Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad de Jaén

Convocatoria ordinaria II, Curso 2015-16
Sistemas Concurrentes y Distribuidos

Nombre: Curso:

Marcar en la siguiente tabla la respuesta correcta. (**4 puntos**)

+0.2 respuesta correcta,
-0.066 por error, (Cada tres errores se descuenta un acierto)
0 repuesta no contestada.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
D	A	D	C	D	D	D	D	B	D	X	B	B	D	B	D	D	B	X	D

1) ¿Cuál de las siguientes afirmaciones es cierta?
   a) La concurrencia es un tipo de paralelismo
   b) El paralelismo y la concurrencia son conceptos que no guardan relación alguna
   c) El paralelismo puede desarrollarse en sistemas monoprocesador
   d) El paralelismo es un tipo de concurrencia

2) Un interbloqueo (deadlock) se produce:
   a) cuando todos los procesos están esperando que ocurra un evento que nunca se producirá
   b) si el resultado de la secuencia depende de la llegada relativa a algún punto crítico en la secuencia
   c) cuando existe un grupo de procesos que nunca progresan pues no se les otorga tiempo de procesador para avanzar
   d) ninguna de las otras respuestas es cierta.

3) En la comunicación asíncrona entre procesos:
   a) Ninguna primitiva de envío o recepción bloquearán a los procesos implicados
   b) La primitiva de envío bloqueará al emisor
   c) Ambas primitivas de envío o recepción bloquearán a los procesos implicados
   d) La primitiva de recepción bloqueará al proceso si no hay datos en el buzón

4) En el mecanismo de RPC, el resguardo o sustituto del procedimiento invocado se crea
   a) en el lado del cliente
   b) La creación de resguardos o stubs no es una técnica de RPC
   c) En el lado del cliente y en el lado del servidor
   d) en el lado del servidor

5) La operación "signal(.)" de un semáforo:
   a) No hará nada con la variable del semáforo
   b) Ninguna de las respuestas es correcta
   c) Incrementará siempre el valor de la variable del semáforo
   d) Si hay procesos bloqueados no incrementará el valor de la variable del semáforo

6) Las variables de condición en un monitor:
   a) Controlan diferentes condiciones dentro del monitor
   b) Garantizan la exclusión mutua de las funciones del monitor
   c) Son como los semáforos dentro del monitor
   d) Son necesarias para poder mantener la sincronización de los procesos dentro del monitor

7) El algoritmo de Dekker:
   a) Presenta situaciones en las que puede no garantizar las propiedades de viveza
   b) Es válido para "n" procesos con ligeras modificaciones
   c) Está orientado a entornos distribuidos
   d) Está orientado a entornos centralizados.

8) La asignación de procesadores físicos a hilos se realiza:
   a) Directamente, asignando la CPU al proceso del que forma parte un único hilo
   b) Indirectamente, asignando los procesadores lógicos a una CPU
   c) Directamente, por parte del planificador del Sistema Operativo
   d) Se hace a dos niveles, un primer nivel para asignar los hilos de usuario a los procesadores lógicos, y un segundo nivel para asignar los procesadores lógicos al procesador o procesadores físicos

9) Dada la siguiente configuración de procesos, determinar la respuesta correcta:

```
process P0
        Begin
                A;
                signal(s);
                B;
        End;
```

```
process P1
        Begin
                C;
                wait(s);
                wait(t);
                D;
        End;
```

```
process P2
        Begin
                E;
                signal(t);
                F;
        End;
```

   a) B se ejecutará siempre después de C
   b) D se ejecutará después de E y A
   c) A se ejecutará antes de F
   d) D se ejecutará siempre después de B y C.

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

12) En el lenguaje Java, si se produce una excepción no programada
   a) No puede evitarse la finalización de la aplicación
   b) Podremos tratar la excepción y evitar la finalización de la aplicación
   c) No podremos tratar esa excepción en el programa
   d) Ninguna de las respuestas es correcta

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

14) El paso de mensajes síncrono permite la comunicación:
   a) Muchos a uno
   b) Uno a muchos
   c) Muchos a muchos
   d) Uno a uno

15) Si en la definición de un método de una clase Java aparece la palabra reservada *synchronized*
   a) Sólo lo ejecutará un hilo a lo largo de la ejecución de la aplicación
   b) Sólo podrá ser ejecutado por un único hilo a la vez
   c) Ninguna de las respuestas es correcta
   d) No puede formar parte de la definición de un método

