Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N2	**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*) La actuación de los descriptores de fichero reduce el rendimiento del sistema.

ii. (*1p*) Cuando se produce una interrupción que no esté bloqueada o enmascarada, el núcleo añade una capa de contexto en la pila de capas de contexto de un proceso.

RESPUESTA:

i) **Falsa**: El proceso pasa el descriptor de fichero a las llamadas al sistema asociadas con operaciones de E/S tales como read o write. El núcleo usa el descriptor para localizar rápidamente el objeto de fichero abierto. De esta forma, apoyándose en estos dos elementos el núcleo solamente necesita realizar una vez (durante la ejecución de open) y no en cada operación de E/S con el fichero, tareas tales como la búsqueda de la ruta de acceso o el control de acceso al fichero. Esto supone una mejora en el rendimiento del sistema. (***Apartado 1.8.4*).**

ii) **Verdadero.** Los motivos por los que el núcleo añade una capa de contexto en la pila de capas de contexto de un proceso son: cuando se produce una interrupción, cuando el proceso realiza una llamada al sistema; cuando se produce un cambio de contexto. (***Apartado 3.6.2***).

2. (*2p*). Enumere y describa las características comunes de los mecanismos IPC del System V.

RESPUESTA:

(Apartado 6.4.1.a). Los *mecanismos IPC del System V* están implementados en el sistema como una unidad y comparten características comunes, entre las que se encuentran:

1. Cada tipo de mecanismo IPC tiene asignada *una tabla en el espacio de memoria del núcleo* de tamaño fijo. Por lo tanto en el núcleo existen tres tablas relacionadas con los mecanismos IPC: una para semáforos, otra para mensajes y una tercera para la memoria compartida.
2. Cada tabla asignada a un tipo de mecanismo IPC posee un número de entradas configurable. Cada entrada contiene información relativa a una instancia de dicho mecanismo IPC o canal IPC.
3. Cada entrada de la tabla tiene asignada una *llave numérica*, que permite controlar el acceso a dicha instancia del mecanismo IPC.
4. Cada entrada de la tabla asociada a un tipo de mecanismo IPC tiene asignado un índice IT para su localización dentro de la tabla.
5. Cada entrada de la tabla asociada a un tipo de mecanismo IPC tiene almacenada una estructura ipc_perm que presenta la siguiente definición:

```
Structm ipc_perm{
ushort uid;        Identificador de usuario del proceso propietario del recurso.
ushort gid;        Identificador de grupo del proceso propietario del recurso.
ushort cuid;       Identificador de usuario del proceso creador del recurso.
ushort cgid;       Identificador de grupo del proceso creador del recurso.
```

`ushort mode;`	Modo de acceso (permisos de lectura, escritura y ejecución para el usuario, el grupo y otros usuarios).
`ushort seq;`	Número de secuencia. Es un contador que mantiene el núcleo y que se incrementa siempre que se cierra una instancia o canal de un mecanismo IPC. Este contador es necesario para identificar los canales abiertos e impedir que mediante una elección aleatoria del identificador de canal, un proceso pueda adquirirlo.
`key_t key;`	Llave de acceso.
`}`	

6. Cada entrada de la tabla asociada a un tipo de mecanismo IPC, además de la estructura ipc_perm, tiene almacenada también otras informaciones como por ejemplo el pid del último proceso que ha utilizado la entrada y la fecha de la última actualización o acceso.
7. Cada instancia de un mecanismo IPC tiene asignado un descriptor numérico NIPC elegido por el núcleo, que la referencia de forma única y que será utilizado para localizar la instancia rápidamente cuando se realicen operaciones sobre ella.
8. Cada tipo de mecanismo IPC dispone de una llamada al sistema tipo get [shmget (memoria compartida), semget (semáforos) y msgget (colas de mensajes)] que permite crear una nueva instancia de un determinado tipo de mecanismo IPC o acceder a alguna ya existente.
9. Cada tipo de mecanismo IPC dispone de una llamada al sistema tipo ctl [shmctl (memoria compartida), semctl (semáforos) y msgctl (colas de mensajes)] que permite acceder a la información administrativa y de control de una instancia de un mecanismo IPC.

