Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N	**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. (*1 p*). Explique **razonadamente** si las siguientes afirmaciones son verdaderas o falsas:

a) (*0.5 p*) `specfs` mantiene un `nodo-s` común por cada proceso que quiere acceder a un dispositivo.
b) (*0.5 p*) En el UNIX SVR3, para tratar un fallo de validez el proceso invoca al manipulador de fallos de validez, que necesita como argumento de entrada la dirección virtual ha provocado dicho fallo.

RESPUESTA:

a) Es **FALSA** ya que `specfs` mantiene un único nodo común por cada dispositivo para centralizar todas las operaciones de acceso al mismo (evitando por ejemplo que un proceso cierre un dispositivo mientras otro proceso lo está usando) (ver 9.5.4 El `nodo-s` común y en particular la figura 9.4).
b) Es **FALSA**. El núcleo es quien invoca al manipulador de fallos de validez (Apartado 7.7.1).

2. (*2 p*). Responda razonadamente las siguientes preguntas:

a) (*1 p*). Indique en detalle el funcionamiento y sintaxis del programa `fsck`.
b) (*1 p*). Señale las inconsistencias que revisa `fsck`.

RESPUESTA:

(Apartado 8.9 del libro de texto)

a) Para evitar errores, con cierta frecuencia se debe revisar el sistema de ficheros para verificar su consistencia y asegurar que todos sus bloques son accesibles. Esto se consigue con el programa `fsck`. Su sintaxis es la siguiente:

```
$ fsck [-opciones] [sistema...]
```

`fsck` revisa y repara de forma interactiva las posibles inconsistencias que encuentra en los sistemas de ficheros UNIX. En el caso de que no existan inconsistencias, `fsck` informa sobre el número de ficheros, número de bloques usados y número de bloques libres de que dispone el sistema. Si el sistema presenta inconsistencias, `fsck` proporciona mecanismos para corregirlo.

`fsck` revisa los sistemas de ficheros que se le indican en la línea de órdenes. Si no se especifica ningún sistema, `fsck` revisa los sistemas que se especifican en la tabla de montaje.

Si `fsck` encuentra un fichero o directorio cuyo directorio padre no puede determinarse, colocará el fichero huérfano en el directorio `lost+found` perteneciente al sistema que se está revisando. Puesto que el nombre del fichero se registra en su directorio `home` y éste es desconocido, a la hora de guardarlo en `lost+found` se nombrará con su número de `nodo-i`.

Aparte de revisar un sistema de ficheros recién creado `fsck` se utiliza principalmente para revisar sistemas estropeados por alguna causa accidental como una parada imprevista del sistema. El núcleo mantiene copias en memoria tanto del superbloque como de algunos `nodos-i` (*`nodos-im`*). Además el acceso a disco se realiza a través de la caché de buffers de bloques de disco. Esto crea inconsistencias de contenido entre el disco y la memoria. Estas inconsistencias se corrigen periódicamente con la intervención de los procesos demonio `syncer` o `update` que se encargan de invocar a la llamada al sistema `sync` para actualizar el disco con la memoria. Si por cualquier circunstancia el sistema deja de funcionar antes de que se produzca una actualización, la próxima vez que se intente utilizar esos sistemas de ficheros, será necesario repararlo dentro de la posible con la ayuda de `fsck`.

b) Las inconsistencias que revisa `fsck` son las siguientes:

- Bloques reclamados por más de un `nodo-i` o la lista de bloques libres.
- Bloques reclamados por un `nodo-i` o la lista de bloques libres, pero que están fuera del rango del sistema.
- Contadores de enlaces incorrectos.
- Número de bloques demasiado grande y tamaño de directorios inadecuados.
- Formato inadecuado para los `nodos-i`.
- Bloques no registrados por nadie (nodos-i, lista de bloques libres, etc.).
- Revisión de los directorios en busca de ficheros que apuntan a `nodos-i` no asignados o números de `nodo-i` fuera de rango.
- Existencia en el superbloque de más bloques para `nodos-i` de los que hay en el sistema de ficheros.
- Formato incorrecto de la lista de bloques libres.
- Total de bloques libres o contador de `nodos-i` incorrecto.

