

A decorative graphic on the right side of the page. It features three blue circles of different sizes, each composed of concentric rings of varying shades of blue. Two thin blue lines intersect at a point between the top two circles, extending towards the top-left and bottom-right corners of the page. A third thin blue line extends from the bottom-right corner towards the bottom-right circle.

Documentación de laboratorio

Curso primavera 2011-2012

Este documento contiene las sesiones a realizar durante las clases de laboratorio. Las sesiones incluyen el trabajo previo y el trabajo a realizar durante la sesión

Profesores SO-Departamento AC
08/02/2012

Índice de sesiones

Sesión 1: Introducción.....	3
Sesión 2: Introducción.....	16
Sesión 3: Procesos.....	22
Sesión 4: Procesos.....	28
Sesión 5: Procesos.....	34
Sesión 6: Gestión de Memoria	40
Sesión 7: Gestión de Memoria II	44
Sesión 8: Gestión de Entrada/Salida I	48
Sesión 9: Gestión de Entrada/Salida III	55
Sesión 10: Sistema de Ficheros	59

Sesión 1: Introducción

Preparación previa

1. Objetivos

El objetivo de esta sesión es que aprendáis a desenvolveros en el entorno de trabajo de los laboratorios. Veremos que algunas operaciones se pueden hacer tanto con comandos interactivos como utilizando el gestor de ventanas. Nos centraremos en la práctica de algunos comandos básicos y en la utilización del manual online (`man`) que encontraréis en todas las máquinas Linux.

2. Habilidades

- Ser capaz de utilizar las páginas de `man`.
- Ser capaz de utilizar comandos básicos de sistema para modificar/navegar por el sistema de ficheros: `cd`, `ls`, `mkdir`, `cp`, `rm`, `rmdir`, `mv`.
- Conocer los directorios especiales `."` y `.."`.
- Ser capaz de utilizar comandos básicos de sistema y programas de sistema para acceder a ficheros: `less`, `cat`, `grep`, `gedit` (u otro editor).
- Ser capaz de modificar los permisos de acceso de un fichero.
- Ser capaz de consultar/modificar/definir una variable de entorno.
- Ser capaz de utilizar algunos caracteres especiales de la Shell (intérprete de comandos):
 - `&` para ejecutar un programa en segundo plano (ejecutar en background).
 - `>` para guardar la salida de un programa (redireccionar la salida).

3. Conocimientos previos

En esta sesión no se requieren conocimientos previos.

4. Guía para el trabajo previo

4.1. Acceso al sistema

En los laboratorios tenemos instalado Ubuntu 10.04.LTS. Para comenzar, ejecutaremos lo que llamamos una *Shell* o un intérprete de comandos. Existen varios intérpretes de comandos, en el laboratorio utilizaréis Bash (GNU-Bourne Shell), pero en general nos referiremos a él como Shell. La mayoría de las cosas que explicaremos en esta sesión podéis leerlas en el manual de Bash (ejecutando el comando **`man bash`**). Encontraréis varios usuarios creados para que podáis hacer las pruebas que os pedimos fácilmente. Los usernames de los usuarios son: "alumne", "so1", "so2", "so3", "so4" y "so5". El password es "sistemas" para todos ellos.

Para ello ejecutaremos el “Terminal” y se nos abrirá una ventana similar a la de la imagen. Una *Shell* es un programa que el S.O. nos ofrece para poder trabajar en un modo de texto interactivo. Este entorno puede parecer menos intuitivo que un entorno gráfico, pero es muy sencillo y potente.

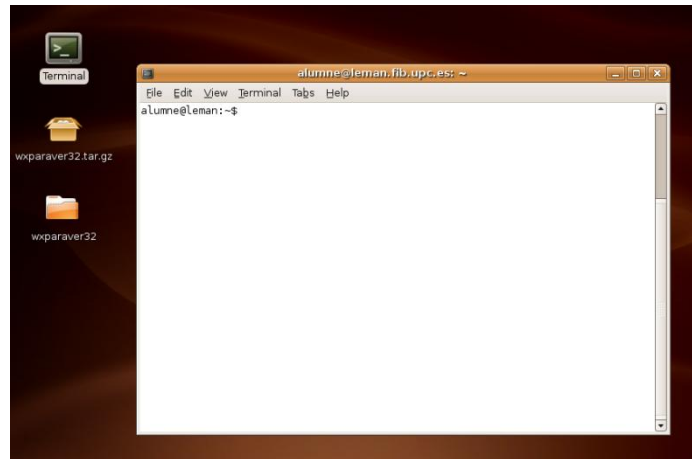


Figura 1 Ventana de la shell

El texto que aparece a la izquierda junto con el cursor que parpadea es lo que se conoce como *prompt* y sirve para indicar que la Shell está lista para recibir nuevas órdenes o comandos. Nota: en la documentación de laboratorio utilizaremos el carácter # para representar el prompt e indicar que lo que viene a continuación es una línea de comandos (para probar la línea NO DEBÉIS ESCRIBIR #, sólo el comando que aparece a continuación).

El código de la Shell se podría resumir de la siguiente manera:

```
while(1){
    comando=leer_comando();
    ejecutar_comando(comando);
}
```

Existen dos tipos de comandos: comandos externos y comandos internos. Los **comandos externos** son cualquier programa instalado en la máquina y los **comandos internos** son funciones implementadas por el intérprete de comandos (cada intérprete implementa los suyos, los hay comunes a todos ellos y los hay propios).

4.2. Comandos para obtener ayuda

En Linux, existen dos comandos que podemos ejecutar de forma local en la máquina para obtener ayuda interactiva: el comando **man**, que nos ofrece ayuda sobre los comandos externos, y el comando **help**, que nos ofrece ayuda sobre los comandos internos (como parte de la instalación, se instalan también las páginas del manual que podremos consultar a través del man). Los comandos internos

- **Leed la guía de cómo utilizar el man** de Linux que podéis encontrar al final de esta sección (“Utilización del manual”). A continuación, **consultad el man** (`man nombre_comando`) de los siguientes comandos. En concreto, para cada comando debéis leer y entender perfectamente: la SYNOPSIS, la DESCRIPTION y las opciones que os comentamos en la columna “Opciones” de la tabla.

Para leer con el man	Descripción básica	Opciones
man	Accede a los manuales on-line	
ls	Muestra el contenido del directorio	-l, -a
alias	Define un nombre alternativo a un comando	
mkdir	Crea un directorio	
rmdir	Elimina un directorio vacíos	
mv	Cambia el nombre de un fichero o lo mueve a otro directorio	-i
cp	Copia ficheros y directorios	-i
rm	Borra ficheros o directorios	-i
echo	Visualiza un texto (puede ser una variable de entorno)	
less	Muestra ficheros en un formato apto para un terminal.	
cat	Concatena ficheros y los muestra en su salida estándar	
grep	Busca texto (o patrones de texto) en ficheros	
gedit	Editor de texto para GNOME	
env	Ejecuta un comando en un entorno modificado, si no se le pasa comando, muestra el entorno	
chmod	Modifica los permisos de acceso a un fichero.	

- **Utilizad el comando help** para consultar la ayuda sobre los siguientes comandos internos:

Para consultar con el help	Descripción básica	Opciones
help	Ofrece información sobre comandos internos de la Shell	
export	Define una variable de entorno	
cd	Cambia el directorio (carpeta) actual	
alias	Define un nombre alternativo a un comando	

- **Acceded a la página del man para el bash** (ejecutando el comando `man bash`) y buscad el significado de las variables de entorno PATH, HOME y PWD (nota: el carácter “/” sirve para buscar patrones en las páginas del man. Utilizadlo para encontrar directamente la descripción de estas variables).

Utilización del manual

Saber utilizar el manual es básico ya que, aunque durante el curso os explicaremos explícitamente algunos comandos y opciones, el resto (incluido llamadas a sistema) deberéis buscarlo vosotros mismos en el manual. El propio man es auto contenido en este sentido, ya que podéis ejecutar para ver todas sus opciones:

```
# man man
```

La información del manual está organizada en secciones. La sección 2, por ejemplo, es la de llamadas a sistema. Las secciones que podemos encontrar son:

- 1. comandos
- 2. llamadas a sistema
- 3. llamadas a librerías de usuario o del lenguaje
- 4. etc.

La información proporcionada al ejecutar el man es lo que se conoce como “página de man”. Una “página” suele ser el nombre de un comando, llamada a sistema o llamada a función. Todas las páginas de man siguen un formato parecido, organizado en una serie de partes. En la Figura 2 tenéis un ejemplo de la salida del man para el comando ls (hemos borrado alguna línea para que se vean las principales partes). En la primera parte podéis encontrar tanto el nombre del comando como la descripción y un esquema (SYNOPSIS) de su utilización. En esta parte podéis observar si el comando acepta opciones, si necesita algún parámetro fijo u opcional, etc.

LS(1)	User Commands	LS(1)
NAME ls - list directory contents		
SYNOPSIS ls [OPTION]... [FILE]...		
DESCRIPTION List information about the FILES (the current directory by default). Sort entries alphabetically if none of -cftuSUX nor --sort. Mandatory arguments to long options are mandatory for short options too. -a, --all do not ignore entries starting with .		
SEE ALSO The full documentation for ls is maintained as a Texinfo manual. If the info and ls programs are properly installed at your site, the command info ls should give you access to the complete manual.		
ls 5.93	November 2005	LS(1)

Figura 2 man ls (simplificado)

La siguiente parte sería la descripción (DESCRIPTION) del comando. Esta parte incluye una descripción más detallada de su utilización y la lista de opciones que soporta. Dependiendo de la instalación de las páginas de man también podéis encontrar aquí los códigos de finalización del comando (EXIT STATUS). Finalmente suele haber una serie de partes que incluyen los

autores de la ayuda, la forma de reportar errores, ejemplos y comandos relacionados (SEE ALSO).

En la Figura 3 tenéis el resultado de ejecutar “man 2 write”, que corresponde con la llamada a sistema write. El número que ponemos antes de la página es la sección en la que queremos buscar y que incluimos en este caso ya que existe más de una página con el nombre write en otras secciones. En este caso la SYNOPSIS incluye los ficheros que han de ser incluidos en el programa C para poder utilizar la llamada a sistema en concreto (en este caso unistd.h). Si fuera necesario “linkar” vuestro programa con alguna librería concreta, que no fueran las que utiliza el compilador de C por defecto, lo normal es que aparezca también en esta sección. Además de la DESCRIPTION, en las llamadas a función en general (sea llamada a sistema o a librería del lenguaje) podemos encontrar la sección RETURN VALUE (con los valores que devuelve la función) y una sección especial, ERRORS, con la lista de errores. Finalmente también encontramos varias secciones donde aquí destacamos también la sección de NOTES (aclaraciones) y SEE ALSO (llamadas relacionadas).

WRITE(2)	Linux Programmers Manual	WRITE(2)
NAME	write - write to a file descriptor	
SYNOPSIS	<pre>#include <unistd.h> ssize_t write(int fd, const void *buf, size_t count);</pre>	
DESCRIPTION	write() writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data. Note that not all file systems are POSIX conforming.	
RETURN VALUE	On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately. If count is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect. For a special file, the results are not portable.	
ERRORS	EAGAIN Non-blocking I/O has been selected using O_NONBLOCK and the write would block. EBADF fd is not a valid file descriptor or is not open for writing. Other errors may occur, depending on the object connected to fd.	
NOTES	A successful return from write() does not make any guarantee that data has been committed to disk. In fact, on some buggy implementations, it does not even guarantee that space has successfully been reserved for the data. The only way to be sure is to call fsync(2) after you are done writing all your data.	
SEE ALSO	close(2), fcntl(2), fsync(2), ioctl(2), lseek(2), open(2), read(2), select(2), fwrite(3), writev(3)	

Figura 3 man 2 write (simplificado)

El man es simplemente una herramienta del sistema que interpreta unas marcas en ficheros de texto y las muestra por pantalla siguiendo las instrucciones de esas marcas. Las tres cosas básicas que tenéis que saber son:

- Normalmente una página de man ocupa varias pantallas, para ir avanzando simplemente hay que apretar la barra espaciadora.

- Para ir una pantalla hacia atrás podéis apretar la letra **b** (back).
- Para buscar un texto e ir directamente podéis usar el carácter “/” seguido del texto. Por ejemplo “/SEE ALSO” os llevaría directo a la primera aparición del texto “SEE ALSO”. Para ir a la siguiente aparición del mismo texto simplemente utilizad el carácter **n** (next).
- Para salir del man utilizad el carácter **q** (quit)

5. Bibliografía

- Capítulo 2: Operating-System Structures
- Guía de BASH shell:
 - De la asignatura ASO (en catalán):
<http://docencia.ac.upc.edu/FIB/ASO/files/lab10-2q/aso-lab-bash-guide.pdf>
 - En inglés: <http://tldp.org/LDP/abs/html/index.html>
 - Tutorial de uso de Shell: http://www.ant.org.ar/cursos/curso_intro/c920.html

Ejercicios a realizar en el laboratorio

- Las prácticas se realizarán en un sistema Ubuntu 10.04 LTS
- Tenéis a vuestra disposición una imagen del sistema igual a la de los laboratorios para poder preparar las sesiones desde casa. La imagen que os ofrecemos es para VMPlayer:
 - http://downloads.vmware.com/d/info/desktop_downloads/vmware_player/3_0
 - Imagen: <http://softdocencia.fib.upc.es/software/ubuntu10.tgz>
- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicaréis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo: .
- Las líneas del enunciado que empiezan por el carácter “#” indican comandos que tenéis que probar. Vosotros NO tenéis que escribir el carácter #.
- Para entregar: fichero sesion01.tar.gz



#tar zcfv sesion01.tar.gz entrega.txt

1. Navegar por los directorios (carpetas en entornos gráficos)

Podréis observar que la gran mayoría de los comandos básicos en Linux son de 2 ó 3 letras que sintetizan la operación a realizar (en inglés por supuesto). Por ejemplo, para cambiar de directorio (change directory) tenemos el comando `cd`. Para ver el contenido de un directorio (list directory) tenemos el comando `ls`, etc.

En unix los directorios están organizados de forma jerárquica. El directorio base es la raíz (representada por el carácter `/`) y a partir de ahí cuelgan el resto de directorios del sistema, en

el que se sitúan archivos y directorios comunes para todos los usuarios y archivos. Además, dentro de esta jerarquía, cada usuario suele tener asignado un directorio (*home directory*), pensado para que actúe como base del resto de sus directorios y ficheros. Cuando un usuario inicia un terminal, su directorio actual de trabajo pasa a ser su home directory. Para modificar el directorio actual de trabajo puede usar el comando `cd`, que le permite navegar por toda la jerarquía de ficheros.

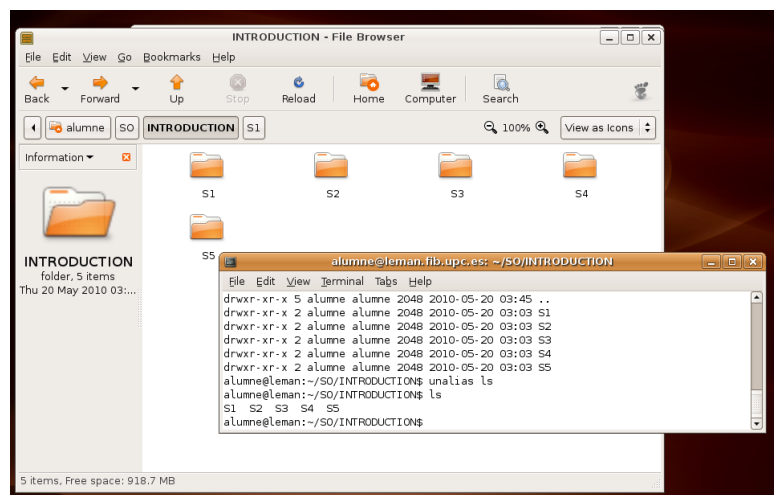
A continuación, realiza los siguientes ejercicios utilizando los comandos que consideres más oportunos:

1. Crea los directorios para las 5 primeras sesiones de la asignatura con los nombres S1, S2, S3, S4 y S5.



PREGUNTA 1: ¿Qué comandos habéis utilizado para crear los directorios S1..S5?