3. (*1 p*). Describe el resultado de la siguiente operación:

mutsem=semget(2471, 3, IPC_CREAT | 0666);

RESPUESTA:

La orden mutsem=semget(2471,3, IPC_CREAT | 0666) crea **un array** llamado mutsem que contiene **tres semáforos generales** asociados a la **clave 2471** con permisos de lectura y escritura para todos los usuarios, (***apartado 6.4.2b***). Si la llamada al sistema tiene éxito "mutsem" contendrá un número entero que identifica al array de semáforos y permite hacer operaciones sobre él. Si se produce algún fallo semget devuelve -1.

4. (*2p*)**.** El ladrón de páginas de un cierto sistema UNIX debe transferir a un dispositivo de intercambio 25 páginas del proceso A, 100 páginas del proceso B, 50 páginas del proceso C y 75 páginas del proceso D. En una operación de escritura puede transferir 30 páginas al dispositivo de intercambio.

a) (*1.5 p*). Determinar la secuencia de operaciones de intercambio de páginas que tendría lugar si el ladrón de páginas examina las páginas de los procesos en el orden C, B, D y A.
b) (*0.5 p*). ¿Cuál es el último proceso que transfiere completamente todas sus páginas al dispositivo de intercambio?

RESPUESTA:

a). (**Apartado 7.3.4**). La secuencia completa de operaciones es la siguiente:

1) El ladrón de páginas asigna espacio para 30 páginas y transfiere al dispositivo de intercambio 30 páginas del proceso C. Le quedan por transferir 20 páginas del proceso C, 100 páginas del proceso B, 75 páginas del proceso D y 25 páginas del proceso A.

2) El ladrón de páginas asigna espacio para otras 30 páginas y transfiere al dispositivo de intercambio 20 páginas del proceso C y 10 páginas del proceso B. Luego ha transferido todas las páginas del proceso C y le quedan por transferir 90 páginas del proceso B, 75 páginas del proceso D y 25 páginas del proceso A.

3) El ladrón de páginas asigna espacio para otras 30 páginas y transfiere al dispositivo de intercambio 30 páginas del proceso B. Le quedan por transferir 60 páginas del proceso B, 75 páginas del proceso D y 25 páginas del proceso A.

4) El ladrón de páginas asigna espacio para otras 30 páginas y transfiere al dispositivo de intercambio 30 páginas del proceso B. Le quedan por transferir 30 páginas del proceso B, 75 páginas del proceso D y 25 páginas del proceso A.

5) El ladrón de páginas asigna espacio para otras 30 páginas y transfiere al dispositivo de intercambio 30 páginas del proceso B. Luego ha transferido todas las páginas del proceso B. Le quedan por transferir 75 páginas del proceso D y 25 páginas del proceso A.

6) El ladrón de páginas asigna espacio para otras 30 páginas y transfiere al dispositivo de intercambio 30 páginas del proceso D. Le quedan por transferir 45 páginas del proceso D y 25 páginas del proceso A.

7) El ladrón de páginas asigna espacio para otras 30 páginas y transfiere al dispositivo de intercambio 30 páginas del proceso D. Le quedan 15 páginas del proceso D y 25 páginas del proceso A.

8) El ladrón de páginas asigna espacio para otras 30 páginas y transfiere al dispositivo de intercambio 15 páginas del proceso D y 15 páginas del proceso A. Luego ha transferido todas las páginas del proceso D. Le quedan por transferir 10 páginas del proceso A.

9) El ladrón de páginas guarda las 10 páginas que le restan por intercambiar del proceso A en la lista de páginas de intercambio y no las intercambiará hasta que la lista este llena.

b) El proceso D es el último que finaliza su transferencia completa de todas sus páginas, pues en el último paso el ladrón de páginas guarda las 10 páginas que le restan por intercambiar del proceso A en la lista de páginas de intercambio y no las intercambiará hasta que la lista esté llena.

