Examen – Sistemas Concurrentes –Grupo B **24-01-2018**

Apellidos: [redacted] **Nombre:** [redacted] **Grupo Prácticas:** [redacted]

El examen consta de varias partes: (A) Tema 1, (B) Tema 2 y (C) Temas 3 y 4. Quien tenga todo suspenso ha de hacer el examen completo y quien tenga alguna parte suspensa (o quiera subir nota de esa parte), contestará a las preguntas correspondientes a esa parte.

I. Ejercicios (Hasta **4,0** puntos)

A (Tema 1)

1. Semáforos: (a) Se invierten las operaciones **sem_wait(...)** en el proceso consumidor: ¿qué operaciones concurrentes dejarían de cumplirse? Justificar la respuesta.

```
1.  …
2.  void productor(void *arg){
3.    for(unsigned i=0;i<Max_produc;
    i++){
4.    int dato= producir_dato(i);
5.    sem_wait(huecos);
6.    sem_wait(mutex);
7.    //introducir dato y actualizar
8.    sem_signal(mutex);
9.    sem_signal(datos);
10.   }
11. }
```

```
…
1.  void consumidor(void *arg){
2.    for(unsigned i=0;i<Max_consum;
    i++){
3.  int dato;
4.  sem_wait(mutex);//orden CORRECTO:
5.  sem_wait(datos);//sem_wait(datos);
6.                  //sem_wait(mutex);
7.  // extraer dato y actualizar
8.  sem_signal(mutex);
9.  sem_signal(huecos);
10.   }
11. }
```

(b) Se invierten las operaciones sem_signal(...) en el proceso productor : ¿qué operaciones concurrentes dejarían de cumplirse? Justificar la respuesta.

```
1.  …
2.  void productor(void *arg){
3.  for(unsigned i=0;i<Max_produc;
    i++){
4.    int dato= producir_dato(i);
5.    sem_wait(huecos);
6.    sem_wait(mutex);
7.    //introducir dato y actualizar
8.    sem_signal(datos);
9.    sem_signal(mutex);
10.   }
11. }
```

```
…
1.  void consumidor(void *arg){
2.  for(unsigned i=0;
     i<Max_consum;i++){
3.    int dato;
4.    sem_wait(datos);
5.    sem_wait(mutex);
     //extraer dato buffer y actualizar
6.    sem_signal(mutex);
7.    sem_signal(huecos);
8.   }
9.  }
```

B (Tema 2)

2. Monitores: suponer un sistema básico de paginación de un sistema operativo multiusuario que proporciona 2 operaciones: adquirir(positive n) y liberar() para que los procesos de usuario puedan obtener las páginas de memoria que necesiten para su computación y, posteriormente, dejarlas libres para que puedan ser utilizadas por otros usuarios. Cuando un proceso invoca la operación adquirir (N), si no hay memoria disponible para atenderla, dicha petición quedaría pendiente hasta que exista un número suficiente de páginas libres en memoria para satisfacerla completamente. Llamando a la operación liberar(), un proceso de usuario convierte en disponibles para el sistema todas las páginas de memoria que tuviera asignadas.

Suponemos que los procesos de usuario adquieren y liberan páginas todas ellas del mismo tamaño (1K) a un área de memoria con estructura de cola y que nunca presenta el problema conocido como fragmentación de memoria (las páginas se mantienen contiguas y sin huecos en la cola).

Se pide programar un monitor que incluya las operaciones anteriores suponiendo semántica de señales *desplazantes urgentes* (SU).

(a) Resolverlo suponiendo orden FIFO estricto para atender las llamadas a la operación `adquirir(N)` de los procesos.
(b) Relajar la condición anterior y resolverlo atendiendo las llamadas según "primero la de menor número de páginas pedido" (SJF).

C (Temas 3 y 4)

3. Planificación de tareas de tiempo real: para el conjunto de tareas cuyos datos se muestran a continuación:
(a) Determinar cuántas veces interfiere la tarea t1 a las demás tareas.
(b) Calcular el tiempo de respuesta de cada tarea

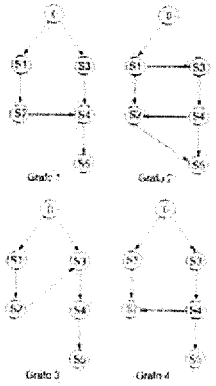
	Ci	Ti	Di
T1	1	3	2
T2	3	6	5
T3	2	13	13

Manifiesto abiertamente mi odio incondicional a capel.