3. (*2 p*). Responde razonadamente las siguientes preguntas:

a) (*1 p*) ¿Qué son las hebras? Señala los tipos de hebras que existen.

b) (*1 p*) Describe qué son las hebras de núcleo.

RESPUESTA:

(Apartado 4.9 libro de texto)

a) La hebra es un objeto dinámico que representa un punto de control en el proceso y que ejecuta una secuencia de instrucciones. Los recursos (espacio de direcciones, ficheros abiertos, credenciales de usuario, cuotas,...) son compartidos por todas las hebras de un proceso. Además cada hebra tiene sus objetos privados, tales como un contador de programa, una pila y un contador de registro. Un proceso UNIX tradicional tiene una única hebra de control. Los sistemas multihebras como son SVR4 y Solaris extienden este concepto permitiendo más de una hebra de control en cada proceso.

En función de sus propiedades y usos se distinguen tres tipos diferentes de hebras:

- *Hebras del núcleo*: son objetos primitivos no visibles para las aplicaciones.
- *Procesos ligeros*: son hebras visibles al usuario que son reconocidas por el núcleo y que están basadas en hebras del núcleo.
- *Hebras de usuario*: Son objetos de alto nivel no visibles para el núcleo. Pueden utilizar procesos ligeros, si éstos son soportados por el núcleo, o pueden ser implementadas en un proceso UNIX tradicional sin un apoyo especial por parte del núcleo.

b) Una hebra del núcleo no necesita ser asociada con un proceso de usuario. Es creada y destruida internamente por el núcleo cuando la necesita. Se utiliza para ejecutar una función específica como por ejemplo, una operación de E/S o el tratamiento de una interrupción. Comparte el código del núcleo y sus estructuras de datos globales. Además posee su propia pila del núcleo. Puede ser planificada independientemente y utiliza los mecanismos de sincronización estándar del núcleo, tales como `sleep()` y `wakeup()`.

Las hebras del núcleo resultan económicas de crear y usar, ya que los únicos recursos que consumen son la pila del núcleo y un área para salvar el contexto a nivel de registros cuando no se están ejecutando. También necesitan de alguna estructura de datos para mantener información sobre planificación y sincronización. Asimismo, el cambio de contexto entre hebras del núcleo se realiza rápidamente.

Las hebras del núcleo no son un concepto nuevo. Los procesos del sistema tales como el ladrón de páginas en los núcleos de UNIX tradicionales son funcionalmente equivalentes a las hebras del núcleo.

4. (*2 p*) Se tiene un directorio con un único fichero de texto "`disnak.txt`" cuyo contenido es "`manantial`". El propietario del fichero invoca las siguientes acciones en un terminal:

```
$ ln disnak.txt texto.txt
$ ln -s disnak.txt notas.txt
$ rm disnak.txt
```

Responde razonadamente a las siguientes preguntas:

i) (1 *p*) ¿Qué es un enlace duro? ¿Qué es un enlace simbólico?
ii) (*0.5 p*) ¿Cuánto vale el contador de referencias de "`notas.txt`"?
iii) (*0.5 p*) Señala la sentencia que hay que invocar para que se imprima el contenido original del fichero `disnak.txt`, ("manantial").

RESPUESTA:

i) Un enlace duro es cada uno de los nombres que se asignan a un determinado `nodo-i` en un sistema de ficheros. Un fichero no se borra mientras exista al menos un enlace duro al mismo. El contador de referencias lleva la cuenta del número de enlaces duros (nombres) que hay asignados a un determinado `nodo-i`.

Un enlace simbólico por el contrario es tan solo un fichero que contiene la ruta de acceso a otro archivo y por lo tanto no incrementa el contador de referencias del `nodo-i` del fichero referenciado (Ver apartado 8.5, Pág. 347 y siguientes).

ii) El fichero `notas.txt` recién creado tiene el contador de referencias a 1, puesto que es un enlace simbólico, esto es, un fichero con su propio nodo-i que solo tiene una referencia (`notas.txt`).

iii) El usuario primeramente crea un enlace duro a `disnak.txt` (usando `ln`) cuyo nombre es `texto.txt`. Esto incrementa el contador de referencias de dicho `nodo-i` (que en caso de no existir más referencias pasaría a valer 2).

