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. Los términos Shell o intérprete de órdenes, llamadas al sistema, tiempo compartido e interrupción software o Trap son imprescindibles cuando se habla de un sistema operativo como UNIX. De una definición breve de cada uno de ellos e indique para qué son útiles.

(1.2 puntos)

1	a) Shell o intérprete de órdenes (defina e indique su utilidad) Se trata de una aplicación software que lee la línea de órdenes la analiza y la ejecuta. Para ello debe existir un archivo con código ejecutable que se corresponda con la orden a ejecutar. Se trata de una utilidad que se proporciona con el sistema operativo aunque no forma parte de él. Este software realiza llamadas al sistema tanto para crear procesos como para acceder a los dispositivos de E/S. Se ejecuta como un proceso de usuario. Útil para trabajar desde el terminal y poder acceder a muchos de los recursos del sistema informático
	b) LLamadas al sistema (defina e indique su utilidad) Las llamadas al sistema son la interfaz que ofrece el sistema operativo para poder solicitar sus servicios y acceder a los recursos hardware y software del sistema . Las llamadas al sistema provocan la intervención de sistema operativo y el cambio de modo usuario a modo núcleo del procesador. En modo núcleo se puede ejecutar el juego completo de instrucciones (tanto las instrucciones privilegiadas como no privilegiadas) y acceder a todos los dispositivos de E/S, memoria, etc. Las llamadas al sistema son útiles para solicitar servicios al sistema operativo y acceder a recurso que están protegidos.
	c) Tiempo compartido (defina e indique su utilidad) Así se definen aquellos sistemas operativos que además de permitir la multiprogramación o ejecución concurrente de proceso, permiten la interacción usuario máquina. Se trata de sistemas operativos que permiten al usuario trabajar delante de la máquina, lo que hace el trabajo más eficiente
	d) Interrupción software o trap (defina e indique su utilidad) Se trata de una interrupción que está provocada por una instrucción o una llamada al sistema. Es útil para provocar la intervención del sistema operativo, conlleva el paso a modo protegido.

2. Dado el programa Prueba.c cuyo archivo ejecutable ha sido generado como "Prueba".

```
 1 /*** Prueba.c ***/
 2 #include " los necesario .. stdio.h stdlib.h, unistd.h"
 3
 4 int main(int argc, char *argv[]) {
 5   pid_ t val;
 6
 7  if (argc==1)
 8   { if (execl("/bin/ls", "ls","-la",NULL)<0)
 9     { printf ("Message 1\n",1);  exit(1); }
10   }
11  else if (argc==2)
12   { val= fork();
13     if (execl("/bin/cat", "cat",argv[1],NULL)<0)
14     {printf("Message 2\n");  exit(2); }
15      }
16
17   while (wait(NULL)!=-1)  printf ("waiting \n");
18   printf("Message 3\n");
19   exit(0);
20 }
```

Indique de forma justificada los procesos creados y el parentesco que existe entre ellos, así como la información que muestran en la salida estándar al lanzar la ejecución con:

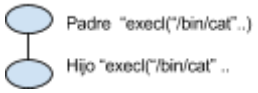
(1.5 punto=0,75+0,75**)**

2

a) $ ./Prueba

Al ejecutar ./Prueba (sin argumentos) se cumple la condición de argc==1 con lo que se genera un único proceso que cambia su imagen de memoria por el código /bin/ls y muestra el resultado de ejecutar la orden "ls -la" en pantalla. Muestra en la salida estándar el contenido de directorio actual en formato largo.

b) $ ./Prueba Prueba.c

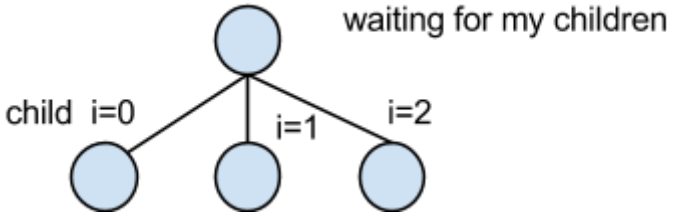


Al ejecutar "./Prueba Prueba.c" se cumple la condición de argc==2 con lo que se generan dos proceso un padre y un hijo. Como no se comprueba el valor devuelto por la llamada fork(),ambos procesos ejecutan "execl("/bin/cat","cat",argv[1],NULL)" que cambia su imagen de memoria por el código /bin/cat y muestran el contenido del archivo Prueba.c. Es decir en pantalla aparece dos veces el contenido del archivo Prueba.c.