II. Preguntas de respuesta alternativa (Hasta **3,0** puntos)

A (Tema 1)

1.Señalar los grafos de sincronización garantizan que las variables del siguiente programa concurrente consiguen valores no-indefinidos cuando el programa termine:



S_1: x^2

S_2: $m1 = a * cuad$

S_3: $m2 = bx$

S_4: $z = m1 + m2$

```
{ a==b==c==z==y==1, cuad == m1 == m2 == indefinido, x == 2}
S0::cobegin  S1;S2;S3;S4;S5 coend;
S1::cuad= x*x; S2::m1 = a*cuad; S3::m2= b*x; S4::z= m1+m2; S5::y=z + c;
```

a. Grafo 1
b. Grafo 2
c. Grafo 3
d. Grafo 4

2.Para los siguientes triples: (a) indicar cuál de ellos posee una *poscondción correcta*, es decir, que lo convierta en *demostrable* según la *Lógica de Programas* estudiada. (b) Escribir las *precondiciones* que conviertan en demostrables a los demás triples (sin alterar la *poscondición* escrita).

```
a. { i > 0 } i= i - 1; { i ≥ 1 }
b. { a > 0 } a= a - 7; { a > -6 }
c. { i < 10 } i= 2*i + 1 { 2*i < 9}
d. { Verdad } c= a + b; c= c/2; { c= (a + b)/2 }
```

3.Indicar cuál de las siguientes definiciones se aplica al concepto de programa concurrente *correcto* (las demás definiciones no consiguen explicar este concepto de una forma totalmente satisfactoria)

a. Si se puede demostrar que el programa cumple las propiedades de seguridad y vivacidad de los procesos
b. Si todos los procesos acceden a recursos compartidos con *justicia* y se obtiene un resultado secuencial correcto
c. Si se puede demostrar que todos los procesos del programa cumplen todas las propiedades concurrentes siguientes: *seguridad*, *vivacidad* y *equidad*
d. Si podemos demostrar que todos los procesos terminan sus cálculos y, por tanto, es equivalente a una determinada secuencia de ejecución secuencial

4.¿Cuál de las posibles salidas del programa que se indican más abajo será la correcta?

```
Process p1(){
  x+= 10;}
Process p2(){
  if (x > 100)
   cout<<x<<endl<<flush;
  else cout<<x-50;<<endl
<<flush;}
int main(){
  int x= 100;
  cobegin p1(); p2(); coend;}
```

a. Puede llegar a imprimir '60' Si solo se puede 1, la b.
b. El valor final de la vraiable 'x' es indeterminado si no se programan instrucciones atómicas en este código
c. Imprime '50' pero en ese caso el valor final de la vraiable 'x' será siempre '100'
d. La ejecución de programa imprimirá siempre '50' ó '100'

5. El siguiente código pretende ser un protocolo correcto para resolver el problema de lectores/escritores con semáforos binarios. Indicar cuál de las afirmaciones es la correcta.

```
Semaphore mutex=1, lector=1,
escritor=1;  //semáforos FIFO
int no_lectores=0,
no_escritores=0;

process lector (i: int){
do{
  lector.wait();
  mutex.wait();
    no_lectores++;
    if (no_lectores==1)
        escritor.wait();
  mutex.signal();
 lector.signal();
<<<<LEER>>>>
  mutex.wait();
    no_lectores--;
    if(no_lectores==0)
       escritor.signal();
  mutex.signal();
}while(true);
}
process escritor(j: int){
  do{
    lector.wait();
    escritor.wait();
<<<<ESCRIBIR>>>>
   escritor.signal();
   lector.signal();
while(true);
}
cobegin for(i=1;
i<n;i++)lector(i); for(i=1;
i<m;i++)escritor(i) coend;
```

a. El protocolo anterior resuelve el problema con prioridad a los lectores
b. El protocolo anterior no resuelve realmente el problema de lectores y escritores
c. El protocolo anterior resuelve el problema intentando no favorecer siempre a los procesos de un mismo tipo (lectores o escritores)
d. El protocolo anterior resuelve el problema con prioridad a los escritores

B (Tema 2)

6. Seleccionar la única respuesta correcta de las siguientes.

a. Un programa con procesos concurrentes y programado con monitores no puede implementarse utilizando procesadores multinúcleo pues estos poseen caches privados
b. La protección automática de las variables permanentes declaradas dentro de un monitor hace innecesaria la demostración de la propiedad concurrente denominada seguridad de los programas concurrentes
c. La planificación FIFO obligatoria de las colas de los monitores asegura en todo caso que siempre se satisfaga la propiedad de vivacidad de los procesos de un programa concurrente
d. Como consecuencia de haber realizado una llamada a un procedimiento de un monitor, un proceso concurrente se puede suspender y salir del monitor