La segunda sentencia crea un enlace simbólico (`ln -s`) de nombre `notas.txt` hacia el mismo archivo `disnak.txt;` pero esto no afecta en absoluto al `nodo-i` del archivo `disnak.txt`.

Finalmente la tercera sentencia borra el nombre `disnak.txt`, decrementando el contador de referencias de su `nodo-i` pero no elimina el archivo, puesto que todavía existe un enlace duro que apunta hacia él (`texto.txt`).

Por tanto, La sentencia que hay que invocar es

"`cat texto.txt`"

Puesto que el fichero original cuyo contenido era "`manantial`" sigue existiendo y es accesible a través del enlace duro "`texto.txt`".

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*) El programa es compilado bajo el nombre `Examen` y se invoca la orden "`$ mknod yAS p`" sin producirse ningún error. Describir el funcionamiento así como la salida obtenida salida cuando se invoca a continuación "`$ ./Examen yAS`".

```
        #include <stdio.h>
        #include <string.h>
        #include <stdlib.h>
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>
        #include <unistd.h>
        #define MAX 256

[1]     void main(int argc, char *argv[])
             {
[2]          if (argc!=2)        {
                  printf("Argumentos incorrectos \n");
                  exit(1);
                  }
             int fd, pid;
             char buffer[MAX];
[3]          if((fd=open(argv[1],O_RDWR))==-1){perror("Error open:"); exit(2);}
[4]          if((pid=fork())==-1) {perror("Error en fork:"); exit(2);}
             if(pid==0)
                  {
[5]               if(read(fd,buffer,4)>0)
                       {
                       printf("%s", buffer);
                       }
                  else
                       {
                       perror("error read:");
                       close(fd);
                       exit(3);
                       }
                  }
             else
                  {
[6]               sleep(2);
                  printf("\n%s D", argv[0]);
                  fflush(stdout); //fuerza impresion imediata
[7]               if(write(fd,argv[1],strlen(argv[1])+1)<0)
                       {
                       perror("error write:");
[8]                    close(fd);
                       exit(4);
                       }
[9]               wait();
                  printf("O.\n\n");
                  }
             close(fd);
             }
```

RESPUESTA

Apartado a):

[1] Se declara la función principal (`main`) que admite como entada un array de cadenas `argv[]` de longitud `argc`, siendo el primer parámetro de entrada el nombre con el que ha sido invocado el fichero ejecutable.

[2] Si el número de argumentos es distinto de dos (el nombre del ejecutable y un argumento adicional) se muestra por la salida estándar el mensaje "`Argumentos incorrectos`" seguido de un salto de línea.

[3] La llamada al sistema `open()` abre el fichero cuyo nombre es pasado como segundo argumento a la función (`argv[0]`) en modo de lectura y escritura (`O_RDWR`) y almacena en `fd` un número entero denominado descriptor de fichero que permite realizar posteriormente operaciones sobre el mismo. Si se produce un error la llamada al sistema devuelve -1, esto hace que se cumpla la condición del `if`. Si esto ocurre, se ejecutará primeramente `perror()` que imprime "`Error open:`" seguido del descriptor del error que se ha producido en la llamada al sistema anterior (almacenado en la variable global `errno`), a continuación la llamada al sistema `exit()` terminaría el proceso devolviendo un código de terminación 2.

[4] La llamada al sistema `fork()` crea un proceso hijo al que devuelve 0, mientras que al padre le devuelve el pid del hijo que se almacena en la variable `pid`. En caso de producirse un error devolverá `-1`, cumpliéndose entonces la condición de la cláusula `if` que imprimirá "`Error en fork:`" seguido del descriptor del error y terminara el proceso.

[5] La sentencia `read()` es una llamada al sistema que intenta leer cuatro bytes del fichero cuyo descriptor es `fd` y los almacena en la variable `buffer`. La función devuelve el número de bytes leídos. Si la función ha leído algún carácter se ejecutaría la cláusula del `if`. En caso contrario el `else` siguiente.

