Examen Sistemas Concurrentes y Distribuidos
21 de Mayo de 2015

Apellidos, Nombre:
Grupo (M/T):

1) A continuación tienes 16 preguntas con cuatro posibles respuestas cada una. Por cada pregunta sólo una de las cuatro respuestas es correcta. Cada pregunta acertada vale 0,25 puntos, cada respuesta errada descuenta 0,06 puntos y las preguntas no contestadas no suman ni restan. Se ruega que rellene la siguiente tabla con sus respuestas, sólo se corregirán las respuestas que estén indicadas en dicha tabla. (4 puntos)

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**

1. La relación existente entre procesos e hilos es:
   a) Los recursos podrán ser asociados tanto a los procesos como a los hilos
   b) Los procesos son estructuras *ligeras* mientras que los hilos son estructuras *pesadas*
   c) El Sistema Operativo debe manejar la misma información que para el mantenimiento de los procesos
   d) **Los hilos están asociados al proceso que los crea**

2. La posibilidad que nos permite un sistema multihilo es:
   a) No ofrece ninguna ventaja sobre un sistema multiproceso
   b) Son un elemento presente en todos los Sistemas Operativos
   c) **Permite una mejor paralelización de un problema sin necesidad de crear nuevos procesos**
   d) Ninguna de las respuestas es correcta

3. Para poder seguir la ejecución de un hilo será necesario almacenar:
   a) **Al menos la información de contexto y pila**
   b) Una cantidad de información similar a la necesaria para gestionar un proceso
   c) La información de contexto, pila y recursos asignados
   d) Ninguna de las respuestas es correcta

4. El algoritmo de Peterson frente al de Dekker:
   a) No tiene el problema de espera ocupada que sí tiene el de Dekker
   b) Es más eficiente que el algoritmo de Dekker
   c) Tiene una mejor solución para el problema de sincronización entre procesos
   d) **Ninguna de las respuestas es correcta**

5. En términos de eficiencia:
   a) Los algoritmos de espera ocupada son más eficientes que los semáforos
   b) La eficiencia de los semáforos depende exclusivamente de la CPU
   c) Los monitores son más eficientes que los semáforos
   d) **A priori, no puede determinarse qué técnica de sincronización es la más eficiente**

6. ¿Qué son las condiciones de Bernsein?
   a) Indican si dos o más procesos pueden ejecutarse concurrentemente
   b) **Determinan si un conjunto de instrucciones pueden ejecutarse concurrentemente**
   c) Sirven para determinar las secciones críticas de los procesos
   d) Ayudan a la sincronización de los procesos

7. La característica principal de un monitor es:
   a) Solucionan el problema de la sincronización entre procesos concurrentes
   b) Ninguna de las respuestas es correcta
   c) Sólo hay un proceso en el monitor en cada momento
   d) **Todas las funciones se ejecutan en exclusión mutua**

8. La gestión de los procesos bloqueados en un semáforo:
   a) **Debe ser siempre FIFO para evitar la inanición**
   b) Mediante el uso de semáforos, los procesos no pasan a estado bloqueado
   c) El Sistema Operativo desbloqueará los procesos en función de la prioridad
   d) Puede ser FIFO o LIFO

9. En el problema del productor/consumidor resuelto mediante semáforos:
   a) Ninguna de las respuestas es correcta
   b) Sólo es necesario garantizar la exclusión mutua al buffer compartido
   c) Los procesos productores deben sincronizarse entre sí para garantizar la corrección del problema
   d) **Los procesos productores deben sincronizarse con los procesos consumidores para garantizar la corrección del problema**

10. El problema del interbloqueo:
    a) No es un problema que se da en la programación concurrente
    b) Se resuelve mediante el uso de monitores
    c) **Ninguna de las respuestas es correcta**
    d) Se resuelve mediante el uso de semáforos

11. Los monitores requieren de la utilización y definición de dos tipos de procesos:
    a) Proceso monitor y proceso principal
    b) Procesos padres y procesos hijo
    c) **Procesos activos y procesos bloqueados**
    d) Procesos bloqueados y procesos bloqueantes

12. En la instrucción de espera selectiva `select`, el proceso que la ejecuta se bloquea si:
    a) La instrucción `select` no genera bloqueo del proceso
    b) **No existe ningún mensaje en los buzones/canales que se manejan**
    c) No disponga de alternativa `else`
    d) No se cumple ninguna de las guardas, si las tuviera

