EEE1: Ejercicio de Evaluación

5 de Noviembre de 2018

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 7 cuestiones, cuya valoración se indica en cada una de ellas.**

---

1. En cada apartado, defina brevemente el primer término que se cita y relacione con él la sentencia en lenguaje C u orden del sistema que se da a continuación.

(1.5 puntos=0.5+0.5+0.5)

1	a) Llamada al sistema (defina)
	Las llamadas al sistema son un mecanismo para solicitar servicios al sistema operativo (SO). Se trata de una interfaz entre las aplicaciones y el núcleo del SO que facilita el acceso a los recursos hardware y software del sistema. Al ejecutar una llamada al sistema se provoca una interrupción software o Trap que cambia el modo de ejecución de modo usuario a modo núcleo e interviene el SO
	printf ("Este mensaje se imprime en la salida estándar \n"); //en un programa (relacione)
	Para que las llamadas al sistema puedan ser invocadas de forma cómoda y portable desde las aplicaciones escritas en C, se proporcionan una serie de funciones como printf() que forman parte de las librerías (bibliotecas) estandarizadas del lenguaje C.

	b) Modos de ejecución usuario y núcleo (defina)
	Son distintos modos de funcionamiento del procesador. Cuando el sistema ejecuta un programa de usuario, en modo usuario, se restringe la ejecución de ciertas instrucciones relacionadas con el acceso a algunos recursos críticos. Acceder a estos recursos es responsabilidad exclusiva del núcleo del SO que se ejecuta en modo núcleo, en el cual no se aplican estas restricciones. Se entra en modo núcleo siempre que se invoca un servicio del núcleo a consecuencia de una excepción o interrupción.
	b= a / 0; //división por cero en un programa de usuario (relacione)
	La ejecución de esta sentencia provoca una división por cero. Se trata de un error que provocará una excepción que hará que la ejecución se interrumpa, el procesador salga del modo usuario, entre en modo núcleo y se invoque a la rutina de servicio del SO correspondiente para el tratamiento de este error.

	c) Tiempo compartido (defina) Los sistemas de tiempo compartido son sistemas multiprogramados en los que además se posibilita la interacción directa entre el usuario y la máquina . $ ls -la | wc -l (relacione) Mediante esta línea de órdenes se lanzan dos procesos en foreground o interactivamente simultáneamente (que intercambian un flujo de información entre ellos). Estos procesos progresan concurrentemente, alternándose en el uso de la CPU gracias al mecanismo de tiempo compartido.

2. **D**ado el programa ***experiment.c*** cuyo archivo ejecutable es "***experiment***".

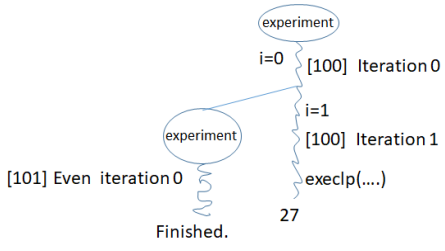
```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/wait.h>
 5
 6 int main(int argc, char *argv[]) {
 7   pid_t  pid;
 8   int    i, N=4;
 9
10   for (i=0; i<N; i++)
11   { printf("[%d] Iteration %d\n", getpid(),i);
12     if (i%2 == 0)
13      {   pid = fork();
14          if (pid == 0) {
15            printf("[%d] Even iteration (%d)\n", getpid(),i);
16            break; }
17          else if (pid == -1) {
18            printf ("Error creating the process\n");
19             exit(-1); }
20       }else
21         if (execlp("wc","wc","-l","experiment.c",NULL)<0)
22              printf("Error running wc\n");
23   }
24  while (wait(NULL)!=-1);
25   printf("[%d] Finished.\n", getpid());
26   exit(0);
27 }
```

Nota: ***a % b*** devuelve el resto de la división entera ***a / b***.