2. Si abris el *File Browser* del Ubuntu, y vais a la misma “carpeta” que estáis en la shell, deberíais ver algo similar a lo de esta imagen:



3. Cambiad al directorio S1.
4. Listad el contenido del directorio. Aparentemente no hay nada. Sin embargo, hay dos “**ficheros ocultos**”. Todos los ficheros que en Unix empiezan por el carácter “.” son **ficheros ocultos**, y suelen ser especiales. Consultad que opción debéis añadir al comando para ver todos los ficheros. Los ficheros que se ven ahora son:
 - Fichero de tipo directorio “.”: Hace referencia al mismo directorio en el que estáis en ese momento. Si ejecutáis (`#cd .`) veréis que seguís en el mismo directorio. Más adelante veremos qué utilidad tiene.
 - Fichero de tipo directorio “..”: Hace referencia al directorio de nivel inmediatamente superior al que estamos. Si ejecutáis (`# cd ..`) veréis que cambias al directorio de donde veníais.
 - Fijaros que estos ficheros ocultos no aparecen en el entorno gráfico, si entráis en la carpeta S1 no aparecen estos ficheros ocultos.



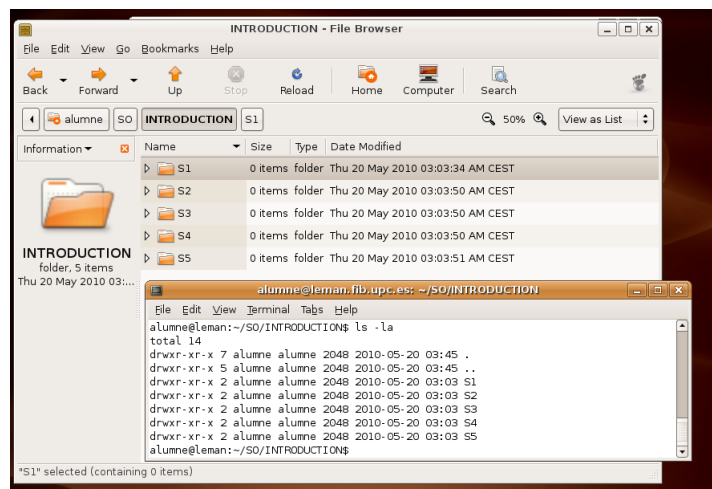
PREGUNTA 2: ¿Qué comando utilizáis para listar el contenido de un directorio? ¿Qué opción hay que añadir para ver los ficheros ocultos?

5. Las opciones de los comandos normalmente se pueden acumular. Mirad en el manual que opción hay que incluir para ver información extendida sobre los ficheros y probadlo.



PREGUNTA 3: ¿Qué opción hay que añadir a ls para ver información extendida de los ficheros? ¿Qué campos se ven por defecto con esa opción?

6. Cuando utilizamos muy a menudo una configuración específica se suele usar lo que se conoce como “alias”. Consiste en definir un pseudo-comando que la Shell conoce. Por ejemplo, si vemos que el comando ls siempre lo ejecutamos con las opciones “-la”, podemos redefinir “ls” como un alias.
 - #alias ls='ls -la'
 - Ejecutad ahora ls y comprobad como la salida incluye las opciones -la
7. Podemos ver una información similar en el entorno gráfico. Mirad como modificar el File Browser para conseguirlo. Deberíais ver algo similar a lo mostrado en la siguiente figura:



- Las columnas que se muestran aquí son las seleccionadas por defecto pero se pueden cambiar en la opción View.



PREGUNTA 4: ¿Qué opciones de menú habéis activado para extender la información que muestra el File Browser?

8. Desde la Shell borrad algunos de los directorios que habéis creado, comprobad que realmente no aparecen y volved a crearlos. Mirad como se hace lo mismo en el entorno gráfico.



PREGUNTA 5: ¿Qué secuencia de comandos habéis ejecutado para borrar un directorio, comprobar que no está y volver a crearlo?

Comandos básicos del sistema para acceder a ficheros

1. Cread un fichero. Para crear un fichero que contenga cualquier texto tenemos varias opciones, por ejemplo abrir el editor y escribir cualquier cosa:
 - #gedit test
2. Para poder ejecutar cualquier comando veréis que tenéis que abrir otra Shell porque la que teníais está bloqueada por el editor (esto sucede sólo al abrir el primer fichero, no si ya tenéis abierto el editor). Siempre tenéis esa opción. Sin embargo, tenemos una opción mucho mejor que consiste en ejecutar el editor en **segundo plano** (o en *background*). Cuando le indicamos a la Shell que ejecute un programa en segundo plano, ejecuta el programa pero nos devuelve el control para poder seguir trabajando. Para ejecutar un comando en segundo plano añadimos al final de la línea del comando el carácter especial "&". Por ejemplo:
 - #gedit test &
3. Para ver de una forma rápida el contenido de un fichero, sin abrir otra vez el editor, tenemos varios comandos. Aquí mencionaremos 2: cat y less. Añadid al fichero test 3 o 4 **páginas** de texto (cualquier cosa). Probad los comandos cat y less.
 - Nota: si el fichero que creáis no es lo suficientemente grande no veréis ninguna diferencia.



PREGUNTA 6: ¿Qué diferencia hay entre el comando cat y less?

4. Copiad el fichero test varias veces (añadiendo un número diferente al final del nombre de cada fichero destino, p.ej. "test2") ¿Qué pasaría si el fichero origen y destino tuvieran el mismo nombre? Mirad en el man la opción "-i" del comando cp. ¿Qué hace? Crea un alias del comando cp (llámalo cp) que incluya la opción -i.



PREGUNTA 7: ¿Para qué sirve la opción -i del comando cp? ¿Cuál es la orden para hacer un alias del comando cp que incluya la opción -i?

5. Probad de borrar alguno de los ficheros que acabáis de crear y a cambiarle el nombre a alguno. Haced un alias con la opción -i del comando rm (llamadlo rm). Comprobad también la opción -i del comando mv.



PREGUNTA 8: ¿Qué hacen las opciones -i y -f del comando rm? ¿Y la opción -i del mv? Escribe la orden para hacer un alias del comando rm que incluya las opciones -i y -f.

6. Otro comando que puede seros muy útil es el comando grep. El comando grep nos permite buscar un texto (explícito o mediante un patrón) en uno o más archivos. Añadid una palabra en uno de los ficheros que habéis copiado y probad el comando grep. Por ejemplo, añadimos la palabra "hola" a uno de los ficheros y hacemos la prueba:
 - #grep hola test test1 test2 test3 test4

El comando `ls -l` también permite ver los permisos que tiene un fichero. En UNIX, los permisos se aplican a tres niveles: el propietario del fichero (u), los usuarios del mismo grupo (g), y el resto de usuarios (o). Y también hacen referencia a tres modos de acceso: lectura (r), escritura (w) y ejecución (x). Por ejemplo, si en el directorio actual sólo tenemos el fichero `f1`, y este fichero tiene permiso de lectura y escritura para el propietario del fichero, sólo lectura para los miembros de su grupo y sólo lectura para el resto de usuarios de la máquina, la ejecución del comando daría la siguiente salida:

```
# ls -la
drwxr-xr-x 26 alumne alumne 884 2011-09-15 14:31 .
drwxr-xr-x 3 alumne alumne 102 2011-09-15 12:10 ..
-rw-r--r-- 1 alumne alumne 300 2011-09-15 12:20 f1
```

La primera columna de la salida indica el tipo de fichero y los permisos de acceso. El primer carácter codifica el tipo de fichero (el carácter 'd' significa directorio y el carácter '-' significa fichero de datos). Y a continuación el primer grupo de 3 caracteres representan, en este orden, si el propietario tiene permiso de lectura (mediante el carácter 'r') o no lo tiene (y entonces aparece el carácter '-'), si tiene permiso de escritura (carácter 'w') o no puede escribir (carácter '-') y si tiene o no permiso de ejecución (carácter 'x' o carácter '-'). El segundo grupo de 3 caracteres son los permisos que tienen los miembros del grupo de propietario y el último grupo de 3 caracteres son los permisos de acceso que tienen el resto de usuarios de la máquina.

Estos permisos se pueden modificar mediante el comando `chmod`. El comando `chmod` ofrece varias maneras para especificar los permisos de acceso, una manera muy sencilla consiste en indicar primero los usuarios que se van a ver afectados por el cambio de permisos, cómo queremos cambiar esos permisos (añadir, quitar o asignar directamente) y la operación afectada. Por ejemplo el comando:

```
#chmod ugo+x f1
```

Modificaría los permisos de `f1`, activando el permiso de ejecución sobre `f1` para todos los usuarios de la máquina.

El comando:

```
#chmod o-x f1
```

Modificaría los permisos de `f1` quitando el permiso de ejecución para los usuarios que no son el propietario ni pertenecen a su grupo.

Y el comando:

```
#chmod ug=rwx f1
```

Haría que los permisos de acceso a `f1` fueran exactamente los indicados: lectura, escritura y ejecución para el propietario los miembros de su grupo.

Modificad los permisos del fichero de test para dejar solamente los de escritura para mi (propietario del fichero), mi grupo y el resto de usuarios, e intentad hacer un cat.

Volved a modificar los permisos de test dejando solamente los de lectura para mi, mi grupo y el resto de usuarios e intentad borrarlo.



PREGUNTA 9: ¿Qué opciones de chmod habéis utilizado para dejar solamente los permisos de escritura? ¿Qué resultado a devuelvo cat? ¿Qué opciones de chmod habéis utilizado para dejar solamente los permisos de lectura? ¿Habéis conseguido borrarlo?

Variables de entorno

Los programas se ejecutan en un determinado entorno o contexto: pertenecen a un usuario, a un grupo, a partir de un directorio concreto, con una configuración de sistema en cuanto a límites, etc. Más detalles del contexto o entorno de un programa los veréis en el tema de procesos. En esta sesión introduciremos las **variables de entorno**. Las variables de entorno son similares a las constantes que podéis definir en un programa, pero están definidas antes de empezar el programa y normalmente hacen referencia a aspectos de sistema (directorios por defecto por ejemplo) y marcan algunos aspectos importantes de su ejecución, ya que algunas de ellas son utilizadas por la Shell para definir su funcionamiento. Se suelen definir en mayúsculas, pero no es obligatorio. Estas variables pueden ser consultadas durante la ejecución de los programas mediante funciones de la librería de C. Para indicarle a la Shell que queremos consultar una variable de entorno debemos usar el carácter \$ delante del nombre de la variable, para que no lo confunda con una cadena de texto cualquiera. También podemos definir o modificar una variable de entorno utilizando el siguiente comando (para modificaciones no se usa el \$ antes del nombre): `export NOMBRE_VARIABLE=valor` (sin espacios). Para consultar el significado de las variables que define la Shell, puedes consultar el man de la Shell que estés utilizando, en este caso bash (man bash, apartado Shell Variables).

1. Ejecuta el comando “env” para ver la lista de variables definidas en el entorno actual y su valor.
2. Para ver el valor de una variable en concreto utiliza el comando echo:
 - i. `#echo $USERNAME`
 - ii. `#echo $PWD`
3. Algunas variables las actualiza la Shell dinámicamente, por ejemplo, cambia de directorio y vuelve a comprobar el valor de PWD. ¿Qué crees que significa esta variable?
4. Comprueba el valor de las variables PATH y HOME.



PREGUNTA 10: ¿Cuál es el significado de las variables de entorno PATH, HOME y PWD? ¿Qué valor tienen?

5. Añade el directorio “.” en la variable PATH. Fíjate en el carácter separador de directorios. Comprueba que, aparte del directorio “.”, la variable PATH aún contiene los directorios que tenía originalmente (no queremos perderlos).
 - i. `#export PATH=$PATH:.`



PREGUNTA 11: La variable PATH es una lista de directorios, ¿Qué carácter hacer de separador? ¿Cómo has añadido un directorio nuevo al PATH (sin perder lo que ya tenías)?



PREGUNTA 12: ¿El directorio en el que estas, está definido en la variable PATH? ¿Qué implica esto?

6. Define dos variables nuevas con el valor que quieras y comprueba su valor.

Mantenemos los cambios: fichero .bashrc

Los cambios que hemos hecho durante esta sesión (excepto los que hacen referencia al sistema de ficheros) se perderán al finalizar la sesión (definición de alias, cambios en el PATH, etc). Para que no se pierdan, hemos de insertar estos comandos en el fichero de configuración de sesión que utiliza la Shell. El nombre del fichero depende de la Shell que estemos utilizando. En el caso de Bash \$HOME/.bashrc. Cada vez que iniciamos una sesión, la Shell se configura ejecutando todos los comandos que encuentre en ese fichero.

7. Edita el fichero \$HOME/.bashrc y añade la modificación del PATH que te hemos pedido en el apartado anterior. Añade también la definición de un alias para que cada vez que ejecutemos el comando ls se haga con la opción -m. Para comprobar que has modificado bien el .bashrc ejecuta los siguientes comando:

```
#source $HOME/.bashrc
#ls
```

Y comprueba que la salida del ls se corresponde con la de la opción -m. El comando source provoca la ejecución de todos los comandos que hay el fichero que le pasamos como parámetro (es una manera de no tener que reiniciar la sesión para hacer que esos cambios sean efectivos).

- Nota: en el entorno de trabajo de las aulas de laboratorio, el sistema se arranca utilizando REMBO. Es decir, se carga una nueva imagen del sistema y por lo tanto todos vuestros ficheros se pierden y los ficheros de configuración del sistema se reinician mediante una copia remota. Eso significa que si reiniciáis la sesión empezaréis a trabajar con el fichero .bashrc original y no se conservarán vuestros cambios.

Algunos caracteres especiales útiles de la Shell

En la sección anterior ya hemos introducido el carácter &, que sirve para ejecutar un comando en segundo plano. Otros caracteres útiles de la Shell que introduciremos en esta sesión son:

- *: La Shell lo substituye por cualquier grupo de caracteres (excepto el "."). Por ejemplo, si ejecutamos (#grep prueba t*) veremos que la Shell substituye el patrón t*

por la lista de todos los ficheros que empiezan por la cadena “t”. Los caracteres especiales de la Shell se pueden utilizar con cualquier comando.

- >: La salida de datos de los comandos por defecto está asociada a la pantalla. Si queremos cambiar esta asociación, y que la salida se dirija a un fichero, podemos hacerlo con el carácter “>”. A esta acción se le llama “redireccionar la salida”. Por ejemplo, `ls > salida_ls`, guarda la salida de `ls` en el fichero `salida_ls`. Probad de ejecutar el comando anterior. A continuación, probadlo con otro comando pero con el mismo fichero de salida y comprobad el contenido del fichero `salida_ls`.
- >>: Redirecciona la salida de datos de un comando a un fichero pero en lugar de borrar el contenido del fichero se añade al final de lo que hubiera. Repite el ejemplo anterior pero con “>>” en lugar de “>” para comprobar la diferencia.



PREGUNTA 13: ¿Qué diferencia hay entre > y >>?

Hacer una copia de seguridad para la siguiente sesión

Dado que en los laboratorios se carga una nueva imagen cada vez que arrancamos el ordenador, es necesario hacer una copia de seguridad de los cambios realizados si queremos conservarlos para la siguiente sesión. Para guardar los cambios podéis utilizar el comando `tar`. Por ejemplo, si queréis generar un fichero que contenga todos los ficheros del directorio `S1`, además del fichero `.bashrc`, podéis ejecutar el siguiente comando desde vuestro directorio `HOME`:

```
#tar zcvf S1.tar.gz S1/* .bashrc
```

Finalmente debéis guardar este fichero en un lugar seguro. Para ello podéis utilizar el servidor Albanta que ofrece la FIB. En Albanta tenéis una cuenta de usuario por alumno con el `username/password` que usáis en la FIB. Para copiar el fichero en Albanta debéis hacer lo siguiente:

```
#sftp tu\_username@albanta.fib.upc.edu
>put S1.tar.gz
>quit
```

Sesión 2: Introducción

Preparación previa

Objetivos

En esta sesión es practicar con todo lo relacionado con la creación de ejecutables. En este curso utilizaremos lenguaje C. Practicaremos la corrección de errores tanto de makefiles como de ficheros C y la generación de ejecutables a partir de cero: ficheros fuente, makefile y librerías.

