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. *(2 p)* Explique **razonadamente** si las siguientes afirmaciones son verdaderas o falsas.

i) *(1 p)* El núcleo puede emplear sus mecanismos de protección para proteger unas hebras de usuario de otras.
ii) *(1p)* El uso de las colas de mensajes en la transferencia de grandes cantidades de datos no afecta al rendimiento del sistema.

RESPUESTA:

i) FALSO (*V. 4.9.4 Hebras de usuario, p. 216*)

Las hebras de usuario poseen varias limitaciones, principalmente debidas a la separación total de información entre el núcleo y las librerías de hebras. Puesto que el núcleo no sabe de la existencia de las hebras de usuario, no puede usar sus mecanismos de protección para proteger una hebras de otras. Cada proceso tiene su propio espacio de dirección, que el núcleo protege de accesos no autorizados de otros procesos. Las hebras de usuario no disfrutan de esta protección, operan en el espacio de direcciones que es propiedad del proceso. La librería de hebras debe suministrar mecanismos de sincronización, los cuales requieren de la cooperación entre las hebras.

ii) FALSO (*V.r 6.4.3 Colas de mensajes, p. 295*)

Las colas de mensajes son útiles para transferir pequeñas cantidades de datos. Sin embargo si hay que transferir grandes cantidades de datos el rendimiento del sistema se deteriora. Esto es debido a que la transferencia de un mensaje requiere dos operaciones de copia de datos en memoria: la primera del espacio de direcciones del proceso emisor a un buffer interno del núcleo y la segunda de dicho buffer al espacio de direcciones del proceso receptor.

2 (*1.5 p*) Supóngase que el fichero `/etc/group` contiene, entre otras, las siguientes líneas:

```
root:*:0:
estudiantes:*:10: TUTOR, ESTUDIANTE1
profesores:x:50:DIRECTOR
```

a) (0.25 p) ¿Cuántos miembros tiene el grupo root?
b) (0.25 p) ¿Cuál es el GID del grupo profesores?
c) (0.25 p)¿Qué usuarios tienen acceso en el grupo estudiantes?
d) (0.75 p) ¿Tiene contraseña de entrada el grupo profesores? ¿Y el grupo estudiantes? En caso afirmativo, ¿dónde está almacenada?

RESPUESTA:
(V. 2.5 Gestión de usuarios)

APARTADO a)
El grupo `root` es un grupo especial del sistema reservado para la cuenta `root`. Este grupo consta de un único miembro, el superusuario.

APARTADO b)

El identificador numérico (GID) del grupo `profesores` es 50.

APARTADO c)
A este grupo tienen acceso el usuario `TUTOR` y el usuario `ESTUDIANTE1`.

APARTADO d)
El grupo `estudiantes` no tiene contraseña de entrada, se encuentra deshabilitada por el carácter `*`.
El grupo `profesores`, si tiene contraseña. La contraseña de acceso se encuentra almacenada en el fichero `/etc/shadow`.

3. *(1.5 p)* Responda razonadamente a las siguientes cuestiones:
a) (*0.75 p)* ¿Qué son los *callouts*? Relaciona algunos usos de los *callouts.*
b) *(0.75 p)* Describe las formas de implementar la lista de *callouts.*

RESPUESTA:

(V. 5.3.2 Callouts)

APARTADO a)

Los *callouts* son un mecanismo interno del núcleo que le permite invocar funciones transcurrido un cierto tiempo. Un *callout* típicamente almacena el nombre de la función que debe ser invocada, un argumento para dicha función y el tiempo en tics transcurrido el cual la función debe ser invocada. Los usuarios no tienen ningún control sobre este mecanismo.

Los callouts se pueden utilizar para la invocación de tareas periódicas tales como: la transmisión de paquetes de red, ciertas funciones de administración de memoria y del planificador, la monitorización de dispositivos para evitar la pérdida de interrupciones, etc.

El núcleo mantiene una *lista de callouts*. La organización de esta lista puede afectar el rendimiento del sistema, si existen varios callouts pendientes, ya que la lista es comprobada por la rutina de tratamiento de la interrupción del reloj a un *npi* elevado en cada tic de reloj. En consecuencia, la rutina debe intentar optimizar el tiempo de comprobación. Por el contrario, el tiempo requerido para insertar un nuevo callout dentro de la lista es menos crítico puesto que la inserción típicamente ocurre a un *npi* más bajo y con mucha menos frecuencia que una vez cada tic.

APARTADO b)

Existen varias formas de implementar la lista de callouts. Un método usado en BSD4.3 es ordenar la lista en función del tiempo que le resta al callout para ser invocado. A este tiempo comúnmente se le denomina *tiempo de disparo*. Cada entrada de la lista de callouts almacena la diferencia entre el tiempo de disparo de su callout asociado y el tiempo de disparo del callout asociado a la entrada anterior. El núcleo decrementa el tiempo de la primera entrada de la lista en cada tic de reloj y lanza el callout si el tiempo alcanza el valor 0.

