UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela Universitaria de Informática
Facultad de Informática

Calificación
1
2
3

SISTEMAS OPERATIVOS

Convocatoria de septiembre, 12 de septiembre de 2007

Nombre	Titulación
SOLUCIONES	

Dispone de tres horas para completar el examen

---

1 (5 puntos) **Test.** Marque las opciones correctas de cada apartado. En caso de que existan varias opciones ciertas, se considerará como correcta la más completa o precisa. Las preguntas no contestadas no puntúan; las mal contestadas puntúan negativamente restando un tercio de su valor. Marque la opción correcta rodeándola con un círculo. Si se equivoca, tache la respuesta equivocada y rodee la opción correcta. Escriba con tinta. Las preguntas respondidas con lápiz o con varios círculos no tachados se considerarán no contestadas.

1. Un sistema multiprogramado:
   a) permite gestionar varias tareas interactivas
   b) evita el posible monopolio de la CPU por un proceso del sistema
   c) debe realizar el control de la sincronización
   d) **ninguna de las afirmaciones es correcta

2. Si varios procesos concurrentes se ejecutan en un sistema, ocurre que:
   a) ** para su ejecución compiten por el uso de uno o de varios procesadores
   b) se ejecutan sobre un sistema operativo multiusuario
   c) comparten el mismo código
   d) necesitan sincronizarse para acceder a los datos que comparten

3. El bloque de control de proceso (BCP) contiene, al menos:
   a) el grado de multiprogramación de un proceso
   b) el código, los datos y la pila de un proceso
   c) **el estado y el identificador de un proceso
   d) todas las afirmaciones anteriores son correctas

4. ¿cuál de las siguientes afirmaciones es correcta?
   a) Los sistemas multiprogramados son sistemas de tiempo compartido que permiten conmutar los trabajos presentes en el sistema.
   b) **Los sistemas de tiempo compartido son sistemas multiprogramados en los que se reparte el uso de la CPU entre los procesos existentes a intervalos regulares de tiempo.
   c) Un sistema de tiempo real es un sistema con tiempos medios de respuesta óptimos
   d) Ninguna de las afirmaciones anteriores es correcta

5. Supongamos que para resolver cierto problema de concurrencia empleamos un monitor y variables condición. En este caso:
   a) Una operación ***wait*** sobre una de las variables condición empleadas en dicho monitor provocará que el proceso que la realice se bloquee siempre que no exista otro proceso en el monitor previamente.
   b) Una operación ***signal*** sobre una de las variables condición empleadas en dicho monitor provocará el desbloqueo de otro proceso que haya realizado una operación ***wait*** sobre cualquiera de las variables condición del monitor.
   c) Una operación ***signal*** nunca provoca el bloqueo del proceso que la realiza.
   d) **Todas las anteriores son falsas.

6. ¿Cuál de las siguientes afirmaciones es correcta?
    a) El modo dual de operación se ha diseñado para que los programas de usuario obtengan acceso pleno a todos los recursos del sistema con total libertad.
    b) **Cuando se produce una interrupción, la CPU conmuta a modo privilegiado/supervisor.
    c) Las instrucciones de desactivación de interrupciones se ejecutan normalmente en modo de usuario.
    d) Las instrucciones lectura del reloj del sistema normalmente deberían ser operaciones privilegiadas.

7. A un planificador de procesos que tiene la cola vacía llegan al mismo tiempo ocho procesos de idéntica duración. Entonces:
    a) Si se utiliza SJF, todos los procesos tendrán el mismo tiempo de retorno
    b) Si se utiliza un Round Robin con cuanto igual a 1 u.t., se obtendrá un mejor tiempo de retorno medio que con el SJF
    c) Si se utiliza un Round Robin con cuanto igual a 8 u.t., se obtendrá un mejor tiempo de retorno medio que con el SJF
    d) **Si se utiliza FCFS, cada proceso tendrá un tiempo de espera diferente a los demás

8. ¿Qué cambio de estado no se puede dar durante la vida de un proceso?
    a) de «preparado» a «en ejecución»
    b) de «en ejecución» a «preparado»
    c) **de «preparado» a «bloqueado»
    d) de «bloqueado» a «preparado»

9. ¿Qué dos características no se pueden combinar en un algoritmo de planificación?
    a) expulsividad y prioridades
    b) prioridades y rodajas de tiempo limitadas
    c) expulsividad y envejecimiento (*aging*)
    d) **todas las anteriores combinaciones son posibles