Habilidades

- Ser capaz de crear ejecutables dado un código en C:
 - Creación de ejecutables y utilización de makefiles sencillos.
- Ser capaz de crear programas en C desde cero:
 - Definición de tipos básicos, tablas, funciones, condicionales y bucles.
 - Utilización de punteros.
 - Formateado de los resultados de los programas.
 - Programas bien estructurados, claros, robustos y bien documentados.
 - Creación y modificación de makefiles sencillos: añadir reglas nuevas y añadir dependencias.
 - Aplicación de sangrías a código fuente en C.

Conocimientos previos

- Programación básica: tipos de datos básicos, tablas, sprintf.
- Programación media: punteros en C, acceso a los parámetros de la línea de comandos. Fases del desarrollo de un programa en C: Preprocesador/Compilador/Enlazador (o linkador)
- Uso de makefiles sencillos
- Uso del comando indent.

Guía para el trabajo previo

- Leer las páginas de man de los siguientes comandos. Estos comandos tienen múltiples opciones, leed con especial detalle las que os comentamos en la columna “Opciones” de la siguiente tabla.

Para leer en el man	Descripción básica	Opciones
make	Utilidad para automatizar el proceso de compilación/linkaje de un programa o grupo de programas	
gcc	Compilador de C de GNU	-c, -o, -l, -L, -D, -l

ar	Crea/modifica/extrae ficheros de -a -t archivos=(librerías)
sprintf	Conversión de formato almacenándola en un búffer
indent	Indentación de ficheros fuente

- Leer la documentación básica del lenguaje de programación C : definición de tipos de datos, variables, funciones, condicionales, bucles, etc.
- Leer la documentación del lenguaje de programación C: arrays, punteros.
- Crea la carpeta \$HOME/S2 y sitúate en ella para realizar los ejercicios.
- Bajate el fichero S2.tar.gz, descomprímelo con `tar xzfv S2.tar.gz` para obtener los ficheros de esta sesión.
- Analiza detenidamente y entiende los ficheros `listaParametros.c` y `punteros.c`.
- Comprueba el número máximo de caracteres que necesitamos para representar un `int` como string comprobando el valor de la constante `INT_MAX` definida en el fichero `/usr/include/limits.h`. Crea un fichero C, llamado `numeros.c`, que defina una constante, que llamaremos `LONG_INT_IN_CHARS`, que sea igual al número de dígitos de `INT_MAX`.
- Añade al fichero anterior una función que compruebe que un string que recibe como parámetro sólo contiene caracteres ASCII entre el '0' y el '9' (además del '\0' al final y potencialmente el '-' al principio para los negativos). La función ha de comprobar que el parámetro puntero no sea NULL. La función devuelve 1 si el string representa un número y tiene como mucho `LONG_INT_IN_CHARS` cifras, y 0 en cualquier otro caso. La función debe tener el siguiente prototipo:
 - `int esNumero(char *str);`
- Haz otra función que convierta un carácter a número (1 cifra). La función asume que el carácter se corresponde con el carácter de un número.
 - `unsigned int char2int(char c);`
- Utiliza `indent` para indentar el fichero `numeros.c` (`#indent numeros.c`).
- **PARA ENTREGAR: previo02.tar.gz**
#tar zcfv previo02.tar.gz numeros.c

Bibliografía


- Tutorial de programación en C (<http://www.elrincondelc.com/cursoc/cursoc.html>).

Bibliografía complementaria

- Programación en C:
 - Programming Language. Kernighan, Brian W.; Ritchie, Dennis M. Prentice Hall
 - Curso interactivo de C: http://labsopa.dis.ulpgc.es/cpp/intro_c.
- Makefiles:
 - <http://es.wikipedia.org/wiki/Make>
 - <http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>
- GNU Coding Standards

- <http://www.gnu.org/prep/standards/standards.html>, especialmente el punto: <http://www.gnu.org/prep/standards/standards.html#Writing-C>

Ejercicios a realizar en el laboratorio

- Para todos los ejercicios, se asume que se modificará el makefile cuando sea necesario y se probarán todos los ejercicios que se piden.
- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo: .
- **Para entregar: sesion02.tar.gz**

```
#tar zcfv sesion02.tar.gz entrega.txt makefile_1 listaParametros.c makefile_4  
mis_funciones.h mis_funciones.c suma.c punteros.c makefile_5 words.c  
suma_floats.c makefile
```

Solucionando problemas con el makefile

1. Modifica el makefile para que funcione
2. Modifica el makefile para que la regla listaParametros tenga bien definidas sus dependencias
3. Modifica el makefile para que el fichero de salida no sea a.out sino listaParametros



Crea una copia del makefile, llamándola makefile_1, para entregarla.

Solucionando problemas de compilación

4. Soluciona todos los errores de compilación que aparezcan
5. Añade el directorio "." a la variable PATH (en el caso que no lo hayas hecho ya en la sesión anterior) de forma que se encuentren los ejecutables que estén en el directorio actual.



PREGUNTA 2: ¿Qué error has solucionado primero, el primero de la lista o el último?

Comprobando el número de parámetros

A partir de este ejercicio, haremos que nuestros programas sean robustos en cuanto al número de parámetros. Modifica el programa anterior para que compruebe que por lo menos hay 1 argumento y en caso contrario llame a una función Usage(). Recordad que la variable argc del main considera el nombre del ejecutable como argumento, nosotros queremos comprobar que hay 1 parámetro además del nombre del ejecutable. La función Usage() es una función que debéis programar vosotros para que escriba un mensaje por pantalla con el formato de uso del programa y una línea de descripción de su funcionalidad.

a. void Usage(void);

```
#listaParametros a b
El argumento 1 es a
El argumento 2 es b
#listaParametros
Usage: listaParametros arg1 [arg2..argn]
Este programa escribe por su salida la lista de argumentos que recibe
```

Procesado de parámetros

6. Crea una copia de numeros.c (trabajo previo) llamada suma.c.
7. Modifica suma.c añadiendo una función mi_atoi que reciba como parámetro un string y devuelva un entero correspondiente al string convertido a número. Esta función asume que el string no es un puntero a NULL, pero puede representar un número negativo. Si el string no contiene un número correcto, o tiene una longitud superior a LONG_INT_IN_CHARS, el resultado es indefinido.
 - a. int mi_atoi(char *s);
8. Modifica suma.c para que se comporte de la siguiente manera: Comprueba que todos los parámetros sean números, si alguno no lo es da un mensaje de error indicando cuales no lo eran. Si todos son números, los convierte a int, los suma y escribe el resultado. Modifica también el makefile para que cree el fichero ejecutable suma. La siguiente figura muestra un ejemplo del funcionamiento del programa:

```
#suma 100 2 3 4 100
La suma es 209
#suma -1 1
La suma es 0
#suma 100 a
Error: el parámetro "a" no es un número
```



Crea una copia del makefile, llamándola makefile_4, para entregarla.

Usamos el preprocesador de C: Dividimos el código (#include)

Queremos separar las funciones auxiliares que vamos creando de los programas principales, de forma que podamos reutilizarlas cuando las necesitemos. En C, cuando queremos encapsular una función o un conjunto de funciones, lo normal es crear dos tipos de ficheros:

- Ficheros ***"include"***. Son ficheros de texto con extensión ***".h"*** que contienen **prototipos** (cabeceras) de funciones y **definiciones de tipos** de datos. Estos ficheros son **"incluidos"** por el preprocesador mediante la directiva **#include <fichero.h>** en el lugar exacto en que aparece la directiva, por lo que la posición es importante. Añadir un

fichero “.h” es equivalente a copiar y pegar el código del fichero en el lugar donde está la directiva #include. Lo correcto es poner en los ficheros include sólo lo que queramos que tenga una visibilidad global.

- Ficheros auxiliares, **ficheros objeto o librerías**. Estos ficheros contienen las **definiciones de variables** globales que necesiten las funciones auxiliares (si necesitan alguna) y la implementación de estas funciones. Podemos ofrecer el fichero “.c” directamente, el fichero objeto ya compilado (si no queremos ofrecer el código fuente) o, en caso de tener más de un fichero objeto, juntarlos todos en una librería (archive con el comando ar).
9. Separa las funciones auxiliares realizadas en los ejercicios anteriores que creas que puedan ser útiles para otros programas en un fichero aparte (mis_funciones.c) y crea un fichero de cabeceras (mis_funciones.h), donde definas las cabeceras de las funciones que ofreces, e inclúyelo en el programa suma.c. Añade una pequeña descripción de cada función junto al prototipo (ańádelo como comentario). Modifica el makefile añadiendo una nueva regla para crear el fichero objeto (obj) y modifica la regla del programa suma para que ahora se cree utilizando este fichero objeto. Añade las dependencias que creas necesarias.
 10. Modifica la función Usage de forma que como mínimo el programa suma reciba 2 parámetros.



PREGUNTA 3: Crea una copia del makefile, llamándola makefile_5, para entregarla. ¿Qué opción has tenido que añadir al gcc para generar el fichero objeto? ¿Qué opción has tenido que añadir al gcc para que el compilador encuentre el fichero mis_funciones.h?

Trabajando con punteros en C

1. Mira el código del programa punteros.c. Modifica el makefile para compilarlo y ejecútalo. Analiza el código para entender todo lo que hace.
2. Crea un programa llamado words.c que acepta un único parámetro. Este programa cuenta el número de palabras que hay en el string pasado como parámetro. Consideraremos que empieza una nueva palabra después de: un espacio, un punto, una coma y un salto de línea (\n). El resto de signos de puntuación no se tendrán en cuenta. Un ejemplo de funcionamiento sería:

```
#words hola
1 palabras
#words "Esta es una frase."
4 palabras
#words "Este parámetro lo trata" "este parámetro no lo trata"
4 palabras
```

3. Modifica el makefile para compilar y montar words.c.

Formateando los resultados para mostrarlos

printf es una función de la librería estándar de C para formatear un texto y almacenar el resultado en un búffer. El primer parámetro es el búffer, de tipo char*, y el segundo parámetro es una cadena de caracteres que especifica el texto a guardar así como el formato a usar. Posteriormente se tiene que utilizar la llamada al sistema write para escribir este buffer por la salida estándar (canal 1) o por la salida estándar de error (canal 2). Un ejemplo de utilización de printf podría ser (utilizad el man para consultar el comportamiento de las funciones que no conozcas):

```
char buffer[256];
printf(buffer, "Este es el ejemplo número %d\n", 1);
write(1, buffer, strlen(buffer));
```

Que muestra por la salida estándar: "Este es el ejemplo número 1".

1. Crea una copia de tu programa suma.c llamándola suma_floats.c
2. Modifica suma_floats.c para que los números que se les pasa como parámetros sean números en coma flotante en formato ascii en vez de enteros. Para transformar un número en coma flotante en formato ascii a un float se utiliza la función strtod (consulta el man).
3. Modifica el makefile para compilar y montar suma_floats.c
4. Modifica suma_floats.c para que utilice printf+write en vez de fprintf para mostrar los resultados.

Sesión 3: Procesos

Preparación previa

Objetivos

Los objetivos de esta sesión son practicar con las llamadas a sistema básicas para gestionar procesos, y los comandos y mecanismos básicos para monitorizar información de kernel asociados a los procesos activos del sistema.

Habilidades

- A nivel de usuario BÁSICO:
 - Ser capaz de hacer un programa concurrente utilizando las llamadas a sistema de gestión de procesos: fork, execlp, getpid, exit, waitpid.
 - Entender la herencia de procesos y la relación padre/hijo.
- A nivel de administrador BÁSICO:
 - Ser capaz de ver la lista de procesos de un usuario y algún detalle de su estado mediante comandos (ps, top).
 - Empezar a obtener información a través del pseudo-sistema de ficheros /proc.

Guía para el trabajo previo

- Lee las páginas de manual de las llamadas getpid/fork/exit/waitpid/execlp. Entiende los parámetros, valores de retorno y funcionalidad básica asociada a la teoría explicada en clase. Fíjate también en los *includes* necesarios, casos de error y valores de retorno. Consulta la descripción y las opciones indicadas del comando ps y del pseudo-sistema de ficheros /proc.

Para leer en el man	Descripción básica	Parámetros/argumentos principales que practicaremos
getpid	Retorna el PID del proceso que la ejecuta	
fork	Crea un proceso nuevo, hijo del que la ejecuta	
exit	Termina el proceso que ejecuta la llamada	
waitpid	Espera la finalización de un proceso hijo	
execlp	Ejecuta un programa en el contexto del mismo proceso	
perror	Escribe un mensaje del último error producido	
ps	Devuelve información de los procesos	-a, -u, -o
proc	Pseudo-file system que ofrece información de datos del kernel	cmdline, cwd, environ exe, status

- Crea el directorio del entorno de desarrollo para esta sesión (\$HOME/S3).
- Descarga el fichero S3.tar.gz y descomprímelo en el directorio que has creado para obtener los ficheros de esta sesión (tar zxvf S3.tar.gz).
- Crea un fichero de texto llamado previo.txt y escribe en él la respuesta a todas las preguntas.
- **Analiza el código** de los programas de ejemplo y el fichero Makefile.ejemplos
 - El fichero Makefile.ejemplos está preparado para compilar todos los programas excepto ejemplo_fork7.c
- **Compila todos los programas**, excepto ejemplo_fork7, usando el fichero **Makefile.ejemplos** (ver fichero README_S3).
- **Ejecuta ejemplo_fork1**
 - Escribe en el fichero previo.txt los mensajes que aparecen en pantalla y explica qué proceso muestra cada uno (padre o hijo) y por qué.
- **Ejecuta ejemplo_fork2**
 - Escribe en el fichero previo.txt los mensajes que aparecen en pantalla y explica qué proceso muestra cada uno (padre o hijo) y por qué.
- **Ejecuta ejemplo_fork3**
 - Escribe en el fichero previo.txt los mensajes que aparecen en pantalla y explica qué proceso muestra cada uno (padre o hijo) y por qué.
- **Ejecuta ejemplo_fork4**
 - ¿En qué orden aparecen en pantalla los mensajes? ¿Qué proceso acaba antes la ejecución?
 - **Modifica el código de este programa** para que el padre no escriba el último mensaje de su código hasta que su hijo haya acabado la ejecución.
- **Ejecuta ejemplo_fork5**
 - Escribe en el fichero previo.txt los mensajes que aparecen en pantalla y explica qué proceso muestra cada uno (padre o hijo) y por qué.
 - **Modifica el código de este programa**, para que el proceso hijo modifique el valor de las variables variable_local y variable_global antes de imprimir su valor. Comprueba que el padre sigue viendo el mismo valor que tenían las variables antes de hacer el fork.
- **Ejecuta ejemplo_fork6**, redireccionando su salida estándar a un fichero
 - Describe el contenido del fichero de salida
 - ¿Podemos asegurar que si ejecutamos varias veces este programa el contenido del fichero salida será exactamente el mismo? ¿Por qué?
- **Modifica el fichero Makefile.ejemplos** para añadir la compilación de ejemplo_fork7.c y utilízalo para compilarlo ahora.
 - ¿Porqué no compila el programa ejemplo_fork7.c? ¿Tiene algo que ver con el hecho de crear procesos? ¿Cómo se puede modificar el código para que escriba el valor de la “variable_local”?
- **Ejecuta ejemplo_exec1**

- Describe el comportamiento de este programa. ¿Qué ves en pantalla?
¿Cuántas veces aparece en pantalla el mensaje con el pid del proceso? ¿A qué se debe?
- **Ejecuta ejemplo_exec2**
 - Describe el comportamiento de este código. ¿Qué mensajes aparecen en pantalla? ¿Cuántos procesos se ejecutan?
- **Consulta en el man** a qué sección pertenecen las páginas del man que habéis consultado. Además, apunta aquí si se ha consultado alguna página adicional del manual a las que se han pedido explícitamente.
- **PARA ENTREGAR: previo03.tar.gz**
#tar zcfv previo03.tar.gz Makefile.ejemplos ejemplo_fork4.c ejemplo_fork5.c ejemplo_fork7.c previo.txt

Bibliografía

- Transparencias del Tema 2 (Procesos) de SO-grau.
- Capítulo 3 (Processes) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.