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

17) 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 disponga de alternativa "else"
   c) No se cumple ninguna de las guardas, si las tuviera
   d) No existe ningún mensaje en los buzones/canales que se manejan

18) Dado el siguiente código podemos afirmar que:

```
public void funcion2() {
        Rectangle rect;
        synchronized(rect){
                rect.width+=2;
        }
        rect.height-=3;
}
```

a) Todas las afirmaciones son FALSAS.
b) La anchura width del rectángulo rect no puede ser modificada por varios hilos a la vez.
c) La altura height del rectángulo no puede ser modificada por varios hilos a la vez.
d) La anchura width del rectángulo rect puede ser modificada por varios hilos a la vez.

20) En las llamadas a procedimiento remoto (RPC), la invocación al resguardo del cliente:
a) No requiere de conexión entre cliente y servidor
b) Siempre genera el bloqueo del proceso que realiza la invocación
c) La invocación se realiza siempre de un módulo que se encuentra en otro sistema
d) Debe garantizar que existe concordancia entre los parámetros

PREGUNTAS DE TEORÍA Y PROBLEMAS

1) Propiedades que han de cumplirse para garantizar la corrección de programas concurrentes (**1 punto**)

2) Implementar mediante el uso de monitores el gestor de páginas de memoria de un sistema operativo. Dicho gestor asignará las páginas de memoria de los diferentes procesos que se ejecutan en el sistema, garantizando que las páginas asignadas al proceso en cada solicitud deberán ser contiguas. El gestor ha de proporcionar las operaciones de *int solicitar (int n)* (devolverá la dirección de inicio de la primera página asignada) y *void liberar (int n)*, siendo n el número de páginas de memoria solicitadas.

   Proponer la implementación del monitor básico. Realizar las modificaciones a dicha solución para añadir las siguientes funcionalidades:

   - incorporar la operación *int libres()* que devuelva el número de páginas de memoria disponibles; modificar la forma en que se realiza la solicitud de páginas por parte de los procesos para no solicitar páginas en caso de que no haya suficiente número disponible-
   - priorizar las solicitudes de menor número de páginas. (**1,5 puntos**)

Supondremos para la solución básica que un proceso no puede realizar dos peticiones de páginas sin antes liberar las páginas solicitadas previamente. Además, en las peticiones se indicará el proceso que las realiza. Cada vez que se liberen páginas, el primer proceso bloqueado en la petición de asignación tratará de completar la reserva de páginas. En caso de que no se pueda, volverá a estado bloqueado.

El Gestor de memoria contendrá como estructuras de datos principales dos vectores:

- El primero, de tamaño NPAG, se usará para indicar el estado de las páginas; así, un valor 0 en la celda *i* significa que el bloque *i* está libre; un valor *k* en la celda *i* significa que el bloque de memoria *i* está asignado al proceso *k*.
- El segundo vector, de tamaño NPROC, se usará como índice para los procesos; un valor *i* en la celda *k* significa que la secuencia de bloques asignados al proceso *k* comienza en la página *i*. Un valor -1 en la celda significa que el proceso no tiene asignados bloques.

