PROGRAMACIÓN CONCURRENTE. EXAMEN PARCIAL **20-04-10**

Apellidos:_____ **Nombre:** _____

1 **[2,0]** El algoritmo de la figura 1 pretende resolver el problema de la *exclusión mutua para n-procesos*, para lo cual sólo utiliza n variables `flag` que pueden tomar 3 posibles valores: (`pasivo`, `solicitando`, `en_SC`), 1 variable `turn: 0..n-1` y la variable `j` local a los procesos. Las variables globales del algoritmo se inicializan a `pasivo` y `0`.

Se pide:

a) **[1,0]** Obtener el máximo número de turnos que un proceso no pasivo ha de esperar para entrar en sección crítica en el caso de planificación de procesos más desfavorable para él.

b) **[1,0]** Explicar las diferencias entre el algoritmo de la figura y el de Knuth y qué efecto tienen en la propiedad de equidad en el acceso de los procesos.

```
                Pi
<<resto de instrucciones>>
(1)  repeat
(2)  flag[i]:= solicitando;
(3)  j:= turn;
(4)  while j≠i do
(5)   if flag[j]≠pasivo then j:= turn;
(6)      else j:= (j+1) mod n;endif;
(7)  enddo;
(8)   flag[i]:= en_SC;
(9)  j:= 0;
(10) while(j<n)∧(j=ivflag[j]≠ en_SC) do
(11)   j:=j+1 enddo;
(12) until (j≥n ∧ (turno=i ∨
     flag[turno]=pasivo));
(13) turno:=i;
(14) <<sección crítica>>
(15) j:= (turno+1) mod n;
(16) while(j≠turno) ∧(flag[j]=pasivo) do
(17)   j:=(j+1) mod n;
(18) turno:= j;
(19) flag[i]:= pasivo;
     ...
```

Figura 1

2 **[1,25]** Un programa simple con un solo bucle transforma un vector inicial (**a**[1..n]) = **A**[1..n] (k:1..n: **a**[k]= **A**[k]) de tal forma que al final de su ejecución el valor de cada elemento del vector será **a**[k]= $\Sigma_{j=1}^{k}$ **A**[j] (cada elemento del vector **a** contiene la suma de todos los elementos anteriores a él en el array inicial **A**). Utilizar el invariante (I.B.) que se indica para demostrar la corrección del bucle cuyo bloque principal de instrucciones se da más abajo.

I.B.:{ $\Lambda_{k=1}^{i-1} a[k] = \Sigma_{m=1}^{k} A[m]$ }

```
i:=1 ;
s:= 0 ;
while ( i ≤ n ) do
        begin
        s:= s + a[i];
        a[i]:= s;
        i:= i + 1;
        end;
end do;
```

3 **[1,25]** Con qué tipo(s) de semántica de señales (SA, SX, SW, SU, SC) sería *correcto* (cumple propiedades de seguridad y vivacidad) el siguiente monitor. (a) Justificar.

```
Monitor Recursos;
var disponibles: int;
    c, b: signal;
procedure obtener(i);
  begin
  if (!disponibles) {
     c.signal();
     b.wait();
  }
     disponibles--;
     if (b.queue()) b.signal();
  end;
 end;

procedure suspender();
  begin
   if (disponibles >0)
      c.wait();
   end;
procedure rellenar();
begin
 disponibles+= M;
 b.signal();
end;
begin
  disponibles=M;
end;
```

(b) ¿Qué modificaciones sería necesario realizar en el código de los procedimientos del monitor anterior para que fuera correcto tanto para señales desplazantes como para no-desplazantes?

Procedure P();
begin
if (s=0) then c.wait();
else s:=s-1;
end;

Procedure V();
begin
if (c.queue()) then c.signal()
else s:=s+1;
end;

if (s=0) then

while (s==0)
c.wait();
s:=s-1;
NO

if c.queue()
c.signal()
else s:=s+1

4 **[0,75]** Con respecto a los axiomas de la operación **wait** de los monitores: explicar por qué el axioma de la operación **c.wait** de las señales con semántica *desplazante* ha de ser diferente del correspondiente para las señales *SC*. porque para los señales desplazantes, tras la ejecución de la operación wait(), se puede interrumpir la ejecución del procedimiento y cuando continúe su ejecución en el futuro no tiene porqué cumplirse el invariante. Es decir, para este tipo de señales no tiene porqué cumplirse el I.M. en la poscondición de la operación wait().

5 **[0,75]** Escribir el código de un monitor con tres procedimientos: **P()**, **V()**, **Inicio(m:0..MAXINT)** que implemente las operaciones *wait(s)* y *signal(s)* de un semáforo general para cualquier tipo de señal de los monitores y que satisfaga el invariante: $\{0 \leq s\} \wedge \{s = s_0 + nV + nP\}$. Se ha de segurar que nunca se pueda producir *robo de señal* por parte de un proceso que entre al monitor cuando se ha señalado a un proceso bloqueado en una variable condición, que simula la cola del semáforo que se pretende implementar.

