UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Departamento de Informática de Sistemas y Computadoras (DISCA)

fSO

Ejercicio de Evaluación
05 de Noviembre de 2012

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

APELLIDOS		**NOMBRE**		Grupo
DNI		**Firma**		

- **No desgrape las hojas.**
- **Conteste exclusivamente en el espacio reservado para ello.**
- **Utilice letra clara y legible. Responda de forma breve y precisa.**
- **El examen consta de 9 cuestiones, en cada una de ellas se indica su valoración.**

SOLUCIONES

---

1) ¿Se puede crear una aplicación de usuario cuyo código no incluya ninguna llamada al sistema? Justifique su respuesta. **0,75 puntos**

1	*No sería posible.* *Las aplicaciones de usuario necesitan acceder a datos del disco (read/write) y mostrar información por pantalla. Se trataría de una aplicación que no accede a los recursos hardware de la máquina.* *Se podría crear una aplicación que en ejecución, no realizara ningún acceso de entrada/salida, además dicha aplicación no podría terminar su ejecución normal con exit(). La única utilidad de un programa así sería la de ocupar tiempo de CPU (recuerde el programa "tragón" de practicas)*

---

2) Durante la inicialización o arranque del sistema operativo de un computador de sobremesa moderno, el procesador ¿podría encontrarse en modo usuario? Justifique la respuesta. **0,75 puntos**

2	*No podría.* *Durante el arranque del sistema es necesario cargar el núcleo del sistema operativo en memoria, para ello se debe acceder al disco o memoria secundaria y por tanto es necesario ejecutar instrucciones de E/S. Las instrucciones de E/S son privilegiadas y solo se ejecutan en modo núcleo.* *En UNIX tampoco sería posible, en el caso hipotético de que el código del sistema operativo estuviese siempre cargado en memoria principal. Al accionar el interruptor se ejecuta un programa que reside en memoria ROM, cuya misión es cargar en la memoria RAM otro programa con capacidad para localizar en el disco duro el código del núcleo del sistema operativo, llevarlo a memoria y posteriormente cederle el control. A partir de este momento el núcleo se encarga de inicializar los diferentes drivers, y crear un primer proceso que será el encargado de crear el resto de los procesos. La creación de procesos sólo se puede llevar a cabo en modo núcleo.*

3) El siguiente código corresponde al archivo ejecutable generado con el nombre "Ejemplo1".

```
/*** Ejemplo1***/
#include "todas_las_cabeceras_necesarias.h"
main()
{ int i=0;
  pid_t pid, pid2;
  while (i<2)
  { pid=fork()
     switch(pid)
      {case (-1):{printf("Error creando hijo\n");
                  break;}
       case (0):{pid2=fork();
                 printf("Hijo %i creado\n",i);
                 sleep(10);
                 exit(0);}
       default: {printf("Padre\n");
                 sleep(5);}
     }
    i++;
  }
  exit(0);
}
```

Suponga que "Ejemplo1" se ejecuta correctamente:

a) Indique de forma razonada, el número de procesos que creará y dibuje el árbol de procesos generado.

b) Indique de forma justificada, si pueden producirse procesos zombies y/o huérfanos.

1.5 puntos

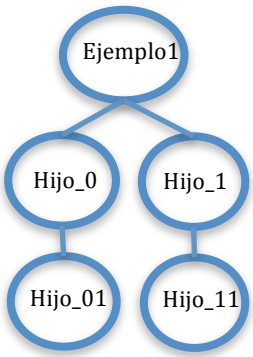
3

a)

Se crean un total de 5 procesos al ejecutar el código ejemplo1.

El proceso Ejemplo1 crea dos procesos hijos: con la llamada pid=fork() e i=0 crea Hijo0 a continuación ejecuta la opción "default" e incrementa i, vuelve a repetir el bucle y con i=1 crea otro proceso Hijo1.

Además tanto el Hijo0 como Hijo1 se convierten en padres ya que ellos ejecutan el código de la opción "case(0)" dentro del switch, donde aparece otra llamada fork(), tanto Hijo01, como Hijo11 y sus padres (Hijo0,Hijo1) ejecutan exit() y finalizan.



b) *En el código Ejemplo1, no aparece la llamada wait(). Los procesos padres no esperan a sus procesos hijos y por tanto podrían quedar procesos huérfanos, si los procesos padre finalizan su ejecución antes que sus hijos.*

sleep() hace que un proceso se suspendan voluntariamente.