13. En la llamada a procedimiento remoto:
    a) Los dos sistemas deberán tener una misma arquitectura
    b) Se utilizará el mismo lenguaje de programación para codificar los procesos
    c) **Ninguna de las respuestas es correcta**
    d) Deberá ser el mismo Sistema Operativo en las máquinas remotas

14. En la comunicación síncrona entre procesos:
    a) El emisor espera siempre al receptor antes de iniciar la transmisión
    b) **El primero que alcanza la primitiva de comunicación deberá esperar hasta que el otro alcance la suya antes de iniciar la transmisión**
    c) El receptor espera siempre al emisor antes de iniciar la transmisión
    d) Ni emisor ni receptor esperan antes de iniciar la transmisión

15. La llamada a un procedimiento remoto:
    a) **Permite la ejecución de un procedimiento presente en un proceso remoto dentro de un Sistema Distribuido**
    b) Es un tipo de comunicación habitual en Sistemas Distribuidos
    c) Es un elemento necesario en la estructura de los Sistemas Distribuidos
    d) Ninguna de las respuestas es correcta

16. En la comunicación asíncrona entre procesos:
    a) No se requiere ningún tipo de identificación
    b) El *buffer* sólo se comparte entre emisor y receptor
    c) No hay necesidad de buffer en la transmisión
    d) **Ninguna de las respuestas es correcta**

1) (1 punto) ¿Qué se entiende por un programa concurrente correcto?

2) (1 punto) ¿Cuál sería el principal inconveniente de disponer de un esquema de comunicación mediante paso de mensajes y no disponer de la sentencia de espera selectiva?

3) (1 punto) Supongamos que tenemos un Buffer representado por un array de N enteros que es compartido por dos procesos concurrentes. El primer proceso realizará la siguiente acción de forma repetida: Generará entre 1 y 4 números aleatorios y los almacenará en el Buffer. El segundo proceso realizará la siguiente acción de forma repetida: Recogerá los números que ha generado el primer proceso para cada acción. Resolver el problema mediante el uso de semáforos. Razonar la solución aportada.

4) (1,5 puntos) Resolver el problema de la exclusión mutua entre N procesos suponiendo que imponemos las siguientes restricciones: la comunicación a través de un buzón es unidireccional y sólo se permiten establecer relaciones de comunicación de tipo muchos a uno. Razonar la solución aportada.

5) (1,5 puntos) Supongamos un aeropuerto donde hay tres pistas. Una de las pistas se dedica exclusivamente a aterrizajes, otra pista para despegues y la tercera se puede utilizar para el despegue y el aterrizaje con prioridad al aterrizaje. Diseñar un monitor que permita a la torre de control del aeropuerto poder gestionar las pistas. Razonar la solución aportada.

Soluciones de los proglemas

Primer problema

Es una modificación del problema clásico del productor consumidor. Para ello, almacenamos en el Buffer todos los números que genere el productor en una pasada y añadimos al Buffer el número de elementos que se generaron en esa pasada. Por tanto, si x es el número de elementos que se generaron en una pasada, debemos asegurarnos que hay *x+1* huecos en el Buffer antes de insertar. El consumidor, primero deberá obtener x para saber los elementos que debe retirar del Buffer y liberar *x+1* elementos del Buffer al finalizar su ejecución.

```
Variables compartidas
  buffer : Array [1..N] de enteros;
  frente, cola : enteros; índices del buffer
  mutex : semaáforo inicializado a 1; exclusión mutua
  lleno: semáforo incializado a 0; para los elementos del buffer
  vacio: semáforo inicializado a N; para los huecos del buffer

Proceso Productor
Begin
   Repetir
      numElmRonda = Random(4); Genera un número entre 1 y 4
      producir( elmentos[], numElmRonda);

      Para i = 1 hasta numElmRonda + 1
         wait(vacio); Nos aseguramos que hay huecos en el Buffer
      Fin_Para

      wait(mutex)
      Para i = 1 hasta numElmRonda
         buffer[frente] = elmentos[i];
         frente = (frente + 1) MOD N;
      Fin_Para
      buffer[frente] = numElmRonda;
      frente = (frente + 1) MOD N;
      signal(mutex)

      Para i = 1 hasta numElmRonda + 1
        signal(lleno); Indicamos los elementos producidos
      Fin_Para
   Para siempre
End
```

```
Proceso Consumidor
Begin
   Repetir
      wait(lleno)
      wait(mutex)
      numElmRonda = buffer[cola]; Número de elementos a consumir
      cola = (cola + 1) MOD N;
      signal(mutex)

      Para i = 1 hasta numElmRonda
         wait(lleno); Elementos que hay que consumir
      Fin_Para

      wait(mutex)
      Para i = 1 hasta numElmRonda
         elmentos[i] = buffer[cola];
         cola = (cola + 1) MOD N;
      Fin_Para
      signal(mutex)

      Para i = 1 hasta numElmRonda + 1
        signal(vacio); Generamos los huecos en el buffer
      Fin_Para

      consumir( elementos[], numElmRonda);
   Para siempre
End
```

