Examen Sistemas Concurrentes y Distribuidos
3 de Junio de 2013

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. Cada respuesta errada descuenta un tercio del valor de una pregunta acertada. 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. ¿Cuándo hablamos que dos o más procesos son concurrentes?
   a) Cuando tenemos al menos tantas unidades de procesamiento como procesos.
   b) **Es suficiente si las instrucciones de los procesos se intercalan en la ejecución.**
   c) Cuando se ejecutan en ordenadores diferentes.
   d) Sólo en el caso de ejecución paralela.

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

3. En los programas concurrentes:
   a) Podemos determinar de forma clara el orden de ejecución de las diferentes instrucciones que lo componen.
   b) El tiempo empleado para terminar la ejecución siempre es la misma
   c) **Se pueden producir resultados diferentes para el mismo conjunto de datos de entrada.**
   d) Ninguna de las anteriores es correcta.

4. La ejecución concurrente de varios procesos implica:
   a) La necesidad de múltiples unidades de procesamiento.
   b) Que existan múltiples programas dentro del sistema.
   c) **Una arquitectura del Sistema Operativo que la permita.**
   d) Un sistema Operativo Monoprogramado.

5. La exclusión mutua entre diferentes procesos garantiza:
   a) **El acceso seguro a la información compartida entre procesos.**
   b) No es necesario garantizar la exclusión mutua entre procesos.
   c) Sólo es necesaria en Sistemas Distribuidos.
   d) El acceso seguro a los recursos compartidos.

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

7. Los semáforos:
   a) Están presentes en todas la herramientas de programación.
   b) Las herramientas de programación garantizan su uso correcto para solucionar el problema de la exclusión mutua
   c) Las herramientas de programación garantizan su uso correcto para solucinar el problema de la sincronización entre procesos
   d) **Son herramientas de programación para el uso de los programadores en los problemas de concurrencia.**

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

9. El problema del interbloqueo:
   a) Se resuelve mediante el uso de semáforos.
   b) Se resuelve mediante el uso de monitores.
   c) No es un problema que se de en la programación concurrente
   d) **Todas las anteriores son falsas.**

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

11. En los sistemas distribuidos debemos:
    a) Debemos garantizar la exclusión mutua de las secciones críticas.
    b) **Debemos garantizar la correcta sincronización de los procesos.**
    c) Debemos garantizar el acceso de los procesos a los recursos locales.
    d) Todas las respuestas son correctas.

12. Las variables de condición de los monitores:
    a) **Garantizan la sincronización de los procesos.**
    b) Garantizan la exclusión mutua de los procesos.
    c) No son variables propias de los monitores.
    d) Garantizan tanto la sincronización como la exclusión mutua de los procesos.

13. La sentencia *resume* de un monitor:
    a) Tiene la misma lógica de funcionamiento que la operación *signal* de un semáforo.
    b) Permite bloquear a un proceso en el monitor dentro de una variable de condición.
    c) Sólo se aplica a una variable de condición del monitor si hay procesos bloqueados en la misma.
    d) **Liberará a un proceso bloqueado en una variable de condición del monitor.**

14. En la comunicación directa entre procesos es necesario:
    a) Conocer el destinatario del mensaje.
    b) Conocer el remitente del mensaje.
    c) No se requiere ningún tipo de identificación.
    d) **El emisor debe conocer al destinatario y el receptor al remitente**.

15. En el problema del productor/consumidor si la primitiva de envío no bloquea al productor:
    a) El emisor deberá asegurarse que el consumidor esté disponible.
    b) **Deberemos utilizar un buzón de tamaño indefinido**.
    c) No hay solución posible con esa suposición de partida.
    d) Ninguna de las anteriores es correcta.

16. La utilización de un canal:
    a) **Establecerá el tipo de información que se transmitirán emisor y receptor en una comunicación síncrona**.
    b) Establecerá el tipo de sincronización necesaria en la comunicación.
    c) Permitirá el almacenamiento de información para la comunicación entre procesos.
    d) Ninguna de las anteriores es correcta.

2) (1 punto) ¿Cuál es la ventaja de la concurrencia en los sistemas monoprocesador?

3) (1 punto) Indicar las condiciones exigidas a una correcta solución al problema de la exclusión mutua.

Todos los problemas deberán justificarse adecuadamente para que sean corregidos. No se aceptarán soluciones que carezcan de una justificación.

4) (1 punto) Diseñar un proceso controlador que provoque que los dos primeros procesos que lo invoquen sean suspendidos y el tercero los despieste, y así cíclicamente. Resolver el problema mediante paso de mensajes síncronos con canales.

