DISEÑO Y ADMINISTRACIÓN DE SISTEMAS OPERATIVOS **Septiembre 2013**

Material permitido: **NINGUNO.** 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. Explique **razonadamente** si las siguientes afirmaciones son verdaderas o falsas:

a) (*1 p*) La llamada al sistema `msgrcv` permite que un proceso pueda enviar un mensaje.

b) (*1 p*) Un nodo-i se modifica solamente cuando se cambia el contenido de un archivo.

Solución:

a) FALSA. La llamada al sistema `msgrcv` permite que un proceso pueda extraer un mensaje de una determinada cola de mensajes.

b) FALSA. Las modificaciones en el nodo-i pueden producirse sin alterar el contenido del archivo (por ejemplo si se accede al archivo, se crea un enlace duro o se cambian los permisos de acceso).

2. *(1.5 p)* Describe las principales limitaciones que presentan las señales.

Solución:

Como mecanismo IPC, las señales poseen varias limitaciones:

- Las señales resultan costosas en relación a las tareas que suponen para el sistema. El proceso que envía la señal debe realizar una llamada al sistema; el núcleo debe interrumpir al proceso receptor y manipular la pila de usuario de dicho proceso, para invocar al manipulador de la señal y posteriormente poder retomar la ejecución del proceso interrumpido.
- Tienen un ancho de banda limitado, ya que solamente existen 32 tipos de señales distintas.
- Una señal puede transportar una cantidad limitada de información.

3. (*1.5 p*) Supóngase la siguiente orden tecleada por un usuario desde el intérprete de comandos (`$`) de un sistema UNIX:

```
$ man 2 mount
```

a) (*0.75 p*) Explique **razonadamente** si dicha orden está bien escrita o contiene algún error de sintaxis.

b) (*0.75 p*) Si considera que la orden está bien escrita explique su significado. Por el contrario, si considera que está mal escrita indique como se debería escribir correctamente.

Solución:

a) La orden está bien escrita.

b) Esta orden muestra en la salida estándar (por defecto la pantalla) la página de la segunda sección del manual de ayuda asociada a `mount` que describe a la llamada al sistema que tiene ese mismo nombre.

4. (*2.5 p*) Supóngase un computador con una memoria principal de capacidad C_{Mp}=4 MiB y un tamaño de página S_P= *4* KiB. Calcular el contenido en binario y en decimal de cada uno de los campos en que se descompondría la dirección física DIR_F=*2020220*. Suponer que cada posición de memoria contiene una palabra y que esta tiene un tamaño de 1 byte.

Ayuda: 1 MiB=2^{20} bytes, 1 KiB=2^{10} bytes.

Solución:

El tamaño *n* de una dirección de memoria es el número de bits que se necesitan para codificar el número total de posiciones direccionables de memoria.

Puesto que la capacidad es $C_{Mp}=2^{22}$ bytes, supuesto que cada posición de memoria contiene una palabra y que ésta tiene un tamaño de 1 bytes, entonces:

$n = \log_2 2^{22} = 22$ bits.

Por otro lado el número total de marcos de página, se calcularía con la ecuación:

$$N_{TM} = \frac{C_{Mp}}{S_P} = \frac{2^{22}}{2^{12}} = 2^{10} = 1024$$ marcos de página.

El tamaño *m* del campo "*Número j de marco de página*" se puede obtener de la siguiente forma:

$m = \log_2 N_{TM} = \log_2 2^{10} = 10$ bits.

Y el tamaño *d* del campo DES_F, se obtiene entonces como:

$d = n - m = 22 - 10 = 12$ bits.

Pasando a binario DIR_F se obtiene:

DIR_F=0111101101 001101111100

Luego *j*= 0111101101= 493 y DES_F = 001101111100= 892

5. (*2.5 p*) Conteste razonadamente a los siguientes apartados:

a) (*1 p*) Explicar el significado de las sentencias enumeradas (**[ ]**) de este programa destacando la diferencia entre **[2]** y **[3]**.

b) (*1.5 p*) El programa una vez compilado se convierte en el ejecutable `pr1`. Explicar el funcionamiento del programa cuando se invoca desde la línea de comandos (`$`) de la siguiente manera: `$ ./pr1 archivo1 archivo2`.

```
     #include <fcntl.h>
     #include <stdio.h>
     #include <stdlib.h>
     void main(int np, char* a[])
     {
     int fd1, fd2;
     FILE* fd3;
     char buf[1], mensaje[]="nemaxe";

     if(np==3)
     {
[1]  fd1=creat(a[1],0600);
[2]  fd2=open(a[1],O_RDONLY);
[3]  fd3=fopen(a[2],"w");

[4]  if (fd1==-1 || fd2==-1 || fd3==NULL)
          {printf("Error creando los ficheros");}
     else
          {
[5]            write(fd1,mensaje,6);
               close(fd1);

               lseek(fd2,0,SEEK_END);
```

```
                while(0<=lseek(fd2,-1,SEEK_CUR))
                {
[6]                   read(fd2,buf,1);
[7]                   lseek(fd2,-1,SEEK_CUR);
                      fwrite(buf,1,1,fd3);
                }

          }
[8]  close(fd2);
     fclose(fd3);

     }
     else {printf("Argumentos incorrectos");}

     }
```

Solución:

a) El significado de las sentencias numeradas es el siguiente:

[1]: La llamada al sistema `creat` crea un fichero cuyo nombre es el segundo argumento pasado a la función `main` a partir de la ruta actual con permisos de lectura y escritura para el usuario que ejecuta el programa (que es el propietario del fichero recién creado). Si el archivo ya existía simplemente borra su contenido. La llamada al sistema devuelve un identificador `fd1` del objeto de fichero abierto recién creado.

