Prueba Objetiva Temas I, II y III

Nombre: SOLUCIONES	13-01-2021	GIADE: GIM:

Preguntas de respuesta alternativa: 35%

Seleccionar la única respuesta correcta de cada una de las cuestiones siguientes; cada una bien contestada tiene una puntuación de (**0,5**)[1].

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* $x = y + a \parallel y = x + b$ *coend*
   (b) $P_2 ::$ *cobegin* $x = y + a \parallel y = f(x) + b$ *coend*
   (c) ✓ $P_3 ::$ *cobegin* $x = y + a \parallel y = a + b$ *coend*
   (d) $P_4 ::$ *cobegin* $x = x/y \parallel y = a + x$ *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

[1] si hay duda respecto de la elección en alguna pregunta, se sugiere explicar en un folio aparte la razón de la elección realizada (indicando el número de cuestión a la que se alude)

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%

1. 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 ≡ "Siempre ocurrirá que eventualmente se cumpla muchas veces" (indica una propiedad de "seguridad")

   (d) Alguna vez en el futuro y, desde entonces, para siempre ≡ "Eventualmente ocurrirá que se cumplirá siempre desde entonces" (indica una propiedad de "vivacidad")

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

2. Considerando el siguiente programa concurrente:

```
cobegin S::<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}
```

```
{x== 7 \/ x==3 \/ x==4 \/ x==0}
<x= x + 2>;
{x== 9 \/ x==5 \/ x==6 \/ x==2}
```

```
{x== 6 \/ x==2 \/ x==4 \/ x==0}
<x= x + 3>;
{x== 9 \/ x==5 \/ x==7 \/ x==3}
```

```
{x== 5 \/ x==2 \/ x==3 \/ x==0}
<x= x + 4>;
{x== 9 \/ x==6 \/ x==7 \/ x==4}
```

Aplicando la regla de la *no-interferencia*, la precondición de la sentencia `cobegin` ha de ser equivalente a la conjunción de las precondiciones : {x== 0}

Aplicando la regla de la *no-interferencia*, la poscondición de la sentencia `coend` ha de ser equivalente a la conjunción de las poscondiciones : {x== 9}

3. 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?

   Se puede producir el entrelazamiento de las instrucciones de un procedimiento del monitor entre varios procesos del programa si en los procedimientos del referido monitor se han utilizado variables condición y operaciones de sincronización `c.wait(), c.signal()`.

Resolución de problemas: 25%

(Lo realizarán los **alumnos de *GIADE***)

1. 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.

```
Monitor CuentaAhorrosOrdenLlegadaConPrioridad;
var saldo, contador: integer;
    cola: condition;
```

```
procedure retirar(cantidad: positive
    );
 var ticket: integer;
 begin
   ticket= contador; contador ++ ;
   if (cola.queue()) then
      cola.wait(ticket);
   while cantidad > saldo do begin
     cola.wait(ticket);
   end;
   saldo -= cantidad;
   cola.signal();
end;
```

```
procedure depositar(cantidad:
    positive)
begin
 saldo += cantidad;
 cola.signal();
end
begin
  saldo= CANTIDAD_INICIAL;
  contador= 0;
end;
```

2. 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).

```
process P(i: 1..N);
  var c: char;
  begin
  //Iniciar todos los envios
  for j:=1 to N do
    if i!= j then begin
       c:= ProduceCaracter();
       Send(c, P[j]);
    end;
  //Ahora se hacen todas las recepciones
  for j:= 1 to N-1 do
    select
      for k:= 1 to N when i != k Receive(c, P[k]) do
        print c ;
      enddo
    end select
end;
```

(Lo realizarán los **alumnos de *GIM***)

1. 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.

   Invariante del monitor : $Memoria\_disponible \leq M_0 \wedge peticiones\_pendientes \rightarrow obtener.queue() \wedge Memoria\_disponible \geq 0$

   Condiciones de sincronización:

   - $Memoria\_disponible > 0$ (señalar *obtener*)
   - $\neg\, peticiones\_pendientes$ (señalar *esperar*)

```
Monitor MemoriaFIFO
const M0;
var mem_disponible: integer;
       peticiones_pendientes: boolean;
       esperar, obtener: condicion;
```

```
procedure solicitar;
  begin
     if (peticiones_pendientes or
        obtener.queue()) then
         esperar.wait();
     peticiones_pendientes:= true;
 end;
 procedure adquirir(n: integer);
 begin
    while (mem_disponible < n) do
       obtener.wait();
    mem_disponible -= n;
   peticiones_pendientes:= false;
   if esperar.queue() then esperar.
      signal();
end;
```

```
procedure liberar(n: integer);
begin
   mem_disponible += n;
   if obtener.queue() then
       obtener.signal();
end;
```

```
begin mem_disponible:= M0; peticiones_pendientes:= false; end;
```

```
Monitor MemoriaSJF;
const M0;
var mem_disponible: integer;
       peticiones_pendientes: boolean;
       esperar, obtener: condicion;
```

```
procedure adquirir(n: integer);
 begin
    while (mem_disponible < n) do
       obtener.wait(n);
    mem_disponible -= n;
   if obtener.queue() then
       obtener.signal();
end;
```

```
procedure liberar(n: integer);
begin
   mem_disponible += n;
   if obtener.queue() then
       obtener.signal();
end;
```

```
begin mem_disponible:= M0; peticiones_pendientes:= false; end;
```

2. 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.

```
Process Cajero{
   ENTRY sacar(num_cliente: integer, cantidad: in/out integer);
   ENTRY ingresar(num_cliente: integer, cantidad: integer);
   var efectivo: integer:= 100;
          saldo: array[NUM_CLIENTES]:= {10, ..., 10};
 do{
   select
      when (efectivo > 0) accept sacar(num_cliente, cantidad) do
        if (efectivo >= cantidad and saldo[num_cliente] >= cantidad)
           begin
             efectivo:= efectivo-cantidad;
             saldo[num_cliente]:= saldo[num_cliente] - cantidad;
          end
        else cantidad:= -1;
      when (efectivo > 0) accept ingresar(num_cliente, cantidad) do
            saldo[num_cliente]:= saldo[num_cliente] + cantidad;
      when (efectivo = 0)
            sleep (60*60);
            efectivo:= efectivo + 100;
   end
 }while (true)
}
```