Ejercicios a realizar en el laboratorio

- Para todos los ejercicios, se asume que se probarán todos los ejercicios que se piden.
- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo:
- **Para entregar: sesion03.tar.gz**



#tar zcfv sesion03.tar.gz entrega.txt makefile myPS.c myPS2.c myPS3.c

Mutación de procesos: myPs.c

1. Crea un programa llamado myPS.c que cree tantos procesos hijos como parámetros se le pasen por la línea de comandos. Estos parámetros serán identificadores de usuario como, por ejemplo, “alumne” o “root”. Cada uno de los hijos creados ejecutará la función hazPS cuyo código es:

```
/* Ejecuta el comando ps -u username mediante la llamada al sistema execlp */
/* Devuelve: el código de error en el caso de que no se haya podido mutar */
int hazPS(char *username)
{
    execlp("ps", "ps", "-u", username, (char*)NULL);
    return errno;
}
```

2. El proceso padre tiene que bloquearse entre la creación de un hijo y el siguiente hasta que se pulse una tecla (usad la función getchar()).
3. El programa tiene que escribir un mensaje diferente en el caso del proceso padre y de los hijos, y en todos los casos debe contener el pid del proceso que lo escribe. Pej: “Soy el PADRE y mi PID es xxx” y “Soy un proceso hijo y mi PID es yyy”.
4. Crea un makefile, que incluya las reglas “all” y “clean”, para compilar y montar el programa myPS.c.



PREGUNTA 1: ¿Cómo puede saber un proceso el pid de sus hijos? ¿Qué llamada pueden utilizar los procesos para consultar su propio PID?

5. A partir de ahora se incluirá siempre, **para TODAS las llamadas a sistema**, la comprobación de errores. Utiliza la llamada `perror` (man 2 perror) para escribir un mensaje que describa el motivo que ha producido el error. Además, en caso que el error sea crítico, como por ejemplo que falle un fork o un execlp, tiene que terminar la ejecución del programa. La gestión del error de las llamadas a sistema puede hacerse de forma similar al siguiente código:

```
int main (int argc, char *argv[])
{
...
    if ((pid=fork())<0) error_y_exit("Error en fork",1);
...
}
void error_y_exit(char *msg,int exit_status)
{
    perror(msg);
    exit(exit_status);
}
```



PREGUNTA 2: ¿En qué casos se ejecutará cualquier código que aparezca justo después de la llamada `execlp` (En cualquier caso/ En caso que el `execlp` se ejecute de forma correcta/ En caso que el `execlp` falle)?

Consulta de la información de los procesos en ejecución: `myPS.c`

6. Comprueba la información que el S.O. exporta a través del pseudo-sistema de ficheros `proc`, que está ubicado en `/proc`. Puedes consultar la información utilizando el manual (`man proc`)
7. Para este ejercicio vamos a utilizar dos terminales de la shell. En una ejecuta `myPS` con un solo `username` como parámetro y no pulses ninguna tecla para que el proceso padre se quede bloqueado. En la segunda ventana ves al directorio `/proc` y comprueba que aparece un directorio cuyo nombre es el PID del proceso padre. Entra en ese directorio y mira la información extendida (permisos, propietario, etc) de los ficheros del directorio.



PREGUNTA 3: ¿Qué directorios hay dentro de `/proc/PID_PADRE`?



PREGUNTA 4: ¿Qué opción de `ls` has usado? ¿A qué ficheros “apuntan” los ficheros `cwd` y `exe`? ¿Cuál crees que es el significado de `cwd` y `exe`?



PREGUNTA 5: Apunta el contenido de los ficheros `environ`, `status` y `cmdline`. Compara el contenido del fichero `environ` con el resultado del comando `env`. ¿Qué relación ves? Busca en el contenido del fichero `status` el estado en el que se encuentra el proceso y apúntalo en el fichero de respuestas.

8. Entra ahora en el directorio `/proc` del proceso hijo, mira la información extendida de ese directorio



PREGUNTA 6: ¿A qué ficheros “apuntan” los ficheros `cwd` y `exe`?

PREGUNTA 7: ¿Puedes mostrar el contenido de los ficheros `environ`, `status` y `cmdline`? ¿En qué estado se encuentra el hijo?

Ejecución secuencial de los hijos: myPS2.c

1. Crea una copia de myPS.c, llamada myPS2.c, con la que trabajarás en este ejercicio. Modifica, también, el makefile para poder compilar y montar myPS2.c.
2. Lo correcto en un programa que crea procesos, es hacer que el proceso padre espere a que sus procesos hijos terminen y que compruebe su estado de finalización (llamada waitpid) para que se libere la memoria que el kernel reserva para cada proceso internamente (PCB). El lugar donde se produce la espera es determinante para generar un código secuencial (todos los procesos hijos se crean y ejecutan de 1 en 1) o concurrente (todos los procesos hijos se crean y se ejecutan de forma potencialmente paralela, dependiendo de la arquitectura en la que lo ejecutemos).

En este caso queremos hacer un **código secuencial**. Para ello utilizaremos la llamada al sistema waitpid entre una creación de proceso y la siguiente, de forma que aseguramos que no tendremos 2 procesos hijos ejecutándose a la vez.

3. Modifica el programa myPS2.c de forma que se fuerce a que los procesos se creen y ejecuten de forma secuencial (sólo puede haber 1 proceso hijo ejecutándose). Sustituye en el código del padre la llamada a getchar por la llamada a waitpid.
4. Modifica los datos que muestra el comando ps (modifica los parámetros de la llamada a execlp) para que no sea la información por defecto. Añade el ppid (el pid del proceso padre), el nombre del ejecutable y la utilización de cpu. ("man ps" para ver las opciones de ps que hay que usar)
5. Comprueba que la información que el ps muestra para el proceso padre es la misma que podemos ver a través de /proc (fichero stat).

Ejecución concurrente de los hijos: myPS3.c

1. Crea una copia de myPS.c, llamada myPS3.c, con la que trabajarás en este ejercicio. Modifica también el makefile para poder compilar y montar myPS3.
2. Modifica el programa myPS3.c para que primero se creen **todos** los hijos y luego el proceso padre se quede esperando una tecla (así mientras no pulsemos la tecla, todos los hijos que vayan acabando se quedarán en estado *zombie*). Al pulsar la tecla, el padre realizará las llamadas a waitpid necesarias para esperar a sus hijos en un segundo bucle.
3. Ejecuta myPS3 con varios usernames y deja al padre bloqueado antes del bucle de waitpids. En otra ventana comprueba que en /proc aparece un directorio para cada uno de los hijos.



PREGUNTA 8: Comprueba el fichero status de /proc/PID_PADRE /proc/PID_HIJO(s). ¿En qué estado está el proceso padre? ¿En qué estado están los hijos?



4. Para comprobar el efecto de la ejecución concurrente, y ver que la planificación del sistema genera resultados diferentes, ejecuta varias veces el comando myPS3 con los mismos parámetros y guardar la salida en diferentes ficheros. Comprobar si el orden en que se ejecutan los ps's es siempre el mismo.

PREGUNTA 9: ¿Qué has hecho para guardar la salida de las ejecuciones de myPS3?

Sesión 4: Procesos

Preparación previa

1. Objetivos

Durante esta sesión introduciremos la gestión de eventos entre procesos como mecanismo de comunicación y sincronización entre procesos. También se trabajarán aspectos relacionados con la concurrencia de procesos.

2. Habilidades

- Ser capaz de reprogramar/esperar/enviar eventos utilizando la interfaz de Unix entre procesos. Practicaremos con: signal/pause/alarm/kill.
- Ser capaz de enviar eventos a procesos utilizando el comando kill.

3. Conocimientos previos

Los signals o eventos pueden ser enviados por otros procesos o enviados por el sistema automáticamente, por ejemplo cuando acaba un proceso hijo (SIGCHLD) o acaba el temporizador de una alarma (SIGALRM).

Cada proceso tiene una tabla en su PCB donde se describe, para cada signal, que acción hay que realizar, que puede ser: **Ignorar el evento** (no todos pueden ignorarse), **realizar la acción por defecto** que tenga el sistema programada para ese evento, o **ejecutar una función que el proceso haya definido** explícitamente mediante la llamada a sistema signal. Esta función tiene una cabecera predefinida que es la siguiente:

```
void nombre_funcion( int numero_de_signal_recibido );
```

El hecho de recibir el signal recibido como parámetro nos permite utilizar la misma función para más de un signal y poder distinguir que signal concreto hemos recibido.

4. Guía para el trabajo previo

Con unos ejemplos veremos, de forma sencilla, como programar un evento, como enviarlo, que sucede con la tabla de programación de signals al hacer un fork, etc. Para esta sesión repasa los conceptos explicados en clase de teoría sobre procesos y signals.

- Lee las páginas de man de las llamadas a sistema signal/kill/alarm/pause
- Lee las páginas de man del comando kill. Prueba la opción -l para ver la lista de signals que hay en el sistema.

Para leer en el man	Descripción básica
---------------------	--------------------

Opciones

signal	Reprograma la acción asociada a un evento concreto
kill (llamada a sistema)	Envía un evento concreto a un proceso

pause	Espera la llegada de un signal	
alarm	Programa el envío de un signal SIGALRM al cabo de N segundos	
sleep	Función de la librería de C que bloquea al proceso durante el tiempo que se le pasa como parámetro	
/bin/kill (comando)	Envía un evento a un proceso	-L
ps	Muestra información sobre los procesos del sistema	-o pid,s,cmd,time
waitpid	Espera la finalización de un proceso	WNOHANG

Bájate el fichero S4.tar.gz y descomprímelo (tar zxvf S4.tar.gz). Compila los ficheros y ejecútalos. En el fichero README que encontraras hay una pequeña descripción de lo que hacen y como compilarlos. Intenta entenderlos y comprender como se usan las llamadas a sistema que practicaremos antes de ir al laboratorio. Los ficheros están comentados de forma que entiendas lo que se está haciendo.

Algunos signals son especialmente útiles, como por ejemplo SIGALRM (alarma, temporizador), SIGCHLD (fin de un proceso hijo), o SIGUSR1/SIGUSR2 (para usarse por el programador). Mírate las acciones por defecto de estos signals.

Realiza las siguientes pruebas antes de ir al laboratorio. Crea un fichero llamado entrega.txt y escribe en él las respuestas a las siguientes preguntas (indicando su número).

1. **Sobre alarm1:** ¿Qué pasa cuando llega otro signal?

1.1. Ejecuta alarm1 en una consola. Haz ps en otra para obtener el pid y envíale un evento “-KILL” utilizando el comando kill. ¿El comportamiento es el mismo que si esperas que llegue el SIGALRM? ¿Recibes un mensaje diferente en el terminal? ¿Existe alguna relación (PADRE/HIJO) entre el proceso que está ejecutando el termina y el proceso alarm1? ¿Con que llamada a sistema el terminal recoge el estado de finalización del proceso que ejecuta alarm1 cuando este termina?

1.2. ¿Es necesario el exit que hay en el código? Si la acción por defecto es acabar el proceso, ¿se ejecuta?

2. **Sobre alarm2:** ¿Podemos enviar eventos desde la consola?

2.1. Ejecuta alarm2 en una consola. Abre otra, consigue su pid y envíale el signal “-ALRM” desde la consola varias veces. ¿Qué ha pasado, el control de tiempo ha funcionado como pretendíamos? ¿Podemos asegurar desde un programa que los signals que recibimos son sólo los que esperábamos? ¿Se puede modificar el comportamiento de cualquier signal? Mira en el man (man alarm) el valor de retorno de la llamada a sistema alarm y piensa como podríamos arreglar el código para detectar cuando un SIGALRM nos llega sin estar relacionado con el temporizador.

3. **Sobre alarm3:** La tabla de programación de signals se hereda

3.1. ¿Quien recibe las alarmas: El padre, el hijo, los dos? ¿Cómo lo has comprobado? Modifica el mensaje que se escribe en la función “funcio_alarm” para que se escribe el PID del proceso que escribe el mensaje, de forma que podamos ver fácilmente que proceso recibe los signals.


4. **Sobre alarm4:** Comprueba que las alarmas son recibidas únicamente por el proceso que las ha generado
 - 4.1. ¿Cuántas alarmas programa cada proceso? ¿Quién recibe cada alarma: El padre, el hijo, los dos? ¿Qué le pasa al padre? ¿Cómo lo has comprobado? Modifica el código de forma que la primera alarma la programe el padre antes de fork (y el hijo no), y observa cómo el hijo se queda esperando en la llamada pause.
5. **Sobre alarm5:** Desactivación del temporizador y envío de signals mediante llamada a sistema. Para desactivar el temporizador se puede utilizar la llamada a sistema alarm pasándole como parámetro un 0.
 - 5.1. Completa el programa ejemplo_alarm5 para limitar el tiempo máximo de espera del padre en el waitpid: si al cabo de 2 segundos el hijo no ha acabado la ejecución el padre deberá enviarle un SIGKILL. Además, al salir del waitpid el padre debe mostrar en salida estándar un mensaje describiendo la causa de la muerte del hijo (ha acabado a tiempo o el padre le ha enviado un SIGKILL) y el tiempo que faltaba para que saltara el temporizador. Asegúrate de que el padre no recibe ningún SIGALRM si el hijo acaba a tiempo.
6. **Sobre ejemplo_waitpid:** Analiza el código de este programa y ejecútalo.

PARA ENTREGAR: previo04.tar.gz
#tar czfv previo04.tar.gz entrega.txt

5. Bibliografía

- Transparencias del Tema 2 (Procesos) de SO-grau.
- Capítulo 21 (Linux) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.

Ejercicios a realizar en el laboratorio

- Para todos los ejercicios, se asume que se probarán todos los ejercicios que se piden y se modificará el makefile para que se puedan compilar y montar los ejecutables.
- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el  símbolo:
- **Para entregar: sesion04.tar.gz**

```
#tar zcfv sesion04.tar.gz entrega.txt makefile ejemplo_alarm2.c ejemplo_alarm3.c  
ejemplo_signal.c ejemplo_signal2.c eventos.c second.c
```

1. SOBRE EL TRABAJO PREVIO

- Sobre alarm2:
 - Reprograma el signal SIGUSR1 y haz que esté asociado a la misma función que la alarma. Modifica la función “funcion_alarma” de forma que actualice los segundos en caso que llegue un signal SIGALRM y que escriba un mensaje en caso que llegue SIGUSR1. Comprueba que funciona enviando signals SIGUSR1 desde la consola utilizando el comando kill.
 - **NOTA:** Recuerda que la función de atención al signal recibe como parámetro el número de signal recibido
- Sobre alarm3:
 - Modifica el código para que la programación del evento SIGALRM (llamada a sistema signal) solo la haga el hijo. ¿Qué le pasa al padre ahora cuando le llega el evento SIGALRM?
- Si se hereda la tabla de acciones asociadas a signals al hacer un fork, ¿qué pasa si hacemos un execlp (y cambiamos el código)? ¿Se mantiene la reprogramación de signals? ¿Se pone por defecto? Modifica el código para incluir una mutación a otro código y haz la prueba. Ten en cuenta que el ejecutable que pongas ha de tener una cierta duración o no dará tiempo a que llegue el signal.
- Sobre ejemplo_waitpid:
 - El programa ejemplo_waitpid, ¿Es secuencial o concurrente? ¿Qué significa el tercer parámetro del waitpid? ¿Qué valores puede tener? (man waitpid)
 - Crea una copia de este programa llamada ejemplo_signal.c. Modifica este programa para que el proceso padre, en vez de hacer un bucle para esperar a todos los procesos hijos, utilice el signal SIGCHLD (notificación de que un hijo ha acabado) para ejecutar el waitpid. ¿Cómo sale ahora la salida del programa ejemplo_signal.c respecto a la de ejemplo_waitpid (mismo orden/diferente orden)?
 - **PISTA:** Recuerda que cuando se está tratando un signal y recibimos otro signal del mismo tipo, se guarda para cuando salgamos de tratar el signal actual, pero si recibimos más, entonces se pierden. Por este motivo

podemos recibir menos eventos tipo SIGCHLD que procesos hijos tengamos. Mira el significado de la opción WNOHANG de la llamada a sistema waitpid.

