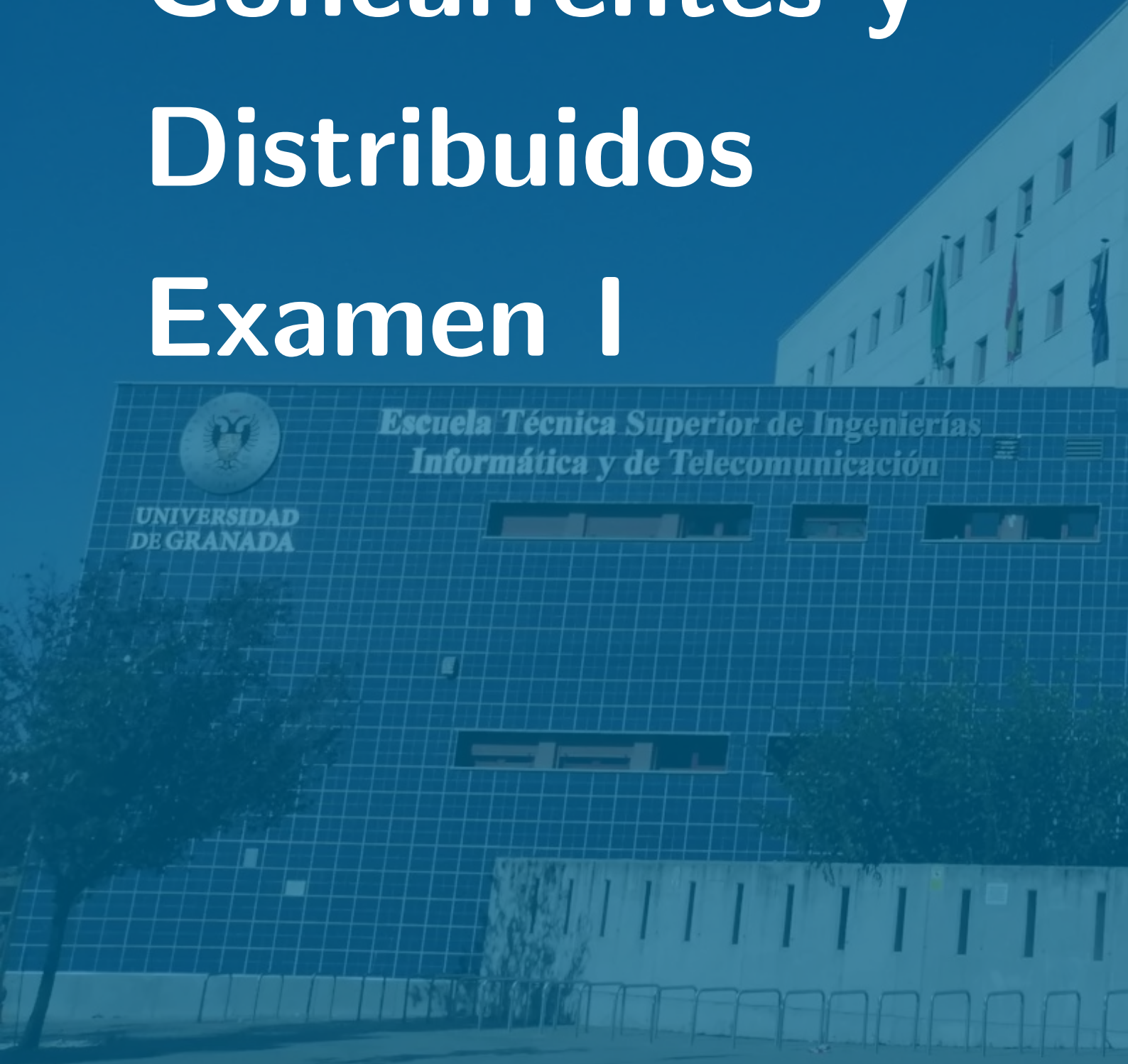
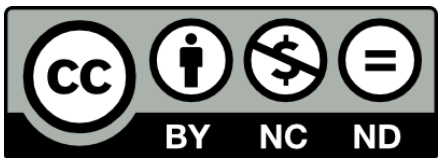
Sistemas Concurrentes y Distribuidos Examen I



Los Del DGIIM, `losdeldgiim.github.io`

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Sistemas Concurrentes y Distribuidos Examen I

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán

Granada, 2024

Asignatura Sistemas Concurrentes y Distribuidos.