7. La interpretación correcta del axioma de la operación de sincronización "c.signal()" con señales desplazantes **{¬ empty(c) ∧ L ∧ C} c.signal() {IM ∧ L }** es una de las siguientes (Nota: IM= *invariante de monitor*, L= *invariante de las variables locales al procedimiento*, C= *condición de sincronización para la variable condición implicada*)

a. El proceso concurrente que ocasiona la ejecución de la operación de sincronización no se suspende salvo que la cola "c" esté vacía

b. El proceso concurrente se suspende en la cola de entrada al monitor cuando ejecuta la operación de sincronización
c. Cuando se ejecuta la operación de sincronización dentro de un procedimiento del monitor, el estado reentrante del procedimiento hace cierta la condición "C"
d. Si la cola no está vacía, el valor de certeza de la condición "C" se transmite a todo el monitor hasta que se desbloquea 1 proceso de la cola

8. Seleccionar la única afirmación correcta con respecto a la demostración de corrección, con las reglas de las operaciones de sincronización desplazantes de los monitores, del "semáforo de Habermann"

```
Monitor Semaforo;
   1. var na, np, nv:integer; c: condicion;
   2. procedure P();                          procedure V();
   3. begin                                   1. begin
   {np= min(na,nv)}                           {np= min(na,nv)}
   4. na:= na+1;                              2.  nv:= nv+1;
   {np= min(na-1,nv)                          {np= min(na,nv-1)
   5. if(na > nv)       then                  3.  if(na > np) then
   {np= min(na,nv)}                           {na>np & np= nv-1}
   6. c.wait();                               4.      c.signal;
   {na>np & np= nv-1}                         {np= min(na,nv)}
   7. else {na<=nv & np=na-1}                 5.  else {np=na & np<nv}
   8. endif;                                  6.  endif;
   {np+1= min(na,nv)}                          {np= min(na,nv)}
   9. np:= np+1;                              7. end;
   {np= min(na,nv)}
   10.     end;
begin  {true} na:=0; nv:=0; np:=0; {IM: np= min(na,nv)}end;
```

a. En la precondición de la instrucción (4) `c.signal()` del procedimiento V() no se satisface el invariante del monitor
b. Antes de la instrucción (4) `c.signal()` del procedimiento V() se ha de satisfacer el invariante del monitor
c. La demostración del procedimiento V() podría ser incompleta porque no se ha tenido en cuenta el caso de que se cumpliera: np > na en la precondición de la instrucción (5) "`else...`" del mencionado procedimiento
d. Los procedimientos serían también demostrables para una variable condición "c" con semántica "señalar y continuar" (SC)

9. La interpretación correcta del axioma de la operación de sincronización c.wait() con señales desplazantes (**{ IM ∧ L } c.wait() { C ∧ L }**) es una de las siguientes

a. Si no se cumple la condición "C", al volver la llamada a la operación de sincronización, el procedimiento del monitor no se ha programado correctamente
b. El proceso que ocasiona la ejecución de la operación se bloquea y deja libre el monitor, cumpliéndose a continuación la condición "C" como precondición de la siguiente instrucción
c. La condición "C" coincide con el invariante global del monitor
d. Si no se cumple la condición "C", al volver la llamada a la operación de sincronización, no podemos afirmar que el invariante del monitor se cumpla

10. Suponiendo que disponemos de un lenguaje de programación con monitores con señales SU, indicar cuál de las siguientes implementaciones del mecanismo de sincronización entre procesos denominado cita es la correcta

a.

```
Monitor Citas1;
Var p, q: condition;
void*acepta_cita();
 begin
   p.wait();
   q.signal();
end;
void*llama()
   begin
   p.signal();
   q.wait();
   end:
cobegin
P1::Citas1.llama();
P2::Citas.acepta_cita() coend;
```

b.

```
Monitor Citas2;
Var p, q: condition;
void*acepta_cita();
  begin
   q.signal();
   p.wait();
end;
void*llama()
   begin
   p.signal();
   q.wait();
   end:
cobegin P1::Citas1.llama();
P2::Citas.acepta_cita() coend;
```

c.

```
Monitor Citas3;
Var p, q: condition;
señalado:boolean;
void*acepta_cita();
     begin
       if( q.empty()) then
   p.wait();
   señalado:= true;
   q.signal();
end;

 void*llama()
   begin
   p.signal();
   if (not señalado)
   q.wait();
   señalado:= false;
   end:
begin señalado:= false end;
cobegin
P1::Citas1.llama();
P2::Citas.acepta_cita() coend;
```

d.

```
Monitor Citas2;
Var p, q: condition;
señalado:boolean;
void*acepta_cita();
     begin
       if( q.empty()) then
   p.wait();
      q.signal();
   señalado:= false;
end;

 void*llama()
   begin
   p.signal();
   if (not señalado)
   q.wait();
   señalado:= true;
   end:
begin señalado:= false end;
cobegin
P1::Citas1.llama();
P2::Citas.acepta_cita() coend;
```

C (Temas 3 y 4)

11. Seleccionar la alternativa verdadera:

a. En ningún caso el proceso receptor se suspenderá al ejecutar la orden **receive(...)** en el *paso de mensajes asíncrono con búfer*.

b. Las condiciones de la *instrucción de espera selectiva* (**select**) han de ser excluyentes entre las distintas alternativas de esta orden.

c. Un orden **select** podría suspender su ejecución incluso si todas las condiciones de sus alternativas se hubieran evaluado como ciertas.

d. El paso de mensajes síncrono no puede implementarse con un búfer.