Con estas suposiciones, el monitor básico será el siguiente:

```
Monitor Gestor
VAR
  memoria = ARRAY [0...NPAG-1] of integer;
  // Array para gestionar el uso de las páginas de memoria
  índice= ARRAY[0..NPROC-1] of integer;
  //índice para acceder a la primera página ocupada por un proceso
  esperaHuecos: condition;
EXPORT
    int solicitar (int id, int n), void liberar (int id, int n)
// ---------------------------------------------------------------------------
int solicitar (int id, int n) {
  int j,cuenta;// Variables auxiliares para recorrer las matrices
  bool huecosEncontrados;
  huecosEncontrados = FALSE;
  REPEAT // Se buscan hasta encontrarlos n huecos consecutivos
    j=0; cuenta=0;
    WHILE j<NPAGS DO {
      IF memoria[j]==0 then cuenta ++;
      ELSE cuenta=0;
      IF (cuenta == n) then BREAK; // Se han encontrado n huecos contiguos
      ELSE j++;
    }
    IF (j == NPAGS) // No se han encontrado n huecos contiguos
      THEN  RESUME (esperaHuecos);
            // Se despierta a otro proceso que estuviera bloqueados
            // a la espera de huecos, si lo hubiere
            delay (esperaHuecos); // Se bloquea este proceso
      ELSE huecosEncontrados = TRUE;
  UNTIL huecosEncontrados
  // Almacenamos el bloque de inicio en el índice
  indice[id]=j;
  // Ahora marcamos los bloques como ocupados por el proceso id
  FOR (j=0; j<n; j++) memoria[índice[id]]=id;
  // Si hay procesos esperando solicitudes, se desbloquean de manera encadenada
  IF NOT empty(esperaHuecos) THEN RESUME (esperaHuecos);
  // Devolvemos la dirección del primer bloque
  RETURN (índice[id]);
}
// ---------------------------------------------------------------------------
void liberar (int id, int n) {
  int j;
  IF (indice[id]!=0) THEN { // El proceso id tiene bloques ocupados
    // Marcar los bloques como libres
    j=0;
```

```
    WHILE (j<n) && (memoria[(indice[id]+j)%NPAGS]==id) DO
      // Liberamos hasta n bloques asignados al proceso id
      memoria[([indice[id]+j)%NPAGS]=0;
    índice[id]=0;
    // Si hay procesos bloqueados, se desbloquean
    IF ¡empty(esperaHuecos) THE resume (esperaHuecos);
}
// ---------------------------------------------------------------------------

BEGIN // Inicialización del monitor
  FOR (i=0; i<NPROC; i++) DO indice[i]=0;
  FOR (i=0; i<NPAGS; i++) DO memoria[i]=0;
END;
```

La implementación del Proceso id sería

```
Process P (int id) {
  int n
  REPEAT
    n = random()*id; // Se genera un valor aleatorio;
    gestor.solicitar (id,n);
    // usar la memoria asignada
    gestor.liberar (id,n);
}
// ---------------------------------------------------------------------------
```

Modifiquemos ahora la solución para llevar el conteo de huecos totales libres y huecos contiguos libres. Para ello, usaremos dos variables y un método privado para actualizar esta segunda variable, El monitor quedaría entonces de la siguiente manera:

```
Monitor Gestor
VAR
  … // Igual que la solución anterior hasta la línea IF NOT empty …
  totalLibres: integer;
EXPORT
  int solicitar (int id, int n), void liberar (int id, int n), int libres();

// ---------------------------------------------------------------------------
int solicitar (int id, int n) {
… Igual que la solución anterior
  // Decrementar el número de bloquesLibres
  totalLibres-= n;
  // Devolvemos la dirección del primer bloque
  RETURN (índice[id]);
}
// ---------------------------------------------------------------------------
void liberar (int id, int n) {
  …. Igual que solución anterior
```

```
    // Actualizar número de bloques libres
    totalLibres += n;
    // Si hay procesos bloqueados, se desbloquean
    IF ¡empty(esperaHuecos) THE resume (esperaHuecos);
}
// ---------------------------------------------------------------------------
int libres () {
  RETURN (totalLibres);
}
// ---------------------------------------------------------------------------
BEGIN // Inicialización del monitor
  …. // Igual que solución anterior
  totalLibres=NPAGS;
END.
La implementación del Proceso id sería ahora
Process P (int id) {
  int n
  REPEAT
    REPEAT
      n = random()*id;
    UNTIL n<gestor.libres();
    gestor.solicitar (id,n);
    // usar la memoria asignada
    gestor.liberar (id,n);
}
```

Incorporamos ahora la priorización de las peticiones atendiendo al número de páginas solicitado. Para ello, modificamos la estructura inicial y creamos una lista ordenada por peticiones. Supondremos los siguientes métodos sobre dicha lista:

- getFirst(): Devuelve el primer elemento de la lista
- deleteFirst () : elimina el primer elemento de la lista
- insert (n): Crea un elemento en la lista con valor n, y lo inserta en la posición adecuada atendiendo al orden; si hay más de un elemento con valor n, lo inserta en orden FIFO.

