Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N1	**Aviso 1:** Todas las respuestas deben estar razonadas. **Aviso 2:** Escriba sus respuestas con una letra **lo más clara posible.** **Aviso 3: No use *Tipp-ex*** o similares (atasca el escáner).

ESTE EXAMEN CONSTA DE 5 PREGUNTAS

1. (*2p*). Explique razonadamente si las siguientes afirmaciones son VERDADERAS o FALSAS

i. (*1p*). Múltiples procesos pueden compartir de forma transparente el mismo fichero.

ii. (*1p*). Cuando se produce una interrupción que no esté bloqueada o enmascarada, el proceso invoca al algoritmo `inthand()`.

RESPUESTA:

i) **Verdadera**. En UNIX, los ficheros son accedidos secuencialmente por defecto. Cuando un usuario abre el fichero, el núcleo inicializa el puntero de lectura/escritura a cero. Cada vez que el proceso lee o escribe datos, el núcleo avanza el puntero en la cantidad de bytes transferidos.

El mantener un puntero de lectura/escritura en el objeto de fichero abierto permite al núcleo aislar unas de otras las diferentes sesiones de trabajo sobre un mismo fichero (ver Figura 1-6). Si dos procesos abren el mismo fichero, o si un proceso abre el mismo fichero dos veces, el núcleo genera un nuevo objeto de fichero abierto y un nuevo descriptor de fichero en cada invocación a open. De esta forma una operación de lectura o de escritura por un proceso producirá el avance de su propio puntero de lectura/escritura y no afectará al del otro. Esto permite que múltiples procesos compartan de forma transparente el mismo fichero. (***Apartado 1.8.4).***

ii) **Falsa.** Es el núcleo quien invoca al algoritmo `inthand()`. (***Apartado 3.7).***

2. (*2p*). Explique los motivos por los que el núcleo puede invocar a un driver de dispositivo. Señale y describa las partes en las que se divide un driver.

RESPUESTA:

(***Apartado 9.4.2***) El núcleo puede invocar a un driver de dispositivo por varios motivos:

- *Configuración.* El núcleo llama al driver cuando se arranca el sistema para comprobar e inicializar el dispositivo.
- *Entrada/Salida*. El subsistema de E/S llama al driver para escribir o leer datos.
- *Control*. El usuario puede hacer peticiones de control tales como la apertura o cierre de un dispositivo o el rebobinado de una cinta magnética.
- *Interrupciones.* El dispositivo genera interrupciones una vez que se ha completado una operación de E/S o se produce algún cambio en el estado del dispositivo.

Las funciones de configuración son llamadas una única vez, cuando el sistema arranca. Las funciones de entrada/salida y control son operaciones síncronas. Son invocadas en respuesta a peticiones de usuario específicas y se ejecutan en el contexto del proceso invocador. La rutina `d_strategy` del driver de modo bloque es una excepción a esta norma. Las interrupciones son eventos asíncronos, el núcleo no puede predecir cuándo ocurrirán y se ejecutan en el contexto de cualquier proceso.

El driver se divide en dos partes:

- *Parte superior del driver*. Contiene las rutinas síncronas. Se ejecutan en el contexto del proceso. Pueden acceder al espacio de direcciones y al *área U* del proceso invocador y pueden poner al proceso a dormir si fuese necesario
- *Parte inferior del driver*. Contiene las rutinas asíncronas. Se ejecutan en el contexto del sistema y usualmente no tienen ninguna relación con el proceso actualmente en ejecución y en consecuencia

no pueden acceder al espacio de direcciones de dicho proceso o a su *área U*. Además, no pueden poner a dormir a ningún proceso puesto que podrían bloquear un proceso no relacionado.