[2,0] PROBLEMA: Suponer que N procesos comparten P impresoras todas iguales (suponer N > P). Se há de programar una clase **PrinterMgr** para gestionar el acceso, no desplazante y en exclusión mutua, a dichas impresoras , por medio de los siguientes métodos:
int peticion(int id) — Llamado por un proceso con **id** : 0 ≤ **id** < N) para solicitar el acceso a una impresora. Bloquea hasta que la impresora esté disponible y devuelve el identificador (0 ≤ **prntId** < P) de la impresora que se le ha concedido.
void liberar(int id, int prntId) — Llamado por un proceso con **id** para dejar libre a la impresora **prntId**. Si hay varios procesos esperando que se quede una impresora libre, entonces la impresora **prntId** se le concederá al proceso que estaba esperando y que posea el menor **id**.

a) **[1,25]** Programar una solución al problema anterior con regiones críticas condicionales.
b) **[0,75]** ¿Cuál es la propiedad de *seguridad* requerida para que este sistema sea correcto? (Indicar el Invariante de Recurso)

[2,0] PROBLEMA: Una familia de n pajarillos hambrientos comen de un mismo plato que inicialmente contiene **F** porciones de pienso. Cada uno de los pajarillos come una porción de pienso cada vez y, de forma repetitiva, duerme durante un rato, volviendo después a comer. Los comportamientos de un pajarillo y del padre se pueden modelar, entonces, como sigue a continuación:

```
1  class Pajarillo implements Runnable
2  {
3  private Plato plato; // Compartido por
     todos los pajarillos y el padre pájaro
4  int mi_Id; // Id  de pajarillo (para las
     salidas )
5
6  public Pajarillo(Plato p, int id)
7  { plato = p; mi_Id = id; }
8
9  public void run()
10 {
11 while (true)
12 {
13 System.out.println("Pajarillo " + mi_Id +
     " está hambriento.");
14 plato.obtieneComida();
15 System.out.println("Pajarillo " + mi_Id +
     " está comiendo.");
16 comer();
17 }
18 }
```

```
19 private void comer() {
21 try                                    {
     Thread.sleep((int)Math.round(Math.ran
     dom()*50)); }
22 catch (InterruptedException e) {}
23 }
24 }
25
26 El padre pájaro duerme hasta que el
     plato está vacío, lo completa y vuelve
     a dormir:

27 class Padre implements Runnable {
28 /** El plato que ha de mantener lleno */
29 private Plato plato;
30 public Padre(Plato p)
31 { plato = p; }
32 public void run() {
33 while (true) {
34 plato.completar();
35 System.out.println("Plato
     completado.");
36 }
37 }
38 }
```

Como se puede observa, la correcta sincronización entre los pajarillos y el padre se programa dentro de la clase **Plato**.
Se pide:
a) **[1,0]** Una implementación en pseudo-java de dicha clase (Plato) como un monitor programado en Java.
b) **[1,0]** Obtener el invariante del monitor que ha de verificar la solución que se proponga para éste y demostrar la corrección de sus procedimientos utilizando las reglas de las operaciones sincronización.
Suponer semántica de señales SC.

(a) PROBLEMA

```
Clase Plato;
{ private num_porciones=F;
  public void synchronized obtieneComida(){
      while (num_porciones==0) {
        this.wait();
      }
      num_porciones--;
      if (num_porciones==0) notifyAll();
  }
```

```
public void synchronized Completador(){
    while (num_porcs>0) {
        this.wait();
    }
    num_porcions=F;
    notifyAll();
}
```

(b)

Pajarillo

[B_1] durmiendo durante un rato
[B_2] comprobar si el plato está (estaba) vacío (num_porciones==0)
[B_3] si (num_porciones==0) "wait until" el padre pájaro complete el plato con F porciones de pienso
[B_4] asignar (num_porciones=F;)
[B_5] Coger 1 porción de pienso del plato (num_porciones--;)
[B_6] Si el plato se ha quedado vacío (num_porciones==0)
[B_7] entonces "informar" al padre pájaro
[B_8] comienza la comida

Padre Pájaro

[P_1] "wait until" un pajarillo encuentre el plato vacío
[P_2] asigna F porciones de pienso (o llena el plato)
[P_3] Informar a todos los pajarillos hambrientos que están esperando para comer que hay comida en el plato.

INVARIANTE: [B_7, P_1] nº Salen de [P_1] ≤ nº Entran [B_7]

[P_3, B_3] $i: 1..M$ Sale [B_3]$_i$ < Entra [P_3] + 1

```
resource impresoras (libres: int := P;
void liberar (int id, int printId) {
  region impresoras →
      libres++; k:=0; libre[printId]=true;
      while (!peticion[k] or k<n) do
        k++;
      end do;
      if (k<n) peticion[k]:=false;
  end region;
}
```

```
libre: array [1..P] of bodean := true;
peticion: array [0..n] of bodean := false)

int peticion (int id) {
  region impresoras →
    if (libres == 0) peticion[id]=true;
  end region;
  region impresoras
   when (libres > 0 or peticion[id]=false)
   →
   k:=0;
   when (! libre[k] and k ≤ P) do
    k++;
   end do;
   if (k ≤ P) then libre[k]:=false
                   libres--
              return k;
```