

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, cuya valoración se indica en cada una de ellas.

1. Un computador tiene una carga constituida por procesos largos con CPU intensiva, procesos largos en los que predomina la entrada-salida y procesos cortos que hacen principalmente uso de la CPU. ¿Mejoran la productividad y el tiempo medio de retorno cuando se pasa de monoprogramación a multiprogramación con planificación FCFS? ¿y cuando se pasa de multiprogramación con FCFS a tiempo compartido con planificación round-robin? Justifique su respuesta rellenando la siguiente tabla:

(1 punto=0,5+0,5)

1	Productividad	Tiempo de retorno medio
De MONOprogramación a MULTiprogramación	(0,25) La productividad MEJORA ya que la multiprogramación permite el solapamiento de las actividades de CPU y de entrada-salida entre varios procesos que se ejecutan de forma concurrente, con esto se consigue que tanto la CPU como los dispositivos de entrada-salida tengan una mayor utilización y por lo tanto el sistema finaliza más trabajos por unidad de tiempo.	(0,25) En un sistema multiprogramado los trabajos no tienen que esperar necesariamente en la cola de entrada al sistema, las esperas se tiene en las colas asociadas a la CPU y los dispositivos de entrada-salida. Estas esperas serán menores que la que se tiene en un sistema monoprogramado debido a la utilización simultánea de la CPU y de los dispositivos de entrada salida que posibilita la multiprogramación. Por lo tanto el tiempo de retorno medio MEJORA.
De MULTiprogramación a tiempo compartido	(0,25) Desde el punto de vista de la productividad un sistema de tiempo compartido es equivalente a un sistema multiprogramado ya que el limitar el tiempo que un proceso puede utilizar la CPU una vez se le asigna no va a modificar el nivel de ocupación de la misma por lo tanto la productividad NO MEJORA.	(0,25) Al limitar el tiempo en que un proceso puede utilizar la CPU cuando se le asigna hace que los trabajos con accesos largos de CPU no la monopolicen, dando opción a los trabajos cortos a ocupar la CPU antes, y por lo tanto con menos espera. La espera que esto provoca en los trabajos con accesos largos de CPU es relativamente cortas por lo que en promedio el tiempo de retorno medio MEJORA

2. Sea un sistema informático dotado de un sistema operativo multiprogramado. Indique de forma razonada la relación existente entre los siguientes conceptos:

(0,5 puntos=0,25+0,25)

2	<p>a) Interrupción y llamadas al sistema</p> <p>La relación existente entre las interrupciones y las llamadas al sistema es que las llamadas al sistema al sistema son un tipo específico de interrupciones denominadas TRAPs</p>
	<p>b) Llamadas al sistema y modos de funcionamiento del procesador</p> <p>Los modos de funcionamiento del procesador ofrecen la posibilidad de que sea solo el sistema operativo el que pueda ejecutar las instrucciones privilegiadas de acceso a los recursos. Para que los procesos de usuario puedan acceder a los recursos deben solicitarlo al sistema operativo a través de la interfaz de llamadas al sistema, las cuales al ser invocadas generan una TRAP haciendo, a través de un cambio de modo, que sea el sistema operativo el que finalmente ejecute el código asociado a la tarea</p>

3. Dado el siguiente código cuyo archivo ejecutable ha sido generado como “Prueba”.

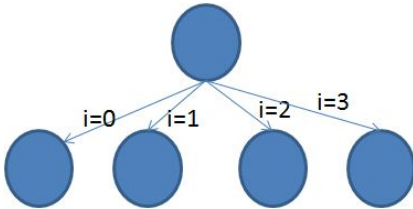
```

1  /*** Prueba.c ***/
2  #include "todos los necesarios"
3  #define N 4
4  main() {
5      int i = 0;
6      int pid;
7
8      for (i = 0; i < N; i++) {
9          pid = fork();
10         if (pid == 0) {
11             printf("Valor i = %d \n", i);
12             sleep(i);
13             exit(1);
14         }
15         sleep(2*i);
16     }
17     exit(0);
18 }

```

Suponga que se ejecuta sin errores e indique de forma justificada:

(1 punto=0,5+0,5)