Curso Académico 2020-21.

Grupo Único.

Descripción Examen parcial de los Temas 1, 2 y 3 de SCD.

Fecha 13-01-2021.

Preguntas de respuesta alternativa: 35 %

Ejercicio 1. Seleccione la única respuesta correcta de cada una de las cuestiones siguientes; cada una bien contestada tiene una puntuación de **0.5**.

1. Un programa concurrente que cumpla la propiedad de *seguridad* para todas sus ejecuciones se considerará correcto si además cumple que:

   (a) Los procesos del programa nunca pueden llegar a una situación de *interbloqueo*.

   (b) Se puede demostrar que sus procesos no sufren *inanición* en ninguna posible ejecución del programa.

   (c) Sus procesos siempre consiguen ejecutar sus instrucciones de forma equitativa o justa.

   (d) Habrá que demostrar la no *inanición* de los procesos y, además, la *vivacidad* (o "liveness").

2. La *hipótesis de progreso finito* asegura que durante la ejecución de cualquier programa concurrente se cumplirá:

   (a) La velocidad de ejecución individual de los procesos está limitada por las características del procesador que ejecute el código generado por el compilador.

   (b) Siempre ha de existir algún proceso activo (ejecutándose) durante la ejecución del programa.

   (c) La propiedad de *ausencia de inanición* de los procesos en todo momento.

   (d) Cualquier proceso del programa, que comienza la ejecución de una instrucción, ha de completar alguna vez la ejecución de esta.

3. Aplicando la propiedad *como máximo una vez* a la evaluación de sentencias como $x = expresion$, que son evaluadas por los siguientes procesos concurrentes ¿cuál de las siguientes evaluaciones se puede considerar que se realiza *atómicamente* (`<x = expresion>`)? (suponer que los valores iniciales de las variables == 0 y que `a` y `b` son constantes):

   (a) P_1: `cobegin x = y + a || y = x + b coend`.

   (b) P_2: `cobegin x = y + a || y = f(x) + b coend`.

   (c) P_3: `cobegin x = y + a || y = a + b coend`.

   (d) P_4: `cobegin x = x / y || y = a + x coend`.

4. Respecto de la orden de espera selectiva (`select`) utilizadas por sistemas con paso de mensajes bloqueante, se poducirá inmediatamente la elección y ejecución si:

   (a) En ese momento solo exista 1 proceso del programa que haya iniciado el envío de mensaje.

   (b) Sea la única orden *potencialmente ejecutable* de dicha instrucción `select`.

(c) Sea *potencialmente ejecutable* y se nombre en la guarda un proceso que ya inició su envío.

(d) Nombre a uno de los procesos del programa que ya iniciaron su envío de mensaje.

5. En relación al problema de la sección crítica para N procesos y su relación con las propiedades de corrección de los programas concurrentes se puede afirmar que:

   (a) Si un algoritmo cumple las 4 *condiciones de Dijkstra*, entonces se puede decir que es una solución *totalmente correcta* a tal problema.

   (b) En el algoritmo de Dijkstra se puede afirmar que ningún grupo de procesos puede ser adelantado indefinidamente por otros procesos que consiguen acceder a la sección crítica repetidamente, impidiéndoles a entrar a esta.

   (c) Con el algoritmo de Knuth el mayor retraso que podría sufrir un proceso *solicitante*, sujeto al peor escenario posible de planificación para él, sería esperar como máximo un número de turnos igual al número de procesos anteriores a dicho proceso en el denominado turno cíclico de acceso a la sección crítica.

   (d) Con el algoritmo de Peterson nunca podría ocurrir que existieran etapas vacías (sin ningún proceso esperando) y anteriores a una etapa a la que se acaba de unir un segundo proceso.

6. Respecto de la demostración de las propiedades de los monitores:

   (a) El invariante de los monitores –para señales de semántica desplazante (SS,SE,SU)– necesariamente ha de cumplirse después de ejecutar la operación de sincronización `c.wait()` en la programación de los procedimientos de un monitor.

   (b) Con semántica de *señales urgentes* (SU) la ejecución de una operación `c.wait()` nunca puede provocar la entrada al monitor de un proceso suspendido en una cola distinta a la cola de entrada al monitor.