El proceso inicial Ejemplo1 se suspende 5 segundos (sleep(5)) después de crear al Hijo_0 y otros 5 segundos después de crear a su Hijo_1, en total 10 segundos. Hijo_0 e Hijo_1 se suspende 10 segundos (sleep(10)), existe una alta probabilidad de que Ejemplo1 finalice su ejecución antes de que Hijo1 acabe. Por tanto Hijo1 queda huérfano y es adoptado por init().

Es bastante improbable que haya procesos zombies, ya que los procesos padres (Ejemplo1, Hijo_0, Hijo_1) aparentemente finalizan antes que sus hijos. No obstante en función del orden de ejecución de los hijos creados dentro del switch (Hijo_01 e Hijo_11) podrían darse zombies durante un periodo muy breve.

4) Indique de forma justificada si son ciertas o falsas las siguientes afirmaciones sobre los estados de los procesos:

a) Sólo los procesos que están en *Ejecución* pueden pasar al estado *Terminado*, una vez finaliza la ejecución de sus instrucciones.
b) El estado *Suspendido* sólo puede ser alcanzado por aquellos procesos que hayan solicitado o estén realizando una operación de E/S bloqueante
c) El cambio de estado de en Ejecución al de *Preparado* sólo es posible en sistemas con planificadores expulsivos

1.0 puntos

4

a)
FALSA
Desde el estado de Preparado y Suspendido un proceso puede pasar a Terminado si se le envía una señal. Un proceso puede finalizar de forma anormal mediante la señal SIGKILL. Los procesos reciben las señales incluso en estado suspendido, la señal SIGKILL no se puede enmascarar.

b)
FALSA
Los procesos pueden pasar al estado suspendido al ejecutar una instrucción bloqueante como wait()o sleep(). También pasan a SUSPENDIDO cuando solicitan un recurso SW que se encuentra ocupado (sem_wait, pthread_mutex_lock)

c)
CIERTA
Con planificadores expulsivos cuando un proceso es expulsado de la CPU, por otro mas prioritario, antes de finalizar su ráfaga de CPU, se inserta en la cola de preparados. En planificadores no expulsivos los procesos sólo abandonan la CPU para ir a los estados de acabado o suspendido.

5) Un sistema dispone de un planificador multicolas a corto plazo (PCP) con tres colas Cola0, Cola1 y Cola2, cuyos algoritmos de planificación son **RR con q=1, SRTF, y FCFS**, respectivamente.
La planificación entre **colas es gestionada con prioridades expulsivas** siendo la más prioritaria la Cola2 y la menos prioritario la Cola0. Cada proceso dispone de un contador de promoción (ContPro) que el sistema mantiene para establecer su promoción entre colas, de manera que los procesos puedan alcanzar colas más prioritarias. Cada vez que un proceso **pasa al estado suspendido** su ContPro se incrementa en 1 (ContPro= ContPro+1). Un proceso es ubicado en la Cola0 si su ContPro es igual a 0, en la Cola1 si su ContPro=1 y en la Cola2 si ContPro>=2. Los procesos que llegan al sistema se les asigna un ContPro=0 y van a la Cola0.
Suponga que las operaciones de E/S se efectúan sobre el mismo dispositivo gestionado con FCFS y que al sistema llegan los procesos mostrados en la tabla:

Proceso	Perfil de ejecución	Instante de llegada	ContPro
A	2 CPU + 2 E/S + 1 CPU + 4 E/S + 1 CPU	0	0
B	1 CPU + 2 E/S + 4 CPU + 2 E/S + 2 CPU	1	0
C	3 CPU + 1 E/S + 1 CPU	2	0
D	2 CPU + 1 E/S+ 1CPU	3	0

a) Rellene la siguiente tabla indicando en cada instante de tiempo donde se encuentran los procesos.
b) Indique la utilización de CPU para esta carga y los tiempos medios de espera y retorno.