Otra posible aproximación sería utilizar también una lista ordenada, pero almacenar el tiempo absoluto de finalización para cada entrada. De esta forma, en cada tic, el núcleo compara el tiempo absoluto actual con el de la primera entrada y lanza el callout cuando los tiempos son iguales.

4. (*2 p*). Supóngase que la lista parcial de i-nodos libres del superbloque está vacía, su i-nodo recordado es 25 y su variable índice puede tomar como máximo el valor 4. Además, existen los siguientes i-nodos libres en la tabla de i-nodos: 95, 45, 88, 37, 32, 30 y 73. Dibuje la lista parcial de i-nodos libres del superbloque una vez que ha sido rellenada por el núcleo. ¿Cuál sería ahora el i-nodo recordado?

RESPUESTA:
(*V. Tema 8)*

El núcleo comenzará a buscar i-nodos libres a partir del i-nodo recordado (25) en orden ascendente, por lo tanto conviene ordenar los números de i-nodo libres del enunciado en esta forma

30 32 37 45 73 88 95

Por otra parte, puesto que la variable índice puede tomar como máximo el valor 4, eso implica que la lista parcial de i-nodos del superbloque va a poder contener como máximo 5 números de i-nodos libres. En consecuencia, el núcleo seleccionará para rellenar la lista de i-nodos libres a los cinco primeros que encuentre, estos son 30, 32, 37, 45 y 73.

Finalmente, la lista parcial de i-nodos libres del superbloque es rellenada por el núcleo colocando los i-nodos en orden descendente, con lo que se tiene:

i-nodo recordado ->	**73**	45	37	32	30
índice	**0**	1	2	3	4

Por lo tanto, el i-nodo recordado es el **73.**

5. *(3 p)* Conteste razonadamente a los siguientes apartados:

a) (2 p) Explicar el significado de las sentencias enumeradas (**[ ]**) del programa que se muestra en la página siguiente.

b) (1 p) Explicar el funcionamiento del programa si se compila en el ejecutable `programa`, se invoca con la orden **.**`/programa DyASO` y no se produce ningún error.

La pregunta continua en la página siguiente

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>
        #include <unistd.h>
        #include <string.h>
        #include <sys/wait.h>

[1]     int main(int argc, char *argv[])
        {
           int fd, pid, salida, status;

           if (argc!=2) {printf("Argumentos incorrectos\n"); exit(1);}
[2]        fd=open("fichero1.sh", O_RDWR|O_CREAT|O_TRUNC, 0666);
[3]        if (fd==-1) {perror("Error open"); exit(2);}
           write(fd,"#!/bin/bash\necho Ejercicio 5 ",29);
[4]        write(fd,argv[1],strlen(argv[1]));
[5]        close(fd);
[6]        chmod("fichero1.sh",0700);

[7]        unlink("fifo");
[8]        if (mknod("fifo",S_IFIFO|0666,0)==-1)
              {perror("Error mknod"); exit(3);}