   (c) Para programar correctamente los monitores que usan variables condición del tipo *señalar y salir* (SS) hay que programar siempre la operación de sincronización `c.signal()` como la última instrucción de los procedimientos.

   (d) Si las señales que usa un monitor tienen semántica *señalar y continuar* (SC), el proceso señalador –tras provocar la ejecución de `c.signal()`– sigue su ejecución dentro del monitor pero el proceso notificado solo sale de cola en la que estuviera suspendido.

7. Respecto de las operaciones de paso de mensajes no-bloqueantes:

   (a) Siempre (incluso son soporte hardware) es necesario programar operaciones de comprobación que indiquen si es seguro acceder a los datos en transmisión antes de que la ejecución de la operación `receive()` devuelva el control al proceso receptor.

(b) Si el proceso receptor está preparado para recibir los datos en transmisión, la ejecución de la operación `receive()` *vuelve* inmediatamente siempre.

(c) El proceso emisor siempre supone que la ejecución de la operación `send()` accederá a datos en un estado inseguro.

(d) Existe un caso en el cual la ejecución de la operación `receive()` no detiene al proceso receptor aunque no se hayan terminado de transmitir los datos que han de recibirse.

Cuestiones y ejercicios: 40 %

Ejercicio 2. Sobre la diferencia conceptual existente entre las denominadas *propiedades de seguridad* y *vivacidad* respecto de su validez temporal durante la ejecución de un programa concurrente:

- Clasificar las propiedades de la tabla según el tipo de propiedad al que pertenecen (aparte, justificarlo brevemente).
- ¿En qué instante(s) de la ejecución de los programas se ha de cumplir cada uno de los siguientes tipos de propiedades? Elegir una de las posibilidades de validez temporal para cada una de las propiedades de la siguiente tabla:

  (a) Siempre.

  (b) Alguna vez.

  (c) Intermitentemente.

  (d) Alguna vez en el futuro y, desde entonces, para siempre.

nombre propiedad	**tipo propiedad** (*seguridad*|*vivacidad*)	(**a**|**b**|**c**|**d**)
Exclusión mutua		
Ausencia de interbloqueo		
Alcanzabilidad de la SC		
Productor-consumidor		
No inanición procesos		
Equidad procesoss		
Finalización de los cálculos		
Acceso individual a SC		

Tabla 1: Tabla para el ejercicio 2.

Ejercicio 3. Considerando el siguiente programa concurrente:

```
1 S :: cobegin <x = x+2> || <x = x+3> || <x = x+4> coend
```

Aplicando las reglas de demostración concurrentes estudiadas, demostrar que el siguiente triple de Hoare es un aserto demostrablemente cierto de la Lógica de Programas:

$$\{x = 0\}\ S\ \{x = 9\}$$

Ejercicio 4. Si los procedimientos de un monitor solo pueden ser ejecutados por un proceso de un programa concurrente a la vez, ¿cómo se justifica decir que durante la ejecución de un programa concurrente con un monitor se producirá entrelazamiento de sus instrucciones?

Resolución de problemas: 25 %

Ejercicios para los alumnos de GIADE.

Ejercicio 5. Una cuenta de ahorros es compartida por varias personas. Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo. Queremos usar un monitor para resolver el problema. El monitor debe tener 2 procedimientos: `depositar(c)` y `retirar(c)`. Suponer que los argumentos de las 2 operaciones son siempre positivos, e indican las cantidades a depositar o retirar. El monitor usará la semántica señalar y espera urgente (SU).

El reintegro de fondos a los clientes se hace únicamente según el orden de llegada, es decir, si hay más de un cliente esperando, sólo el primero que llegó puede optar a retirar la cantidad que desea, mientras esto no sea posible, esperarán todos los demás clientes, independientemente de cuanto quiera retirar cada uno. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está ya esperando un reintegro de 300 unidades; si llega después otro cliente que quiere retirar las 200 unidades, debe esperarse. Para resolverlo se pueden utilizar variables condición prioritarias.

Ejercicio 6. Supongamos que tenemos N procesos concurrentes semejantes. Cada proceso produce $N-1$ caracteres (con $N-1$ llamadas a la función `ProduceCaracter()`) y envía cada carácter a los otros $N-1$ procesos. Además, cada proceso debe imprimir todos los caracteres recibidos de los otros procesos. En la solución al problema anterior se ha de garantizar que el orden en el que se imprimen los caracteres es el mismo orden en el que se iniciaron los envíos de dichos caracteres (pista: usa un `select` para recibir).