2.0 puntos (1.25+0.75)

5a

T	Cola 0 RR q=1	Cola 1 SRTF	Cola 2 FCFS	CPU	Cola E/S	E/S	Evento
0	A			A			Llega A , ContPro(A)=0
1	A B			B			Llega B, ContPro(B)=0
2	CA			A		B	LlegaC, ContPro(C)=0 ContPro(B)=1
3	D C			C	A	B	Llega D ContPro(A)=1
4	CD	B		B		A	
5	CD			B		A	
6	CD	B A		A			
7	CD	B		B		A	ContPro(A)=2
8	CD			B		A	
9	CD			D	B	A	ContPro(B)=2
10	D C			C	B	A	
11	CD		A	A		B	
12	CD			D		B	FIN A
13	C		B	B		D	ContPro(D)=1
14	C	D		B			
15	C			D			FIN B
16	C			C			FIN D
17				---		C	ContPro(C)=1
18				C			
19							FIN C
20							
21							

5b

Tiempo medio de espera= 1+1+11+9 /4 =22/4 =5.5
Tiempo medio de retorno= (12-0)+(15-1)+(19-2)+(16-3)/4 = 56/4 =14
Utilización de CPU = 18/19

---

6) Dado el siguiente código cuyo archivo ejecutable ha sido generado con el nombre "Ejemplo2".

```
 1 /*** Ejemplo2***/
 2 #include <stdio.h>
 3 #include <pthread.h>
 4 void *fun_hilo( void *ptr )
 5 { int sec;
 6   sec=(int)ptr;
 7   sleep(sec);
 8   printf("Yo he esperado %d segundos\n",sec);
 9 }
10
11 int main()
12 { pthread_attr_t atrib;
13   pthread_t hilo1, hilo2, hilo3;
14
15   pthread_attr_init( &atrib );
16   pthread_create( &hilo1, &atrib, fun_hilo, (void *)30);
17   pthread_create( &hilo2, &atrib, fun_hilo, (void *)1);
18   pthread_create( &hilo3, &atrib, fun_hilo, (void *)10);
19   pthread_join( hilo3, NULL)
20 }
21
```

Indique las cadenas que imprime el programa en la terminal tras su ejecución. Justifique su respuesta

1.0 puntos

6

Yo he esperado 1 segundos
Yo he esperado 10 segundos

El hilo main únicamente espera con join() a hilo3. Por tanto el main terminara inmediatamente después de que termine el hilo3. Cuando el hilo main acaba se liberan todos los recursos del proceso, memoria principal, etc. y por tanto finalizan todos los hilos, independientemente de si han acabado de ejecutar todas sus instrucciones o no.
Dada la diferencia de tiempos en los sleep() que ejecutan los hilos, hilo3 finaliza antes que hilo1, por lo que nunca saldrá el mensaje del hilo1 que se suspende durante 30 segundos.

---

7) El siguiente código C corresponde a un proceso con dos hilos que comparten memoria y se han de sincronizar utilizando el **mecanismo de espera activa**. Para diseñar su **protocolo de entrada y salida** a la sección crítica utilizan las variables compartidas **flag** y **turn** declaradas.

```
1  //***** Ejemplo 3 **//
2  #include <pthread.h>
3  …
4  int flag[2];
5  int turn;
6
7  void* thread(void* id)
8  { int i;
9    i = (int)id;
10  while ( 1 ) {
11
12     remaining_section();
13
14     /**b)Asigne valores a flag y a turn**/
15     while(/**c)Condición del Protocolo de Entrada**/);
16
17     critical_section();
18
19     flag[i] = 0;
20   }
21 }
22 int main() {
23   pthread_t th0, th1;
24   pthread_attr_t atrib;
25   pthread_attr_init(&atrib);
26   /**a)Inicialice la variable flag **/
27   pthread_create(&th0, &atrib, thread, (void *)0);
28   pthread_create(&th1, &atrib, thread, (void *)1);
29   pthread_join(th0, NULL);
30   pthread_join(th1, NULL);
31 }
```

Se pide:

a) Reemplace "`/**a) Inicialice la variable flag **/`" por el código C correspondiente.

b) Reemplace "`/**b) Asigne valores a flag[i] y a turn**/`" por el código C correspondiente.

c) Reemplace "`/**c) Condicion del protocolo de entrada**/`" por el código C correspondiente.