12. Seleccionar la alternativa verdadera:

a. El algoritmo RMS nos dice que la *planificabilidad* (las tareas consiguen ejecutar sus unidades antes de expire su plazo máximo de respuesta) de un conjunto de N tareas periódicas es imposible si la utilización conjunta del procesador supera el límite: $U = N * (2^{\frac{1}{N}} - 1)$.

b. Si se utiliza asignación estática de prioridades a las tareas, entonces podemos afirmar que dicho conjunto de tareas es *planificable* independientemente de la fase de cada tarea

c. El plazo de respuesta máximo (D) para cada tarea no depende del instante de su próxima activación.

d. El algoritmo de planificación denominado RMS nunca puede proporcionarnos un esquema de planificación de tareas con aprovechamiento del 100% del tiempo del procesador

13. Seleccionar las afirmaciones correctas respecto de la ejecución de las alternativas de la espera selectiva **select** que poseen algunos lenguajes de programación distribuida

a. Si hay guardas ejecutables en las alternativas, se selecciona no *determinísticamente* una entre las que poseen una orden **send** ya iniciada

b. Si no hay guardas ejecutables pero sí las hay potencialmente ejecutables, la instrucción de espera selectiva se suspende hasta que un proceso nombrado inicie una operación **send**

c. Si en las alternativas no se ha programado ninguna sentencia de entrada (**receive**), se selecciona *no determinísticamente* una cualquiera de éstas para su ejecución y la espera selectiva termina

d. La espera selectiva nunca puede levantar una excepción en el programa donde se programe

14. Seleccionar la afirmación correcta respecto de la espera selectiva con guardas indexadas

a. Las condiciones de las alternativas de la espera selectiva no pueden depender de argumentos de entrada (**arg**) de las sentencias de entrada (**receive (var arg)**)que se programen en éstas

b. El índice que se programa para replicar una alternativa no puede depender de valores límite (inicial, final) no conocidos en tiempo de compilación del programa

c. En la instrucción de espera selectiva no se pueden combinar alternativas indexadas con otras alternativas normales no indexadas

d. Un conjunto de N procesos emisores cada uno envía caracteres al resto de los procesos de dicho conjunto, es decir, cada proceso recibe (N-1) mensajes de los demás, entonces es imposible programar los procesos individuales con instrucciones de espera selectiva porque dicha orden se ejecuta 1 vez y termina

15. Seleccionar la afirmación correcta

a. La instrucción de espera selectiva no es necesaria en los lenguajes de programación porque todo se puede programar con operaciones de paso de mensajes no bloqueantes

b. **MPI_Probe** o comprobación bloqueante de mensaje es redundante con **MPI_Wait**

c. **MPI_Send** podría volver sin esperar la ejecución de la operación de recepción concordante

d. **MPI_Recv** podría volver sin esperar la ejecución de la operación de envío concordante

III. Cuestiones y preguntas (Hasta **3,0** puntos):

A (Tema 1)

1. Explicar el significado de las propiedades de corrección de los programas concurrentes y cuándo podemos afirmar que un programa concurrente es correcto y en qué *grado de corrección* estaría

B (Tema 2)

2. Programar el mecanismo de sincronización denominado *barrera* con monitores. Se necesita programar un procedimiento del monitor para implementar la espera de los procesos que esperan y otro para desbloquear a todos, que invoca el último proceso esperado

C (Temas 3 y 4)

3. Explicar el problema (seguridad) que tendría el siguiente código, que programa 2 operaciones de paso de mensajes no bloqueante y sin búfer con respecto a los valores finales de las variables x e y, que aparecen señaladas con (**) el código mostrado:

```
int main(int argc, char*argv[]){
 int rank, size, vecino, x, y;
 MPI_Status status;
 MPI_request request_send, request_recv;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 y=rank*(rank+1);
 if (rank%2) vecino= rank+1;
 else vecino= rank - 1;
     //Para rellenar
```

```
 //Las siguientes operaciones pueden aparecer en cualquier orden
 MPI_Irecv(&x,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_recv);
 MPI_Isend(&y,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_send);
```

```
 //(**)
 x= y + 7;
```

¿Qué funciones de OpenMPI programarías en el espacio en blanco ("para rellenar") que resolviera el problema identificado?