Segundo Problema

Para la solución se necesitará un proceso controlador que será el encargado de conceder los permisos de entrada en las secciones críticas a los diferentes procesos. El proceso controlador atenderá una solicitud, enviará una confirmación para que ese proceso pueda entrar en la sección crítica. Posteriormente el controlador tendrá que quedar a la espera de que el proceso termine su sección crítica para que pueda atender otra petición.

```
Buzones
  entradaSC : buzón para indicar su intención de entrar en la SC
  salidaSC: buzón para indicar la finalización de la SC
  pasoSC: buzón individual para cada proceso para entrar en la SC

Proceso P(id)
Begin
  ......
  mensaje.id = id;
  mensage.cuerpo = generarMensaje();

  Enviar(entradaSC, mensaje);
  Recibir(pasoSC[id], mensaje);

  // SECCIÓN CRÍTICA

  Enviar(salidaSC, mensaje);
  ......
End

Proceso Controlador
Begin
  Repetir
     Recibir(entradaSC, mensaje);
     Enviar(pasoSC[mensaje.id], mensaje);
     Recibir(salidaSC, mensaje);
     //Ya se puede atender a otra solicitud de otro proceso
  Para siempre
End
```

Tercer Problema

Para el diseño del monitor se necesitarán funciones para:

- Permiso de aterrizaje, devuelve la pista
- Finalización del aterrizaje, se pasa la pista
- Permiso para el despegue, devuelve la pista
- Finalización de despegue, se pasa la pista

Además las pistas las representaremos en un array donde se indicará si está ocupada o libre.

- Pista 1, aterrizaje
- Pista 2, despegue
- Pista 3, aterrizaje/despegue

```
Monitor ControlAeropuerto
Variables
  pistas: Array [1..3] para indicar si está ocupada o libre;
  peticionAterrizaje : variable condición para solicitudes pendientes
  peticionDespegue: variable condición para solicitudes pendientes

Interfaz
  solicitudAterrizaje() : idPista;
  finAterrizaje(idPista);
  solicitudDespegue(): idPista;
  finDespegue(idPista);

Función solicitudAterrizaje() : idPista
Begin
  Si (pistas[1] == ocupada) Y (pistas[3] == ocupada)
    delay(peticionAterrizaje);
  Fin_Si

  Si pistas[1] == libre
    pistas[1] = ocupada;
    return 1;
  Si no
    pista[3] == ocupada;
    return 3;
  Fin_Si
End
```

```
Procedimiento finAterrizaje( idPista )
Begin
  pistas[idPista] == libre;
  Si (idPista == 3) Y vacia(peticionAterrizaje)
    resume(peticionDespegue)
  Si no
    resume(peticionAterrizaje)
  Fin_Si
End

Función solicitudDespegue() : idPista
Begin
  Si pistas[2] == ocupada
    delay(peticionDespegue)
  Si no Si (pistas[3] == ocupada) O ((pista[3] == libre) Y (NO vacia(peticionAterrizaje)))
    delay(peticionDespegue)
  Fin_Si

  Si pistas[2] == libre
    pistas[2] = ocupada;
    return 2;
  Si no
    pista[3] == ocupada;
    return 3;
  Fin_Si
End

Procedimiento finDespegue( idPista )
Begin
  pistas[idPista] == libre;
  Si (idPista == 3) Y (NO vacia(peticionAterrizaje))
    resume(peticionAterrizaje)
  Si no
    resume(peticionDespegue)
  Fin_Si
End
```