1.0 puntos

7

a)

```
flag[0]=0;  flag[1]=0;
```

b)

```
flag[i] = 1;
turn =(i+1)%2;
```

c)

```
while (flag[(i+1)%2]==1) && (turn==(i+1)%2);
```

---

8) Indique todos los posibles valores que puede alcanzar la variable x tras la ejecución concurrente de los procesos A, B y C. Justifique su respuesta indicando para cada uno de valores el orden de ejecución de las distintas secciones.

// Variables compartidas
int x=1;
Semáforos S1=3, S2=0, S3=0;		

// Proceso A	//Proceso B	//Proceso C

P(S1)
P(S3)
x = 2*x + 1; // sección 1
V(S2)
	P(S1)
	P(S2)
	x = x*3; // sección 2
	V(S1)
		P(S1)
		x = x + 2; // sección 3
		V(S3)
		P(S1)
		x = x + 3; // sección 4

1.0 puntos

8

Existen al menos cuatro posibles ordenes de ejecución que son:

X=1

C P(S1), sección3, V(S3) -> A P(S1)P(S3) sección1, V(S2) -> B P(S1), P(S2) sección2 , V(S1) -> C P(S1) sección4 → valor de x=24

*x=3 → x= 3*2 +1 =7 → x= x*3= 21 → x= x+3=24*

x=1

C P(S1),sección3,V(S3), P(S1), sección4 → A P(S1), P(S3), sección1 → Proceso B suspendido en el semáforo S1

x=3 →x=6 → x=13

x=1

C P(S1), sección3,V(S3)→ A P(S1), P(S3), sección1, V(S2)→ C P(S1)sección4 → Proceso B suspendido en el semáforo S1

x=3 →x=7 → x=10

x=1

C P(S1), sección3,V(S3)→B P(S1)→ C P(S1), sección4→ A P(S1)A se Suspende en S1→ B P(S2), B suspendido en el semáforo S2

x=3 →x=7

x=1

C P(S1), sección3,V(S3)P(S1) seccion4→B P(S1)P(S2) B suspendido en S2→A P(S1),A suspendido en el semáforo S1

x=3 →x=7

9) Complete en "Ejemplo4" el código de las funciones fth_UNO y fth_DOS, con las operaciones sobre **semáforos** necesarias, para que al ejecutarlo muestre por pantalla de manera ordenada, los diez primeros números enteros, es decir, " 0 1 2 3 4 5 6 7 8 9" .

```
 1 /*** Ejemplo4***/
 2 #include <stdio.h>
 3 #include <pthread.h>
 4
 5 void *fth_UNO( void *ptr )
 6 { int i;
 7   for (i=1; i<10; i+=2)
 8   {
 9     printf("%d ",i);
10
11   }
12 }

13 void *fth_DOS( void *ptr )
14 { int i;
15 for (i=0; i<10; i+=2)
16   {
17     printf("%d ",i);
18
19   }
20 }

20 int main()
21 { pthread_attr_t atrib;
22   pthread_t th1, th2;
23
24   pthread_attr_init( &atrib );
25   pthread_create(&th1,&atrib,fth_UNO, NULL);
26   pthread_create(&th2,&atrib,fth_DOS, NULL);
27   pthread_join( th1, NULL);
28   pthread_join( th2, NULL);
29   printf ("\n");
30 }
```

El hilo "th1" debe mostrar los números impares y "th2" los pares. Para sincronizar la ejecución de los hilos utilice tantos semáforos como sea necesario y diga su valor de inicialización. Utilice la nomenclatura P y V.

1.0 puntos

9

Semáforo: sinc, smutex;
sinc=0; smutex=1;

```
void *fth_UNO( void *ptr )
{ int i;
  for (i=1; i<10; i+=2)
  { P(sinc);
    printf("%d ",i);
    V(smutex);
  }
}
```

```
void *fth_DOS( void *ptr )
{ int i;
for (i=0; i<10; i+=2)
  { P(smutex);
    printf("%d ",i);
    V(sinc);
  }
}
```