencia}(P_1, P_2)}{\{x = 0 \wedge y = 0 \wedge n > 0\}\ \textbf{cobegin}\ P_1 \parallel P_2\ \textbf{coend}\ \{x = 1 \wedge y = n\}}$$

Q.E.D.

1.3 Problema 3: Exclusión Mutua con Aforo n y Cola FIFO

Ejercicio 3: Cerrojo de Capacidad n con Orden Estricto de Llegada

Se diseña una solución de exclusión mutua en memoria compartida donde pueden entrar como máximo n procesos simultáneamente en la sección crítica, con $n \geq 2$. En esta versión:

1. Se debe garantizar una **cola FIFO** para procesos que esperan cuando el aforo está completo.
2. El cerrojo permite hasta n procesos dentro, pero la cola usa un **único contador protegido**.
3. Al salir un proceso, solo se desbloquea a un proceso de la cola, no a todos.

Tareas:

(a) Dar pseudocódigo de entrada y salida utilizando variables en memoria compartida e instrucciones atómicas.

(b) Demostrar las propiedades de corrección: exclusión mutua de aforo ($N_{sc} \leq n$), vivacidad y orden FIFO.

Solución Detallada

1. Diseño Algorítmico (Ticket Lock Generalizado con Aforo):
Para garantizar orden FIFO estricto sin despertar a todos los procesos (evitando el efecto rebaño o *thundering herd*), empleamos el principio del algoritmo de la panadería / cerrojo de ticket:

- Una variable atómica `ticket_distribuidor` reparte turnos correlativos mediante `fetch_and_add`.
- Una variable `turno_servicio` indica qué turno tiene derecho a entrar a la sección crítica.
- Una variable `aforo_disponible` (inicializada a n) contabiliza las plazas libres.
- Para evitar despertar a todos, cada proceso puede esperar en una bandera individual en memoria o comprobar su turno directamente mediante espera activa ordenada: `mi_turno <turno_concedido`.

2. Pseudocódigo Formal de Entrada y Salida:

```
// Constante de aforo maximo
const int N_MAX = n; // con n >= 2

// Variables compartidas en memoria
atomic<int> ticket_distribuidor = 0; // Entrega turnos correlativos
atomic<int> turno_salida = 0;        // Contador de procesos que han
    salido
atomic<int> dentro = 0;              // Numero actual de procesos en SC
    (<= n)

// Protocolo de Entrada para el proceso Pi:
void entrar_SC() {
    // 1. Obtener numero correlativo de llegada de forma atomica:
```

```
    int mi_ticket = fetch_and_add(&ticket_distribuidor, 1);

    // 2. Condicion de entrada FIFO con aforo:
    // Un proceso con 'mi_ticket' solo puede ingresar si han salido suficientes procesos
    // de forma que su posicion en la cola no supere el aforo n:
    // Condicion: mi_ticket < turno_salida + N_MAX
    while (mi_ticket >= turno_salida.load(memory_order_acquire) + N_MAX) {
        // Espera activa (spin) o suspension controlada.
        // Solo el primer proceso de la cola superara esta condicion en cuanto
        // un proceso saliente incremente turno_salida.
    }

    // 3. Proceso ingresa formalmente a la SC:
    fetch_and_add(&dentro, 1);
}

// Protocolo de Salida para el proceso Pi:
void salir_SC() {
    // 1. Decrementar contador de procesos dentro:
    fetch_and_sub(&dentro, 1);

    // 2. Desbloquear exactamente al siguiente proceso de la cola FIFO:
    // Al incrementar 'turno_salida' en 1, exactamente UN unico proceso
    // (aquel con mi_ticket == turno_salida_anterior + N_MAX) vera su guarda cumplida:
    fetch_and_add(&turno_salida, 1);
}
```

3. Demostración de Propiedades de Corrección:

- **Seguridad (Aforo Máximo $N_{sc} \leq n$):**
  Sea k el valor actual de `turno_salida`. Por definición del bucle de espera, cualquier proceso con ticket T que haya ingresado a la sección crítica satisface obligatoriamente:
  $$T < k + n.$$
  Asimismo, como los procesos que ya han abandonado la sección crítica corresponden a los tickets $T < k$, los procesos que se encuentran simultáneamente dentro de la sección crítica pertenecen estrictamente al rango de tickets:
  $$k \leq T < k + n.$$
  La cardinalidad de este intervalo entero es:
  $$N_{sc} = |\{T \in \mathbb{N} \mid k \leq T \leq k + n - 1\}| = (k + n - 1) - k + 1 = \mathbf{n}.$$
  Por consiguiente, en cualquier instante temporal $N_{sc} \leq n$, garantizando la exclusión mutua de capacidad.