10. Un sistema de memoria paginado utiliza direcciones lógicas de 32 bits y tamaño de página 1 kbyte. Cada entrada en la tabla de páginas ocupa 4 bytes. ¿cuál es el número máximo de marcos de página que se necesitarán para almacenar por completo una tabla de páginas en memoria?
    a) $4 \cdot 2^{32}$ marcos
    b) $4 \cdot 2^{22}$ marcos
    c) ** $16 \cdot 2^{10}$ marcos
    d) $4 \cdot 2^{10}$ marcos

direcciones lógicas de 32 bits

tamaño de página de 1 Kbyte=2^{10} bytes → se necesitan 10 bits para el desplazamiento

El número máximo de páginas será de $2^{(32-10)}= 2^{22}$ páginas

La tabla de páginas ocupará como máximo 2^{22}.4 bytes= 16Mbytes

En total haría falta 16Mbytes/1Kbyte=16.2^{10} páginas o marcos

11. Tenemos un sistema de gestión de memoria paginada de un nivel que utiliza una TLB. Se tienen los siguientes tiempos medios: tiempo de acceso a la TLB: 3 nseg; tiempo medio de acceso a memoria principal: 42 nseg. Se desea que el tiempo de medio de acceso a memoria a partir de una dirección lógica sea inferior a 48 nseg. ¿Cuál es la tasa de aciertos mínima de la TLB que lo garantizaría?
    a) ** 93%
    b) 97%
    c) 85%
    d) 90%

t acierto= 3+42=45ns

t fallo=3+42+42=87ns

87 *(1-p)+45*p<48

42p>39 → p>0.928 → tasa de aciertos mayor que 93%

12. Cuando realizamos una operación ***V*** sobre un semáforo:
    a) Se despertará siempre al primer proceso que realizó una operación ***P***, y por tanto, el proceso que realiza la ***V*** se bloquea ya que no es posible que continúen su ejecución dos procesos por la propiedad de exclusión mutua.
    b) En caso de existir algún proceso bloqueado, se despertará a uno de ellos según la política definida en la implementación del semáforo, y por tanto, el proceso que realiza la ***V*** se bloquea ya que no es posible que continúen su ejecución dos procesos por la propiedad de exclusión mutua.
    c) **En caso de existir algún proceso bloqueado, se despertará a uno de ellos según la política definida en la implementación del semáforo.
    d) Lo dicho en la (c), y además en caso de no existir ningún proceso bloqueado el valor del semáforo no se alterará.

13. Suponga un archivo perteneciente a un sistema de archivos tipo FAT32 cuyos bloques de datos poseen un tamaño de 512 bytes. Asumiendo que: el archivo ya ha sido abierto con éxito; el puntero de acceso al archivo posee el valor 0; y la FAT se encuentra totalmente cargada en memoria principal, ¿cuántos accesos a disco se requerirán para leer un registro de tamaño de 1024 bytes que comienza en el byte 2048 del archivo?
    a) un acceso
    b) **dos accesos
    c) cuatro accesos
    d) seis accesos

14. Suponga un sistema de archivos que soporta las políticas básicas de gestión del espacio en disco. En este hipotético sistema cuando un usuario crea un archivo, especifica qué política se aplica. Para un archivo que contendrá las copias de seguridad de los archivos propiedad del usuario ¿qué tipo de política elegiría?
    a) **contigua
    b) encadenada
    c) indexada
    d) segmentada

15. Suponga un sistema de archivos que utiliza una política de gestión del espacio en disco encadenada con doble encadenamiento que contiene 65536 bloques de sistema de archivo, teniendo cada bloque un tamaño de 1024 bytes. Si un archivo posee 7 bloques, ¿cuál será su tamaño efectivo máximo?
    a) 7168
    b) ** 7140
    c) 7154
    d) 7156

65536 = 2^{16} , por lo que se necesitan 16 bits (2 bytes) para direccionar los 65536 bloques. Dado que nos indican que la gestión es encadenada con doble encadenamiento, en cada bloque de 1024 bytes se emplearan 4 bytes (2 bytes para cada puntero) para el encadenamiento de bloques, quedando por tanto 1020 bytes efectivos para almacenar datos en cada bloque. Por tanto, en un archivo de 7 bloques se podrán almacenar 1020*7= 7140 bytes de datos.