- Recuerda también que no debemos bloquearnos en el código de una función de atención a un signal.

- Crea una copia de ejemplo_signal.c con el nombre ejemplo_signal2.c. Modifica este programa para que los procesos hijos muestren su PID y se bloqueen esperando la recepción de un signal. El padre, una vez creados todos los procesos hijos, les enviará a todos ellos el signal SIGUSR1 (acción por defecto terminar). Además, cuando acabe uno de los procesos hijos, el padre mostrará el PID más el motivo por el cual este proceso ha acabado (macros WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG). ¿Qué valor de finalización presenta cada proceso hijo?

- PISTA: Ten en cuenta que la llamada pause() bloquea al proceso hasta que llegue un signal que el proceso tenga capturado, pero si el signal llega antes de que el proceso ejecute el pause(), el proceso podría bloquearse indefinidamente.
- PISTA: Si gestionamos el fin de los procesos hijos reprogramando el SIGCHLD, podría pasar que se pierdan eventos. Consulta el ejemplo cada_segundo_sigchld.c
- Create una función que, utilizando las macros anteriores, escriba en PID del proceso que ha terminado y un mensaje indicando si ha terminado por un exit o un signal y en cada caso en exit_status o signal_code. Teneis un ejemplo de cómo usar estas macros en las transparencias del T2.



2. PROTECCIÓN EN PROCESOS

Los procesos no pueden enviar eventos a procesos de otros usuarios. En el sistema hay varios usuarios creados (so1, so2, so3, so4 y so5) el password de todos es “sistemas”.

- Ejecuta el programa alarm2 en un terminal y comprueba su PID. Abre una nueva sesión y cambia al usuario so1 (ejecuta #su so1). Intenta enviar eventos al proceso que está ejecutando alarm2 desde la sesión iniciada con el usuario so1. ¿Puedes enviárselos?, ¿qué error da?



3. GESTIONAMOS EVENTOS

Haz un programa, llamado eventos.c, que tenga un contador global del tiempo que lleva ejecutándose (en segundos). Además, si el proceso recibe un SIGUSR1 pondrá a cero el contador de segundos y si recibe un SIGUSR2 escribirá por pantalla el valor del contador. Haz que todos los signals sean atendidos por la misma función. Envía los signals desde la línea de comandos y comprueba que funciona correctamente.

4. COMPORTAMIENTO POR DEFECTO

La reprogramación de un signal se mantiene durante toda la vida del proceso. Por esta razón, a veces, es necesario forzar el comportamiento por defecto de los signals en el caso de que no nos interese procesar más eventos.



- Crea un programa llamado `second.c`. Este programa reprograma los signals `SIGALRM`, `SIGUSR1` y `SIGUSR2` para que ejecuten una misma función que muestre un mensaje por pantalla con el número de signal recibido y re programe el signal con el comportamiento por defecto. De esta forma, la segunda vez que se reciba el mismo signal se ejecute el comportamiento por defecto de ese signal. ¿Qué mensaje muestra el Shell cuando se envía por segunda vez el mismo signal?
-

Sesión 5: Procesos

Preparación previa

1. Objetivos

Los objetivos de esta sesión son: entender las implicaciones de la gestión de procesos en el rendimiento de un sistema multiusuario y ser capaz de medir e influir en el tiempo de ejecución de un programa según la carga de la máquina y el cambio en las prioridades del sistema.

2. Habilidades

- Entender cómo influyen las distintas políticas de planificación y prioridades en los programas de usuario.
- Entender cómo influye la carga del sistema en el tiempo de ejecución de los programas.
- Ser capaz de ver la lista de los procesos de todos los usuarios para detectar posibles problemas en el sistema.

3. Guía para el trabajo previo

- Antes de la sesión, consultad el `man` (`man nombre_comando`) de los siguientes comandos. En concreto, para cada comando debéis leer y entender perfectamente la SYNOPSIS y la DESCRIPTION.

Para leer en el <code>man</code>	Descripción básica	Opciones
<code>nice</code>	Ejecuta un programa modificándole la prioridad de planificación	
<code>top</code>	Muestra información de los procesos en el sistema y información global del estado del sistema	
<code>uptime</code>	Muestra cuanto tiempo lleva ON el sistema y la carga media	
<code>w</code>	Muestra quien está conectado y que está haciendo	
<code>proc</code>	Pseudo-file system que ofrece información de datos del kernel	<code>cpuinfo</code>
<code>time</code>	Ejecuta un programa y mide el tiempo que tarda en ejecutarse	

1. En Linux la política de planificación de procesos está basada en Round Robin con prioridades: es decir, la cola de Ready está ordenada en función de la prioridad que se ha asignado a los procesos. Los usuarios pueden **bajar** la prioridad de sus procesos ejecutándolos mediante el comando **nice** (sólo root puede subir la prioridad de cualquier proceso).
2. Los comandos **top**, **uptime** y **w** nos muestran información del sistema. Útiles de cara a hacer un control sencillo de la carga del sistema.

- El comando `top` nos muestra todos los procesos del sistema y detalles como el %CPU que consumen, la prioridad que tiene, el usuario, etc. Además, nos muestra información global del sistema como la carga media. Este valor no es puntual sino que es la media de los últimos 1, 5 y 15 minutos
- El comando `w` nos muestra información muy resumida pero informa de quién está conectado.
- El comando `uptime` ofrece información estadística del sistema.

```

so3@leman.fib.upc.es: ~
File Edit View Terminal Tabs Help
alumne@leman.fib.upc.es: ... so3@leman.fib.upc.es: ~ root@leman.fib.upc.es: /ho...
top - 06:59:05 up 2:12, 4 users, load average: 1.99, 1.38, 0.76
Tasks: 87 total, 3 running, 84 sleeping, 0 stopped, 0 zombie
Cpu(s): 67.1% us, 1.0% sy, 31.9% ni, 0.0% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 514952k total, 366680k used, 148272k free, 16172k buffers
Swap: 0k total, 0k used, 0k free, 199796k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 5746 sol        25   0  1412   264   208  R   66.9   0.1   3:05.75  fib
 5749 sol        35  10  1412   264   208  R   32.0   0.1   1:29.14  fib
 4205 root        15   0 41688  18m 6820  S    0.7   3.6  19:10.03 Xorg
 4917 alumne     15   0 60788  16m 12m   S    0.3   3.2   0:04.62 nautilus
    1 root        16   0  1568   532   460  S    0.0   0.1   0:00.91 init
    2 root       RT    0     0     0     0  S    0.0   0.0   0:00.00 migration/0
    3 root       34  19     0     0     0  S    0.0   0.0   0:00.00 ksoftirqd/0
    4 root       RT    0     0     0     0  S    0.0   0.0   0:00.00 watchdog/0
    5 root       10  -5     0     0     0  S    0.0   0.0   0:00.22 events/0
    6 root       10  -5     0     0     0  S    0.0   0.0   0:00.02 khelper
    7 root       12  -5     0     0     0  S    0.0   0.0   0:00.00 kthread
    9 root       10  -5     0     0     0  S    0.0   0.0   0:01.10 kblockd/0
   10 root       20  -5     0     0     0  S    0.0   0.0   0:00.00 kacpid
  104 root       15   0     0     0     0  S    0.0   0.0   0:00.15 pdflush
  105 root       15   0     0     0     0  S    0.0   0.0   0:00.61 pdflush
  107 root       19  -5     0     0     0  S    0.0   0.0   0:00.00 aio/0
  106 root       15   0     0     0     0  S    0.0   0.0   0:00.25 kswapd0

```

Figura 4 Ejemplo de salida del comando `top`

La Figura 4 nos muestra un ejemplo de la salida del comando `top`. En la parte superior hay el resumen del sistema y luego la lista de tareas. En este caso hemos remarcado la carga del sistema y los dos procesos que estábamos ejecutando. La columna **PR** nos muestra la prioridad de los procesos. En Linux un valor alto de la prioridad indica menos prioridad. Podemos ver como el proceso con **PR=25** recibe más %CPU (66.9) que el proceso con **PR=35** (32.0).

3. A la hora de interpretar la información que nos da `top` y `uptime` es necesario tener en cuenta el número de unidades de computación (CPU's, cores, etc.) que tenemos en la máquina, ya que eso determina el número de programas que podemos tener en ejecución al mismo tiempo. Por ejemplo, si tenemos 2 cores podría pasar que 2 procesos al mismo tiempo estuvieran consumiendo un 80% de cpu. El fichero `/proc/cpuinfo` contiene información sobre las cpu's que tenemos en nuestra máquina.
4. En esta sesión os proporcionamos varios **scripts** para facilitar la ejecución de los programas. Un script no es más que un fichero texto que contiene un conjunto de comandos que pueden ser interpretados por la Shell. La primera línea de un script

debe indicar la Shell que se quiere utilizar para interpretar el contenido del fichero. Esto es importante porque además de comandos en los scripts se pueden utilizar estructuras de control (condicionales, bucles,...) y cada Shell define su propia sintaxis para estas estructuras. Los scripts se pueden ejecutar como cualquier otro programa, pero como son ficheros de texto y por defecto estos ficheros se crean sin permiso de ejecución, es necesario comprobar si tienen ese permiso activado y si no lo tienen debemos activarlo mediante el comando `chmod` (ver sesión 2).

5. El comando **`/usr/bin/time`** sirve para ejecutar un programa midiendo su tiempo de ejecución (consulta su página del manual: `man time`). Nota: el intérprete de comandos Bash tiene un comando interno que también se llama `time` y es el que se ejecutará por defecto si no ponemos el path completo de nuestro comando.
6. **Familiarízate con el entorno que usarás en la sesión:** bájate el fichero `S5.tar.gz` y descomprímelo (`tar zxvf S5.tar.gz`).

- a. En este paquete tienes el fichero `fibonacci.c` que contiene un programa de cálculo y que lo utilizaremos para ver el impacto que tiene en su ejecución la carga del sistema y las prioridades. Utiliza el `makefile` para compilar el programa y ejecútalo con diferentes parámetros para ver cómo funciona:

- i. **Utiliza el comando `/usr/bin/time`** para medir el tiempo que tarda Fibonacci en ejecutar Fibonacci si se le pasan los parámetros 10, 20, 30, 40 y 50 y apunta ese tiempo en el fichero `previo.txt`.

- b. En el paquete tenéis también dos scripts para automatizar la ejecución de varias instancias de Fibonacci. El script `FIB` recibe como parámetro cuántos programas Fibonacci queremos lanzar y los ejecuta de forma concurrente (en background), midiendo el tiempo que tarda cada uno en acabar. El script `BAJA_PRIO_FIB` también ejecuta concurrentemente el número de programas Fibonacci que le pasamos como parámetro y mide su tiempo de ejecución, pero en este caso utiliza el comando `nice` para ejecutarlos con menos prioridad. Por ejemplo, si ejecutamos el comando:

```
# ./FIB 2
```

Es equivalente a ejecutar los siguientes comandos de forma consecutiva:

```
# /usr/bin/time ./fib 45&
# /usr/bin/time ./fib 45&
```

- i. Anota en el fichero `previo.txt` con qué valor de `nice` se ejecuta fibonacci desde el script `BAJA_PRIO_FIB`. Para averiguarlo, consulta en la página del manual el comportamiento de `nice`.
 - c. También os damos el fichero `Sesion5.ods` que es una hoja de cálculo de OpenOffice que iréis rellenando durante la sesión.
- **PARA ENTREGAR:** `previo05.tar.gz`
`#tar zcfv previo05.tar.gz previo.txt`

4. Bibliografía

- Transparencias del Tema 2 (Procesos) de SO-grau
- Capítulo 5 (Scheduling) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.

Ejercicios a realizar en el laboratorio

- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicaréis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo:



- Para entregar: `sesion05.tar.gz`

`#tar zcfv sesion05.tar.gz entrega.txt Sesion5.ods`

¿Cuántas CPU's o cores tengo?

Accede al fichero `/proc/cpuinfo` para averiguar cuantas CPU's o cores tienes en la máquina en la que trabajas. Esta información te ayudará a interpretar los resultados sobre el consumo de CPU de los procesos.



- **Pregunta 1:** Apunta en el fichero entrega.txt el número de unidades de cálculo que tienes en la máquina.

Ejecución de 1 aplicación "sola". Medimos el tiempo.

- Para este ejercicio utilizaremos 2 terminales. En uno de ellos ejecuta el comando `top` y en el otro ejecuta 5 veces en secuencial el comando (cuando acaba uno lanza el siguiente):
 - `# ./FIB 1`
- Comprueba en la salida del comando `top` que cada ejecución recibe aproximadamente toda la CPU. Calcula el tiempo medio, el máximo y el mínimo (tiempo real) de las 5 ejecuciones.



Pregunta 2: Apunta el tiempo medio, máximo y mínimo para una instancia en la Tabla 1 de la hoja de cálculo adjunta.

5. Impacto de entorno multiproceso. Carga del sistema, tiempo de ejecución y prioridades.

- Repite el mismo experimento primero para 2 instancias concurrentes (ejecutando el comando `./FIB 2`) y luego para 5 instancias (ejecutando el comando `./FIB 5`).
- Observa con el comando `top` como se reparte la CPU en cada uno de los casos. Observa cómo se intenta hacer un reparto "justo" y como el tiempo de ejecución de todos los procesos es incrementado en la misma proporción. Para interpretar estos resultados ten en cuenta el número de CPUs que tienes en la máquina.

- Comprueba que las prioridades de los procesos son las mismas.



Pregunta 3: Anota para cada experimento los tiempos medio, máximo y mínimo en la Tabla 1 de la hoja de cálculo adjunta.

6. Impacto de entorno multiproceso y multiusuario

- Los experimentos anteriores se han hecho con el mismo usuario. En este ejercicio veremos cómo afecta lo que hace un usuario al resto.
- Para este experimento utilizaremos 3 terminales. Mantén en ejecución el comando top en un terminal. Abre un nuevo terminal y utiliza el comando su para cambiar a un nuevo usuario (so1 por ejemplo):
 - #su so1
- Ahora ejecuta como el usuario “so1” el comando ./FIB 5 para generar una carga alta. Y al mismo tiempo, en otro terminal y como el usuario inicial (“alumne”) , ejecuta 5 veces de forma secuencial el comando ./FIB 1.



Pregunta 4: ¿Cómo se ve afectado el tiempo de ejecución del proceso?



Pregunta 5: ¿Qué %CPU ha asignado el sistema a cada proceso?



Pregunta 6: La asignación, ¿ha sido por proceso o por usuario? ¿Crees que es posible que 1 usuario sature el sistema con este criterio?

7. Impacto de la prioridad en el tiempo de ejecución

El comando nice permite bajar la prioridad a los procesos de un usuario de forma que se asigne más tiempo de CPU a los más prioritarios. Vamos a comprobar como influye la prioridad en el reparto de la CPU.