(1.4 puntos=0,7+0,7)

a) Suponga que las llamadas al sistema no producen error e indique de forma justificada el número de procesos creados y el parentesco que existe entre ellos al ejecutar *experiment*

En la iteración ***i=0***, ***i es*** par cunmpliendose la condición (línea 12), el padre hace fork() y crea un proceso hijo que saldrá del bucle (break) y finaliza. En la segunda iteración, ***i=1***, no se cumple la condición y el padre hace el ***execlp()*** cambiando su código por el de ***wc***, que tras ejecutarlo finaliza. Por tanto, sólo se crean dos procesos: el proceso inicial (padre) y un proceso hijo de éste.



b) Suponga que el PID del proceso *experiment* es 100, y que el sistema operativo asigna los PIDs 101, 102, … a sus posibles procesos hijos. Escriba una de las posibles secuencias que se mostraría en pantalla al ejecutar el programa.

[100] Iteration 0
[101] Even iteration 0
[101] Finished.
[100] Iteration 1
27 //Resultado de ejecutar la orden "wc -l experiment.c"

3. **D**ado el programa ***proc.c*** cuyo archivo ejecutable es "***proc***".

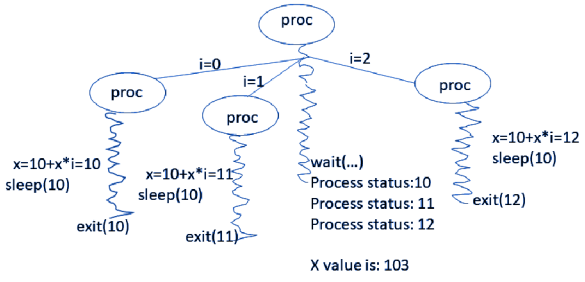
```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #define N 3
4  int X=1;  //global variable
5
6  int main(int argc,char *argv[]) {
7     pid_t pid;
8     int   i,status;
9
10    for (i=0; i<N; i++) {
11       pid = fork();
12       if (pid==0) {
13              X=10+X*i;
14              sleep(10);
15              exit(X);
16       }
17    }
18    X=100+i;
19    while (wait(&status)>0)
20       printf("Process status: %d\n", status/256);
21
22    printf("X value is: %d \n",X);
23    return(0);
}
```

Nota: ***a % b*** devuelve el resto de la división entera ***a / b***.

(1.2 puntos=0,5+0,5+0.2)

a) Indique de forma justificada los procesos que se crean al ejecutar *proc*, el parentesco existente entre ellos y los mensajes que se imprimen en el terminal.

proc crea 4 procesos, un padre y 3 hijos.
Todos los hijos hacen exit(X) (línea 15), y finalizan sin crear nuevos procesos, mientras que el padre espera a todos los hijos (línea 19). Lo que muestra *proc* en el terminal es:
Process status: 10
Process status: 11
Process status: 12
X value is: 103



b) Dada las instrucciones de las líneas 13, 14, 15 y 19, marque el estado(s) en que podría estar el proceso(s) tras ejecutar dicha instrucción y antes de ejecutar la siguiente instrucción

Línea	Suspendido	Preparado	En ejecución	Terminado	Zombie	Huerfano
13		X	X			
14	X					
15				X		
19	X	X	X			

c) Indique de forma justificada si puede haber o no condición de carrera en el acceso a la variable global X

Al crear el proceso con fork(), cada proceso tendrá su propia copia de la variable X, X no es compartida, y por tanto no hay problemas de condición de carrera en el acceso a X.

