

## Examen de Programación Concurrente

Febrero 2007

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería del Software

### Normas

Este examen es un cuestionario tipo test que consta de **9 preguntas** en **6 páginas**. La puntuación total del examen es de **10 puntos**. La duración total es de **dos horas**. El examen debe contestarse en las **mismas hojas**. No olvidar rellenar **apellidos, nombre y número de matrícula** en cada hoja.

**Sólo hay una respuesta válida por pregunta excepto para la pregunta 9.** Para el resto, 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.

Las soluciones se proporcionarán antes de la revisión. Las calificaciones se darán a conocer el **19 de febrero**. La revisión del examen tendrá lugar el **21 de febrero**.

### Cuestionario

- (1 puntos) 1. Supóngase en un programa concurrente un proceso se queda ejecutando indefinidamente dentro del cuerpo de una entrada (*entry*) de un objeto protegido. **Se pide** señalar la respuesta correcta:
- (a) ☐ El resto de procesos podrían acceder al recurso y ejecutar otras entradas.
  - (b) ☐ La suposición es falsa: la exclusión mutua garantiza que el proceso nunca pueda quedar bloqueado.
  - (c) ☐ La exclusión mutua garantiza que cualquier otro proceso quedará bloqueado al ejecutar cualquier otra operación del objeto protegido.
  - (d) ☐ La exclusión mutua garantiza que cualquier otro proceso que ejecute una función (*function*) del objeto protegido no quedará bloqueado al acceder en modo lectura.
- (1 puntos) 2. Supóngase que una condición de sincronización (*CPRE*) de un recurso compartido depende del propio recurso y de un parámetro de entrada (*x*) que puede tomar un conjunto de valores discreto. Supóngase que dicho recurso va a ser implementado como un objeto protegido. **Se pide** señalar la respuesta correcta según la metodología utilizada en la asignatura:
- (a) ☐ Bastará con crear una entrada (*entry*) para dicha operación que compruebe directamente la condición de sincronización en la guarda.
  - (b) ☐ Siempre es viable crear una familia de entradas indexada con el tipo del parámetro *x* y reencolar (*requeue*) directamente desde la entrada (*entry*) que representa la operación.
  - (c) ☐ Siempre es necesario crear una familia de entradas indexada por el número máximo de procesos del sistema y reencolar desde la entrada (*entry*) que representa la operación.
  - (d) ☐ Ninguna de las otras respuestas es correcta.
- (1 puntos) 3. ¿Es posible tener parámetros de salida en las cláusulas **accept** de *rendez-vous*?
- (a) ☐ Sí.
  - (b) ☐ No.

- (1 puntos) 4. El siguiente código muestra el esquema de una tarea servidora y ha sido generado siguiendo, presuntamente, la metodología de la asignatura para *rendez-vous* y paso de mensajes síncrono. Se muestra el detalle de la única operación del recurso cuya condición de sincronización depende de un parámetro.

```

...
BloqOp : Cola_De_TX.Cola;
NumBloqOp : Natural := 0;
NumProcesados : Natural;
Pend : DatoPendiente; -- Registro con un TX y un Channel
begin
...
Cola_De_TX.Crear(BloqOp);
loop
  select
    ...
  or
    when True => -- CPRE: P(X)
      accept Op (X : in TX;
                 C : in out Channel) do
        Insertar (BloqOp, (X, C));
        NumBloqOp := NumBloqOp + 1;
      end Op;
  or
    ...
end select;

-- Bucle de desbloqueo
NumProcesados := NumBloqOp;
for I in 1 .. NumProcesados loop
  Primero(BloqOp, Pend);
  Borrar(BloqOp);
  if P(Pend.X) then
    «Sentencias para conseguir la postcondición»;
    Send(Pend.C, True);
    NumBloqOp := NumBloqOp - 1;
  else
    Insertar(BloqOp, Pend);
  end if;
end loop;
end loop;

```

Se pide señalar la respuesta correcta:

- (a) ☐ La declaración del parámetro C de tipo Channel no compilará porque **accept** no admite parámetros de entrada/salida.
- (b) ☐ La aplicación de la metodología es perfecta y no hay ningún problema.
- (c) ☐ El bucle de desbloqueo puede ser incorrecto desde el momento en el que podrían quedar peticiones de ejecución de la operación sin atender a pesar de que su condición de sincronización fuera cierta.
- (d) ☐ Es necesario utilizar una cola auxiliar en la que ir volcando los datos de los procesos cuyos datos no cumplen la condición de sincronización.

- (1 puntos) 5. Ada 95 prohíbe el uso de parámetros de las entradas (*entries*) de los objetos protegidos en las guardas. **Se pide** señalar la respuesta correcta:

- (a) ☐ Se prohíbe porque nadie ha encontrado una forma de generar código que automáticamente pudiera comprobar dichas condiciones.
- (b) ☐ Se prohíbe por una cuestión de eficiencia.
- (c) ☐ Se prohíbe para que la ejecución de otra operación del recurso por parte de otro proceso no viole la propiedad de exclusión mutua.
- (d) ☐ Se prohíbe porque el objeto protegido no puede conocer el valor de dicho parámetro antes de la ejecución del cuerpo la entrada (*entry*).

