Examen de Programación Concurrente - Clave: b

Septiembre 2008

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería del *Software*

Normas

Este examen es un cuestionario tipo test que consta de **9 preguntas** en **4 páginas**. La puntuación total del examen es de **10 puntos**. La duración total es de **una hora y media**. El examen debe contestarse en las **hojas de respuestas**. No olvidéis rellenar **apellidos, nombre, DNI y clave del examen**.

Sólo hay una respuesta válida por pregunta. Toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada. Toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas ofrecidas en la misma.

La solución al examen se proporcionará antes de la revisión. Las calificaciones se darán a conocer el **9 de septiembre**. La revisión del examen tendrá lugar el **11 de septiembre**.

Cuestionario

(1 punto) 1. Dado el siguiente código:

```
while C loop null; end loop;
```

donde `C` es una expresión booleana que puede contener variables, **se pide** marcar la afirmación correcta:

(a) Podría utilizarse para realizar espera activa como parte de un protocolo de exclusión mutua en un sistema de paso de mensajes.

(b) Podría utilizarse para realizar espera activa como parte de un protocolo de exclusión mutua en un sistema de memoria compartida.

(c) Es equivalente a un bucle infinito que no termina.

(d) Ninguna de las otras afirmaciones es correcta.

(1 punto) 2. Supóngase que se quiere implementar un programa que lea un número natural menor que 1000 y cree exactamente tantas tareas como dicho número indique. **Se pide** marcar la afirmación correcta:

(a) Es posible hacerlo en Ada usando punteros a tareas.

(b) Es posible hacerlo en Ada pero la única forma de hacerlo es creando un array de 1000 tareas.

(c) No es posible realizar tal programa en Ada puesto que la creación de tareas es estática y debe ser decidida en tiempo de compilación.

(d) No es posible realizar tal programa en Ada porque hacen falta punteros a tareas y Ada no tiene.

(1 punto) 3. El siguiente código va a ser invocado simultáneamente por varios procesos:

```
entry MCD (X, Y : in     Positive;
           Res  :    out Positive)
when True is
   A : Natural := X;
   B : Natural := Y;
begin
   while B > 0 loop
      if A > B then
         A := A - B;
      else
         B := B - A;
      end if;
   end loop;
   Res := A;
end MCD;
```

Se pide marcar la afirmación correcta:

(a) Las variables `A` y `B` no son compartidas por los procesos que invocan el código.

(b) Cada iteración del bucle interno va a ser realizada por un proceso distinto.

(c) Es necesario asegurar exclusión mutua en cada acceso a las variables `A` y `B`.

(d) Es necesario asegurar exclusión mutua al ejecutar las asignaciones a las variables `A` y `B`.

(1½ puntos) 4. Supóngase un programa concurrente ejecutando las tareas `A` y `B` mostradas a continuación:

```
task body A is
begin
   while C loop
      Signal (M);
      C := True;
      Wait (M);
   end loop;
end A;
```

```
task body B is
begin
   loop
      Wait (M);
      C := False;
      Signal (M);
   end loop;
end B;
```

La variable `M` es un semáforo binario inicializado a `0`, `C` es una variable booleana inicializada a `True` y ambas variables están compartidas entre las dos tareas. **Se pide** marcar la afirmación correcta:

(a) En algún momento la tarea `A` puede salir del bucle de ejecución, pero no es seguro que esto ocurra.

(b) Tanto la tarea `A` como la tarea `B` ejecutarán los bucles que aparecen en el código sin quedarse nunca bloqueadas y sin que ninguna salga del bucle.

(c) Es completamente seguro que en algún momento la tarea `A` saldrá del bucle de ejecución.

(d) La tarea `A` acabará quedándose bloqueada en algún momento con toda seguridad.

(1 punto) 5. Supóngase que un proceso P ha ejecutado una sentencia **`select`** que tiene únicamente ramas **`accept`** (no tiene ni ramas **`else`** ni **`delay`**). **Se pide** marcar la afirmación correcta:

(a) El proceso puede no haber aceptado ninguna entrada *rendez-vous*.

(b) Debía haber al menos un proceso, diferente de P, que invoca una de las entradas de *rendez-vous*.

(c) El proceso puede haber aceptado simultaneamente dos entradas *rendez-vous*.

(d) Al menos dos guardas de la **`select`** eran ciertas.