[2]: La llamada al sistema `open` abre el archivo que se acaba del crear en la llamada anterior devolviendo **otro** objeto de fichero abierto `fd2` para trabajar en modo de solo lectura (`O_RDONLY`). En este momento existen dos objetos de fichero abierto que apuntan al mismo archivo pero que pueden usarse de forma independiente. Esta sentencia es una llamada al sistema lo que significa que el programa está trabajando en contacto directo con el sistema operativo.

[3]: La llamada a la función de librería estándar ANSI C `fopen` abre un archivo cuyo nombre es el tercer argumento de la función `main` en modo de escritura (`w`). Hay que hacer notar que esta función de librería, es parte del lenguaje C y proporciona un API de acceso a los archivos que es **independiente del sistema operativo** mientras que la llamada al sistema [2] es totalmente dependiente del sistema operativo. Cuando se llama `fopen`, la librería se encarga de llamar internamente a `open` para cumplir con su función sin que el programador necesite conocer los detalles de las llamadas al sistema.

[4]: Esta sentencia comprueba los descriptores de fichero para ver si los archivos se han abierto correctamente, en caso de error en la llamada al sistema `open` el descriptor de fichero será `-1`, mientras que en el caso de la función de librería `fopen` el resultado será un puntero nulo (`NULL`).

[5]: La llamada al sistema `write` escribe el mensaje "`nemaxe`" (cuya longitud son 6 bytes y que se encuentra almacenado en el buffer `mensaje)` en el fichero cuyo descriptor es `fd1`.

[6]: La llamada al sistema `read` lee un byte del descriptor de fichero `fd2` y lo almacena en `buf` .

[7]: La llamada al sistema `lseek` hace retroceder 1 posición el puntero de lectura del archivo desde la posición actual (`SEEK_CUR`).

[8]: La llamada al sistema `close` cierra el objeto de fichero abierto al que corresponde el descriptor `fd1`;

b) El funcionamiento del programa es el siguiente:

En primer lugar se comprueba que el número de argumentos de la función es 3, (en caso contrario se mostraría el mensaje "`Argumentos incorrectos`" y finalizaría el programa). Hay que recordar que los argumentos que recibe la función `main` en este caso son: `pr1` (el nombre del ejecutable que ha sido llamado) `archivo1` y `archivo2`.

Como la comprobación es correcta se crea un archivo vacío cuyo nombre será `archivo1` y un objeto de archivo abierto cuyo descriptor es `fd1`. A continuación se vuelve a abrir el mismo archivo, creando así un segundo objeto de archivo abierto cuyo descriptor es `fd2`, esta vez en modo de solo lectura. Por último se abre (y en caso de no existir se crea) un segundo archivo, `archivo2` en modo de lectura devolviendo un puntero a un objeto de tipo `FILE`, llamado `fd3` que es un descriptor de archivo para las funciones de librería ANSI C.

A continuación se comprueba que los archivos se han abierto correctamente, en caso negativo se muestra el mensaje de error "`Error cerrando ficheros`".

Si la apertura ha sido correcta se utiliza el descriptor de fichero `fd1` para escribir el mensaje "`nemaxe`" en el `archivo1` y después se cierra el objeto de fichero abierto cuyo descriptor es `fd1`.

Aunque se ha cerrado el objeto de fichero abierto con identificador `fd1`, el fichero `archivo1` sigue abierto y es accesible a través del objeto de fichero abierto cuyo identificador es `fd2`. Para ello se posiciona el puntero de lectura/escritura al final del fichero `archivo1` con la llamada al sistema `lseek` y la opción `SEEK_END`.

A continuación se entra en un bucle en el cual se usa el descriptor de archivo `fd2` recorre `archivo1` desde el final hasta el principio escribiendo el contenido en orden inverso en `archivo2`.

En la condición del bucle se hace retroceder el puntero de lectura/escritura y se comprueba si se ha terminado el fichero (la posición de lectura es <0) y en caso afirmativo el bucle termina. En caso negativo se lee un byte del fichero y se almacena en `buf`, después se hace retroceder una posición el puntero de lectura/escritura para que vuelva a apuntar al mismo sitio que antes de la lectura y por último se escribe el contenido de `buf` en `archivo2`.

Al terminar el programa se cierran todos los objetos de fichero abiertos que quedan usando la llamada al sistema `close` para cerrar `fd2` y la función de librería `fclose` para cerrar `fd3`. Cuando el programa termina hay dos archivos cuyo contenido es el siguiente:

Archivo1: `nemaxe`
Archivo2: `examen`