Solución:

Utilizaremos dos buzones de sincronización:

- pidoPermiso
- permisoConcedido

La solución para los diferentes procesos vendrá parametrizada por un valor que nos lo identificará.

```
Proceso Proc(i)

Repetir
   ...
   pidoPermiso[i] ! any
   permisoConcedido[i] ? any
   ...
ParaSiempre

Proceso Controlador

variables
   identificadores : array [1..3] de enteros;
   // almacena el identificador del proceso bloqueado
   cont1,cont2 : entero

cont1=0
Repetir
   cont1++
   Select
      Para cont2 = 1 hasta NUMEROPROCESOS replicate
         pidoPermiso[cont2] ? any
         identificadores[cont1] =cont2
   or
      terminate
   finSelect

   Si cont1 == 3 entonces
       Para cont2 = 1 hasta 3 Hacer
          permisoConcedido[identificadores[cont2]] ! any
       FinPara
       cont1 = 0
    FinSi
```

```
ParaSiempre
```

5) (1,5 punto) Tenemos un aparcamiento donde hay una capacidad para 500 coches. El acceso al aparcamiento es mediante una rampa donde sólo puede pasar un único coche para entrar o para salir del aparcamiento. Mientras haya coches en un sentido los coches del sentido contrario tendrán que esperar:

- Resolver el problema mediante la utilización de un monitor.
- Basándonos en el problema anterior dar una solución a los problemas de inanición que puedan presentarse.

Solución:

Diseñaremos el monitor sin dar preferencia a ninguno de los tipo de coches. Primero se mostrará la solución que no resuelve el problema de inanición.

El monitor se diseñará con dos procedimientos para cada coche, un primer procedimiento que comprobará que el coche pueda acceder a la rampa y un segundo procedimiento que complete el paso de la rampa por parte del coche.

```
Monitor Aparcamiento

export accesoRampaBajada, accesoRampaSubida,finRampaBajada, finRampaSubida;

Variables
    capacidadAparcamiento : Entero;
    nCochesSubida, nCochesBajada: Entero; // número de coches de un tipo
    cochesSubida, cochesBajada : Variable condicion;

procedimiento accesoRampaBajada()

    Si ((nCochesSubida > 0) O (capacidadAparcamiento = 0))
        delay(cochesBajada); // Rampa ocupada
    FinSi

    nCochesSubida++;
    capacidadAparcamiento--;

    //Liberamos más coches si hay capacidad para ello
    Si (capacidadAparcamiento > 0)
        resume(cochesBajada)
    FinSi

procedimiento accesoRampaSubida()

    Si (nCochesBajada > 0)
        delay(cochesSubida); // Rampa ocupada
    FinSi

    nCochesSubida++;
    capacidadAparcamiento++;

    // Si hay coches de subida esperando los liberamos
    resume(cochesBajada)

proceso finRampaBajada()

    nCochesBajando--; // Y están en el aparcamiento
    Si (nCochesBajando = 0)
    // Liberamos la rampa
        resume(cochesSubiendo)
    FinSi

proceso finRampaSubida()

    nCochesSubiendo--; // Y están en el aparcamiento
    Si (nCochesSubiendo = 0)
    // Liberamos la rampa
        resume(cochesBajando)
    FinSi

Inicializacion Monitor
    nCochesSubiendo = nCochesBajando = 0;
    capacidadAparcamiento = 500;
Fin Inicializacion Monitor

Proceso CocheBajada

    // Inciciamos el acceso al aparcamiento
    Aparcamiento.accesoRampaBajada()

    // El coche se encuentra en el aparcamiento
    Aparcamiento.finRampaBajada()

    aparcarCoche()

Proceso CocheSubida
```

```
    // Inciciamos el acceso al aparcamiento
    Aparcamiento.accesoRampaSubida()

    // El coche se encuentra en el aparcamiento
    Aparcamiento.finRampaSubida()

    abandonarAparcamiento()
```