4. En un sistema de tiempo compartido su planificador a corto plazo consta de dos colas, una gestionada con SRTF (ColaS), en la que en caso de empate se utiliza el orden de llegada como criterio de desempate, y otra gestionada con Round Robin (ColaR) con un quantum de 2 unidades de tiempo (q=2 ut). Los procesos nuevos y los procedentes de E/S siempre van a la ColaR. La ColaR es la más prioritaria y un proceso es degradado a la cola ColaS tras consumir un quantum q en CPU. La gestión intercolas es por prioridades expulsivas. Todas las operaciones de E/S se realizan en un único dispositivo con política de servicio FCFS.
En este sistema se solicita ejecutar el siguiente grupo de trabajos:

Proceso	Instante de llegada	Ráfagas de CPU y E/S
A	0	5 CPU + 2 E/S + 1 CPU
B	1	4 CPU + 4 E/S + 4 CPU
C	2	3 CPU + 1 E/S + 5 CPU

(2.0puntos =1.4+ 0.6)

4 a) Indique diagrama de uso de CPU, rellenando la tabla adjunta. Los procesos aparecerán en las colas ordenados en base al criterio de planificación, de forma que el proceso de la derecha será el primero. Además en cada instante de tiempo t ponga entre paréntesis (ej. (A)), en la cola de la que provenga, el proceso al que asigne la CPU.

T	ColaR-->	ColaS-->	CPU	Cola E/S-->	E/S	Evento
0	(A)		A(4)			Llega A
1	B(4)		A(3)			Llega B
2	C(3)(B)	A(3)	B(3)			Llega C
3	C(3)	A(3)	B(2)			
4	(C)	A(3)B(2)	C(2)			
5		A(3)B(2)	C(1)			
6		A(3)B(2)(C)	C(0)			C es el más corto
7		A(3)(B)	B(1)		C(0)	
8	(C)	A(3)B(1)	C(4)			
9		A(3)B(1)	C(3)			
10		C(3)A(3)(B)	B(0)			
11		C(3)(A)	A(2)		B(3)	empate: orden cola
12		C(3)	A(1)		B(2)	
13		C(3)	A(0)		B(1)	
14		(C)	C(2)	A(2)	B(0)	
15	(B)	C(2)	B(3)		A(1)	
16		C(2)	B(2)		A(0)	
17	(A)	B(2)C(2)	A(0)			
18		B(2)(C)	C(1)			Fin A, empate
19		B(2)	C(0)			
20			B(1)			Fin C
21			B(0)			
22						Fin B
23						

4 b) Indique los tiempos de espera y de retorno de cada proceso

	T.espera	T.retorno
A	**9**	18-0 = **18**
B	**9**	22-1 = **21**
C	**9**	20-2 = **18**

/*******Valenciá **********/

En este sistema se solicita ejecutar el siguiente grupo de trabajos:

Proceso	Instante de llegada	Ráfagas de CPU y E/S
A	0	5 CPU + 2 E/S + 1 CPU
B	1	4 CPU + 4 E/S + 4 CPU
C	2	2 CPU + 1 E/S + 5 CPU

(2.0puntos =1.4+ 0.6)

4 a) Indique diagrama de uso de CPU, rellenando la tabla adjunta. Los procesos aparecerán en las colas ordenados en base al criterio de planificación, de forma que el proceso de la derecha será el primero. Además en cada instante de tiempo t ponga entre paréntesis (ej. (A)), en la cola de la que provenga, el proceso al que asigne la CPU.

T	ColaR-->	ColaS-->	CPU	Cola E/S-->	E/S	Evento
0	(A)		A(4)			Llega A
1	B(4)		A(3)			Llega B
2	C(2)(B)	A(3)	B(3)			Llega C
3	C(2)	A(3)	B(2)			
4	(C)	A(3)B(2)	C(1)			
5		A(3)B(2)	C(0)			
6		A(3)(B)	B(1)		C(0)	B más corto
7	(C)	A(3)(B)	C(4)			
8		A(3)B(1)	C(3)			
9		c(3)A(3)(B)	B(0)			
10		C(3)(A)	A(2)		B(3)	empate: orden cola
11		C(3)	A(1)		B(2)	
12		C(3)	A(0)		B(1)	
13		(C)	C(2)	A(2)	B(0)	
14	(B)	C(2)	B(3)		A(1)	
15		C(2)	B(2)		A(0)	
16	(A)	B(2)C(2)	A(0))			
17		B(2)(C)	C(1)			FIN A, empate
18		B(2)	C(0)			
19		B(2)	B(1)			Fin C
20			B(0)			
21						Fin B
22						
23						

