

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2560 — Fundamentos de Lenguajes de Programación — 1' 2026

Interrogación 2 – Pauta

1. Explica cómo implementar monitores usando semáforos. Tienes que implementar exclusión mutua entre los métodos del monitor —mediante un semáforo `e`; y sincronización por condición entre los procesos que llaman a los métodos del monitor. En particular, tienes que desarrollar lo siguiente:

   a) Un *código de entrada* que es ejecutado después de que un proceso llama a un método del monitor, pero antes de que el proceso empiece a ejecutar el cuerpo del método.

   b) Un *código de salida* que es ejecutado justo antes de que el proceso retorne del cuerpo del método.

   c) Código que implementa `wait` y `signal` sobre las variables de condición —vamos a suponer sólo una variable de condición. Recuerda que una variable de condición es básicamente una cola FIFO, inicialmente vacía, de procesos suspendidos más un contador del número de procesos en la cola, inicialmente en 0. Para suspender procesos, considera que, además de `e`, hay un semáforo por cada proceso (un semáforo privado para cada proceso).

   Respuesta:

   Usamos la disciplina *signal and continue* (SC): el proceso que llama a `signal` sigue ejecutando dentro del monitor; el proceso despertado debe competir de nuevo por entrar (no recibe el monitor de inmediato).

   Variables globales:

```
semaphore e = 1;
int contador = 0;
queue cola;
semaphore s[1..n] = 0;
```

   a) Código de entrada (lo ejecuta el proceso p al llamar a cualquier método del monitor):

```
P(e);
```

   b) Código de salida (lo ejecuta el proceso p justo antes de retornar del cuerpo del método):

```
V(e);
```

   c) wait y signal (los ejecuta el proceso p dentro del cuerpo del método):

```
wait():
    contador = contador + 1;
    cola.enqueue(p);
    V(e);
```

```
    P(s[p]);
    P(e);

signal():
    if (contador > 0) {
        contador = contador - 1;
        q = cola.dequeue();
        V(s[q]);
    }
```

Notar que con SC, `signal` no garantiza que el proceso despertado ejecute de inmediato: solo lo habilita para volver a pedir `e`, por lo que al reanudar debe re-chequear la condición que esperaba (típicamente con un `while` en vez de un `if` al usar `wait`).

Puntaje:

- **(2 ptos)** Por el uso de un único semáforo mutex entre métodos del monitor
- **(2 ptos)** La implementación de wait
- **(2 ptos)** La implementación de signal.

2. ¿Qué hace la siguiente función en Scheme? Explica clara y precisamente.

```
(define (x lis)
  (cond
    ((null? lis) 0)
    ((not (list? (car lis)))
      (cond
        ((eq? (car lis) #f) (x (cdr lis)))
        (else (+ 1 (x (cdr lis))))))
    (else (+ (x (car lis)) (x (cdr lis))))))
```

Respuesta:

La función `x` recorre `lis` recursivamente y **cuenta cuántos elementos atómicos distintos de `#f`** hay en total, contando también los elementos dentro de las sublistas anidadas (a cualquier nivel de profundidad).

En detalle:

- Si `lis` es vacía, no hay nada que contar: retorna 0.
- Si el primer elemento (`car lis`) no es una lista (es un átomo):
  - si ese átomo es `#f`, no se cuenta (se "salta"), y se sigue contando en el resto de la lista (`cdr lis`);
  - si no es `#f`, se cuenta 1 más lo que se cuente en el resto de la lista.
- Si el primer elemento sí es una lista, se cuenta recursivamente lo que haya dentro de esa sublista, más lo que haya en el resto de `lis`.

En otras palabras: `x` cuenta el número de hojas (átomos) distintas de `#f` en el árbol que representa `lis`.

Puntaje:

- **(1 pto)** Por el caso base de la recursión.
- **(2.5 pto)** Por la identificación/interpretación de las condiciones.
- **(2.5 ptos)** Por el funcionamiento del paso recursivo.

3. Responde una sola: a) o b).

   a) Considera el siguiente comando: `for [i = 1 ... N] S;`

      1 pt. Reescribe este comando usando un *loop* `while` más comandos explícitos de asignación a la variable cuantificadora `i`.