[9]        if ( (pid=fork())==-1 ) {printf("Error fork\n"); exit(4);}
           if ( pid==0 )
               {
[10]            salida=system("./fichero1.sh > fifo");
                if (salida==0) printf("¡Correcto!\n");
                unlink("fifo");
                unlink("fichero1.sh");
                }
           else
                {
[11]            execl("/bin/cat","cat","fifo",NULL);
                perror("Error execl");
                printf("¡Incorecto!\n");
                exit(5);
                }
           exit(0);
        }
```

RESPUESTA

APARTADO a)

[1] Se declara la función principal `main` que recibe `argc` indicando el número de parámetros con el que ha sido invocada y un vector de cadenas `argv` que contiene cada uno de los parámetros recibidos.

[2] `open` abre el fichero `"fichero1.sh"` con permiso de lectura y escritura (`O_RDWR`) truncándolo, de modo que elimina el contenido anterior del archivo (`O_TRUNC`). Si el fichero no existe lo crea (`O_RDWR`), con la máscara de permisos `0666`. Si no hay errores devuelve el descriptor de fichero `fd`, en caso contrario devuelve `-1`.

[3] Si se ha producido un error en [2] se cumple la cláusula `if` y `perror` imprime "`Error signal:`" seguido del descriptor textual del mensaje de error que se encuentra almacenado en la variable `errno`. A continuación se termina el proceso con la llamada al sistema `exit` y el código de retorno `2`.

[4] `write` escribe en el fichero abierto en [2] el segundo argumento de entrada del ejecutable `argv[1]` (recordemos que el primer argumento `argv[0]` es el nombre del ejecutable). El número de bytes a escribir es la longitud de dicho argumento que se obtiene con `strlen`.

[5] La llamada al sistema `close` cierra el fichero cuyo descriptor es `fd`.

[6] La llamada al sistema `chmod` permite cambiar los permisos de acceso al fichero `fichero1.sh` otorgándole permiso de lectura, escritura y ejecución únicamente al propietario del fichero (`0700`).

[7] `unlink` es una llamada al sistema que elimina el fichero `fifo` (en caso de que exista).

[8] `mknod` permite crear un fichero FIFO (`S_IFIFO`) de nombre `fifo` y con máscara de permisos `0666`.

[9] La llamada al Sistema `fork()` crea un proceso hijo que es una copia idéntica del proceso padre. Al proceso padre le devuelve el `pid` del hijo mientras que al proceso hijo le devuelve `0`. En caso de error, `fork` devuelve `-1`, de tal modo que se ejecuta la cláusula `if`, se imprime "`Error fork`" y se termina con un código de retorno `4`.

[10] La llamada al sistema `system` permite ejecutar una orden en un proceso separado como si se teclease en el *shell*. En este caso la orden es "`./fichero1.sh > fifo`" que ejecuta el script `fichero1.sh` y redirige su salida estándar al fichero `fifo`.

[11] `execl` es una llamada al sistema que carga en la memoria el programa ejecutable que se le indica en el primer parámetro, con las opciones que se le pasan a continuación. En este caso se ejecutará el programa `cat` del directorio `bin` (`/bin/cat`) al que se le pasan como argumentos su propio nombre (convenio de UNIX) así como el argumento `fifo`. `NULL` indica que ya no hay más argumentos. Si la llamada al sistema tiene éxito, el contexto y la memoria del proceso que la invoca es sustituida por el nuevo ejecutable `cat`.

APARTADO b)

El funcionamiento del programa es el siguiente:

La función principal en primer lugar comprueba que recibe dos argumentos de entrada (el nombre de la función y el argumento adicional `DyASO`), puesto que así es, no se produce ningún error.

Seguidamente se abre el fichero `fichero1.sh`. Si no se produce ningún error, tal como indica el enunciado, escribe en dicho fichero el texto "`#!/bin/bash`" seguido de un salto de línea y en la siguiente línea se escribe "`echo Ejercicio 5`".

A continuación se escribe el segundo argumento de entrada que es `DyASO`. Hecho esto el proceso cierra el fichero y le otorga permisos de lectura, escritura y ejecución para el propietario del fichero con chmod. De este modo, en el directorio actual se habrá creado un shell script de nombre `fichero1.sh` con el siguiente contenido:

```
#!/bin/bash
echo Ejercicio 5 DyASO
```

Seguidamente, se borra (si es que existe) el fichero FIFO de nombre `fifo` usando la llamada al sistema `mknod`. Tal como indica el enunciado no se producen errores por lo que el proceso continua. A continuación se realiza un `fork` que bifurca el proceso original en un proceso padre y un proceso hijo:

-HIJO: El hijo utiliza `system` para ejecutar la orden "`./fichero1.sh > fifo`" en un proceso nuevo. Esta orden ejecuta el fichero `./fichero1.sh` recién creado enviado su salida estándar (`Ejercicio 5 DyASO`) al fichero FIFO.

-PADRE: El padre ejecuta la orden "`cat fifo`" mediante la llamada al sistema `execl`. Esta orden machaca el contexto del proceso padre con el nuevo ejecutable cat. Por lo que, si no se producen errores (tal como indica el enunciado) las órdenes que hay a continuación nunca se ejecutan y **NO** se imprime ¡Incorecto! en la salida estándar.

-HIJO: Una vez que termine de ejecutarse la orden invocada con `system`, el proceso el proceso hijo recibe la condición de salida del subshell que queda almacenada en la variable `salida`. Al no haberse producido ningún error la condición de salida es cero y por lo tanto se imprime "`¡Correcto!`" seguido de un salto de línea. Finalmente se eliminan con `unlink` los ficheros `fifo` y `fichero1.sh`.

De este modo la salida completa del programa es:

```
Ejercicio 5 DyASO
¡Correcto!
```

NOTA: Sea cual sea el orden en el que el SO planifique los procesos padre e hijo, la salida es siempre la misma ya que la lectura o escritura del fichero FIFO produce un bloqueo hasta que haya (al menos) un proceso lector y otro escritor. De este modo, el hijo quedará suspendido hasta que el padre lea el contenido de la tubería.

NOTA2: Es importante destacar la diferencia entre `system` y `exec` (en cualquiera de sus variantes `execv execl`…). `system` utiliza un proceso separado y el proceso original queda a la espera de que el proceso nuevo termine para continuar la ejecución. Por el contrario `exec` machaca el contexto del proceso que la invoca que queda sustituido por el nuevo ejecutable, de modo que las órdenes que hay a continuación de `system` se ejecutarán mientras que las que siguen a `exec` no lo harán.