- Mantén la ejecución del comando top en un terminal y ejecuta de forma simultánea, con el mismo usuario, los comandos ./FIB 1 y BAJA_PRIO_FIB 2. Mide los tiempos de ejecución de cada fibonacci. Repite el experimento 5 veces y anota en la Tabla 2 (de la hoja de cálculo) el tiempo de ejecución medio para FIB y BAJA_PRIO_FIB en la fila “2 instancias”.
- Comprueba, mediante el comando top, como se han asignado diferentes prioridades a cada uno y como se observa un reparto de %CPU similar al de la Figura 4
- Repite el experimento anterior, lanzando simultáneamente ./FIB 1 y BAJA_PRIO_FIB 5. Anota los tiempos de ejecución en la fila “5 instancias”, para estos experimentos.
-



Pregunta 7: ¿Cómo se ve afectado el tiempo de ejecución de FIB respecto al número de instancias de BAJA_PRIO_FIB?



Pregunta 8: ¿Qué %CPU ha asignado el sistema a cada proceso con 2 y 5 instancias de BAJA_PRIO_FIB?

- Sin embargo, las prioridades sólo afectan a la compartición de la CPU si los procesos que compiten por ella tienen diferente prioridad. Para comprobarlo, ejecuta en un terminal el comando `top` y en otro lanza el comando `BAJA_PRIO_FIB 5`. Anota en el fichero `entrega.txt` la prioridad de los procesos y el tiempo medio de ejecución y comprueba que ese tiempo es similar entre ellos. Ahora ejecuta el comando `FIB 5` varias veces y anota en el fichero `entrega.txt` la prioridad de los procesos y el tiempo de ejecución. Comprueba que aunque se ejecutan con mayor prioridad que en el caso anterior, el tiempo de ejecución de cada uno también es similar al que tenían cuando los has ejecutado con baja prioridad.
- **Pregunta 9:** Anota en el fichero `respuestas.txt` el tiempo medio de ejecución y la prioridad de cada proceso Fibonacci cuando se han lanzado con baja prioridad (`BAJA_PRIO_FIB`) y cuando se han lanzado con prioridad normal (`FIB`).



Sesión 6: Gestión de Memoria

Preparación previa

Objetivos

- Comprender la relación entre el código generado por el usuario, el espacio lógico del proceso que ejecutará ese programa y el espacio de memoria física ocupado por el proceso.
- Entender la diferencia entre enlazar un ejecutable con librerías estáticas o dinámicas.
- Entender el funcionamiento de algunos comandos básicos que permiten analizar el uso de la memoria que hacen los procesos.
- Entender el comportamiento de funciones de la librería de C y de llamadas a sistema simples que permiten modificar el espacio de direcciones lógico de los procesos en tiempo de ejecución (memoria dinámica).

Habilidades

- Ser capaz de relacionar partes de un binario con su espacio de direcciones lógico en memoria.
- Saber distinguir las diferentes regiones de memoria y en que región se ubica cada elemento de un proceso.
- Entender el efecto de malloc/free/sbrk sobre espacio de direcciones, en particular sobre el heap.

Conocimientos previos

- Entender el formato básico de un ejecutable.
- Programación en C: uso de punteros.
- Saber interpretar y consultar la información disponible sobre el sistema y los procesos en el directorio /proc.

Guía para el trabajo previo

- Antes de la sesión, consultad el man (`man nombre_comando`) de los siguientes comandos. En concreto, para cada comando debéis leer y entender perfectamente: la SYNOPSIS, la DESCRIPTION y las opciones que os comentamos en la columna “Opciones” de la tabla.

Para leer en el man	Descripción básica	Opciones a consultar
gcc	Compilador de C	-static
nm	Comando que muestra la tabla de símbolos del programa	
objdump	Comando que muestra información sobre el fichero objeto	-d

/proc	Contiene información sobre el sistema y los procesos en ejecución	/proc/[pid]/maps
malloc	Función de la librería de C que valida una región de memoria lógica	
free	Función de la librería de C que libera una región de memoria lógica	
sbrk	Llamada a sistema que modifica el tamaño de la sección de datos	

- En la página web de la asignatura (<http://docencia.ac.upc.edu/FIB/grau/SO>) tenéis el fichero S6.tar.gz que contiene todos los ficheros fuente que utilizaréis en esta sesión. Créate un directorio en tu máquina, copia en él el fichero S6.tar.gz y desempaquéalo (tar xzfv S6.tar.gz).
- Crea un fichero de texto llamado previo.txt y contesta en él a las siguientes preguntas.
- Practica el uso de nm y objdump con los siguientes ejercicios aplicados al siguiente código (que puedes encontrar en el fichero mem1_previo.c):

```
#include<stdio.h>
#include<stdlib.h>
void suma (int op1, int op2, int *res){
    *res = op1 + op2;
}

int j;
main(int argc, char *argv[]){
    int i;
    int s;
    i=atoi(argv[1]);
    j=atoi(argv[2]);
    suma(i,j,&s);
    fprintf(stdout,"suma de %d y %d es %d\n",i,j,s);
}
```

- Utiliza el comando nm sobre el ejecutable y apunta en el fichero “previo.txt” la dirección asignada a cada una de las variables del programa. **¿Es posible saber la dirección de las variables “i” y “j”? ¿Por qué? ¿En qué zona de memoria están reservadas estas variables? Busca también la dirección asignada a la función “suma”.**
- **Modifica el programa anterior (llámalo mem1_previo_v2.c)** para que la variable “s” esté definida en el programa principal como puntero a entero (int *s) y asígnale una dirección válida antes de llamar a la función “suma”. Para ello utiliza la función malloc para reservar suficiente espacio para almacenar un entero y asígnale la dirección de esa región a la variable. Adapta el resto del código a este cambio para que siga funcionando.

- Utiliza el comando gcc para compilar el fichero mem1_previo.c enlazándolo con librerías estáticas. **Indica en el fichero “previo.txt” que comando has utilizado.**
- **PARA ENTREGAR: previo06.tar.gz**
 - **#tar zcfv previo06.tar.gz previo.txt mem1_previo_v2.c**

Bibliografía

- Transparencias del Tema 3 (Memoria) de SO-grau
- Capítulo 8 (Main Memory) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.

Ejercicios a realizar en el laboratorio

- Para cada pregunta que se crea un nuevo fichero de código se tiene que modificar el Makefile para que lo compile y monte el ejecutable.
- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo:
- **PARA ENTREGAR: sesion06.tar.gz**
 - **#tar zcfv sesion06.tar.gz entrega.txt Makefile mem1_v2.c mem1_v3.c**



Compilación estática y dinámica

El fichero mem1.c contiene un programa que reserva memoria durante la ejecución. Analiza su contenido antes de responder a las siguientes preguntas.

1. Compila el programa enlazando con las librerías dinámicas del sistema (compilación por defecto) y guarda el ejecutable con el nombre mem1_dynamic. Ejecuta el comando nm sobre el ejecutable.



PREGUNTA 1 ¿Qué variables aparecen en la salida del nm? ¿Qué dirección ocupa cada una? Indica a que región pertenece cada variable según la salida del nm.

2. Compila ahora el programa, enlazando con las librerías estáticas del sistema, y guarda el ejecutable con el nombre mem1_static. Compara ahora los ejecutables mem1_dynamic y mem1_static de la siguiente manera:
 - Utilizando el comando nm para ver los símbolos definidos dentro del ejecutable.
 - Utilizando el comando objdump con la opción -d para ver el código traducido.
 - Viendo el tamaño del ejecutable resultante.



PREGUNTA 2: Para los dos ejecutables, observa su tamaño, la salida de los comandos nm y objdump ¿En qué se diferencian los dos ejecutables?

1. Ejecuta en background las dos versiones del ejecutable y compara el contenido del fichero maps del /proc para cada uno de los procesos.



PREGUNTA 3: Observa el contenido del fichero maps del /proc para cada proceso y apunta para cada región la dirección inicial, dirección final y etiqueta asociada. ¿Qué diferencia hay entre el maps de cada proceso?

1. Ejecuta en background el programa mem1 sin pasarle ningún parámetro. Accede al directorio del /proc que contiene la información sobre el proceso y observa el fichero maps. Este fichero contiene una línea para cada región reservada. La primera columna nos indica la dirección inicial y final de la región (en hexadecimal). La diferencia entre ambos valores nos da el tamaño de la región. Busca en la página del man para *proc* el formato de la salida del fichero maps y el significado del resto de campos.



PREGUNTA 4 ¿A qué región de las descritas en el maps pertenece cada variable y cada zona reservada con malloc? Apunta la dirección inicial, dirección final y el nombre de la región.

2. Ejecuta en background el programa pasándole los parámetros 3, 5 y 100 (son 3 ejecuciones). Observa el maps del /proc para comparar el tamaño de las zonas de memoria en función del número de regiones reservadas con malloc.



PREGUNTA 5: Para cada ejecución apunta el número de mallocs hechos y la dirección inicial y final del heap que muestra el maps del /proc. ¿Cambia el tamaño según el parámetro de entrada? ¿Por qué?

3. Crea una copia del fichero mem1.c llamada mem1_v2.c.
4. Añade un free al final de cada iteración del bucle de reserva de regiones en mem1_v2.c y vuelve a ejecutar en background (con el parámetro 100), observando el fichero maps del /proc y el tamaño de la zona que alberga las regiones reservadas con malloc.



PREGUNTA 6: ¿Cuál es el tamaño del heap? ¿Cambia el tamaño según el resultado de la pregunta anterior? ¿Por qué?

5. Crea una copia de mem1.c llamada mem1_v3.c. Modifica el código de mem1_v3.c substituyendo la llamada a la función malloc por la llamada a sistema sbrk y ejecutalo en background, usando los mismos parámetros que en el apartado 6.3. Observa el fichero maps del /proc y el tamaño del heap que alberga las regiones reservadas con sbrk.



PREGUNTA 7: Para cada ejecución apunta el número de mallocs hechos y la dirección inicial y final del heap que se muestra en el maps del /proc. ¿Cambia el tamaño del heap respecto al observado en la pregunta 6.3? ¿Por qué?

6. Modifica el código anterior para que implemente (además de la reserva) la liberación de memoria utilizando sbrk (llamada a sistema) en vez de malloc (librería de C).

Sesión 7: Gestión de Memoria

Preparación previa

Objetivos

- Entender la diferencia entre espacio lógico del proceso y la memoria física que ocupa.
- Entender el concepto de acceso ilegal a memoria.
- Entender los efectos de no liberar la memoria no necesaria.
- Entender efectos de sobrecargar la memoria de la máquina.

Habilidades

- Saber cómo reservar y liberar memoria dinámica y sus implicaciones en el sistema.
- Ser capaces de obtener información sobre los procesos en ejecución a través del `/proc`.
- Ser capaces de detectar y corregir errores en el uso de memoria de un código.
- Relacionar el rendimiento del sistema con el uso de memoria que hacen los procesos.

Conocimientos previos

- Programación usando memoria dinámica.
- Uso de punteros.
- Saber interpretar y consultar la información disponible sobre el sistema y los procesos en el directorio `/proc`.

Guía para el trabajo previo

- Antes de la sesión, consultad el `man` (`man nombre_comando`) de los siguientes comandos. En concreto, para cada comando debéis leer y entender perfectamente: la SYNOPSIS, la DESCRIPTION y las opciones que os comentamos en la columna “Opciones” de la tabla.

Para leer en el man	Descripción básica	Opciones a consultar
<code>getpagesize</code>	Devuelve el tamaño de página del sistema	
<code>free</code>	Muestra la cantidad de memoria de la máquina	<code>-b</code>
<code>vmstat</code>	Muestra estadísticas sobre el uso de la memoria	<code>-n</code>
<code>top</code>	Muestra información sobre el sistema y los procesos en ejecución	comando interactivo: <code>f</code>
<code>/proc</code>	Contiene información sobre el sistema y los procesos en ejecución	<code>/proc/[pid]/stat</code>
<code>sbrk</code>	Llamada a sistema que (potencialmente) modifica el tamaño de la sección de datos	<code>incremento=0</code>
<code>getrusage</code>	Consulta el uso de los recursos	

- En la página web de la asignatura (<http://docencia.ac.upc.edu/FIB/grau/SO>) tenéis el fichero S7.tar.gz que contiene todos los ficheros fuente que utilizaréis en esta sesión. Créate un directorio en tu máquina, copia en él el fichero S7.tar.gz y desempaquéalo (tar xzf S7.tar.gz).
- Crea un fichero de texto llamado previo.txt y contesta en él a las siguientes preguntas.
- El fichero mem2_1.c contiene un código que tiene un error en el uso de un puntero. Ejecuta el programa y comprueba que error aparece. Modifica el código (en el fichero mem2_1_v2.c) para que cuando el programa genere el signal de tipo SIGSEGV (segmentation fault), la rutina de atención al signal muestre un mensaje de error por pantalla y acabe la ejecución del programa.
- Consulta el man del *proc* e indica, en el fichero “previo.txt”, en qué fichero del /proc, y en qué campo del fichero, se muestra el número de fallos de página del proceso (minor page faults y major page faults).
- Ejecuta el comando vmstat de manera que la salida se ejecute cada segundo. ¿Qué opción has utilizado? ¿En qué columnas se muestra la cantidad de swap-in y swap-out? Contesta a las preguntas en el fichero “previo.txt”.
- **PARA ENTREGAR: previo07.tar.gz**
 - **#tar zcfv previo07.tar.gz previo.txt mem2_1_v2.c**

Bibliografía

- Transparencias del Tema 3 (Memoria) de SO-grau
- Capítulo 8 (Main Memory) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.
- Capítulo 9 (Virtual Memory) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.

Ejercicios a realizar en el laboratorio

- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo:
- **PARA ENTREGAR: sesion07.tar.gz**
 - **#tar zcfv sesion07.tar.gz entrega.txt mem2_2_v2.c**



1. Accesos incorrectos

El fichero mem2_2.c contiene un código que tiene un error en el uso de un puntero (diferente del error que había en el fichero mem2_1.c visto en el estudio previo). Ejecuta el programa y comprueba que realmente no funciona.

1. Modifica el código (llámalo mem2_2_v2.c) para que cuando el proceso reciba el signal de tipo SIGSEGV se imprima en la salida estándar un mensaje indicando:
 - a. El valor de la dirección de la variable p.
 - b. El contenido de esta variable (dirección a la que apunta el puntero).

- c. La dirección dónde finaliza el heap del proceso.



PREGUNTA 1: ¿Qué error contiene el código del programa? ¿Por qué el programa no falla en las primeras iteraciones? Explica cómo se podría solucionar el error.

2. Reservando memoria

Analiza el código que contiene el fichero `mem2_3.c` y compílalo. Comprueba también el significado de cada uno de los dos parámetros de entrada del programa.

1. Ejecuta el programa pidiéndole que no libere la región reservada al final de cada iteración.



PREGUNTA 2: ¿Detectas algún patrón en la cadena de tiempos mostrada? ¿A que crees que es debido? (PISTA: consulta en el man la llamada `getpagesize`)

2. Ejecuta ahora el programa pidiéndole que libere la región reservada al final de cada iteración.



PREGUNTA 3: ¿Qué diferencias observas con el apartado 2.1? ¿A qué crees que se deben las diferencias?

3. El area de swap

Analiza el código que contiene el fichero `mem2_4.c` y compílalo. Haz los siguientes ejercicios:

1. Mediante el comando `free` consulta la cantidad de memoria física, en bytes, de la máquina (supongamos que el valor es F bytes).



PREGUNTA 4: ¿Cuánta memoria física tiene el sistema (F) en bytes?

2. **Ejecución con un sólo proceso:** Utiliza dos shells diferentes. En una lanza el comando `vmstat` para que se ejecute periódicamente (por ejemplo, cada segundo) y en otra lanza el programa `mem2_4` con los parámetros que se indican en la siguiente tabla (las ejecuciones tienen que ser secuenciales, es decir no lances el siguiente `mem2_4` hasta que hayas eliminado el anterior):

Ejecución	Tamaño Región	Número Procesos	Número Iteraciones
Ejecución 1	$F/4$	1	1
Ejecución 2	$F/4$	1	4

Para cada ejecución observad el tiempo de ejecución de cada bucle de acceso y el número de swap-in y swap-out que va reportando el `vmstat`. Cuando acabe el bucle que cuenta recorridos sobre el vector (justo antes de entrar en el bucle infinito), consulta para ese proceso el número de fallos de página que ha provocado su ejecución (minor page faults y major page faults), accediendo al fichero `stat` de su directorio correspondiente en el `/proc`. Observa como con un único proceso no se pone en marcha el mecanismo de swap del sistema.