      5 pts. Luego, a partir del *axioma de asignación* y de la *regla del comando while*, desarrolla una regla de inferencia para el comando `for`.

   b) Considera que los triples $\{P1\}$ $S1$ $\{Q1\}$ y $\{P2\}$ $S2$ $\{Q2\}$ son ambos verdaderos y son libres de interferencia. Considera además que S1 contiene un comando `await` de la forma `<await (B) T>`, en que `B` es una expresión booleana y `T` es un comando. Sea S1' igual a S1, excepto que el comando `await` es reemplazado por el par de comandos

      ```
      while (not B);   —no hay un comando supeditado al while
      T;               —el mismo comando T anterior
      ```

      Responde las siguientes preguntas de manera independiente entre ellas:

      i) ¿Es $\{P1\}$ $S1'$ $\{Q1\}$ verdadero? Explica clara y precisamente si es verdadero siempre, algunas veces, o nunca.

      ii) ¿Son los triples $\{P1\}$ $S1'$ $\{Q1\}$ y $\{P2\}$ $S2$ $\{Q2\}$ libres de interferencia? Explica clara y precisamente si lo son siempre, algunas veces, o nunca.

Respuesta:

a) i) Reescritura con `while`:

```
i := 1;
while (i <= N) do
    S;
    i := i + 1;
```

a) ii) Sea $P(i)$ el predicado invariante a iterar para el ciclo. Utilizando el axioma de asignación $\{P[E/x]\}$ $x := E$ P sobre i := i+1, obtenemos $\{P(i+1)\}$ i:=i+1 $\{P(i)\}$. Combinando esto con la premisa del cuerpo del ciclo $\{P(i) \wedge i \leq N\}$ S $\{P(i+1)\}$ y aplicando la regla del while, obtenemos la regla de inferencia para el for:

$$\frac{\{P(i) \wedge i \leq N\}\ S\ \{P(i+1)\}}{\{P(1)\}\ \texttt{for [i = 1 ... N] S}\ \{P(N+1)\}}$$

Nota: Se asume que i no es modificada por S y N no depende de i

b) i) **Siempre verdadero.** La validez de los triples se analiza de forma secuencial, ignorando procesos externos. Secuencialmente, `<await (B) T>` significa "esperar hasta que B sea verdadero y ejecutar T", lo cual es indistinguible de la secuencia while (not B); T. Por lo tanto, si el triple original es verdadero, $S1'$ también lo es.

b) ii) **No siempre (algunas veces).** La diferencia entre `S1` y `S1'` es la **atomicidad**:

`<await (B) T>` se ejecuta como una única acción indivisible (ningún otro proceso puede ejecutar nada entre que se evalúa B y termina T), mientras que en `while (not B); T`, se introduce un nuevo punto

de control entre el while y T. En este punto, el proceso $S2$ puede ejecutar acciones que modifiquen variables de las que dependen B o T, rompiendo potencialmente la propiedad de no interferencia.

Puntaje (a):

- **(1 pto)** Por reescribir el comando.
- **(2 ptos)** Por la aplicación correcta del axioma de asignación para la variable
- **(2 ptos)** Por combinar correctamente las premisas usando la regla de composición secuencial.
- **(1 pto)** Por la estructura final.

Puntaje (b):

- **(2 ptos)** Por cada respuesta (si/no).
- **(1 pto)**Por cada explicación.

4. Considera un juego en un parque de diversiones, en que hay n procesos pasajeros y un proceso carro. Los pasajeros repetidamente esperan para poder dar una vuelta en el carro, que puede llevar hasta C pasajeros, $C < n$. Sin embargo, el carro da una vuelta sólo cuando está lleno.

   a) Escribe el código de los procesos pasajeros y del proceso carro, y escribe un **monitor** para sincronizarlos. El monitor debe tener tres operaciones: `darUnaVuelta()`, que es llamada por los pasajeros; y `subirPasajeros()` y `bajarPasajeros()`, que son llamadas por el carro. Usa la disciplina SC.

   b) Generaliza tu respuesta a a) para usar m procesos carros, $m > 1$. Considera que hay una sola pista para todos los carros, por lo que los carros no pueden adelantarse unos a otros, es decir, deben terminar las vueltas en el mismo orden en que las empiezan.