- **Orden Estricto de Llegada (FIFO):**
  La asignación de tickets mediante la instrucción atómica indivisible `fetch_and_add` establece un orden total estricto sobre las solicitudes: si el proceso P_i solicita entrar antes que P_j, entonces $\text{ticket}_i < \text{ticket}_j$. Como la condición de desbloqueo ticket $<$ turno_salida$+n$ es monótona no decreciente con respecto a `turno_salida`, P_i obtendrá permiso de entrada estrictamente antes o simultáneamente dentro del mismo cupo de n plazas que P_j, impidiendo cualquier adelantamiento arbitrario.

- **Desbloqueo Unitario (Sin Efecto Rebaño):**
  Cada ejecución de `salir_SC()` incrementa `turno_salida` en exactamente 1 unidad. El valor de la cota superior turno_salida $+ n$ avanza de M a $M + 1$. En consecuencia, únicamente el proceso que posea ticket $= M$ verá evaluada su guarda como verdadera, despertando a un único proceso a la vez.

1.4 Problema 4: Monitor CentroRecursos con Prioridad Acumulada

Ejercicio 4: Monitor para Dos Tipos Incompatibles con Prioridad Dinámica

Se desea un monitor `CentroRecursos` para gestionar el acceso de procesos de dos tipos (**Tipo 0** y **Tipo 1**) bajo las siguientes reglas:

1. Solo procesos del mismo tipo pueden estar dentro simultáneamente (aforo máximo $M > 0$).
2. Nunca pueden coincidir procesos de tipos distintos en el centro.
3. Si hay procesos esperando de ambos tipos, tiene prioridad el tipo que haya acumulado más esperas totales a lo largo de la ejecución.
4. Al terminar, un proceso saliente despierta a un proceso del tipo con mayor prioridad acumulada.

El monitor exporta los procedimientos: `entrar(int tipo)` y `salir(int tipo)`.
Tareas:

(a) Diseñar el monitor especificando variables internas, colas de condición y contadores.

(b) Explicar cómo se aplica la prioridad dinámica sin inconsistencias.

(c) Justificar formalmente la exclusión mutua, ausencia de interbloqueo y vivacidad.

2

Examen Parcial 2021

Metadatos de la Relación y Fuentes:

Ficha de la Convocatoria: Examen Parcial de SCD 2021 (Temas 1 y 2).
• **Contenido evaluado:** Monitor de Cuenta Bancaria con política estricta de retirada FIFO (evitando inanición de solicitudes grandes), y verificación axiomática concurrente formal de 3 procesos sumadores con el Teorema de No Interferencia de Owicki-Gries.

2.1 Problema 1: Monitor Cuenta Bancaria con Retiradas FIFO Estrictas

Ejercicio 1: Cuenta Bancaria con Saldo No Negativo y Turno de Retirada

Se desea implementar un monitor `CuentaBancaria` que gestione el saldo de una cuenta compartida mediante las operaciones:

- `depositar(int c)`: Incrementa el saldo en c unidades ($c > 0$) y despierta procesos bloqueados.
- `retirar(int c)`: Disminuye el saldo en c unidades ($c > 0$), garantizando en todo momento que el saldo nunca se vuelva negativo (saldo ≥ 0).

Requisitos de Concurrencia:

1. Las solicitudes de retirada deben ser satisfechas en **estricto orden de llegada (política FIFO)**. Si el primer cliente de la cola desea retirar una cantidad superior al saldo actual, debe esperar a que se realicen nuevos depósitos; ningún cliente que haya llegado más tarde puede adelantarlo, aunque solicite una cantidad menor.
2. No debe haber inanición ni interbloqueo.