---

2 (2,5 puntos) Tenemos un sistema de memoria virtual paginada, con estas características:

- tamaño de página: 1024 bytes
- marcos disponibles en memoria física: 5
- algoritmo de reemplazo: LRU **local**

Dos procesos A y B se están ejecutando en el sistema, que le asigna un máximo de tres marcos a A y un máximo de dos marcos a B. Sus tablas de páginas contienen ahora mismo la siguiente información:

	Proceso A	
	marco	bit validez
0		i
1	4	v
2		i
3	3	v
4		i
5		i
6		i
7		i

	Proceso B	
	marco	bit validez
0		i
1	0	v
2		i
3		i
4	2	v

Las últimas referencias a memoria que han ocurrido en el sistema son estas **direcciones lógicas**: (B:1500), (A:3333). Tras ello, se realizan los siguientes accesos: (B:2109), (A:207), (A:4444), (B:2222), (A:2345), (B:3471), (A:88), (B:2000), (A:1055), (B:1093), (B:5008), (A:7815).

Se pide aplicar el algoritmo de reemplazo para esta secuencia de accesos y mostrar lo siguiente *(nota: cada apartado puntúa por un tercio del valor de la pregunta)*.

a. Cómo quedan las tablas de páginas de A y B al final de la secuencia.
b. Calcular las direcciones físicas a las que dan lugar las direcciones lógicas de la secuencia.
c. Calcular cuántos fallos de página se provocan e indicar cuáles son los accesos responsables de esos fallos.

a.

	Proceso A	
	marco	bit validez
0	1	v
1	4	v
2		i
3		i
4		i
5		i
6		i
7	3	v

	Proceso B	
	marco	bit validez
0		i
1	2	v
2		i
3		i
4	0	v

b.

Dirección lógica (página:desplazamiento)	Dirección Física
B:2109 -> 2:61	2048 + 61 = 2109
A:207 -> 0:207	1024 + 207 = 1231
A:4444 ->4:348	4096 + 348 = 4444
B:2222 -> 2:174	2048 + 174 = 2222
A:2345 -> 2:297	3072 + 297 = 3369
B:3471 -> 3:399	0 + 399 = 399
A:88 -> 0:88	1024 + 88 = 1112
B:2000 -> 1:976	2048 + 976 = 3024
A:1055 -> 1:31	4096 + 31 = 4127
B:1093 -> 1:69	2048 + 69 = 2117
B:5008 -> 4:912	0 + 912 = 912
A:7815 -> 7:647	3072 + 647 = 3719

c.

accesos	**valor inicial**	**B:2109 2:61**[i]	**A:207 0:207**	**A:4444 4:348**	**B:2222 2:174**	**A:2345 2:297**	**B:3471 3:399**
marco 0	B:1	B:1	B:1	B:1	B:1	B:1	**B:3**
marco 1			**A:0**	A:0	A:0	A:0	A:0
marco 2	B:4	**B:2**[ii]	B:2	B:2	B:2	B:2	B:2
marco 3	A:3	A:3	A:3	A:3	A:3	**A:2**	A:2
marco 4	A:1	A:1	A:1	**A:4**[iii]	A:4	A:4	A:4
FP[iv]		x	x	x		x	x

(continuación)

accesos	**A:88 0:88**	**B:2000 1:976**	**A:1055 1:31**	**B:1093 1:69**	**B:5008 4:912**	**A:7815 7:647**
marco 0	B:3	B:3	B:3	B:3	**B:4**	B:4
marco 1	A:0	A:0	A:0	A:0	A:0	A:0
marco 2	B:2	**B:1**	B:1	B:1	B:1	B:1
marco 3	A:2	A:2	A:2	A:2	A:2	**A:7**
marco 4	A:4	A:4	**A:1**	A:1	A:1	A:1
FP		x	x		x	x

Por tanto se generan en total nueve fallos de página.

(i)Dirección lógica en formato página:desplazamiento.

(ii)Se sustituye la página lógica 4 ya que el ultimo acceso de B se realizó a la página lógica 1 (dirección lógica 1500 según dato del enunciado).

(iii)Se sustituye la página lógica 1 ya que los ultimos accesos de B se realizaron a las páginas lógicas 3 y 0.