Respuesta:

Como estamos trabajando con SC (Signal-and-Continue), el proceso que ejecuta `signal()` continúa dentro del monitor, por lo que los procesos despertados pueden perder su condición frente a procesos nuevos (*barging*). Para solucionar esto, utilizamos un sistema de **cupos exactos** en lugar de booleanos simples.

a) Implementación con 1 Carro

Código de los Procesos:

```
while (true) {
    Parque.darUnaVuelta();
}

while (true) {
    Parque.subirPasajeros();
    Parque.bajarPasajeros();
}
```

Monitor `Parque`**:**

```
monitor Parque {
    int esperando = 0;
    int a_bordo = 0;
    int cupos_subir = 0;
    int cupos_bajar = 0;

    cond c_pasajeros;
    cond c_viaje;
    cond c_carro_espera;
    cond c_carro_lleno;
    cond c_carro_vacio;

    void darUnaVuelta() {
        esperando++;

        if (esperando >= C && cupos_subir == 0) {
            signal(c_carro_espera);
```

```
        }

        while (cupos_subir == 0) {
            wait(c_pasajeros);
        }

        cupos_subir--;
        esperando--;
        a_bordo++;

        if (a_bordo == C) { signal(c_carro_lleno); }

        while (cupos_bajar == 0) {
            wait(c_viaje);
        }

        cupos_bajar--;
        a_bordo--;

        if (a_bordo == 0) { signal(c_carro_vacio); }
    }

    void subirPasajeros() {
        while (esperando < C) { wait(c_carro_espera); }

        cupos_subir = C;
        for (int i = 0; i < C; i++) { signal(c_pasajeros); }

        while (a_bordo < C) { wait(c_carro_lleno); }
    }

    void bajarPasajeros() {
        cupos_bajar = C;
        for (int i = 0; i < C; i++) { signal(c_viaje); }

        while (a_bordo > 0) { wait(c_carro_vacio); }
    }
}
```

b) Generalización a m Carros

Para extender la solución a m carros con una única pista, garantizamos el orden de **entrada** al carro con un turno dentro del monitor (`turno_subir`), y el orden de **término** de la vuelta con una cadena de semáforos manejada por los propios procesos carro, fuera del monitor. Así evitando que un proceso carro quede bloqueado con `P()` mientras retiene el mutex del monitor, lo cual bloquearía innecesariamente a todos los demás procesos.

Semáforos globales (fuera del monitor):

```
semaphore s[0..m-1];
s[m-1] = 1;
s[i] = 0; // para todo i != m-1
```

Código de los Procesos:

```
while (true) {
    ParqueMultiple.subirPasajeros(id);
    P(s[(id - 1 + m) % m]);
    ParqueMultiple.bajarPasajeros(id);
    V(s[id]);
}
```

Monitor `ParqueMultiple`**:**

```
monitor ParqueMultiple {
    int esperando = 0;

    int a_bordo[m] = {0};
    int cupos_subir[m] = {0};
    int cupos_bajar[m] = {0};

    int turno_subir = 0;

    cond c_pasajeros;
    cond c_viaje[m];
    cond cola_carros_subir[m];
    cond c_carro_lleno[m];
    cond c_carro_vacio[m];

    void darUnaVuelta() {
        esperando++;
        if (esperando >= C && cupos_subir[turno_subir] == 0) {
            signal(cola_carros_subir[turno_subir]);
        }

        while (cupos_subir[turno_subir] == 0) { wait(c_pasajeros); }

        int mi_carro = turno_subir;

        cupos_subir[mi_carro]--;
```

```
        esperando--;
        a_bordo[mi_carro]++;

        if (a_bordo[mi_carro] == C) { signal(c_carro_lleno[mi_carro]); }

        while (cupos_bajar[mi_carro] == 0) { wait(c_viaje[mi_carro]); }

        cupos_bajar[mi_carro]--;
        a_bordo[mi_carro]--;

        if (a_bordo[mi_carro] == 0) { signal(c_carro_vacio[mi_carro]); }
    }

    void subirPasajeros(int id) {
        while (id != turno_subir || esperando < C) {
            wait(cola_carros_subir[id]);
        }

        cupos_subir[id] = C;
        for (int i = 0; i < C; i++) { signal(c_pasajeros); }

        while (a_bordo[id] < C) { wait(c_carro_lleno[id]); }

        turno_subir = (turno_subir + 1) % m;
        signal(cola_carros_subir[turno_subir]);
    }

    void bajarPasajeros(int id) {
        cupos_bajar[id] = C;
        for (int i = 0; i < C; i++) { signal(c_viaje[id]); }

        while (a_bordo[id] > 0) { wait(c_carro_vacio[id]); }
    }
}
```

Puntaje (a):

- **(1 pto)** Uso correcto de `while` en las condiciones de espera.
- **(1 pto)** Lógica de sincronización (mantener SC, evitando *busy waiting*).
- **(1 pto)** Control de errores de concurrencia (evitar booleanos compartidos susceptibles a *barging*) y estado de pasajeros.

Puntaje (b):

- **(1 pto)** Manejo correcto de la entrada de los carros.
- **(2 pto)** Uso correcto de los semáforos (inicialización, ubicación, etc).