Las dos partes del driver necesitan sincronizar sus actividades. Si un objeto es accedido por ambas partes, entonces las rutinas de la parte superior deben bloquear las interrupciones (mediante la elevación del *npi*) mientras manipulan el objeto. En caso contrario, el dispositivo podría interrumpir mientras el objeto se encuentra en un estado inconsistente, con lo que el resultado sería impredecible

3. *(1p)*. Indique el resultado de la siguiente ejecución:

```
% chmod g+r prueba
```

RESPUESTA:

El comando `chmod` configura la máscara de modo del archivo prueba para que otros miembros del grupo al que pertenece el archivo puedan leerlo. (***Apartado 2.6.2*).**

Para configurar la máscara de modo de un fichero, su propietario o el superusuario, pueden utilizar el comando `chmod`. Su sintaxis es:

```
chmod {u,g,o,a}{+,-}{r,w,x,s,t} <ficheros>
```

En el comando se indica a qué usuarios afecta: usuario propietario (u), usuarios pertenecientes al grupo al que pertenece el archivo (g), otros usuarios (o), todos los usuarios (a). A continuación se especifica si se están añadiendo permisos (+) o quitándolos (-). Finalmente se especifica qué tipo de permiso se hace referencia lectura (r), escritura (w) o ejecución (x).

4. (*2p).* Explique razonadamente el significado de las sentencias enumeradas ([ ]) del siguiente programa escrito en lenguaje C

```
#include <signal.h>
main()
{
[1] long mask0;
[2] mask0=sigsetmask(sigmask(SIGUSR1)|
                     sigmask(SIGUSR2));
[3] sigblock(sigmask(SIGINT));
[4] sigsetmask(mask0);
}
```

RESPUESTA:

(***Apartado 4.4.3***). En la sentencia **[1]** se declara la variable mask0 de tipo entero largo. En la sentencia **[2]** se invoca a sigsetmask para bloquear la recepción de las señales del tipo SIGUSR1 y SIGUSR2. En la variable mask0 se almacena la máscara de señales original que se tenía especificada antes de invocar a esta llamada al sistema.

En la sentencia **[3]** se invoca a sigblock para añadir las señales del tipo SIGINT al grupo de señales bloqueadas. Finalmente en **[4]** se vuelve a invocar a sigsetmask para restaurar la máscara original de señales, es decir, sin tener bloqueadas a las señales del tipo SIGUSR1, SIGUSR2 y SIGINT.

5. *(3 p)* Conteste razonadamente a los siguientes apartados:
a) *(1.5 p)* Explicar el significado de las sentencias enumeradas (**[ ]**) de este programa.
b) *(1.5 p)* Explicar el funcionamiento del programa asumiendo que el pid del primer proceso creado es 1000, los pids se asignan secuencialmente y no se produce ningún error.

```
     #include <sys/types.h>
     #include <sys/ipc.h>
     #include <sys/sem.h>
     #include <sys/shm.h>
     #include <stdio.h>
     #include <stdlib.h>

     void main()
     {
          int *x;
          int pid, shmid, semid;
[1]       struct sembuf op[1];
[2]       semid=semget(IPC_PRIVATE,1,IPC_CREAT|0600);
          if (semid==-1)
               {perror("fallo en semget");
               exit(1);}
[3]       shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT | 0600);
          if (shmid==-1)
               {perror("fallo en shmget");
               exit(2);}
[4]       x=shmat(shmid,0,0);
          *x=0;
[5]       if ((pid=fork())==-1) exit(2);
               if (pid==0)
               {
[6]                 sleep(5);
                    *x=10;
[7]                 shmdt(x);
                    op[0].sem_num=0;
                    op[0].sem_op=1;
                    op[0].sem_flg=0;
[8]                 if (semop(semid, op, 1) == -1)  perror("Error1:");
                    exit(3);
               }
               else
               {
                    struct sembuf op[1];
                    op[0].sem_num=0;
                    op[0].sem_op=-1;
                    op[0].sem_flg=0;
                    if (semop(semid, op, 1) == -1)  perror("Error2:");
                    printf("El proceso %d hace que x=%d\n",pid,*x);
               }
     }
```

RESPUESTA

Apartado a):

[1] Se crea un vector de estructuras `op` de tipo `sembuf` con un único elemento. Esta estructura permite realizar operaciones con semáforos. El primer campo `sem_num` indica el semáforo sobre el que opera, el campo `sem_op` indica la operación a realizar, por último `sem_flg` permite pasar los *flags* `IPC_NOWAIT` y `SEM_UNDO` a la operación.

[2] La llamada al sistema `semget()` obtiene un vector de semáforos cuyo identificador numérico es `semid` a partir de una clave. El primer argumento es `IPC_PRIVATE` que indica que el semáforo es privado al proceso y sus hijos (no es accesible para otros procesos). El segundo argumento indica que el vector contendrá un único semáforo. Finalmente el tercer argumento indica los *flags*. En este caso `IPC_CREAT|0600` indica que el semáforo deberá crearse (`IPC_CREAT`) y que tendrá permisos de lectura y escritura para el propietario (`0600`).