Sólo será necesario modificar el monitor para evitar la inanición de los diferentes procesos. Para ello definiremos un máximo de coches que podrán acceder al aparcamiento o salir del mismo. En el caso del acceso también se tendrá presente que haya capacidad en el aparcamiento.

```
Monitor Aparcamiento

export accesoRampaBajada, accesoRampaSubida,finRampaBajada, finRampaSubida;

Variables
    capacidadAparcamiento : Entero;
    nCochesSubida, nCochesBajada: Entero; // número de coches de un tipo
    maxSubida, maxBajada : Entero; // Limitará el número de coches en un sentido
    MAX = 10; // Máximo de coches seguidos en un sentido en la rampa;
    cochesSubida, cochesBajada : Variable condicion;

procedimiento accesoRampaBajada()

    Si ((nCochesSubida > 0) O (capacidadAparcamiento = 0) O
        (maxBajada = MAX))
        delay(cochesBajada); // Rampa ocupada
    FinSi

    nCochesSubida++;
    maxBajada++;
    capacidadAparcamiento--;

    // Liberamos más coches si hay capacidad para ello
    // y no superamos el límite
    Si ((capacidadAparcamiento > 0) O (maxBajada < MAX))
        resume(cochesBajada)
    FinSi

procedimiento accesoRampaSubida()

    Si ((nCochesBajada > 0) O (maxSubida = MAX))
        delay(cochesSubida); // Rampa ocupada
    FinSi

    nCochesSubida++;
    maxSubida++; // Un coche más subiendo consecutivamente
    capacidadAparcamiento++;

    // Si hay coches de subida esperando los liberamos
    // Si no alcanzamos el máximo
    Si (maxSubida < MAX)
        resume(cochesBajada)
    FinSi

proceso finRampaBajada()

    nCochesBajando--; // Y están en el aparcamiento
    Si (nCochesBajando = 0)
    // Liberamos la rampa e inicializamos
    // el tope de coches subiendo
        maxSubida = 0;
        resume(cochesSubiendo);
    FinSi

proceso finRampaSubida()

    nCochesSubiendo--; // Y están en el aparcamiento
    Si (nCochesSubiendo = 0)
    // Liberamos la rampa e incializamos
    // el tope de coches bajando
        maxBajada = 0;
        resume(cochesBajando)
    FinSi

Inicializacion Monitor
    nCochesSubiendo = nCochesBajando = 0;
    maxSubida = maxBajada = 0;
    capacidadAparcamiento = 500;
Fin Inicializacion Monitor
```

6) (1,5 puntos) En un aeropuerto existen dos pistas de aterrizaje/despegue y una torre de control. Los aviones que tienen este aeropuerto como base realizan vuelos de reconocimiento. Programe el proceso avión y el proceso torre de tal manera que se eviten las

colisiones en las pistas (una misma pista no puede ser utilizada simultáneamente por más de un avión). Resolver el problema mediante el paso de mensajes asíncronos.

Solución:

El tipo de procesos que tendremos serán los siguientes:

1. Aviones que aterrizan.
2. Aviones que despegan.
3. Controlador para distribuir las pistas entre los aviones.

Vamos a necesitar los siguientes buzones para resolver el problema:

- **solicitarPista**. En este buzón los aviones enviarán su identificativo para que el controlador les asigne una de las dos pistas si están disponibles.
- **pistaAsignada**. Tendremos un array de este tipo de buzones, uno para cada uno de los aviones que están solicitando pista. El controlador le enviará al avión el número de pista donde podrá operar.
- **finUsoPista**. Una vez que el avión haya terminado con el uso de la pista se lo comunicará al controlador para que pueda asignarla a otro avión que la solicite.

Los procesos avión estarán parametrizados por su identificador.

```
Proceso AvionDespegando(id)

Variables
    numPista : Entero; //Número de pista asignada

    enviar(solicitarPista, id);
    recibir(pistaAsignada[id], numPista);
    .....
    completarDespegue();
    ....
    enviar(finUsoPista, numPista);

Proceso AvionAterrizando(id)

Variables
    numPista : Entero; //Número de pista asignada

    enviar(solicitarPista, id);
    recibir(pistaAsignada[id], numPista);
    .....
    completarAterrizaje();
    ....
    enviar(finUsoPista, numPista);

Proceso Controlador

Variables
   pista : array [1..2] de Boolean;
   \\ Indicará si está disponible, Verdadero estará disponible
   avionPista : array [1..2] de Entero;
     \\ Avión asignado a una pista
   nPista : Entero; \\Pista liberada

   Repetir
       Select
           when (pista[1] O pista[2]) \\ Hay pistas disponibles
              Si (pista[1])
                  recibir(solicitarPista, avionPista[1]);
                  pista[1] = Falso;
                  enviar(pistaAsignada[avionPista[1]], 1);
              En otro caso
                  recibir(solicitarPista, avionPista[2]);
                  pista[2] = Falso;
                  enviar(pistaAsignada[avionPista[2]], 2);
              FinSi
        O
           recibir(finUsoPista, nPista);
           pista[nPista] = Verdadero;
        FinSelect
   ParaSiempre
```