3. Complete el código propuesto para que cree un esquema de procesos como el mostrado en la figura de manera que:

(a) Todos los hijos deben permanecer zombies durante aproximadamente 10 segundos y finalizar con un exit(i), donde i es la variable que controla la iteración del bucle de creación de procesos hijos

(b) El padre debe esperar a todos sus hijos y mostrar en el terminal el valor de finalización de exit() de cada hijo.



(1.4 puntos)

```
3  /*** Ejemplo1.c ***/
   #include "todas_los_necesarios.h"
   #define N 3
   int main(int argc, char *argv[]) {
    int i, status;
    pid_t val;

    printf("Parent\n");
     for(i=0; i<N; i++)
       val= fork();
       if ( val==0 ) {
         printf("First generation\n");
         exit(i);
       } /* val**/
     }

     sleep(10);
     while (wait(&status)!=-1)
     {printf ("waiting  status= %d \n", status/256);}

     printf("END \n");
     exit(0);
   }
```

4. Un planificador a corto plazo de un sistema de tiempo compartido consta de dos colas, una gestionada con FCFS (ColaF) y otra gestionada con Round Robin con q=10ut (ColaR). 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 ColaF 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	30 CPU + 10 E/S + 30 CPU + 10 E/S + 30 CPU
B	2	10 CPU + 30 E/S + 10 CPU + 30 E/S + 10 CPU
C	4	20 CPU + 20 E/S + 20 CPU + 20 E/S +20 CPU

a) Indique el diagrama de uso de CPU, rellenando la tabla adjunta. Por coherencia la tabla representa intervalos de 10 ut. **(2.3 puntos =1.3+ 0.6 +0.4)**

	T	ColaR	ColaF	CPU	Cola E/S	E/S	Evento
	0	[C,B](A)		A			Llegan A(0),B(2)y C(4)
	10	C,(B)	A	B			
	20	(C)	A	C		B	B a E/S
	30		C,(A)	A		B	
	40		C	A		B	B sale de E/S
	50	(B)	C	**B**	(A)	A	B "se adelanta" a C
	60	(A)	C	A	(B)	B	
	70		A,(C)	C		B	
	80		(A)	A	C	B	B sale de E/S
	90	(B)	A	**B**		C	B expulsa a A
	100			A		C	Fin de B
	110	(C)		C	(A)	A	
	120	(A)	C	A			
	130		A,(C)	C			
	140		(A)	A	(C)	C	
	150			A		C	
	160	(C)		C			Fin de A
	170			C			
	180						Fin de C

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

	T.espera	T.retorno
A	**50**	160-0 = **160**
B	**8 (0)**	100-2 = **98**
C	**66 (60)**	180-4 = **176**

.

4c) Indique de forma breve y justificada si este planificador da cierta preferencia a los procesos limitados por CPU o a los procesos limitados por E/S

Da preferencia (en el sentido de que esperarán menos en colas de CPU) a los procesos limitados por E/S. Por un lado, al tener ráfagas cortas de CPU, tienen menos probabilidad de ser degradados a la cola FCFS, menos prioritaria. Por otro, si tienen muchas ráfagas de E/S, cada vez que acaban una regresan a la cola RR de CPU, la más prioritaria.

5. Los protocolos de acceso a la sección crítica deben cumplir tres requisitos: Exclusión mutua, progreso y espera limitada. Diga para cada una de las siguientes propuestas si cumplen (SI) o no cumplen (NO) la Exclusión mutua. En el caso de cumplir (SI) la exclusión mutua evalúe también SI cumplen o NO los otros requisitos.