[3] `shmget()` es una llamada al sistema que crea una región de memoria compartida devolviendo un identificador numérico `shmid`. El segundo argumento indica el tamaño de la región de memoria (el necesario para albergar un entero). Los otros argumentos son idénticos a los usados en `semget()`.

[4] Por su parte la llamada al sistema `shmat()` asigna un espacio de direcciones virtuales al segmento de memoria cuyo identificador `shmid` ha sido creado por `shmget`. El segundo argumento indica la dirección virtual del proceso donde se desea que empiece la región de memoria compartida. El tercer argumento es un *flag* que actúa como una máscara de bits que indica la forma de acceso a la memoria. La llamada al sistema devuelve un puntero a la región de memoria compartida `x`.

[5] `fork()` llama al Sistema Operativo para crear un proceso hijo que es una copia del proceso que la invoca (proceso padre). La llamada al sistema devuelve al padre el `pid` del hijo mientras que devuelve 0 al hijo. Si se produce un error `fork()` devuelve -1 lo que haría que se ejecutase la llamada al sistema `exit(2)` que terminaría el proceso devolviendo el código de salida “2”.

[6] `sleep(5)` es una llamada al sistema que induce al proceso a dormir durante 5 segundos.

[7] La llamada al sistema `shmdt(x)` libera el espacio de memoria virtual que estaba ocupando región de memoria compartida apuntada por `x`. De este modo el proceso se desconecta de la región de memoria compartida. Al mismo tiempo decrementa el contador de referencias de dicha región de memoria principal, cuando el contador de referencias llega a cero ningún proceso está usando esa región de memoria y por tanto es posible liberarla.

[8] `semop` toma un vector de operaciones `op` de tipo `sembuf` y las aplica sobre un vector de semáforos cuyo identificador es `semid`. Las operaciones se aplican en secuencia de forma consistente, esto es, o se completan todas con éxito, o se produce un error y no se aplica ninguna. El último argumento indica el número de operaciones que debe realizar (una). Si se produce un fallo `semop` devuelve -1 lo que hace que se ejecute la llamada al sistema `perror` que mostraría por pantalla “`Error1:`” seguido del descriptor del error que se encuentra almacenado en la variable `errno`.

Apartado b):

En primer lugar se crea un vector de semáforos `semid` (que tiene un único elemento) y una región de memoria compartida `shmid`. Si alguna de estas operaciones produce un error este se muestra mediante `perror` y el proceso termina.

A continuación, si no se ha producido ningún error se asocia la región de memoria compartida con el puntero a entero `x` y se inicializa dicha región de memoria con 0 (`*x=0`).

En ese momento el proceso se bifurca (`fork`) dando lugar a un proceso padre (`pid=1000`) y a otro hijo (`pid=1001`). Si se produce un error se termina el proceso.

En caso contrario el hijo ejecuta el bloque de código siguiente al `if` mientras que el padre ejecuta el bloque de código que sigue al `else`.

El hijo espera 5 segundos, a continuación escribe un 10 en la variable entera apuntada por `x` y libera la memoria compartida. Finalmente efectúa una operación `sem_op=1` en el semáforo. Esto hace que el semáforo (inicialmente a cero) pase a valer 1 y desbloquea al proceso padre (si estuviese esperando).

El padre por su parte realiza una operación `sem_op=-1` sobre el mismo semáforo, como se trata de un número negativo quedará a la espera de que el semáforo (inicialmente a cero) pase a valer 1. A continuación mostrará por pantalla:

```
El proceso 1001 hace que x=10
```

Y termina.

NOTA: Sea cual sea el proceso que se planifica en primer lugar, el padre no podrá leer la memoria compartida hasta que el hijo haya abierto el semáforo que inicialmente estaba a cero. Por tanto, hasta que no pasen al menos 5 segundos (que es el tiempo que espera el hijo antes de escribir en la memoria compartida y abrir el semáforo) el padre no podrá imprimir el valor apuntado por `x` por lo que **siempre se imprime x=10**.