(1½ puntos) 6. La instrucción TST (*Test and set*) es típica de algunas arquitecturas. Su comportamiento se basa en la existencia de una variable `C` común a varios procesos. Al ejecutar `TST(L, C)`, donde `L` debería ser una variable local al proceso, se realiza **atómicamente** la siguiente ejecución: `L := C; C := 1`. El siguiente programa concurrente hace uso de dicha instrucción para regular el acceso a una sección crítica:

```
procedure Test_And_Set is
   C : Integer := 0;
   procedure TST (L :    out Integer;
                  C : in out Integer);

   task T1;
   task T2;

   task body T1 is
      L : Integer;
   begin
      loop
         S1;
         loop
            TST (L, C);
            exit when L = 0;
         end loop;
         Sec_Critica;
         C := 0;
      end loop;
   end T1;

   task body T2 is
      L : Integer;
   begin
      loop
         S2;
         loop
            TST (L, C);
            exit when L = 0;
         end loop;
         Sec_Critica;
         C := 0;
      end loop;
   end T2;

begin
   null;
end Test_And_Set;
```

Se pide marcar la afirmación correcta:

(a) Se garantiza la exclusión mutua y la ausencia de inanición sin que haya posibilidad de interbloqueo.

(b) Puede producirse inanición de un proceso.

(c) No se garantiza la propiedad de exclusión mutua.

(d) Se puede producir interbloqueo.

(1 punto) 7. La siguiente es una implementación genérica, utilizando objetos protegidos, de una operación de un recurso compartido cuya condición de sincronización depende de los argumentos de entrada. La función booleana `Condicion` comprueba la condición de sincronización sin alterar el estado del objeto.

```
entry Op (X : in T)
when True is
begin
   if Condicion (X) then
      --  Codigo para alcanzar la postcondicion
      --  de la operacion Op

   else
      requeue Op;
   end if;
```

Se pide marcar la afirmación correcta:

(a) Funcionaría si la condición **when** `True` **is** se cambiase por **when** `Condicion(X)` **is** .

(b) Falla porque permitirá que se ejecute el código correspondiente la operación `Op` en estados en que su precondición no se cumple.

(c) No es correcta porque este código puede bloquear todo el objeto protegido.

(d) Funciona, pero no es óptimo porque se comprueba mediante espera activa si se cumple o no la condición de concurrencia.

(1 punto) 8. Dadas las siguiente tareas (obsérvese que no hay bucles):

```
task body Tarea_1 is
begin
   select
      when True =>
         accept RV_2 do
            Tarea_3.RV;
         end RV_2;
   or when True =>
         accept RV_3 do
            Tarea_2.RV;
         end RV_3;
   end select;
end Tarea_1;
```

```
task body Tarea_2 is
begin
   accept RV do
      null;
   end RV;
   Tarea_1.RV_2;
end Tarea_2;

task body Tarea_3 is
begin
   Tarea_1.RV_3;
   accept RV do
      null;
   end RV;
end Tarea_3;
```

Se pide marcar la afirmación correcta:

(a) Sólo `Tarea_1` termina, las otras dos tareas quedan en interbloqueo.

(b) Sólo `Tarea_2` termina, las otras dos tareas quedan en interbloqueo.

(c) Sólo `Tarea_3` termina, las otras dos tareas quedan en interbloqueo.

(d) Las tres tareas se interbloquearán.

(1 punto) 9. En el código que se muestra a continuación se pretende que la tarea `Cliente` se detenga hasta que se cumpla la condición `Condicion` en la tarea `Servidor`:

```
task body Cliente is
   C : Channel;
begin
   Create (C);
   Servidor.Enviar (C);
   Send (C, True);
   Destroy (C);
end Cliente;
```

```
task body Servidor is
   --  ...

begin
   --  ...
   select
      when Condicion =>
         accept Enviar
           (C : in out Channel) do
            Receive (C, V);
         end Enviar;
   --  ...
   end select;
   --  ...
end Servidor;
```

Se pide marcar la afirmación correcta:

(a) Los procesos se bloquean debido a que el cliente debe hacer un `Receive` y el servidor un `Send`. Es decir, deben intercambiarse las llamadas que hacen ahora.

(b) Se bloquea debido a que el proceso cliente nunca llegará a ejecutar la orden `Send`.

(c) Funciona perfectamente, tal y como se describe en el enunciado.

(d) Se bloquea debido a que el proceso servidor nunca ejecutará la orden `Receive`.