(1.6 puntos)

5		Exclusión mutua	Progreso	Espera Limitada
	Semáforo S=1 //variable compartida, semáforo con cola FIFO Protocolo Entrada → P(S) Sección crítica(); Protocolo de salida -->V(S)	SI	SI	SI

	int llave=0; //variable compartida
	Protocolo de Entrada → while (llave ==1);
	llave=1;
	Sección crítica();
	Protocolo de salida → llave =0;
		NO	NO	NO

	//Solución hardware
	Protocolo de Entrada → DI; // deshabilitar interrupciones
	Sección crítica();
	Protocolo de salida → SI; //habilitar
		SI	SI	NO

	int llave=0; // variable compartida
	Protocolo de Entrada → while (test_and_set(llave));
	Sección crítica();
	Protocolo de salida → llave =0;
		SI	SI	NO

6. Sea una pisicina en la que el número máximo de nadadores es 100 y deben CogerGorro() (obligatorio) de un estante para poder Nadar(). Los hilos *nad* ejecutan la función FuncNadar(). Existe un hilo *rep* reponedor de gorros que ejecuta FuncReponer(). Escriba las operaciones necesarias sobre los semáforos, ya declarados e inicializados, en los puntos del código propuesto de manera que:

- Tanto la función CogerGorro(), como ReponerGorro() se realicen en exclusión mutua.
- Un hilo nad siempre debe ejecutar CogerGorro() antes de Nadar()
- El número de estantes para gorros de baño es de 100
- El hilo rep debe poder ReponerGorro() siempre que haya hueco (números de gorros<100)
- Si se intenta reponer gorro y no hay hueco el hilo debe suspenderse hasta que haya hueco
- Pueden Nadar() a la vez (concurrentemente) un máximo de 100 hilos nadadores (nad)
- Cada vez que un hilo nadador finalice otro que lo solicite debe poder nadar si tiene gorro.
- Si ya hay 100 nadadores en la piscina un hilo que lo solicite debe suspenderse

Nota: Puede utilizar nomenclatura de Disktra P(), V(), o nomenclatura POSIX sem_wait(), sem_post())

(1.0 punto)

6

```
#include <los necesarios...>
 sem_t nadadores, gorros, mutex;    //semáforos declarados

int main(int argc, char *argv[]) {
    pthread_attr_t attr;
    pthread_t nad[300], rep;
    int i;
    pthread_attr_init(&attr);
    sem_init(&nadadores,0,0); sem_init(&gorros,0,100); sem_init(&mutex,0,1);
    pthread_create(&rep, &attr, FuncReponer,NULL);
    for (i = 0; i < 300; i++)   pthread_create(&nad[i], &attr, FuncNadar,NULL);
    for (i = 0; i < 300; i++)   pthread_join(nad[i], NULL);
}
void *FuncNadar() {
    //Complete función según especificaciones del enunciado
    P(gorros)

    P(mutex);
    CogerGorro();
    V(mutex);

    Nadar();
    V(nadadores);
}
void *FuncReponer() {
    //Complete función según especificaciones del enunciado
    while(1) {
        P(nadadores);
        P(mutex);
        ReponerGorro();

        V(mutex);

        V(gorrros);
    }
}
```

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

(1.0 punto)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

pthread_t  th1, th2;
 pthread_attr_t atrib;

void *Func_th1(void *arg)
{ int i,j;
  i= *((int *)arg);
  j=1;
  sleep(10+i);
  pthread_join(th2,NULL);
  printf("th1 is awake\n");
  pthread_exit(&j);
}
```

```
void *Func_th2(void *arg)
{ int i,j;
  i= *((int *)arg);
  j=2;
  sleep(20+i);
  printf("th2 is awake\n");
  pthread_exit(&j);
}
```

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

  pthread_attr_init(&atrib);
  printf("Pthread message: \n");
   i= rand();       //función que proporciona un número aleatorio
   pthread_create(&th1, &atrib, Func_th1,&i);
   pthread_create(&th2, &atrib, Func_th2,&i);

  printf("END \n");
  exit(0);
}
```

7	Pthread message: END No hace pthread_join el hilo main. Y el proceso finaliza al hacer exit(0).