PREGUNTA 5: Rellena la siguiente tabla. ¿En la Ejecución 2 por qué crees que cambia el tiempo de acceso según el número de iteración?

Ejecución	Fallos de página	Mínimo tiempo de bucle de acceso	Máximo tiempo de bucle de acceso
Ejecución 1			
Ejecución 2			

3. **Ejecución con varios procesos:** Utiliza de nuevo dos shells diferentes. En una Shell ejecuta el comando top, que debes configurar para que muestre al menos la siguiente información para cada proceso: pid, comando, memoria virtual, memoria residente, memoria de swap y número de fallos de página. En la otra Shell ejecuta el programa mem2_4 utilizando los siguientes parámetros (las ejecuciones tienen que ser secuenciales, es decir no lances el siguiente mem2_4 hasta que hayas eliminado el anterior):

Ejecución	Tamaño Región	Número Procesos	Número Iteraciones
Ejecución 3	F/4	4	1
Ejecución 4	F/4	6	1
Ejecución 5	F/4	4	4

No hace falta que esperes a que acabe la Ejecución 5. Una vez obtenido el mensaje de la primera iteración para todos los procesos puedes interrumpir la ejecución.



PREGUNTA 6: Rellena la siguiente tabla (suma los fallos de página de todos los procesos de una misma ejecución):

Ejecución	Suma de Fallos de página
Ejecución 3	
Ejecución 4	
Ejecución 5	

Sesión 8: Gestión de Entrada/Salida

Preparación previa

Objetivos

- Entender cómo funciona un device driver
- Entender la vinculación entre dispositivo lógico y operaciones específicas
- Entender el concepto de independencia de dispositivos.
- Entender los mecanismos que ofrece la shell para la redirección y comunicación de procesos.

Habilidades

- Ser capaz de crear y eliminar nuevos dispositivos lógicos
- Ser capaz de cargar y descargar módulos del kernel
- Ser capaz de entender la implementación específica de las operaciones read y write
- Ser capaz de aplicar las ventajas de la independencia de dispositivos.
- Saber redireccionar la entrada y la salida de un proceso desde la shell.
- Saber comunicar dos comandos a través de pipes sin nombre desde la shell.

Guía para el trabajo previo

- Antes de la sesión, consultad el man (`man nombre_comando`) de los siguientes comandos. En concreto, para cada comando debéis leer y entender perfectamente: la SYNOPSIS, la DESCRIPTION y las opciones que os comentamos en la columna “Opciones” de la tabla.

Para leer en el man	Descripción básica	Opciones a consultar
mknod	Comando que crea un fichero especial	c,p
insmod	Comando que inserta un módulo en el kernel	
rmmod	Comando que descarga un módulo del kernel	
lsmod	Comando que muestra el estado de los módulos cargados en el kernel	
sudo	Comando que permite ejecutar un comando como root	
open	Abre un fichero o dispositivo	
write	Llamada a sistema para escribir en un dispositivo virtual	
read	Llamada a sistema para leer de un dispositivo virtual	
siginterrupt	Permite que los signals interrumpan a las	

	llamadas a sistema	
grep	Comando que busca patrones en un fichero o en su entrada estándar si no se le pasa fichero como parámetro	-c
ps	Comando que muestra información sobre los procesos en ejecución	-e, -o
sprintf	Función de la librería de C que genera un string con el formato que se indica como parámetro	
strace	Lista las llamadas a sistema ejecutadas por un proceso	-e, -c

- En la página web de la asignatura (<http://docencia.ac.upc.edu/FIB/grau/SO>) tenéis el fichero S8.tar.gz que contiene todos los ficheros fuente que utilizaréis en esta sesión. Créate un directorio en tu máquina, copia en él el fichero S8.tar.gz y desempaquetalo.
- Contesta a las siguientes preguntas en el fichero “previo.txt”.

1. Redirección de entrada/salida, uso de los dispositivos lógico terminal y pipe

- El fichero es1.c contiene un programa que lee de la entrada estándar carácter a carácter y escribe lo leído en la salida estándar. El proceso acaba cuando la lectura indica que no quedan datos para leer. Compila el programa y, a continuación, ejecútalo de las siguientes maneras para ver cómo se comporta en función de los dispositivos asociados a los canales estándar del proceso:
 - a) Introduce datos por teclado para ver cómo se copian en pantalla. Para indicar que no quedan datos pulsa ^D (Control+D), que es el equivalente a final de fichero en la lectura de teclado. **¿Qué valor devuelve la llamada read después de pulsar el ^D?**
 - b) Crea un fichero con un editor de texto cualquiera y lanza el programa ./es1 asociando mediante la shell su entrada estándar a ese fichero. Recuerda (ver Sesión 1) que es posible redireccionar la entrada (o la salida) estándar de un comando a un fichero utilizando el carácter especial de la shell < (o > para la salida). **Apunta el comando utilizado en el fichero “previo.txt”.**
- Los Shell de Linux permiten que dos comandos intercambien datos utilizando una pipe sin nombre (representada por el carácter '|'). La secuencia de comandos conectados mediante pipes se llama pipeline. Por ejemplo, la ejecución del pipeline:

```
# comando1 | comando2
```

hace que el Shell cree dos procesos y los conecte mediante una pipe sin nombre. Redirecciona la salida estándar del proceso que ejecuta el comando1, asociándola con el extremo de escritura de esa pipe, y redirecciona la entrada estándar del proceso que ejecuta el comando 2, asociándola con el extremo de lectura de la misma pipe. De esta manera, todo lo que el proceso comando1 escriba en su salida

estándar será recibido por el proceso comando2 cuando lea de su entrada estándar.

En tu directorio home, ejecuta el pipeline:

```
#ls -l |grep ^d
```

¿Cuál es el resultado? ¿Qué operación realiza el comando 'grep ^d'?

- Ejecuta un pipeline que interconecte los comandos ps y grep, y que muestre en la salida estándar el PID, el usuario y el nombre de todos los procesos bash que se están ejecutando en el sistema. **Anota el comando en el fichero "previo.txt".**

2. Formato de salida

- En Linux, el interfaz de entrada/salida está diseñado para el intercambio de bytes sin interpretar el contenido de la información.

Es decir, el sistema operativo se limita a transferir el número de bytes que se le indica a partir de la dirección de memoria que se le indica, y es responsabilidad del programador el interpretar correctamente esos bytes, almacenándolos en las estructuras de datos que le interese en cada momento. A la hora de recuperar un dato que se ha guardado en un fichero, el programador deberá tener en cuenta el formato en el que se ha guardado.

El fichero es7.c contiene un programa que escribe en salida estándar un entero usando la representación interna de la máquina. Compílo y ejecútalo redireccionando su salida estándar a un fichero:

```
#./es7 > foo.txt
```

Escribe un programa es7_lector.c que al ejecutarlo de la siguiente manera:

```
# ./es7_lector < foo.txt
```

sea capaz de leer e interpretar correctamente el contenido de este fichero.

- En el caso del dispositivo lógico terminal, el device driver que lo gestiona interpreta todos los bytes que se quieren escribir como códigos ascii, mostrando el carácter correspondiente. El fichero es8.c contiene un programa que escribe dos veces un número por salida estándar: una usando la representación interna de la máquina y otra convirtiendo antes el número a string. Compílo y ejecútalo. **¿Cuántos bytes se escriben en cada caso? ¿Qué diferencias observas en lo que aparece en pantalla?**

3. Asociación de dispositivo lógico y dispositivo físico

- El subdirectorio "deviceDrivers" contiene el código de dos device drivers simples: myDriver1.c y myDriver2.c. Estos device drivers sólo implementan su código de inicialización y de finalización, y la función específica de lectura del dispositivo. Además tenéis un makefile que compila ambos device drivers (utilizando el makefile que viene con la distribución de Linux) y dos scripts que se encargan de instalar y de desinstalar los device drivers.

- Analiza el fichero fuente de los dos device drivers y contesta a las siguientes preguntas:

a) **¿Qué función sirve para implementar el read específico del dispositivo gestionado por cada device driver?**

b) **¿Qué son el major y el minor? ¿Para qué sirven? ¿Qué major y minor utilizan los dos device drivers?**

PARA ENTREGAR: previo08.tar.gz

i. **#tar zcfv previo08.tar.gz es7_lector.c previo.txt**

Bibliografía

- Transparencias del Tema 4 (Entrada/Salida) de SO-grau
- Capítulo 13 (I/O Systems) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.
- A. Rubini y J. Corbet. Linux device drivers, 2nd ed, O'Reilly & Associates, Inc., 2001 (<http://www.xml.com/ldd/chapter/book/>).

Ejercicios a realizar en el laboratorio

- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en **negrita** en medio del texto y marcadas con el símbolo:



- **PARA ENTREGAR: sesion08.tar.gz**
- **#tar zcfv sesion08.tar.gz entrega.txt disp.c**

Redireccionamiento y buffering

En este primer ejercicio vamos a trabajar con el fichero es1 visto en el trabajo previo. Como ya se ha comentado, este fichero contiene un programa que lee de la entrada estándar carácter a carácter y escribe lo leído en la salida estándar. A continuación, realiza los siguientes ejercicios:

1. Ejecuta el comando ps desde un terminal. La columna *TTY* de la salida del ps te dirá qué fichero dentro del directorio /dev representa al terminal asociado al shell que tienes en ejecución. Abre un nuevo terminal y ejecuta de nuevo el comando ps. Observa ahora que el fichero que representa al terminal es diferente.
2. Ejecuta, desde el segundo terminal, el programa es1 redireccionando su salida estándar para asociarla con el fichero que representa al primer terminal. Observa como lo que se escribe en el segundo terminal aparece en el primero.
3. Escribe un comando en la Shell que lance dos procesos que ejecuten el programa es1 y que estén conectados mediante una pipe sin nombre. Introduce unos cuantos caracteres mediante el teclado y pulsa ^D para finalizar la ejecución de los procesos.

4. Crea una copia del programa es1.c llamándola es1_v2.c. Modifica el programa es1_v2.c para que, en vez de leer y escribir los caracteres de uno en uno, lo haga utilizando un búfer (char buffer[256]).
5. Ejecuta los siguientes comandos:

```
# strace -c -e read ./es1_v2 < foo.txt
```

```
# strace -c -e read ./es1 < foo.txt
```

El comando strace, ejecutado de esta manera, ejecuta el programa que se le pasa (en el primer caso ./es1_v2 y en el segundo ./es1) y muestra en salida estándar la información sobre las llamadas a sistema read que realiza. En particular, la columna “calls” muestra el número de veces que se ha utilizado esta llamada a sistema.



PREGUNTA 1: Apunta en el fichero “entrega.txt” los comandos que has utilizado en cada apartado y el resultado de las dos ejecuciones del comando strace. Además entrega el fichero “es1_v2.c” ¿Qué diferencias observas en las dos ejecuciones de strace (fíjate sólo en la columna calls)? ¿Qué influencia puede tener sobre el rendimiento de ambas versiones de código? ¿Por qué?

Formato de salida

Analiza en detalle el código de los ficheros es2.c, es3.c y es4.c y asegurate de entender lo que hacen antes de continuar. A continuación, compílalos utilizando el comando make.

1. Ejecuta dos veces el programa es2 primero poniendo el primer parámetro a 0 y luego a 1 (utiliza el valor que quieras para el segundo parámetro). Redirecciona también la salida estándar del proceso para asociarla a dos ficheros diferentes. Observa el contenido de los dos ficheros generados.



PREGUNTA 2: Explica las diferencias observadas en la salida del programa cuando el primer parámetro vale 0 o 1. ¿Para qué crees que sirve este parámetro?

2. Ejecuta dos veces el programa es3 redireccionando su entrada estándar en cada ejecución para asociarla a cada uno de los ficheros generados en el apartado anterior.



PREGUNTA 3: Explica el motivo de los resultados observados dependiendo del formato fichero de entrada.

3. Ejecuta ahora dos veces el programa es4 de la misma manera que has ejecutado el programa es3 en el apartado anterior.



PREGUNTA 4: Explica las diferencias observadas entre la salida del programa es3 y es4. ¿Por qué la salida es inteligible para uno de los ficheros de entrada y no para el otro?

Ciclo de vida

Analiza el contenido de los ficheros es5.c, es1.c y asegurate de entender su funcionamiento.

1. Compila los dos programas y ejecuta cada uno de ellos en un shell diferente. A continuación ejecuta el siguiente comando:

```
# ./showCpuTime.sh ./es5 ./es1
```

showCpuTime.sh es un script que muestra el tiempo de consumo de CPU de cada uno de los programas pasados como parámetro cada cierto tiempo (cada 2 segundos).

2. Cuando acabe el script, mata los 2 procesos es5 y es1,



PREGUNTA 5: Escribe los valores que ha mostrado el script showCpuTime.sh para cada uno de los procesos y comenta las diferencias entre ellos. ¿A qué se deben?

Enunciado 4

Crea una copia del fichero es6.c llamándola es6_v2.c. Modifica el programa es6_v2.c para reprogramar la gestión del signal SIGINT y que muestre un mensaje por salida estándar informando de que se ha recibido. Utiliza la llamada a sistema siginterrupt para que el read devuelva error al recibir este signal y modifica el programa principal para que después del read se muestre un mensaje en salida estándar indicando el resultado de la operación: read correcto, read con error, o read interrumpido por signal. Comprueba el buen funcionamiento de tu código lanzando el programa y enviándole el signal SIGINT pulsando ^C.



PREGUNTA 6: ¿Qué pasaría si no añadiéramos siginterrupt al código? Entrega el código programado en el fichero es6_v2.c.

Ejercicio sobre Device Drivers

El objetivo de este ejercicio es que comprobéis cómo mantiene Linux la asociación entre dispositivo lógico y dispositivo físico. Es decir, cómo es capaz de traducir la función genérica del interfaz de acceso, en el código específico implementado por los device drivers.

Para ello vamos a utilizar los device drivers que habéis analizado en el trabajo previo y que se encuentran en el directorio deviceDrivers.

1. En el caso de los dos device drivers, el major y el minor están fijos en el código. Dependiendo del sistema podría ser que estos números en concreto ya estuviesen en uso y por lo tanto el driver no se pudiera instalar. Si ocurre esto, sustituye el major en el código por uno que no esté en uso. Debes tener en cuenta además que los majors de estos dos drivers también tienen que ser diferentes ya que ambos van a estar instalados al mismo tiempo. Para ver la lista de drivers del sistema y los majors usados puedes ver el contenido del fichero /proc/devices. .



PREGUNTA 7: ¿Estaban ya en uso el major especificado en el código? En caso afirmativo, ¿qué driver lo estaba utilizando?

2. Ejecuta el siguiente script:

```
#./installDrivers.sh
```

Este script compila y carga en memoria los dos device drivers (myDriver1 y myDriver2). Para ello, usa el comando make, obteniendo los módulos compilados (myDriver1.ko y myDriver2.ko) correspondientes al primer y segundo driver. A continuación utiliza el comando insmod para instalar myDriver1 y miDriver2 (**nota: para instalar/desinstalar un dispositivo es necesario ser root, por eso se utiliza el comando sudo**).

3. Utiliza el comando lsmod para comprobar que los módulos se han cargado correctamente.



PREGUNTA 8: Apunta la línea de la salida de lsmod correspondiente a myDriver1 y myDriver2.

4. Utiliza el comando mknod para crear un dispositivo nuevo, llamado myDevice, de tipo *carácter* (opción c) en tu directorio actual de trabajo con el major y el minor definidos por myDriver1. **Para crear un dispositivo es necesario ser root (ver comando sudo).**



PREGUNTA 9: Apunta la línea de comandos que has utilizado para crear el dispositivo.