4 b) Indique los tiempos de espera y de retorno de cada proceso

	T.espera	T.retorno
A	**8**	17-0 = **17**
B	**8**	19-1 = **18**
C	**9**	21-2 = **19**

5. Dado el siguiente programa ***endth.c*** cuyo ejecutable es ***endth***.

(1.1 puntos=0.8 +0.3)

```
#include <... all headers...>
pthread_t  th1, th2;
pthread_attr_t atrib;
int N=0;

void *Func2(void *arg)
{ int j=2;
  sleep(5);
  printf("END THREAD 2 \n");
  pthread_exit(&j);
}
```

```
void *Func1(void *arg)
{ int j=1;
  pthread_create(&th2, &atrib,Func2,NULL);
  N=N+j;
  sleep(20);
  printf("END THREAD 1 \n");
  pthread_exit(&j);
}
```

```
int main (int argc, char *argv[])
{ int b;

  printf("START MAIN \n");
  pthread_attr_init(&atrib);
  pthread_create(&th1, &atrib, Func1,NULL);
  N= N+10;
  sleep(10);
  pthread_join(th2,(void**) &b);
  printf("END MAIN \n");
  exit(0);
}
```

5 a)**I**ndique las cadenas que imprime el programa en la Terminal tras su ejecución. Justifique su respuesta

START MAIN
END THREAD 2
END MAIN

El hilo main finaliza tras ejecutar exit(), por lo tanto, el proceso (proceso pesado) finaliza sin que de tiempo a th1 a finalizar ejecutar el printf

b)**J**ustifique si hay o no posibilidad de que ocurra una condición de carrera en el acceso a las variables N, j y b del programa.

La única variable compartida por los hilos (th1 y main) es la variable global N y por tanto es la única que podría llegar a producir condición de carrera. Las variables j y b son locales a los hilos y por tanto cada uno tiene su copia.

6. **I**ndique cuáles de las siguientes afirmaciones sobre sección crítica son ciertas y cuáles son falsas. Debe justificar adecuadamente sus respuestas, indicando qué cumplen, qué no cumplen y porqué:

(1.2 puntos=0.4+0.4+0.4)

6

a) Si se utiliza un protocolo basado en semáforos para resolver el problema de la sección crítica, como se muestra a continuación, el valor inicial del semáforo S ha de ser 0

```
Protocolo Entrada   → P(S)
                      Sección crítica();
Protocolo de salida → V(S)
```

FALSO

Para resolver el problema de sección crítica (SC) el semáforo debe estar inicializado a 1, con el fin de que cumpla la condición de exclusión mutua, progreso y espera limitada. De manera que sólo un proceso/hilo pueda estar ejecutando código de sección crítica en cada instante

b) No siempre podemos utilizar la solución hardware basada en las instrucciones de Deshabilitar/Habilitar interrupciones, que plantea el código siguiente:

```
Protocolo Entrada   → DI (Deshabilitar)
                      Sección crítica();
Protocolo de salida → EI (Habilitar)
```

CIERTO

La solución hardware basada en las instrucciones DI/EI solo es válida para hilos del núcleo del sistema operativo, ya que estas instrucciones son privilegiadas y sólo se pueden ejecutarse en modo núcleo. Además está solución no sería adecuada para sistemas con más de una CPU, ya que deberían deshabilitarse en todos los núcleos. Esta solución sólo cumple exclusión mutua y progreso, pero no cumple espera limitada, ya que nos se garantiza ningún orden/criterio de acceso a sección crítica

c) La solución basada en la instrucción *test_and_set* que aparece en el cuadro cumple las condiciones de Exclusión mutua, Progreso y Espera limitada, con independencia del planificador que se utilice

```
//llave es una variable global e inicializada a 0
Protocolo Entrada   → while (test_and_set(&llave));
                      Sección crítica();
Protocolo de salida → llave = 0;
```