[6] La llamada al sistema `sleep(2)` duerme al proceso que la invoca durante dos segundos.

[7] La llamada al sistema `write`, escribe en el fichero cuyo descriptor es `fd` la cadena de caracteres que ha sido pasada como segundo argumento a la función principal (`argv[1]`). El número de caracteres a escribir es igual a la longitud de la cadena de entrada `strlen(argv[1])` incrementada en una unidad (carácter nulo de fin de cadena). Como en el caso de la función `read` **[5]** el `if` verifica que no se produzcan errores en la escritura.

[8] `close(fd)` cierra el fichero cuyo descriptor es `fd`.

[9] La llamada al sistema `wait()` hace que el proceso padre espere a la finalización del proceso hijo (o a la recepción de otra señal) antes de continuar.

Apartado b)

Cuando el usuario invoca la orden "`$ mknod yAS p`" se crea en el directorio actual un fichero FIFO cuyo nombre es `yAS`. A continuación se invoca el programa con la orden "`$ ./Examen yAS`" de modo que el programa recibe dos argumentos de entrada, el primero de ellos es "`./Examen`" y el segundo "`yAS`".

El funcionamiento del programa es el siguiente:

En primer lugar el programa comprueba que el número de argumentos de entrada es 2. En caso contrario mostraría un error "`Argumentos incorrectos`" y finalizaría su ejecución. Como en este caso hay dos argumentos el programa continúa.

A continuación, el programa abre el fichero FIFO `yAS` en modo de lectura y escritura y comprueba si se producen errores. Si los hubiese el programa los mostraría con `perror` y terminaría. Puesto que el usuario ha creado dicho fichero FIFO y no se ha producido error alguno el programa prosigue su ejecución normalmente.

La llamada al sistema `fork()` crea un proceso hijo que es una copia del proceso actual. El proceso hijo recibe 0 en la variable `pid y` por tanto ejecuta el bloque de código siguiente al `if.` Por su parte, el proceso padre recibe el pid del hijo y por lo tanto ejecuta el bloque de código correspondiente al `else.`

El proceso hijo intenta leer 4 bytes del **fichero FIFO** `yAS` y mostrarlos por pantalla. En este punto es importante destacar que aunque la lectura `read()` es totalmente similar a la lectura de un fichero normal **el comportamiento es distinto**. Si se estuviese leyendo un fichero ordinario el programa intentaría leer 4 bytes, pero al estar el fichero vacío devolvería 0 y terminaría.

Sin embargo, puesto que `yAS` **es un fichero FIFO**, el proceso quedará **bloqueado** hasta que haya algún dato que leer. Una vez que el dato esté disponible el proceso hijo mostrará el contenido leído.

Por otra parte, el proceso padre se mantendrá a la espera durante dos segundos y posteriormente escribirá por salida estándar un salto de línea seguido del nombre con el que ha sido ejecutado y a continuación un espacio y una `D.` La sentencia `fflush(stdout)`, tal y como indica el comentario que la acompaña, hace que esta línea sea impresa inmediatamente sin esperar a que haya un salto de línea en el flujo de salida. Esto garantiza que la línea se imprime antes de que el proceso continúe.

A continuación, el padre escribe el segundo argumento con el que ha sido invocada la función (`yAS`) en el fichero FIFO y se queda a la espera de que termine el proceso hijo con `wait().` Cuando esto ocurre imprime `"O."` seguido de dos saltos de línea.

Finalmente tanto el padre como el hijo ejecutan `close(fd)` cerrando el fichero y finalizando su ejecución.

Sea cual sea el orden de planificación de ambos procesos, el hijo permanecerá a la espera hasta que el padre escriba `"yAS"` en el fichero FIFO, por lo que el padre será siempre el primero en escribir en la salida estándar el salto de línea seguido de `"./Examen D".` Además, el padre espera a la terminación del hijo antes de escribir `".O"` de modo que el hijo escribirá `"yAS"` intercalado entre `"./Examen D".` y `".O",` por lo que la salida es siempre:

```
./Examen DyASO.
```