(iv)FP = fallo de página

3 (2,5 puntos) Un desarrollador implementa un programa en C++ para contar cuántos elementos de un vector tienen valor negativo. Como el programa se va a ejecutar en un multiprocesador, decide implementarlo con varios hilos concurrentes que realizan búsquedas parciales en segmentos del vector. Y así queda el código:

```
// El vector donde se realizará la búsqueda
const int N =  10000;
int vector [N];

// Esta variable contendrá la cantidad de valores negativos
// que se vayan encontrando en el vector
int negativos_encontrados = 0;

// Rutina que ejecutan los hilos concurrentes
void buscar_negativos ( int inf, int sup ) {
  int mis_negativos = 0;
  for ( int k=inf; k<sup; k++ ) {
    if ( vector[k] < 0 ) {
      mis_negativos = mis_negativos + 1;
    }
  }
  negativos_encontrados = negativos_encontrados + mis_negativos;
}

// Rutina principal
main () {
  rellenar_vector(); // introduce valores en “vector”
  for ( int i=0; i<N; i+=100 ) {
    Thread* hilo = new Thread();
    hilo->Fork ( buscar_negativos, i, i+100 );
  }
  printf ( “El vector contiene %d valores negativos\n”,
           negativos_encontrados );
}
```

El lanzamiento de los hilos se realiza con una sintaxis similar a la de Nachos. El efecto que tiene cada llamada a **`hilo->Fork`** es que se crea un hilo que ejecuta una llamada a **`buscar_negativos(i,i+100)`**, de forma totalmente concurrente y asíncrona con el resto del código.

Primera parte (1 punto)

¿Cuántos hilos diferentes lanza este programa?

El programa lanza los hilos en el bucle «for» del main, que realiza exactamente N/100 iteraciones, en cada una de las cuales se lanza un nuevo hilo. Como N=10000, se lanzará un total de 100 nuevos hilos, que habría que añadir al hilo principal ya existente.

¿Aumentará la velocidad de ejecución de este programa concurrente, con respecto a una implementación completamente secuencial? (tenga en cuenta que se ejecutará en un multiprocesador). ¿Serviría de algo añadir más hilos?

La velocidad sólo podrá aumentar si se consigue que uno o varios hilos se ejecuten en procesadores diferentes, consiguiendo una ejecución en paralelo de varios flujos del programa. Por tanto, dependemos de que el sistema operativo sea capaz de gestionar adecuadamente el multiprocesador. En cualquier caso, el límite máximo de aumento de velocidad vendrá dado por el número de procesadores físicos del sistema: si hay cuatro procesadores, el lanzamiento de diez o cien hilos no va a reducir más el tiempo de ejecución que si lanzamos sólo tres o cuatro hilos. Incluso podría penalizar el tiempo de ejecución, ya que las actividades de gestión de los hilos (cambios de contexto, etc.) consumen su tiempo.

También habría que analizar el algoritmo para verificar que se puede ejecutar realmente en paralelo. Tal y como está escrito, así es, puesto que cada hilo trabaja

independientemente de los demás con su porción del vector, con lo cual los hilos no tienen que esperar por resultados de sus compañeros.

Segunda parte (1,5 puntos)

Al ejecutar el programa, se imprime por pantalla un número incorrecto, casi siempre 0 (cero), incluso cuando todos los elementos del vector son negativos. El programador se da cuenta de que hay algunos fallos graves en el programa, por una inadecuada sincronización entre los hilos. ¿Cuáles son esos fallos y dónde se encuentran? ¿Cómo se podrían solucionar?

El problema radica en el uso de la variable «negativos_encontrados». Cada hilo concurrente, al final de su búsqueda, actualiza el valor de esa variable con la cantidad de valores negativos que ha encontrado en su porción del vector. Si varios hilos intentan modificar simultáneamente «negativos_encontrados», podemos tener inconsistencias (p.ej. perder modificaciones de algún hilo). Esa línea donde se incrementa la variable es una **sección crítica** que se debe ejecutar atómicamente.

Además de lo anterior, hay un problema grave de sincronización entre el hilo principal (el «main») y los hilos lanzados. El **printf** final escribe el valor de la variable «negativos_encontrados» sin haber comprobado que los hilos que actualizan esa variable han terminado de recorrer el vector. Para que el programa fuera correcto, ese **printf** debería esperar a que *todos* los hilos lanzados en el bucle hayan finalizado. Es un caso típico de operación **join**.

Escriba un esquema de la solución, utilizando la herramienta de sincronización que prefiera (semáforos, cerrojos, variables condición, etc.)