Ejercicios para los alumnos de GIM.

Ejercicio 7. Suponer un sistema básico de asignación de páginas de memoria de un sistema operativo que proporciona 2 operaciones: `adquirir(positive n)` y `liberar(positive n)` para que los procesos de usuario puedan obtener las páginas que necesiten y, posteriormente, dejarlas libres para ser utilizadas por otros procesos del sistema. Cuando los procesos llaman a la operación `adquirir(positive n)`, si no hay memoria disponible para atenderla, la petición quedaría pendiente hasta que exista un número de páginas libres suficiente en memoria. Llamando a la operación `liberar(positive n)`, un proceso convierte en disponibles `n` páginas de la memoria del sistema. Suponemos que los

procesos adquieren y devuelven páginas del mismo tamaño a un área de memoria con estructura de cola y en la que suponemos que no existe el problema conocido como fragmentación de páginas de la memoria.

Se pide definir el invariante y programar un monitor –de acuerdo con él– con las operaciones anteriores suponiendo semántica de señales SU. Resolverlo para los dos casos siguientes: (a) suponiendo orden FIFO estricto para atender las llamadas a la operación de *adquirir* páginas por parte de los procesos del sistema; (b) relajando la condición anterior, resolverlo ahora atendiendo las llamadas según el siguiente orden prioritario: petición pendiente con "menor número de páginas primero" (SJF) y utilizando variables condición prioritarias en este segundo caso.

Ejercicio 8. Se tienen N procesos cliente que interactúan con el proceso servidor de 1 cajero automático.

- Los procesos cliente obtienen dinero del cajero realizando lo siguiente: informan de su identidad al proceso servidor del cajero y solicitan una cantidad de dinero.
- El proceso cajero responde con la cantidad solicitada si el cliente tiene suficiente saldo y dispone de suficiente efectivo; si no, el cajero responde denegando la petición al cliente.
- Para ingresar dinero en el cajero los procesos sólo tienen que identificarse e indicar la cantidad que van a ingresar.
- Suponer que cada cliente tiene inicialmente 10 unidades de saldo y que el cajero posee 100 unidades de efectivo (hasta que las agote y no se incrementan con los ingresos) para servir las peticiones de los clientes.
- Cuando el cajero agota completamente las 100 unidades de efectivo, no podrá servir peticiones de ningún tipo hasta pasada 1 hora, transcurrido ese tiempo se vuelven a reponer las 100 unidades

Se pide: utilizar la orden de selección no determinista para programar el proceso servidor del cajero. Programar, además, un cliente generico que realice ingresos y reintegros aleatorios y repetitivamente para completar la simulación.

Soluciones

Ejercicio 1. Aunque en el examen original no era necesario justificar las respuestas[1]:

1. (b): (a) no es porque situación de interbloqueo es una propiedad de seguridad, (c) tampoco porque la equidad no es necesaria para que el programa se considere correcto y (d) tampoco porque la no inanición y la vivacidad son lo mismo.

2. (d): la ausencia de inanición en (c) es una propiedad de vivacidad y (a) y (b) no tienen nada que ver con progreso finito.

3. (c): ya que en todas se modifican las variables `x` e `y`, pero (c) es el único bloque que utiliza *como máximo una vez* una variable modificada, `y` (el resto utilizan tanto `y` como `x`).

4. (c): ya que en (a) la condición de la guarda puede ser falsa, en (b) el programa se bloqueará y en (d) puede que la sentencia `select` seleccione a otra instruccion con guarda para ejecución.

5. (d): (a) no es porque faltaría probar la vivacidad de la solución (tendríamos solo ausencia de interbloqueo y exclusión mutua), (b) no es porque no comple la propiedad de vivacidad, (c) tampoco porque el mayor retraso es de $f(n) = 2f(n-1)+1$ y (d) sí que es por el Lema 3 que vimos en teoría (si en alguna etapa existen al menos dos procesos, entonces el resto de etapas anteriores contienen como mínimo un proceso, por lo que no pueden existir etapas vacías anteriores a la misma).