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 suponiendo que al primer proceso creado al ejecutarlo se le asigna el pid 1000, los pids se asignan en secuencia, no se produce ningún error y los identificadores textuales de las señales de usuario son `"User defined signal 1" y "User defined signal 2"` respectivamente.

```
#include <signal.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void fun1(int sig)
     {
     printf("%d recibe: %s \n",getpid(),strsignal(sig));
     }
void main()
     {
     int pid;
     signal(SIGUSR1,fun1);
     signal(SIGUSR2,fun1);
```

```
[1]       if ((pid=fork())==-1) {  perror("Error en fork():"); exit(2);}
          if (pid==0)
               {
               printf("Primera ronda\n");
[2]            signal(SIGUSR1,SIG_IGN);
[3]            kill(getppid(),SIGUSR1);
[4]            sleep(1);

               printf("Segunda ronda\n");
[5]            sigblock(sigmask(SIGUSR2));
               signal(SIGUSR1,fun1);
               kill(getppid(),SIGUSR1);
               sleep(1);

               printf("Tercera ronda\n");
[6]            sigsetmask(0);
               sleep(1);
               exit(3);
               }
          else
               pause();
               kill(pid,SIGUSR1);
               kill(pid,SIGUSR2);
[7]            pause();
               kill(pid,SIGUSR1);
               kill(pid,SIGUSR2);
[8]            wait();
               exit(3);
               }
          }
```

RESPUESTA

Apartado a):

[1] La llamada al sistema `fork()` crea 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 `perror` que muestra por pantalla el mensaje "`Error en fork():`" seguido de una descripción de dicho error que se encuentra almacenado en la variable `errno`. A continuación ejecutaría la llamada al sistema `exit(2)` que terminaría el proceso devolviendo el código de salida "2".

[2] `signal()` llama al Sistema Operativo para asociar una señal con una acción que puede ser una función que trate la señal o bien alguna de las constantes `SIG_DFL` (que indica que la acción a realizar es la acción por defecto asociada a dicha señal) o `SIG_IGN` (que indica que la señal será ignorada). En este caso se está asociando la señal de usuario número 1 (`SIGUSR1`) con la acción ignorar (`SIG_IGN`).

[3] La llamada `kill()` permite enviar a un proceso (cuyo `pid` se introduce como primer argumento) una señal (segundo argumento). Puesto que la llamada al sistema `getppid()` devuelve el `pid` del proceso padre, el resultado de esta sentencia es enviar la señal de usuario número 1 (`SIGUSR1`) al padre del proceso que se esté ejecutando en ese momento.

[4] `sleep(1)` es una llamada al SO que duerme al proceso durante 1 segundo.

[5] La llamada al sistema `sigblock()` bloquea las señales definidas en una máscara binaria creada con sigmask. En este caso se bloquea la señal de usuario número 2 `SIGUSR2`.

[6] `sigsetmask()` establece la máscara de señales que están bloqueadas. La máscara es un entero largo asociado a la máscara de señales. Se considera que la señal número j está bloqueada si el j-ésimo bit de la máscara está a 1. Como se pasa un 0 todas las señales quedan desbloqueadas.

[7] `pause()` es una llamada al SO cuyo efecto es que el proceso que la invoca quede a la espera de la recepción de una señal que no ignore y que no tenga bloqueada.

[8] La llamada al sistema `wait()` sincroniza la ejecución del padre con la terminación de un proceso hijo. El padre queda suspendido a la espera de que finalize la ejecución del hijo antes de continuar su ejecución.

Apartado b):
El programa en primer lugar asocia las señales `SIGUSR1` y `SIGUSR2` con la ejecución de la función `fun1` definida antes de la función `main`.