Solución Detallada

1. Crítica al Intento del Alumno (Análisis del Borrador 2021):
En el archivo original del examen, el alumno escribió:

```
// CODIGO ERRONEO DEL ESTUDIANTE:
Cuenta::retirar(int c) {
    int pos = posicion;
    posicion++;
```

```
    if (cola.queue()) { cola.wait(pos); } // cola.queue() no existe en C++
    while (saldo < c) {
        cola.queue(pos); // INVENTADO: queue() es una consulta, no suspende hilos
    }
    saldo -= c;
    cola.signal();
}
```

- **Invención sintáctica:** `cola.queue(pos)` no existe en C++ ni en la biblioteca oficial `scd.h` (en la que las operaciones válidas son `cola.wait()`, `cola.signal()`, `cola.empty()` y `cola.get_nwt()`). Además, `wait()` no recibe parámetros.
- **Bucle infinito / Bloqueo activo del monitor:** Si `saldo <c`, el proceso ejecutaba repetidamente `cola.queue(pos)` sin invocar jamás `wait()`. Al no soltar la exclusión mutua del monitor, impedía que cualquier otro hilo pudiera entrar a ejecutar `depositar(c)`. Esto provocaba un **interbloqueo instantáneo y total**.

2. Implementación Canónica Oficial con `HoareMonitor` (UGR / Semántica SU): Para satisfacer la política estricta FIFO sin adelantos entre retiradas de distinto importe bajo la clase `HoareMonitor`, empleamos el patrón de **condición privada de turno**:

- `cola_espera`: Cola de condición donde esperan por estricto orden de llegada (garantizado FIFO por `scd.h`) los procesos que desean retirar cuando ya hay otro cliente gestionando su turno.
- `cola_turno`: Cola de condición donde espera exclusivamente el cliente que ostenta el turno actual si los fondos actuales no alcanzan para su petición específica (`saldo <c_primero`).

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

class CuentaBancariaFIFO : public HoareMonitor {
private:
    int      saldo;          // Invariante: saldo >= 0
    int      c_primero;      // Cantidad requerida por el proceso en cabeza de turno
    bool     turno_ocupado;  // true si hay un cliente en proceso de retirada
    CondVar cola_espera;     // Cola FIFO para clientes conforme van llegando
    CondVar cola_turno;      // Cola donde espera el cliente en turno si saldo < c_primero

public:
    CuentaBancariaFIFO() : saldo(0), c_primero(0), turno_ocupado(false) {
        cola_espera = newCondVar();
        cola_turno  = newCondVar();
    }

    void depositar(int c) {
        saldo += c;
        // Si hay un cliente en turno esperando fondos y ahora hay saldo suficiente:
```

```
        if (turno_ocupado && saldo >= c_primero) {
            cola_turno.signal(); // En semantica SU, despierta y entra de
                inmediato
        }
    }

    void retirar(int c) {
        // 1. Si ya hay otro cliente con el turno de retirada, esperar en
            cola FIFO:
        if (turno_ocupado) {
            cola_espera.wait();
        }

        // 2. Adquirir el turno de retirada en exclusiva:
        turno_ocupado = true;
        c_primero = c;

        // 3. Si no hay saldo suficiente, esperar bloqueado hasta que
            depositen:
        if (saldo < c) {
            cola_turno.wait();
        }

        // 4. Retirar fondos (saldo >= c plenamente garantizado bajo
            semantica SU):
        saldo -= c;

        // 5. Ceder el turno al siguiente cliente en la cola FIFO de
            espera:
        turno_ocupado = false;
        c_primero = 0;
        cola_espera.signal();
    }
};
```

3. Verificación de Seguridad y Ausencia de Inanición:

- **Seguridad (saldo ≥ 0):** Gracias a la semántica Señalar y Espera Urgente (SU), cuando `depositar()` ejecuta `cola_turno.signal()`, la hebra que esperaba en `cola_turno.wait()` reanuda de manera atómica con la garantía formal de que saldo $\geq c$. Como inicialmente saldo $= 0$, por inducción se preserva saldo ≥ 0 en todo instante.
- **Garantía FIFO y No Inanición:** La cola `cola_espera` de `HoareMonitor` garantiza matemáticamente orden FIFO. Un cliente posterior nunca puede saltar a `cola_turno` ni realizar una retirada mientras el cliente anterior mantenga `turno_ocupado == true`, erradicando completamente el adelantamiento y la inanición de retiradas de cuantía elevada.

2.2 Problema 2: Demostración Axiomática Concurrente de Tres Procesos

Ejercicio 2: Prueba Formal de Cobegin con No Interferencia Exhaustiva

Demostrar mediante la Lógica de Hoare y el Teorema de No Interferencia de Owicki-Gries que el siguiente programa concurrente es correcto:

```
{ x == 0 }
cobegin
     < x = x + 2 >; // Proceso P1 (S1)
 ||
     < x = x + 3 >; // Proceso P2 (S2)
 ||
     < x = x + 4 >; // Proceso P3 (S3)
coend
{ x == 9 }
```

Solución Detallada

1. Deducción Semántica de Estados Accesibles:
Cada proceso suma una constante entera (2, 3 y 4). Dado que la suma total es $2+3+4=9$ y las sumas son conmutativas, el valor intermedio de x antes de que cada proceso ejecute su acción refleja el subconjunto de procesos que ya han finalizado:

- **Para P_1 ($S_1 \equiv \langle x := x+2 \rangle$):** Antes de ejecutar S_1, han podido ejecutarse $\{\emptyset\}$, $\{S_2\}$, $\{S_3\}$ o $\{S_2, S_3\}$. Los valores posibles de x son:

$$0, \quad 0+3=3, \quad 0+4=4, \quad 0+3+4=7.$$

  Por consiguiente: $\text{pre}(S_1) \equiv P_1 \equiv \{x \in \{0,3,4,7\}\}$.
  Tras sumar 2, los valores resultantes son $\{0+2, 3+2, 4+2, 7+2\} = \{2,5,6,9\}$.
  Por consiguiente: $\text{post}(S_1) \equiv Q_1 \equiv \{x \in \{2,5,6,9\}\}$.

- **Para P_2 ($S_2 \equiv \langle x := x+3 \rangle$):** Antes de S_2, han podido ejecutarse $\{\emptyset\}$, $\{S_1\}$, $\{S_3\}$ o $\{S_1, S_3\}$. Los valores posibles de x son:

$$0, \quad 0+2=2, \quad 0+4=4, \quad 0+2+4=6.$$

  Por consiguiente: $\text{pre}(S_2) \equiv P_2 \equiv \{x \in \{0,2,4,6\}\}$.
  Tras sumar 3, los valores resultantes son $\{0+3, 2+3, 4+3, 6+3\} = \{3,5,7,9\}$.
  Por consiguiente: $\text{post}(S_2) \equiv Q_2 \equiv \{x \in \{3,5,7,9\}\}$.

- **Para P_3 ($S_3 \equiv \langle x := x+4 \rangle$):** Antes de S_3, han podido ejecutarse $\{\emptyset\}$, $\{S_1\}$, $\{S_2\}$ o $\{S_1, S_2\}$. Los valores posibles de x son:

$$0, \quad 0+2=2, \quad 0+3=3, \quad 0+2+3=5.$$

  Por consiguiente: $\text{pre}(S_3) \equiv P_3 \equiv \{x \in \{0,2,3,5\}\}$.
  Tras sumar 4, los valores resultantes son $\{0+4, 2+4, 3+4, 5+4\} = \{4,6,7,9\}$.
  Por consiguiente: $\text{post}(S_3) \equiv Q_3 \equiv \{x \in \{4,6,7,9\}\}$.

2. Demostración de las Triples Secuenciales Aisladas:
Aplicando el axioma de asignación para cada proceso:

$$\{Q_i[x/x+c_i]\} \ \langle x := x + c_i \rangle \ \{Q_i\}$$

$$\{x+2 \in \{2,5,6,9\}\}\ S_1\ \{Q_1\} \iff \{x \in \{0,3,4,7\}\}\ S_1\ \{Q_1\} \equiv \{P_1\}S_1\{Q_1\}.$$
$$\{x+3 \in \{3,5,7,9\}\}\ S_2\ \{Q_2\} \iff \{x \in \{0,2,4,6\}\}\ S_2\ \{Q_2\} \equiv \{P_2\}S_2\{Q_2\}.$$
$$\{x+4 \in \{4,6,7,9\}\}\ S_3\ \{Q_3\} \iff \{x \in \{0,2,3,5\}\}\ S_3\ \{Q_3\} \equiv \{P_3\}S_3\{Q_3\}.$$