3	<p>a) El número de procesos que se generan al ejecutarlo y el parentesco existente entre ellos</p> <p>Se crean un total de 5 procesos: 4 proceso hijos y un único padre. Los hijos después de ejecutar el printf y sleep hacen exit(1), por lo tanto no se convierten en padre.</p> 
	<p>b) Indique para cada uno de los procesos generados si podría llegar a estar zombie o huérfano</p> <p>Los hijos se quedan zombies durante un intervalo de tiempo corto. Cuando un hijo finaliza su padre no está esperándolo, sino que está ejecutando código o suspendido. El hijo creado con i=0 es el que más tiempo zombie estará.</p> <p>No hay huérfanos ya que el padre finaliza después de que los hijos hayan finalizado.</p>

4. El siguiente código corresponde al archivo ejecutable generado con el nombre “Ejemplo1”.

```

    /** Ejemplo1.c */
1  #include "todas_las_cabeceras_necesarias.h"
2  #define N 3
3  main() {
4      int i = 0;
5      pid_t pid;
6
7      while (i < N) {
8          pid = fork();
9          if (pid == 0) {
10             printf("Mensaje 1: i = %d \n", i);
11             if (i == N-2) {
12                 execl("/bin/ps", "ps", "-la", NULL);
13                 printf("Mensaje 2: i = %d \n", i);
14                 exit(1);
15             }
16             } else {
17                 printf("Mensaje 3: i = %d \n", i);
18                 while (wait(NULL) != -1);
19             }
20             i++;
21         }
22         printf("Mensaje 4: i = %d \n", i);
23         exit(0);
24     }

```

Suponga que “Ejemplo1” se ejecuta correctamente:

(1.5 puntos=0,75+0,75)

4a

Se generan 6 procesos en total. El proceso original genera un hijo en $i=0$. A partir de ahí cada proceso (padres e hijos) generan un hijo en cada iteración ($i=1$), ($i=2$). Solamente los procesos generados en la iteración ($i=1$) no generan ningún proceso hijo porque ejecutan un `exec` y ejecutan el comando `ps`

4 b	b) Indique qué valores de “i” se muestran al imprimir cada mensaje	
	Mensaje 1	$i=0, i=1, i=1, i=2, i=2$
	Mensaje 2	-
	Mensaje 3	$i=0, i=1, i=1, i=2, i=2$

	Mensaje 4	i=3, i=3, i=3, i=3
--	------------------	---------------------------

5. En un computador se inician simultáneamente dos procesos A y B, cuyo perfil es el siguiente:

A: CPU (2 segundos)

B: CPU (1 segundo) + E/S (1 segundo) + CPU (1 segundo)

La carga debida a los otros procesos del computador es despreciable. ¿Cuál será el tiempo de retorno y el tiempo de espera de cada uno de los procesos bajo las siguientes políticas de planificación?

(0.8 puntos)

5					
		Tiempo de retorno		Tiempo de espera	
		A	B	A	B
	Prioridad expulsiva B>A	4	3	2	0
	Round Robin (q=1 ms)	3	4	1	1

6. Diga en qué estado o estados entre Nuevo (N), Preparado (P), En Ejecución (EE), Suspendido (S) y Terminado (T) puede encontrarse un proceso cuando se dan las situaciones siguientes:

(0.7 puntos)

6		N	P	EE	S	T
	Un proceso que va a solicitar una llamada al sistema			X		
	Un proceso que está accediendo a un dispositivo de E/S				X	
	Un proceso que acaba de finalizar un quantum de CPU		X			
	Un proceso que acaba de finalizar su primera ráfaga de CPU				X	
	Un proceso que acaba de ejecutar la llamada wait()			X	X	
	Un proceso que acaba de ejecutar la llamada exit()					X
	Un proceso que acaba de ser creado con una llamada fork()		X	X		

7. El planificador a corto plazo de un sistema operativo de tiempo compartido gestiona procesos interactivos (PI) y procesos lanzados a través de la red (PR). Este planificador se basa en dos colas: ColaPI y ColaPR. La planificación entre colas es por prioridades expulsivas siendo la ColaPI la más prioritaria. La política de cada cola es: ColaPI algoritmo FCFS y ColaPR algoritmo Round Robin con $q=1$ ut. Los procesos tipo PI siempre van a la ColaPI y los procesos PR siempre van a la ColaPR. Debe considerar que el orden de llegada de los procesos a las colas es: en primer lugar los nuevos, a continuación los procedentes de E/S y por último los que provienen de CPU. Todas las operaciones de E/S se realizan en un único dispositivo de E/S con política de servicio FCFS.