- (1 puntos) 6. El siguiente código muestra el esquema de una tarea servidora y ha sido generado siguiendo, presuntamente, la metodología de la asignatura para *rendez-vous* y paso de mensajes síncrono. Se muestra el detalle de una operación del recurso con un parámetro de salida que se ha decidido implementar mediante la comunicación del mismo través de un canal:

```
...
D : ...;
CResp : ...;
...
begin
...
loop
  select
    ...
  or
    when True => -- CPRE: Cierto
      accept Op (C : in out Channel) do
        «Sentencias para conseguir la postcondición»;
        D := «Cómputo del dato de salida»;
        CResp := C;
      end Entrar;
      Send(CResp,D);
    or
      ...
  end select;
  ...
end loop;
```

**Se pide** señalar la respuesta correcta:

- (a) ☐ El código provoca un interbloqueo entre la tarea servidora mostrada y cualquier cliente que invoque la entrada (*entry*) *Op*.
- (b) ☐ La aplicación de la metodología es perfecta y no hay ningún problema.
- (c) ☐ Es obligado utilizar un parámetro de salida en *Op* para el argumento de salida de la operación del recurso.
- (d) ☐ La comunicación del dato *D* hay que hacerla forzosamente dentro de la **select**.

- (1 puntos) 7. Tenemos el siguiente código que será ejecutado por dos tareas concurrentes. Supondremos que las asignaciones son atómicas y que  $X$  e  $Y$  son variables compartidas entre dichos procesos:

```

procedure Swap is
  Z : ...;
begin
  Z := X;
  X := Y;
  Y := Z;
end Swap;

```

Imaginemos que en el momento en que las dos tareas llaman a `Swap` los valores de las variables compartidas son  $X = 1$ ,  $Y = 2$ . **Se pide:** señalar qué posibles valores pueden tener  $X$  e  $Y$  al final de la ejecución de `Swap`

- (a) ☐  $X=1$ ,  $Y=2$  o  $X=2$ ,  $Y=1$  o  $X=1$ ,  $Y=1$  o  $X=2$ ,  $Y=2$   
 (b) ☐  $X=1$ ,  $Y=2$  o  $X=2$ ,  $Y=1$  (y ningún otro)  
 (c) ☐  $X=1$ ,  $Y=1$  o  $X=2$ ,  $Y=2$  (y ningún otro)  
 (d) ☐  $X=2$ ,  $Y=1$  o  $X=1$ ,  $Y=1$  o  $X=2$ ,  $Y=2$  (y ningún otro)

- (1 puntos) 8. Pretendemos usar semáforos para que el procedimiento que aparece en la pregunta anterior se comporte como si `Swap` fuese atómico, de manera que la única respuesta posible a la pregunta anterior fuese  $X=1$ ,  $Y=2$ . Pero queremos hacerlo encerrando entre secciones críticas la menor cantidad de código posible. **Se pide:** marcar, de entre las siguientes posibilidades, cuál es la que consigue el efecto deseado teniendo la menor cantidad de código posible enmarcado dentro de una sección crítica:

(a) ☐

```

begin
  Wait (Mutex);
  Z := X;
  X := Y;
  Signal (Mutex);
  Wait (Mutex);
  Y := Z;
  Signal (Mutex);
end Swap;

```

(b) ☐

```

begin
  Z := X;
  Wait (Mutex);
  X := Y;
  Y := Z;
  Signal (Mutex);
end Swap;

```

(c) ☐

```

begin
  Wait (Mutex);
  Z := X;
  X := Y;
  Signal (Mutex);
  Y := Z;
end Swap;

```

(d) ☐

```

begin
  Wait (Mutex);
  Z := X;
  X := Y;
  Y := Z;
  Signal (Mutex);
end Swap;

```

- (2 puntos) 9. Se quiere programar una *Barrera*: un tipo de datos con una operación *Esperar* que sirva para sincronizar  $n$  tareas (supondremos fijo este número). Las tareas que llaman a *B.Esperar* (siendo *B : Barrera* una variable compartida) suspenden hasta que la tarea  $n$ -ésima realiza esa misma llamada. En ese momento todas despiertan y pueden continuar la ejecución. Una serie posterior de llamadas a *B.Esperar* repetiría ese esquema de ejecución. Asumiremos que ninguna de las  $n$  tareas que se bloquean (y posteriormente desbloquean) en *B.Esperar* llaman a *B.Esperar* antes de que las  $n$  tareas se hayan desbloqueado.

Supongamos la siguiente implementación de una barrera, basado en objetos protegidos:

```
protected type Barrera is
  entry Esperar;
private
  Por_Llegar : Natural := N_Procs;
end Barrera;

protected body Barrera is
  entry Esperar
    when True is
  begin
    Por_Llegar := Por_Llegar - 1;
    requeue Esperar_Apl;
  end Esperar;
end Barrera;

  entry Esperar_Apl
    when Por_Llegar = 0 is
  begin
    null;
  end Esperar_Apl;
```

El problema que aparece con esta implementación es que tras la primera suspensión y re arranque de las  $n$  tareas, la siguiente invocación a *B.Esperar* seguramente causará un error de ejecución. **Se pide:** reprogramar, en el espacio que sigue, el código anterior para conseguir el comportamiento deseado.

*Probablemente la solución más sencilla pasa por tratar el problema directamente en la entrada Esperar:*

```
entry Esperar
  when True is
begin
  if Por_Llegar = 0 then
    Por_Llegar := N_Procs;
  end if;
  Por_Llegar := Por_Llegar - 1;
  requeue Esperar_Apl;
end Esperar;
```

*La solución que se muestra a continuación es **incorrecta** puesto que sólo un proceso es desbloqueado en la primera tanda y todos quedan bloqueados para siempre en la segunda:*

```
entry Esperar_Apl
  when Por_Llegar = 0 is
begin
  Por_Llegar := N_Procs;
end Esperar_Apl;
```

***NO USAR ESTE ESPACIO***