5. Ejecuta el siguiente comando:

```
#./es1 < myDevice
```



PREGUNTA 10: Anota en el fichero “entrega.txt” el resultado de la ejecución y explica a qué se debe.

6. Ahora borra myDevice y vuelve a crear un dispositivo de tipo carácter con el mismo nombre, pero asociándole el major y el minor definidos por myDriver2.c. Y ejecuta de nuevo el comando del apartado 5.



PREGUNTA 11: Anota el resultado de la ejecución. Explica el motivo de las diferencias observadas comparando la salida de este ejercicio con la de la pregunta 4.

7. Elimina myDevice y ejecuta el siguiente script:

```
#./uninstallDrivers.sh
```

Este script desinstala myDriver1 y myDriver2.

Sesión 9: Gestión de Entrada/Salida

Preparación previa

Objetivos

- Entender las diferencias entre pipes sin nombre, con nombre y sockets.
- Entender el funcionamiento del interfaz de acceso a dispositivos de Unix.

Habilidades

- Ser capaces de comunicar procesos utilizando pipes sin nombre.
- Ser capaces de comunicar procesos utilizando pipes con nombre.
- Ser capaces de comunicar procesos utilizando sockets locales.

Conocimientos previos

- Llamadas a sistema de gestión de procesos

Guía para el trabajo previo

- Antes de la sesión, consultad el man (man nombre_comando) de los siguientes comandos. En concreto, para cada comando debéis leer y entender perfectamente: la SYNOPSIS, la DESCRIPTION y las opciones que os comentamos en la columna “Opciones” de la tabla.

Para leer en el man	Descripción básica	Opciones a consultar
mknod	Comando que crea un fichero especial	P
mknod (llamada al sistema)	Llamada al sistema que crea un fichero especial	
pipe	Llamada a sistema para crear una pipe sin nombre	
open	Abre un fichero o dispositivo	O_NONBLOCK, ENXIO
close	Cierra un descriptor de fichero	
dup/dup2	Duplica un descriptor de fichero	
socket	Crea un socket	AF_UNIX, SOCK_STREAM
bind	Asigna un nombre o dirección a un socket	
listen	Espera conexiones a un socket	
accept	Acepta una conexión en un socket	
connect	Inicia una conexión a un socket	

- Crea una pipe con nombre mediante el comando mknod. A continuación lanza un proceso que ejecute el programa ‘cat’ redireccionando su salida estándar hacia la pipe que acabas de crear. En una shell diferente lanza otro proceso que ejecute también el programa ‘cat’, pero ahora redireccionando su entrada estándar hacia la pipe que

acabas de crear. Introduce datos por teclado, en la primera Shell, y pulsa ^D para indicar el fin. Anota en el fichero “previo.txt” los comandos que has ejecutado.

- ¿Es posible comunicar los dos comandos ‘cat’ desde dos terminales diferentes a través de una pipe sin nombre (por ejemplo, utilizando un pipeline de la shell visto en la sesión anterior)? ¿y desde el mismo terminal? Razona la respuesta.
- Escribe en el fichero “previo.txt” el código que deberíamos añadir a un programa para redireccionar su entrada estándar al extremo de escritura de una pipe sin nombre utilizando las llamadas al sistema close y dup. Imagina que el descriptor de fichero asociado al extremo de escritura de la pipe es el 4.
- En la página web de la asignatura (<http://docencia.ac.upc.edu/FIB/grau/SO>) tenéis el fichero S9.tar.gz que contiene todos los ficheros fuente que utilizaréis en esta sesión. Créate un directorio en tu máquina, copia en él el fichero S10.tar.gz y desempaquéalo (tar xzfv S9.tar.gz).
- El fichero “socketMng.c” contiene unas funciones de gestión básica de sockets (creación, solicitud de conexión, aceptación de conexión y cierre de dispositivos virtuales).
 - Analiza en detalle el código de la función createSocket y serverConnection, y busca en el man el significado de las llamadas a sistema socket, bind, listen y accept.
 - Explica en el fichero “previo.txt” paso a paso lo que hacen estas dos funciones.
- **PARA ENTREGAR: previo9.tar.gz**

#tar xzfv previo9.tar.gz previo.txt

Bibliografía

- Transparencias del Tema 4 (Entrada/Salida) de SO-grau.
- Capítulo 13 (I/O Systems) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.

Ejercicios a realizar en el laboratorio

- A medida que vayas realizando los ejercicios, modifica el Makefile para poder compilar y montar los nuevos programas que se piden.
- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo:



- **PARA ENTREGAR: sesion9.tar.gz**
 - **#tar xzfv sesion9.tar.gz entrega.txt sin_nombre.c lector.c escritor.c escritor_v2.c lector_socket.c escritor_socket.c Makefile**

Pipes sin nombre

Escribe un programa en el fichero “sin_nombre.c” que cree una pipe sin nombre y un proceso hijo, cuyo canal de entrada estándar deberá estar asociado al extremo de lectura de la pipe.

Para hacer la redirección utiliza las llamadas a sistema `close` y `dup`. El proceso hijo deberá mutar su imagen para pasar a ejecutar el comando `'cat'` visto en el trabajo previo. Por su parte, el proceso padre enviará a través de la pipe el mensaje de texto "Inicio" a su hijo, cerrará el canal de escritura de la pipe y se quedará a la espera de que el hijo acabe. Cuando eso suceda, el padre mostrará el mensaje "Fin" por la salida estándar y acabará la ejecución.

1. Ejecuta el programa anterior haciendo que el Shell redireccione la salida estándar del padre a un fichero.



PREGUNTA 1: ¿Qué contiene el fichero al final de la ejecución? ¿Cómo se explica ese contenido?

2. Cambia el código del padre para que no cierre el extremo de escritura de la pipe después de enviar el mensaje.



PREGUNTA 2: ¿Acaba el programa padre? ¿y el hijo? ¿Por qué?.

Pipes con nombre

Escribe dos programas que se comuniquen a través de una pipe con nombre. Uno de ellos (`lector.c`) leerá de la pipe hasta que la lectura le indique que no quedan más datos para leer y mostrará en salida estándar todo lo que vaya leyendo. El otro proceso (`escritor.c`) leerá de la entrada estándar hasta que la lectura le indique que no quedan datos y escribirá en la pipe todo lo que vaya leyendo. Cuando no queden más datos para leer los dos programas deben acabar.



PREGUNTA 3: Si quisiéramos que el lector también pudiera enviar un mensaje al escritor ¿podríamos utilizar la misma pipe con nombre o deberíamos crear otra? Razona la respuesta.

Escribe otra versión del programa escritor en la pipe, llamada `escritor_v2.c`. Este programa al intentar abrir la pipe, si no hay ningún lector de la pipe, mostrará un mensaje por la salida estándar que indique que se está esperando a un lector y a continuación se bloqueará en el open de la pipe hasta que un lector abra la pipe para leer. Consulta el error `ENXIO` en el man de open para ver como implementar este comportamiento.

Sockets

Modifica el código de `"lector.c"` y `"escritor.c"` realizados en el apartado 2 para que, en vez de utilizar pipes con nombre, la comunicación se realice utilizando sockets locales (tipo `AF_UNIX`). El escritor debe realizar el papel de cliente, mientras que el lector hará el papel de servidor. Llama a los nuevos ficheros `"lector_socket.c"` y `"escritor_socket.c"`.

Para realizar el código debes utilizar las funciones proporcionadas en el fichero `socketMng.c` y aprovechar el código que consideres oportuno de los ficheros `exServerSocket.c` y `exClientSocket.c`. El fichero `exServerSocket.c` contiene un programa que actúa como un servidor simple creando un socket (con el nombre que recibe como parámetro) y esperando

peticiones de conexión por parte de algún cliente. El fichero `exClientSocket.c` contiene un programa que actúa como un cliente simple que solicita una conexión al socket que recibe como parámetro.

Sesión 10: Sistema de Ficheros

Preparación previa

Objetivos

- Durante esta práctica realizaremos una serie de ejercicios del tema de Sistema de Ficheros, con la finalidad de poner en práctica los conocimientos adquiridos en las clases de teoría.

Habilidades

- Ser capaz de utilizar comandos y llamadas al sistema básicas para trabajar con el SF.
- Ser capaz de modificar el puntero de lectura/escritura con la llamada lseek.

Conocimientos previos

- Llamadas al sistema de entrada/salida y sistema de ficheros.
- Llamadas al sistema de gestión de procesos.

Guía para el trabajo previo

- Repasar los apuntes de la clase de teoría, especialmente los relacionados con el sistema de ficheros basado en l-nodos.
- Consultad el man (`man nombre_comando`) de los siguientes comandos. En concreto, para cada comando debéis leer y entender perfectamente: la SYNOPSIS, la DESCRIPTION y las opciones que os comentamos en la columna “Opciones” de la tabla.

Para leer en el man	Descripción básica	Opciones a consultar
open/creat	Abre/crea un fichero o dispositivo	O_APPEND, O_TRUNC, “Permisos”
df	Devuelve información sobre el sistema de ficheros	-T, -h, -l, -i
ln	Crea enlaces (links) a ficheros	-s
namei	Procesa una ruta de un fichero hasta encontrar el punto final	
readlink	Lee el contenido de un link simbólico	
stat	Muestra información de control de un fichero	-Z, -f
lseek	Modifica la posición de lectura/escritura de un fichero	SEEK_SET, SEEK_CUR, SEEK_END

1. **Contesta las siguientes preguntas** en el fichero “previo.txt”:

- ¿Cómo podéis saber los sistemas de ficheros montados en vuestro sistema y de qué tipo son? Indica, además, en que directorios están montados.
- ¿Cómo se puede saber el número de inodos libres de un sistema de ficheros? ¿Qué comando has utilizado y con qué opciones?
- ¿Cómo se puede saber el espacio libre de un sistema de ficheros? ¿Qué comando has utilizado y con qué opciones?

2. **Ejecuta los siguientes comandos y responde en el fichero previo.txt** a las siguientes preguntas:

```
# echo "esto es una prueba" > pr.txt
# ln -s pr.txt sl_pr
# ln pr.txt hl_pr
```

- ¿De qué tipo es cada uno de links creados, sl_pr y hl_pr? Ejecuta el comando stat sobre pr.txt, sobre sl_pr y sobre hl_pr. Busca en la salida de stat la información sobre el inode, el tipo de fichero y el número de links y anótala en el fichero previo.txt. ¿Cuál es el número de links que tiene cada fichero? ¿Qué significa ese valor? ¿Qué inode tiene cada fichero? ¿Alguno de los links, sl_pr o hl_pr, tiene el mismo inode que pr.txt? ¿Si es así, qué significa eso?
 - Ejecuta los comandos **cat**, **namei** y **readlink** sobre sl_pr y sobre hl_pr:
 - Apunta el resultado en el fichero previo.txt.
 - ¿Observas alguna diferencia en el resultado de alguno de los comandos cuando se ejecutan sobre sl_pr y cuando se ejecutan sobre hl_pr? Si hay alguna diferencia, explica el motivo.
 - Elimina ahora el fichero pr.txt y vuelve a ejecutar los comandos **stat**, **cat**, **namei** y **readlink** tanto sobre sl_pr como hl_pr.
 - Apunta los resultados en el fichero previo.txt
 - ¿Observas alguna diferencia en el resultado de alguno de los comandos cuando se ejecutan sobre sl_pr y cuando se ejecutan sobre hl_pr? Si hay alguna diferencia, explica el motivo.
 - ¿Observas alguna diferencia respecto a la ejecución de estos comandos antes de borrar el fichero pr.txt? Si hay alguna diferencia, explica el motivo.
3. **Escribe un programa "crea_fichero.c"** que utilizando la llamada al sistema creat cree un fichero llamado "salida.txt" con el contenido "ABCD". Si el fichero ya existía se debe sobrescribir. El fichero creado debe tener permiso de lectura y escritura para el propietario y el resto de usuarios no podrán hacer ninguna operación.
4. **Escribe un programa "insertarx.c"** que inserte en el fichero anterior (salida.txt) la letra X entre el último y el penúltimo carácter. El resultado debe ser "ABCXD".

5. **PARA ENTREGAR: previo10.tar.gz**

- a. **#tar zcfv previo10.tar.gz previo.txt**

Bibliografía

- Transparencias del Tema 5 (Sistema de Ficheros) de SO-grau
- Capítulos 10 y 11 (File-System Interface & File-System Implementation) de A. Silberschatz, P. Galvin y G. Gagne. Operating System Concepts, 8th ed, John Wiley & Sons, Inc. 2009.

Ejercicios a realizar en el laboratorio

- Todas las preguntas que se os hagan las tendréis que contestar en un documento de texto aparte, llamado entrega.txt, en el cual indicareis, para cada pregunta, su número y vuestra respuesta. Este documento se debe entregar a través del Racó. Las preguntas están resaltadas en negrita en medio del texto y marcadas con el símbolo:



- **PARA ENTREGAR: sesion10.tar.gz**

#tar zcfv sesion10.tar.gz entrega.txt append.c invirtiendo_fichero.c insertarx2.c

Hard links y Soft links.

Crea un directorio llamado “A” dentro de tu directorio home (\$HOME). A continuación, entra en el directorio “A” utilizando el comando cd y ejecuta los siguientes comandos, que crean soft y hard links a diferentes ficheros:

```
$ echo “estoy en el directorio A” > D
$ ln -s $HOME/A $HOME/A/E
$ ln -s D $HOME/A/F
$ ln $HOME/A $HOME/A/H
$ ln D $HOME/A/G
```



PREGUNTA 1: Contesta a las siguientes preguntas en el fichero “entrega.txt”

- ¿Cual/es de los comandos anteriores han dado error al intentar ejecutarlos? Explica porqué.
- Explicar el resultado de ejecutar el comando “stat” utilizando como parámetro cada uno de los nombres simbólicos que has creado anteriormente.
- ¿Cuál sería el contenido de los ficheros D, E, F y G? Comenta las diferencias que observas al utilizar los comandos “more” o “cat” para ver el contenido del fichero y el resultado de utilizar el comando “readlink”.
- Escribe en papel los accesos que se realizan cuando se accede a los ficheros: “\$HOME/A/F”, “\$HOME/A/E” y “\$HOME/A/G”. Ahora compáralos con el resultado que obtienes cuando ejecutas el comando “namei” con cada uno de los ficheros anteriores. ¿Si ejecutas “readlink \$HOME/A/F” se realizan los mismos accesos? ¿Cómo influye el hecho de que en unos casos sea una ruta absoluta a un fichero y en otros una ruta relativa?
- Crea un soft link de un fichero a si mismo (un fichero que no exista previamente). Comenta el resultado de mirar su contenido utilizando el comando “cat”. Observa

como controla el sistema la existencia de ciclos en el sistema de ficheros. Ejecuta el comando “namei” y comenta su resultado.

Control del tamaño de los ficheros

- Crea un fichero “file” cuyo contenido es “12345”.
- Ahora implementa un programa, llamado append.c, que añada al final del fichero “file” el contenido del propio fichero. **Pista:** si cuando pruebes este programa tarda más de unos segundos en acabar, mata al proceso y comprueba el tamaño del fichero. Si ese tamaño es más que el doble del tamaño original revisa en el código la condición de fin que le has puesto al bucle de lectura del fichero.



PREGUNTA 2: Entrega el fichero append.c.

Operaciones con lseek

Crea un programa que llamaremos “invirtiendo_fichero” que hará una copia de un fichero que recibe como parámetro pero con el contenido invertido. El nombre del fichero resultante será el del fichero original con la extensión “.inv”.



PREGUNTA 3: Entrega el fichero invirtiendo_fichero.c.

Crea un fichero de datos cuyo contenido sea “123456”. Ahora implementa un código (insertarx2.c) que inserte el carácter “X” entre el “3” y el “4”. De tal manera que el contenido final del fichero sea “123X456”.



PREGUNTA 4: Entrega el fichero insertarx2.c.