Añadimos la siguiente variable

Con estos métodos, la implementación del monitor sería:

```
Monitor Gestor
VAR
  memoria = ARRAY [0...NPAG-1] of integer;
  // Array para gestionar el uso de las páginas de memoria
  índice= ARRAY[0..NPROC-1] of integer;
  //índice para acceder a la primera página ocupada por un proceso
  listaOrdenada= list of integer;
  // Lista ordenada con las peticiones de N° de páginas solicitadas
  esperaHuecos: condition;
```

```
EXPORT
int solicitar (int id, int n), void liberar (int id, int n)
// ---------------------------------------------------------------------------
int solicitar (int id, int n) {
  int j,cuenta;
  bool salirBucle;
  salir= FALSE;
  // Insertamos la petición en la posición apropiada
  cola.insert(n);
  // Comprobamos si la cola de peticiones tiene elementos
  // Mientras haya peticiones de menor número de páginas debe bloquearse el proceso
  WHILE n>cola.getFirst() THEN DELAY (esperaTurno);
  // Puede ocurrir que un proceso sea desbloqueado sin que sea su turno, por lo que
hay que volver a comprobar que es su turno
  // Sacamos la petición de la cola
  cola.deleteFirst();
  // En este punto n es el menor valor de las peticiones pendientes.
  // Buscar n huecos consecutivos. Si no hay n consecutivos deberá bloquearse
  j=0; cuenta=0;
  WHILE j<NPAGS DO {
    IF memoria[j]==0 then cuenta ++;
    ELSE cuenta=0;
    IF (cuenta == n) then BREAK; // Se han encontrado n huecos contiguos
    ELSE j++;
  }
  IF (j == NPAGS) // No se han encontrado n huecos contiguos
    THEN {
          // No hay n huecos disponible
          // Podría invocarse a un método de compactación
          RETURN (-1); // Devolvemos -1 para indicar no disponibilidad de memoria
    }
    ELSE  {
        // Almacenamos el bloque de inicio en el índice
        Índice[id]=j;
        // Ahora marcamos los bloques como ocupados por el proceso id
        FOR (j=0; j<n; j++) memoria[índice[id]]=id;
        // Devolvemos la dirección del primer bloque
        RETURN (índice[id]);
      }
}
// ---------------------------------------------------------------------------
void liberar (int id, int n) {
  … Igual que lo anterior
    totalLibres+=n;
```

```
    // Si hay páginas suficientes para atender la petición de mayor prioridad, se
despierta a un proceso
    IF (totalLibres>cola.getFirst() THEN RESUME (esperaHuecos);
}
// ---------------------------------------------------------------------------


BEGIN // Inicialización del monitor
  FOR (i=0; i<NPROC; i++) DO indice[i]=0;
  FOR (i=0; i<NPAGS; i++) DO memoria[i]=0;
```

La implementación del Proceso id sería

```
Process P (int id) {
  int n
  REPEAT
    n = random()*id; // Se genera un valor aleatorio;
    gestor.solicitar (id,n);
    // usar la memoria asignada
    gestor.liberar (id,n);
}
```

3) Modificar la solución básica del problema anterior, utilizando en este caso semáforos. Deberán implementarse las operaciones de solicitar y liberar bloques, (**1,5 puntos**).

Supondremos que disponemos de las operaciones solicitar y liberar. Las únicas variables compartidas serán el array de asignaciones de páginas y el índice de asignaciones, siendo el resto de variables declaradas como locales, y por tanto no compartidas entre los procesos.

```
void solicitar(int id, int n) {
  bool salir=FALSE;
  int i;
  int cuenta;
  REPEAT
    i=0; cuenta=0;
    WAIT (mutex);
    // Buscamos n huecos contiguos
    WHILE (i<NPAGS-cuenta) && (cuenta<n) DO
      IF (memoria[i]==0) cuenta++;
      ELSE cuenta=0;
    SIGNAL (mutex);
    IF (cuenta<N) THEN WAIT (noContiguos);
    ELSE salir==TRUE;
  UNTIL salir;
  // Marcar las páginas como ocupadas por el proceso.
  //La variable j contiene el valor de la última celda libre
  WAIT (mutex);
  indice[id]=j-n;
  For (i=0; i<n;j++) memoria[indice[id]+i]=id;
```

```
  SIGNAL (mutex);
  return (indice[id]); // Se devuelve la dirección de inicio
}

// ---------------------------------------------------------------------------------------------------------------
void liberar(int id, int n) {
  int j=0;
  WAIT (mutex);
  IF (indice[id]!=0) THEN { // EL proceso id tiene bloques ocupados
    // Marcar los bloques como libres
    WHILE (j<n) && (memoria[(indice[id]+j)%NPAGS]==id) DO
      // Liberamos hasta n bloques asignados al proceso id
      memoria[([indice[id]+j)%NPAGS]=0;
    índice[id]=0;
    SIGNAL (mutex);
    // Si hay procesos bloqueados esperando bloques contiguos, se desbloquean
    SIGNAL (noContiguos);
}
```