FALSO

Precisamente la solución de Test&Set, al utilizar la espera activa dentro del protocolo de entrada a la sección crítica, es muy dependiente del planificador, pudiendo llegar a provocar situaciones de bloqueo del sistema en planificadores no equitativos. Esta solución no cumple el requisito de espera limitada, aunque sí cumple el de exclusión mutua y progreso

7. El siguiente código es una variación del ejercicio de suma de filas visto en la práctica de threads. En concreto, en este programa son los hilos sumadores (función **AddRow**) y no el hilo **main** el que se encargan de sumar a la variable **total_addition** el resultado de la suma de la fila. Para ello, se han realizado las siguientes modificaciones:

- Se ha eliminado de la estructura **row** el elemento **addition**.
- La variable **total_addition** se ha declarado global.
- Se ha declarado también la variable local **local_addition** en **AddRow**.
- Se ha declarado el mutex **m**.

```
1   #include <...all_headers...>
2   #define DIMROW 1000000
3   #define NUMROWS 20
4   typedef struct row{
5        int vector[DIMROW];
6   } row;
7   row matrix[NUMROWS];
8   long total_addition = 0;
9   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
11
12  void *AddRow( void *ptr ) {
13     int k;
14     int local_addition;
15     row *fi;
16     fi=(row *)  ptr;
17     local_addition=0;
18     for(k=0;k<DIMROW;k++) {
19          total_addition += fi->vector[k];
20     }
21  }
22  int main() {
23     int i,j;
24     pthread_t  threads[NUMROWS];
25     for(i=0;i<NUMROWS;i++) {
26          for(j=0;j<DIMROW;j++) {
27              matrix[i].vector[j]=1;
28          }
29     }
30     pthread_attr_init( &atrib );
31     for(i=0;i<NUMROWS;i++) {
32         pthread_create(&threads[i],NULL,AddRow,(void *)&matrix[i]);
33     }
34     for(i=0;i<NUMROWS;i++) {
35        pthread_join(threads[i],NULL);
36     }
37     printf(  "Total addition is: %ld\n",total_addition);
38  }
```

Responda a las siguientes cuestiones.

(1.6 puntos= 0.4+0.6+0.6)

7	a) Identifique, en dicho programa, la(s) línea(s) correspondientes a la sección crítica. La línea con sección crítica (S.C.) es la 19, ya que en ella se accede a una variable global compartida por varios hilos: `(total_addition += fi->vector[k];)`

b) Utilice el mutex **m** para proteger la sección crítica e indique de forma justificada qué desventaja puede surgir a nivel de concurrencia con esta solución.

```
for(k=0;k<DIMROW;k++) {
            pthread_mutex_lock(&m);
            total_addition += fi->vector[k];
            pthread_mutex_unlock(&m);
      }
```

Esta solución es poco eficiente ya que continuamente se debe acceder a la S.C para cada elemento a sumar, con lo que se limita severamente la concurrencia en la ejecución de los hilos

c) Sin tener en cuenta los cambios realizados en el punto b, escriba el código necesario para modificar el programa de manera que: siga realizando la suma dentro del bucle pero usando la variable **local_addition** y después del bucle esta variable se sume a **total_addition**. Si es necesario,en su solución, proteja la sección crítica e indique de forma justificada qué ventaja tiene esta solución respecto a la propuesta en el apartado b.

```
for(k=0;k<DIMROW;k++) {
     local_addition += fi->vector[k];
 }
 pthread_mutex_lock(&m);
 total_addition += local_addition;
 pthread_mutex_unlock(&m);
```

Es una solución más eficiente que la del apartado b). Al acceder únicamente a la variable compartida después de la suma, se reduce drásticamente el número de accesos a la S.C y por lo tanto existe una mayor nivel de concurrencia entre los hilos que suman.