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

Proceso	Instante de llegada	Tipo	Ráfagas de CPU y E/S
AI	0	PI	2CPU+4E/S+1CPU
BR	1	PR	4CPU+1E/S+ 5CPU
CR	3	PR	6CPU+2E/S+ 1CPU
DI	5	PI	3CPU+3E/S+1CPU

Indique el diagrama de uso de CPU, rellenando la tabla adjunta para cada instante de tiempo.

(1.5 puntos)

T	ColaPR	ColaPI	CPU	Cola E/S	E/S	Evento
0		(A)	A			Llega A
1	B		A			Llega B
2	(B)		B		A	
3	B, (C)		C		A	Llega C
4	C, (B)		B		A	
5	B, C	(D)	D		A	llega D
6	B, C	A	D			
7	B, C	A	D			
8	B, C	(A)	A		D	
9	B, (C)		C		D	FIN A
10	C, (B)		B		D	
11	B, C	(D)	D			
12	B, (C)		C			FIN D
13	C, (B)		B			
14	(C)		C		B	
15	C, (B)		B			
16	B, (C)		C			
17	C, (B)		B			
18	B, (C)		C			
19	(B)		B		C	
20			B		C	
21	B, (C)		C			
22	(B)		B			Fin C
23						Fin B
24						

8. El siguiente programa (a completar) crea una imagen `img` en forma de matriz bidimensional (el valor de cada elemento de la matriz corresponde al brillo de cada píxel) y le aplica un procesamiento para hacer su negativo (con ayuda de una función de nombre `Negative`) y otro que la convierte en imagen binaria (mediante una función `Binarize`). Las funciones de hilo `Negative` y `Binarize` están escritas de manera que deben recibir un puntero al primer píxel de una mitad de imagen a transformar (el de índice `[0][0]` para la mitad superior de la imagen y de índice `[ROWS/2][0]` para la inferior) y transforman la mitad de imagen que comienza a partir de ese elemento.

<pre> 1 #include <stdio.h> 2 #include <pthread.h> 3 #define COLUMNS 1920 4 #define ROWS 1080 5 #define byte unsigned char 6 7 byte img[ROWS][COLUMNS]; 8 pthread_attr_t attrib; 9 pthread_t thread_negative[2]; 10 pthread_t thread_binarize[2]; 11 12 void *Negative(void *ptr){ 13 byte *p= (byte*)ptr; 14 int i; 15 for(i=0;i<COLUMNS*ROWS/2;i++,p++){ 16 //do negative of pixel pointed by p 17 *p= 255-*p; 18 } 19 } 20 void *Binarize(void *ptr){ 21 byte *p= (byte*)ptr; 22 int i; 23 for(i=0;i<COLUMNS*ROWS/2;i++,p++){ 24 //binarize pixel pointed by p 25 *p= *p>127? 255 : 0; 26 } 27 } </pre>	<pre> 28 int main() 29 { 30 int x,y,i; 31 //Init image 32 for(y=0;y<ROWS;y++){ 33 for(x=0;x<COLUMNS;x++){ 34 img[y][x]= 35 255*(x+y)/(ROWS+COLUMNS); 36 37 pthread_attr_init(&attrib); 38 39 //Start concurrent Negative threads 40 for(i=0;i<2;i++){ 41 //{ ... } 42 43 //Wait for Negative threads ending 44 for(i=0;i<2;i++){ 45 //{ ... } 46 47 //Start concurrent Binarize threads 48 for(i=0;i<2;i++){ 49 //{ ... } 50 51 // ... 52 } </pre>
--	---

Nota: nomenclatura algunas de las funciones POSIX para hilos

int pthread_attr_init(pthread_attr_t *attr); int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

int pthread_exit(void *exit_status); int pthread_join(pthread_t thread, void **exit_status);

Haciendo uso de las variables ya declaradas en el programa:

(1.5 puntos=0,25+0,25+0,25+0,25+0,5)