4) La Primera Orden está empeñada en localizar al Maestro Jedi Luke Skywalker con el fin de acabar con él. Para ello, Kylo Ren ha organizado todos los sistemas estelares en una matriz NxM, ubicando en cada sistema un droide dotado de sensores de presencia de la Fuerza. El funcionamiento deseado de los sensores de los droides es el siguiente: cada cierto intervalo de tiempo, el droide mide el número de midiclorianos en su sistema estelar, informa del mismo al ordenador central, quien le devuelve el promedio del conteo de midiclorianos almacenados en los sensores de los ocho sistemas estelares vecinos. A partir de esos datos, el droide calcula el valor de presencia de la Fuerza atendiendo a una función secreta almacenada en su memoria. Si el valor calculado supera un umbral, manda un aviso al Líder Supremo Snoke para informar de la posible presencia del traidor.

   Conocedor de nuestra capacidad y sabiduría, Kylo Ren nos ha encargado implementar el sistema informático. Tras un sesudo análisis por nuestra parte hemos decidido implementar la solución mediante alguna técnica de paso de mensajes. Se pide justificar la técnica final adoptada e implementar la solución completa mediante dicha técnica (**2 puntos**)

   Utilizaremos paso de mensajes asíncrono al ser más cómoda la solución. La razón es que existe comunicación “muchos a uno” (droides a ordenador central y droides a Snoke). La comunicación ordenador central a droides deberá ser “uno a uno”, por lo que usaremos un array de tamaño NxM de buzones, donde la celda [i,j] contendrá el buzón que usará el ordenador central para enviar los datos pedidos a cada droide sonda.

   Para la comunicación de las alertas a Snoke podemos utilizar directamente un buzón por el que se comuniquen los droides y Snoke directamente, o podríamos hacer que el controlador recibiera dichas alertas desde los droides y las derivara a Snoke. Optaremos por la primera solución al ser más simple, y así evitamos la introducción de una estructura SELECT en el proceso controlador

   Para empaquetar los datos que se transmiten en los mensajes usaremos un registro cuya estructura es la siguiente:

```
    typedef Struct {
          float nMidi;  // Para almacenar la medida de midiclorianos
          int i,j; // Identificamos al droide que envía el mensaje
        } Medidas;
// ---------------------------------------------------------------------------
```

```
   Process droide (int i, int j, function funcionSecreta) {
   Medidas medida;
    float valorVecinos;
    medida.idI = i; medida.idJ=j;
    REPEAT
      Medida.nMidi= random () * MAX; // toma un valor de midiclorianos
      SEND (informaMedidas,medida); // se informa de la medida de midiclorianos
      RECEIVE (vecinos[i,j],valorVecinos);  // Recibo el valor de las celdas vecinas
      IF (funcionSecreta(i,j,valorVecinos)>UMBRAL) THEN
          // funcionSecreta es la función secreta
        SEND (informaSnoke,medida); // Se informa al líder supremo
      DELAY (tiempoEspera);  // Simula la espera hasta la siguiente medida
    FOREVER
   }
// ---------------------------------------------------------------------------
   PROCESS ordCentral () {
    int i,j;
    float[N,M] valores;
    Medidas medida;
    // Se inicializa la matriz de valores
    FOR (i=0; i<N; i++)
      FOR (j=0; j<M; j++)
        valores[i,j]=0;
    REPEAT
      RECEIVE (informaMedidas, medida) // Se lee un mensaje con datos
      // Actualizar el valor en la tabla de valores
      Valores [medida.idI,medida.idJ]=medida.nMid;
      // Calculamos el promedio de los vecinos
      acum=0;
      FOR (i=-1; i<=1; i++)
        FOR (j=-1; j<=1;j++)
          // Evitamos sumar el valor que se ha recibido;
          IF ((i!=0) && (j!=0)) acum+=valores[(medida.idI+i)%N,(medida.idJ+j)%M];
      acum=acum/8;  // Se calcula la media de los ocho valores
      SEND(vecinos[medida.idI,medida.idJ],acum); // Devuelve el valor al droide
    FOREVER
   }
   // ---------------------------------------------------------------------------
   Process Snoke {
   Medidas medida
    REPEAT
      RECEIVE (informeSnoke,medida);
      enviarTropas (medida.idI,medida.idJ); // Se envían tropas al sistema estelar
    FOREVER
   }
```