A continuación, realiza un `fork()` para crear un proceso hijo. Si se produjese algún error sería mostrado por la llamada a través de la llamada al sistema `perror` terminando la ejecución del programa, pero al no producirse ningún error la ejecución continua.

Puesto que el padre es el proceso inicial que tenía `pid` igual a 1000, el hijo recién creado obtendrá un `pid` igual a 1001 (ya que como indica el enunciado los `pid` se asignan en secuencia). La llamada al sistema `fork` devuelve por lo tanto 1001 al padre (el `pid` de su hijo) y 0 al hijo.

De este modo, el hijo ejecuta el bloque de código siguiente a la sentencia `if (pid==0)`, mientras que el padre ejecuta el código siguiente al `else`.

Independientemente del orden de planificación el resultado es siempre el mismo:

El padre espera mediante la llamada al sistema `pause` a que su hijo le envíe una señal por lo que no prosigue su ejecución hasta recibirla.

El Hijo por su parte imprime:

```
Primera ronda
```

A continuación establece que la señal de usuario 1 debe ignorarse. Luego despierta al padre enviándole una señal `SIGUSR1` y se queda durmiendo durante un segundo `sleep(1)`.

El padre se despierta y en primer lugar en núcleo ejecuta en el contexto del padre la acción asociada a dicha señal, esto es `fun1`, que imprime por pantalla el `pid` del proceso que la invoca seguida de “recibe:” y la descripción textual de la señal recibida:

```
1000 recibe: User defined signal 1
```

Una vez atendida la señal la llamada al sistema `wait()` devuelve -1 (indicando que el proceso ha sido interrumpido por la señal) y el padre continúa ejecutándose. Al hacerlo envía mediante `kill` las señales `SIGUSR1` y `SIGUSR2` a su hijo y vuelve a quedarse a la espera de señales mediante `pause()`.

El hijo ignora la señal `SIGUSR1` (por lo que dicha señal **no se atiende y se pierde**) reaccionando únicamente ante la señal `SIGUSR2`. Al recibirla se invoca `fun1` imprimiendo:

```
1001 recibe: User defined signal 2
```

Después continua con lo que estaba haciendo (dormir durante un segundo) y cuando despierta de `sleep(1)` imprime:

```
Segunda ronda
```

Seguidamente bloquea la señal `SIGUSR2` y vuelve asignar `SIGUSR1` a `fun1`. Hecho esto despierta a su padre con una señal `SIGUSR2` y se pone a dormir durante otro segundo.

El padre despierta invocando primero `fun1` que escribe:

```
1000 recibe: User defined signal 1
```

A continuación, vuelve a enviar las señales `SIGUSR1` y `SIGUSR2` a su hijo y se queda a la espera de que termine con `wait()`.

El hijo recibe la señal `SIGUSR1` y la atiende imprimiendo:

```
1001 recibe: User defined signal 1
```

Como la señal `SIGUSR2` está bloqueada, el hijo no la recibe y continúa durmiendo hasta que termine el segundo. Pero, **a diferencia del caso anterior, la señal no se pierde** sino que se mantiene a la espera hasta que dicha señal se desbloquee**.** Por lo tanto el hijo sigue durmiendo hasta que acabe el segundo. A continuación imprime:

```
Tercera ronda
```

Para finalmente desbloquear todas las señales y ponerse a dormir durante un último segundo. Al hacerlo recibe la señal (que ahora ya no está bloqueada) e imprime:

```
1001 recibe: User defined signal 2
```

Finalmente al terminar el tercer segundo, el hijo despierta y termina, a continuación el padre también se despierta y termina. **En resumen,** la ejecución del proceso produce la siguiente salida:

```
Primera ronda
1000 recibe: User defined signal 1
1001 recibe: User defined signal 2
(pasa un segundo)
Segunda ronda
1000 recibe: User defined signal 1
1001 recibe: User defined signal 1
(pasa otro segundo)
Tercera ronda
1001 recibe: User defined signal 2
(pasa un tercer segundo y termina)
```