6. (d): (a) es falsa porque después de `c.wait()` ha de cumplirse la *condición de sincronización* de la variable `c`, (b) es falsa porque contamos además con la cola de procesos urgentes cuando trabajamos con SU y (c) es falsa porque la tras `c.signal()` podríamos colocar un `wait` sobre otra (o la misma) variable condición. Además, (d) es cierta porque describe el comportamiento de SC.

7. (d).

Ejercicio 2. Rellenamos la tabla, teniendo en cuenta que (a) y (c) hacen referencia a propiedades de seguridad y (b) y (d) a propiedades de vivacidad.

nombre propiedad	**tipo propiedad** (*seguridad*|*vivacidad*)	(**a**|**b**|**c**|**d**)
Exclusión mutua	seguridad	a
Ausencia de interbloqueo	seguridad	a
Alcanzabilidad de la SC	seguridad	a
Productor-consumidor	seguridad	a
No inanición procesos	vivacidad	b
Equidad procesos	vivacidad	b
Finalización de los cálculos	vivacidad	d
Acceso individual a SC	seguridad	c

[1]Salvo para aclarar algunas elecciones.

Ejercicio 3. Hemos de demostrar $\{s = 0\}\ S\ \{x = 9\}$ siendo S:

```
1  cobegin <x = x+2> || <x = x+3> || <x = x+4> coend
```

Para ello, primero consideramos los triples:

$$\{x = 0 \vee x = 3 \vee x = 4 \vee x = 7\}\ \texttt{<x = x+2>;}\ \{x = 2 \vee x = 5 \vee x = 6 \vee x = 9\}$$
$$\{x = 0 \vee x = 2 \vee x = 4 \vee x = 6\}\ \texttt{<x = x+3>;}\ \{x = 3 \vee x = 5 \vee x = 7 \vee x = 9\}$$
$$\{x = 0 \vee x = 2 \vee x = 3 \vee x = 5\}\ \texttt{<x = x+4>;}\ \{x = 4 \vee x = 6 \vee x = 7 \vee x = 9\}$$

Todos ellos ciertos por el axioma de asignación:

$$\{x = 2 \vee x = 5 \vee x = 6 \vee x = 9\}^x_{x+2} \equiv$$
$$\{x + 2 = 2 \vee x + 2 = 5 \vee x + 2 = 6 \vee x + 2 = 9\} \equiv$$
$$\{x = 0 \vee x = 3 \vee x = 4 \vee x = 7\}$$

$$\{x = 3 \vee x = 5 \vee x = 7 \vee x = 9\}^x_{x+3} \equiv$$
$$\{x + 3 = 3 \vee x + 3 = 5 \vee x + 3 = 7 \vee x + 3 = 9\} \equiv$$
$$\{x = 0 \vee x = 2 \vee x = 4 \vee x = 6\}$$

$$\{x = 4 \vee x = 6 \vee x = 7 \vee x = 9\}^x_{x+4} \equiv$$
$$\{x + 4 = 4 \vee x + 4 = 6 \vee x + 4 = 7 \vee x + 4 = 9\} \equiv$$
$$\{x = 0 \vee x = 2 \vee x = 3 \vee x = 5\}$$

Renombrando a los triples anteriores de forma:

$$\{P_1\}\ S_1\ \{Q_1\} \qquad \{P_2\}\ S_2\ \{Q_2\} \qquad \{P_3\}\ S_3\ \{Q_3\}$$

repectivamente. Hemos de probar la no interferencia de cada triple con el resto, por lo que hemos de probar 12 triples de no interferencia:

$$\begin{array}{llll} NI(P_2, S_1) & NI(Q_2, S_1) & NI(P_3, S_1) & NI(Q_3, S_1) \\ NI(P_1, S_2) & NI(Q_1, S_2) & NI(P_3, S_2) & NI(Q_3, S_2) \\ NI(P_2, S_3) & NI(Q_2, S_3) & NI(P_1, S_3) & NI(Q_1, S_3) \end{array}$$

Probaremos uno de ellos y notaremos que la por la forma en la que hemos definido los tres triples anteriores, hemos ya tenido en cuenta las precondiciones y poscondiciones apropiadas para la no interferencia entre cada uno de ellos:

$$NI(P_2, S_1) \equiv \{P_2 \wedge P_1\}\ S_1\ \{P_2\} \equiv$$
$$\{x = 0 \vee x = 4\}\ \texttt{<x = x+2>;}\ \{x = 0 \vee x = 2 \vee x = 4 \vee x = 6\}$$

Como $\{x = 0 \vee x = 4\}$ `<x = x+2>;` $\{x = 2 \vee x = 6\}$ es cierto por el axioma de asignación y se verifica que $\{x = 2 \vee x = 6\} \rightarrow P_2$, tenemos que $NI(P_2, S_1)$ es cierto por la primera regla de la consecuencia.