La sección crítica sobre «negativos_encontrados» se puede resolver con cualquiera de las soluciones de sección crítica que se han visto en la asignatura. Por ejemplo, se puede utilizar un semáforo inicializado a cero para regular el control de la variable:

```
Semáforo mutex = 1; // variable global
...
P (mutex);
negativos_encontrados = negativos_encontrados + mis_negativos;
V (mutex);
```

La sincronización entre hilos lanzados y el «main» también se puede resolver de múltiples formas. Una de ellas la hemos expuesto en las clases prácticas de la asignatura, y consiste en utilizar un contador para saber cuántos hilos han finalizado. A continuación se muestra una solución completa que utiliza esta técnica y resuelve también el acceso concurrente a «negativos_encontrados». Todo ello con semáforos. En azul está el código añadido.

```
// El vector donde se realizará la búsqueda
const int N =  10000;
int vector [N];

// Esta variable contendrá la cantidad de valores negativos
// que se vayan encontrando en el vector
int negativos_encontrados = 0;

// Semáforo para controlar el acceso a negativos_encontrados
Semáforo mutex = 1;

// Semáforo para sincronizar los hilos con el main
Semáforo join = 0;

// Variable que indica cuántos hilos están pendientes de terminar
int hilos_pendientes = N / 100;

// Rutina que ejecutan los hilos concurrentes
void buscar_negativos ( int inf, int sup ) {
  int mis_negativos = 0;
  for ( int k=inf; k<sup; k++ ) {
    if ( vector[k] < 0 ) {
      mis_negativos = mis_negativos + 1;
    }
  }
  // Sección crítica
  P (mutex);
  negativos_encontrados = negativos_encontrados + mis_negativos;
  hilos_pendientes = hilos_pendientes – 1;
  if ( hilos_pendientes == 0 ) {
    V (join);
  }
  V (mutex);
}

// Rutina principal
main () {
  rellenar_vector(); // introduce valores en “vector”
  for ( int i=0; i<N; i+=100 ) {
    Thread* hilo = new Thread();
    hilo->Fork ( buscar_negativos, i, i+100 );
  }
  // Espera a que todos los hilos finalicen
  P (join);
  printf ( “El vector contiene %d valores negativos\n”,
           negativos_encontrados );
}
```

Por último, mostraremos la misma solución utilizando un monitor. El monitor obliga a una mejor organización del código a través de la definición de un tipo abstracto de datos que regula el acceso a la variable «negativos_encontrados». La solución es más elaborada, pero al mismo tiempo más modular.

Como el C++ no soporta monitores, utilizaremos una sintaxis ficticia para especificar el monitor, similar en sintaxis a una clase de C++, pero con la semántica de un monitor. Esta solución es válida con semánticas Hoare y Mesa.

```
// El vector donde se realizará la búsqueda
const int N =  10000;
int vector [N];

// Monitor que encapsula negativos_encontrados
Monitor NegativosEncontrados {

  // Variable condición para sincronizar al hilo principal
  private: Condition join;

  // convertimos la variable global en un campo privado del monitor
  private: int negativos_encontrados;

  // variable que indica cuántos hilos están pendientes de terminar
  private: int hilos_pendientes;

  // Constructor
  public: NegativosEncontrados() {
    negativos_encontrados = 0;
    hilos_pendientes = N / 100;
  }

  // Operación para leer el resultado del programa
  public: int Valor() { return negativos_encontrados; }

  // Operación que invocan los hilos cuando terminan de rastrear el vector
  public: void Incrementa (int cuantos) {
    negativos_encontrados += cuantos;
    hilos_pendientes--;
    if ( hilos_pendientes == 0 ) {
      join.Signal();
    }
  }

  // Operación de sincronización del hilo principal
  public: void Espera () {
    if ( hilos_pendientes > 0 ) {
      join.Wait();
    }
  }
}

// Rutina que ejecutan los hilos concurrentes
void buscar_negativos ( int inf, int sup ) {
  int mis_negativos = 0;
  for ( int k=inf; k<sup; k++ ) {
    if ( vector[k] < 0 ) {
      mis_negativos = mis_negativos + 1;
    }
  }
  // escribe en el monitor los resultados
  NegativosEncontrados.Incrementa (mis_negativos);
}

// Rutina principal
main () {
  rellenar_vector(); // introduce valores en “vector”
  for ( int i=0; i<N; i+=100 ) {
    Thread* hilo = new Thread();
    hilo->Fork ( buscar_negativos, i, i+100 );
  }
  // Espera a que todos los hilos finalicen
  NegativosEncontrados.Espera();
  printf ( “El vector contiene %d valores negativos\n”,
           NegativosEncontrados.Valor() );
}
```