8	<p>a) Escriba la línea de código requerida en la línea 40 para lanzar concurrentemente los hilos Negative necesarios para realizar la transformación de negativo de la imagen completa.</p> <pre>pthread_create(&thread_negative[i], &attrib, Negative, (void*) &img[i*ROWS/2][0]);</pre>
	<p>b) Escriba la línea de código necesaria en la línea 44 para la espera de los hilos Negative.</p> <pre>pthread_join(thread_negative[i], NULL);</pre>
	<p>c) Escriba el código necesario en la línea 50 para conseguir la adecuada terminación del programa.</p> <pre>pthread_exit(NULL); o bien for(i=0;i<2;i++) pthread_join(thread_binarize[i], NULL);</pre>
	<p>d) ¿Cuál es el máximo número de hilos pertenecientes a este proceso que pueden llegar a ejecutarse concurrentemente?</p> <p>3 hilos: el de la función main y los dos hilos Negative (o los dos Binarize, cuando los Negative hayan acabado)</p>
	<p>e) ¿Se completaría igualmente la transformación de la imagen de forma correcta si no se añadiera el código de espera de las líneas 43 y 44? Tenga en cuenta que es indiferente en cuanto a la imagen final obtenida hacer primero el negativo de un píxel y luego su binarización que lo contrario (el resultado sería el mismo).</p> <p>Es necesario esperar para asegurar que no haya hilos Negative y Binarize ejecutándose concurrentemente trabajando sobre la misma mitad de imagen, ya que podrían intentar operar simultáneamente sobre un mismo elemento de la matriz de imagen, posibilitándose la aparición de condiciones de carrera. Además, si c) se solucionó con la segunda opción (esperas para los hilos Binarize), eliminar las esperas de los Negative podría hacer que éstos no terminaran de ejecutarse y por tanto no se completara correctamente la transformación.</p>

9. El objetivo del siguiente código es sumar una matriz de NxM elementos, y para ello se crean N hilos, donde cada hilo suma una fila de la matriz, actualizando al finalizar, la variable suma_total.

(1.5 puntos=0,75+0,75)

1	float Mat[N][M];	19	int main() {
2	float suma_total = 0;	20	pthread_attr_t attr;
3	int llave = 0;	21	pthread_attr_init(&attr);
4		22	pthread_t t[N];
5	void *suma_fila (void *arg) {	23	int i;
6	int col;	24	
7	int fila = (int)arg;	25	for (i = 0; i < N; i++) {
8	float suma;	26	pthread_create(&t[i],
9	while(test_and_set(&llave));		&attr, suma_fila, i);
10		27	}
11	for (col = 0; col < M; col++) {	28	for (i = 0; i < N; i++) {
12	suma = suma + Mat[fila][col];	29	pthread_join(t[i], NULL);
13	}	30	}
14		31	printf("Suma = %f\n", suma_total);
15	suma_total = suma_total + suma;	32	}
16	llave = 0;		
17			
18	}		

- a) Indique cuáles de las siguientes sentencias son verdaderas (V) y cuáles falsas (F) (Nota: Un error penaliza una respuesta correcta).

9a		V/F
	La solución planteada garantiza que el código está libre de condiciones de carrera	V
	Para evitar posibles condiciones de carrera, es imprescindible proteger las líneas 11 a 13 para evitar que se pueda acceder a la variable Mat de forma concurrente	F
	La ejecución de este programa, creará N hilos de ejecución, y realizará de forma concurrente la suma de las filas	F
	La variable llave indica el número de hilos que están en la sección crítica	V
	La función test_and_set(&llave) consulta y cambia el valor de llave de forma atómica	V

- b) Se desea modificar el código anterior, utilizando un semáforo S para sincronizar las distintas actividades. El código resultante debe realizar la suma de la filas de forma concurrente. Indique, primero, que líneas se han de comentar, para eliminar la solución basada en test_and_set. Después, debe indicar las instrucciones que añadiría y su número de línea, incluyendo la declaración e inicialización del semáforo. Puede utilizar la notación de Dijkstra o la de POSIX

9 b	<p>Se comentan las líneas 3,9,16.</p> <p>Se añaden las siguientes líneas:</p> <p>Línea 4: Sem_t S; ó Sem_t S(1)</p> <p>Línea 14: sem_wait(S) ó P(S)</p> <p>Línea 17: sem_post(S) ó V(S)</p> <p>Línea 24: sem_init(&S, 0,1) ó S=1; ó se omite si se inicia con 1 en línea 4.</p>
-----	---