Probando todos los triples de no interferencia, llegamos a que los tres triples anteriores están libres de interferencia, por lo que podemos aplicar la regla de la composición concurrente, llegando a que el triple:

$$\{P_1 \wedge P_2 \wedge P_3\}\ S\ \{Q_1 \wedge Q_2 \wedge Q_3\} \equiv \{x = 0\}\ S\ \{x = 9\}$$

es cierto.

Ejercicio 4. Durante la ejecución de un programa concurrente con un monitor se producirá entrelazamiento de sus instrucciones si dentro del monitor hemos usado variables tipo condición y operaciones `wait` y `signal` sobre las mismas, de forma que troozos de código de procedimientos se puedan ejecutar entre el código de un procedimiento distinto, como consecuencia del bloqueo del proceso que ejecutaba dicho procedimiento.

Ejercicios para GIADE.

Ejercicio 5. Para resolver el ejercicio, hemos hecho uso de una variable permanente auxiliar `contador`, para asignar los turnos a los clientes que quieran extrar de su cuenta.

```
1  monitor Cuenta;
      var saldo, contador : integer;
          cola : condition;

5     begin
         saldo := 0;
         contador := 0;
      end

10    procedure depositar(c : integer);
      begin
         {deposito c y dejo pasar al siguiente}
         saldo := saldo + c;
         cola.signal();
15    end

      procedure retirar(c : integer);
      var ticket : integer;
      begin
20       {coge su ticket}
         ticket := contador;
         contador := contador + 1;

         {si hay alguien delante, espera}
25       if cola.queue() then
            cola.wait(ticket);

         {si soy el primero, espero hasta que el saldo incremente}
         while saldo < c do begin
30          cola.wait(ticket);
         end
```

```
      {me llevo lo que quiero y dejo pasar al siguiente}
      saldo := saldo - c;
      cola.signal();
   end
end
```

Ejercicio 6. Se trata de un ejercicio de programación distribuida.

Ejercicios para GIM.

Ejercicio 7. Resolvemos los dos apartados:

(a) Presentamos la solución a la primera opción, la asignación FIFO, definiendo antes el invariante:

$$\{IM\} \equiv \{0 \leqslant paginas_{ocupadas} \leqslant libres\_iniciales \wedge (libres = 0 \iff paginas_{ocupadas} = libres\_iniciales)\}$$

La solución es similar que para el Ejercicio 5:

```
monitor AsignacionFIFO(libres_iniciales : positive);
   var libres : integer;
       cola, ventanilla : condition;

   begin
      libres := libres_iniciales;
   end

   procedure liberar(n : positive);
   begin
      {Se liberan las páginas}
      libres := libres + n;
      ventanilla.signal();
   end

   procedure adquirir(n : positive);
   var ticket : integer;
   begin
      {Si hay otro antes, espera}
      if ventanilla.queue() then
         cola.wait();

      {Espera mientras no haya páginas libres}
      while libres < n do begin
         ventanilla.wait();
      end

      {Reserva sus páginas}
      libres := libres - n;
      cola.signal();
   end
end
```

(b) Si ahora realizamos la asignación según SJF (*Shortest Job First*), usando variables condición prioritarias:

```
1  monitor AsignacionSJF(libres_iniciales : positive);
      var libres : integer;
          cola : condition;

5     begin
         libres := libres_iniciales;
      end

      procedure liberar(n : positive);
10    begin
         {Se liberan las páginas}
         libres := libres + n;
         cola.signal();
      end
15
      procedure adquirir(n : positive);
      begin
         {Espera mientras no haya páginas libres}
         while libres < n do begin
20          cola.wait(n);
         end

         {Reserva sus páginas}
         libres := libres - n;
25       cola.signal();
      end
   end
```

Ejercicio 8. Se trata de un ejercicio de programación distribuida.