3. Matriz Exhaustiva de No Interferencia de Owicki-Gries:
Debemos verificar formalmente las **12 condiciones cruzadas**:

$$\mathrm{NI}(A, S_k) \equiv \{A \wedge \mathrm{pre}(S_k)\}\ S_k\ \{A\}, \quad \text{con } A \in \{P_i, Q_i\},\ i \neq k.$$

Aserción A	**Acción** S_k	**Precondición** $A \wedge \mathbf{pre}(S_k)$	**Transformación y Estado Final** $\in A$
$P_2 = \{0,2,4,6\}$	$S_1(+2)$	$P_2 \wedge P_1 = \{0,4\}$	$0+2=2 \in P_2,\ 4+2=6 \in P_2$ ✓
$P_3 = \{0,2,3,5\}$	$S_1(+2)$	$P_3 \wedge P_1 = \{0,3\}$	$0+2=2 \in P_3,\ 3+2=5 \in P_3$ ✓
$Q_2 = \{3,5,7,9\}$	$S_1(+2)$	$Q_2 \wedge P_1 = \{3,7\}$	$3+2=5 \in Q_2,\ 7+2=9 \in Q_2$ ✓
$Q_3 = \{4,6,7,9\}$	$S_1(+2)$	$Q_3 \wedge P_1 = \{4,7\}$	$4+2=6 \in Q_3,\ 7+2=9 \in Q_3$ ✓
$P_1 = \{0,3,4,7\}$	$S_2(+3)$	$P_1 \wedge P_2 = \{0,4\}$	$0+3=3 \in P_1,\ 4+3=7 \in P_1$ ✓
$P_3 = \{0,2,3,5\}$	$S_2(+3)$	$P_3 \wedge P_2 = \{0,2\}$	$0+3=3 \in P_3,\ 2+3=5 \in P_3$ ✓
$Q_1 = \{2,5,6,9\}$	$S_2(+3)$	$Q_1 \wedge P_2 = \{2,6\}$	$2+3=5 \in Q_1,\ 6+3=9 \in Q_1$ ✓
$Q_3 = \{4,6,7,9\}$	$S_2(+3)$	$Q_3 \wedge P_2 = \{4,6\}$	$4+3=7 \in Q_3,\ 6+3=9 \in Q_3$ ✓
$P_1 = \{0,3,4,7\}$	$S_3(+4)$	$P_1 \wedge P_3 = \{0,3\}$	$0+4=4 \in P_1,\ 3+4=7 \in P_1$ ✓
$P_2 = \{0,2,4,6\}$	$S_3(+4)$	$P_2 \wedge P_3 = \{0,2\}$	$0+4=4 \in P_2,\ 2+4=6 \in P_2$ ✓
$Q_1 = \{2,5,6,9\}$	$S_3(+4)$	$Q_1 \wedge P_3 = \{2,5\}$	$2+4=6 \in Q_1,\ 5+4=9 \in Q_1$ ✓
$Q_2 = \{3,5,7,9\}$	$S_3(+4)$	$Q_2 \wedge P_3 = \{3,5\}$	$3+4=7 \in Q_2,\ 5+4=9 \in Q_2$ ✓

Tabla 2.1: Verificación completa de las 12 ternas de no interferencia de Owicki-Gries.

Como se aprecia en la tabla, en los 12 casos la ejecución de cualquier acción externa sobre cualquier estado accesible preserva rigurosamente el aserto.
4. Composición Concurrente Global: Habiéndose demostrado la no interferencia global, aplicamos la regla de composición concurrente:

$$\text{Precondición Global} \equiv P_1 \wedge P_2 \wedge P_3 = \{0,3,4,7\} \cap \{0,2,4,6\} \cap \{0,2,3,5\} = \{\mathbf{0}\},$$
$$\text{Poscondición Global} \equiv Q_1 \wedge Q_2 \wedge Q_3 = \{2,5,6,9\} \cap \{3,5,7,9\} \cap \{4,6,7,9\} = \{\mathbf{9}\}.$$

Concluimos formalmente que:

$$\boxed{\{x=0\} \quad \textbf{cobegin} \quad \langle x := x+2\rangle \parallel \langle x := x+3\rangle \parallel \langle x := x+4\rangle \quad \textbf{coend} \quad \{x=9\}}$$

Q.E.D.