

A decorative graphic on the right side of the page. It features three concentric blue circles of different sizes, each with a lighter blue ring around it. Two thin blue lines intersect at a point between the top two circles. A large, partially visible circle is at the bottom right.

Documentación de laboratorio

Curso otoño 2016-2017

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

Índice de sesiones

Sesión 1: El intérprete de comandos: shell	3
Sesión 2: El lenguaje C.....	17
Sesión 3: Procesos.....	26
Sesión 4: Comunicación de procesos.....	33
Sesión 5: Gestión de Memoria	40
Sesión 6: Análisis de rendimiento	46
Sesión 7: Gestión de Entrada/Salida	54
Sesión 8: Gestión de Entrada/Salida	63
Sesión 9: Sistema de Ficheros	67
Sesión 10: Concurrencia y Paralelismo	71

Sesión 1: El intérprete de comandos: shell

Preparación previa

1. Objetivos

El objetivo de esta sesión es aprender a desenvolverse 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. Tenemos varios usuarios creados para que se puedan hacer pruebas que involucren a varios usuarios. Los usernames de los usuarios son: "alumne", "so1", "so2", "so3", "so4" y "so5". El password es "sistemas" para todos ellos.

Para comenzar, ejecutaremos lo que llamamos una *Shell* o un intérprete de comandos. 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.

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 se pueden consultar en el manual de Bash (ejecutando el comando **man bash**).

Para ejecutar una Shell basta con ejecutar la aplicación “Terminal”. Con esta aplicación, se abre una nueva ventana (similar a la de la imagen) donde se ejecuta una nueva Shell.

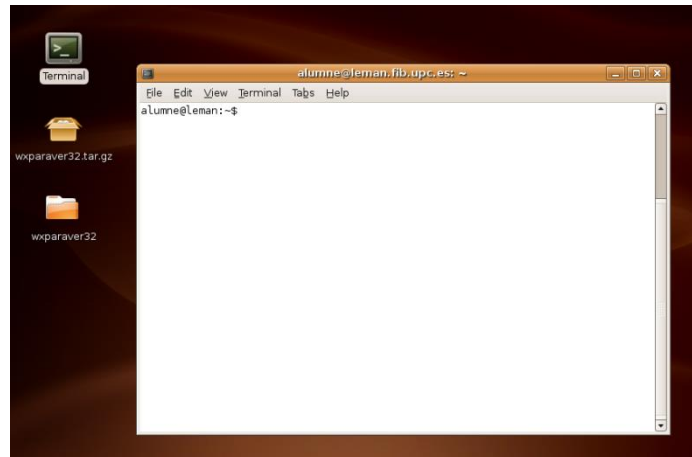


Figura 1 Ventana de la shell

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 (como parte de la instalación, se instalan también las páginas del manual que

podremos consultar a través del `man`), y el comando **help**, que nos ofrece ayuda sobre los comandos internos.

- **Lee la guía sobre cómo utilizar el man** de Linux que tienes al final de esta sección (“Utilización del manual”). A continuación, **consulta el man** (`man nombre_comando`) de los siguientes comandos. En concreto, para cada comando debes leer y entender perfectamente: la SYNOPSIS, la DESCRIPTION y las opciones que aparecen en la columna “Opciones” de la tabla.

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

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

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

- **Accede a la página del man para el bash (ejecutando el comando `man bash`)** y busca el significado de las variables de entorno `PATH`, `HOME` y `PWD` (nota: el carácter “/” sirve

para buscar patrones en las páginas del man. Utilízalo 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) deberás buscarlo tú mismo en el manual. El propio man es auto contenido en este sentido, ya que para ver todas sus opciones puedes ejecutar:

```
# 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 tienes 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 puedes encontrar tanto el nombre del comando como la descripción y un esquema (SYNOPSIS) de su utilización. En esta parte puedes 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 es 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 puedes 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 tienes 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” tu 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		
#include <unistd.h>		
ssize_t write(int fd, const void *buf, size_t count);		
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 cuatro cosas básicas que tienes 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 puedes apretar la letra **b** (back).
- Para buscar un texto e ir directamente puedes usar el carácter **/** seguido del texto. Por ejemplo **/SEE ALSO** os lleva directo a la primera aparición del texto **SEE ALSO**. Para ir a la siguiente aparición del mismo texto simplemente utiliza el carácter **n** (next).
- Para salir del man utiliza el carácter **q** (quit).

5. Bibliografía

La documentación que os damos en este cuaderno normalmente es suficiente para realizar las sesiones, pero en cada sesión os daremos alguna referencia extra.

- Guía de BASH shell:
 - De la asignatura ASO (en catalán):
<http://docencia.ac.upc.edu/FIB/grau/ASO/files/lab/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 14 LTS
- Tienes a tu disposición una imagen del sistema igual a la de los laboratorios para poder preparar las sesiones desde casa. La imagen es para VMPlayer:
 - https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0
 - Imagen: <http://softdocencia.fib.upc.es/software/Ubuntu14.tar.gz>
- Contesta en un fichero de texto llamado entrega.txt todas las preguntas que aparecen en el documento, indicando para cada pregunta su número y tu 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 tienes que probar. NO tienes que escribir el carácter **#**.
- **Para entregar: fichero sesion01.tar.gz**

```
#tar zcfv sesion01.tar.gz entrega.txt
```

Navegar por los directorios (carpetas en entornos gráficos)

Podrás observar que la gran mayoría de los comandos básicos en Linux son 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. 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.

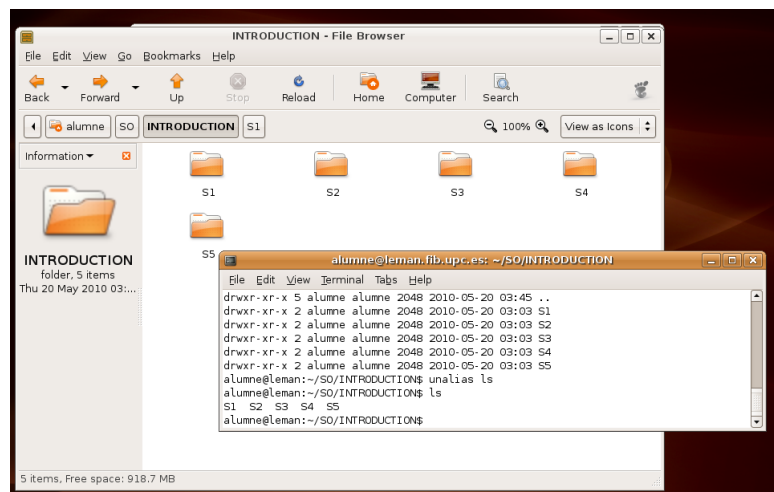
A continuación, realiza los siguientes ejercicios utilizando los comandos que consideres más oportunos:

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 abres el *File Browser* del Ubuntu, y vas a la misma “carpeta” en la que estás en la Shell, deberías ver algo similar a lo de esta imagen:



3. Cambia el directorio actual de trabajo al directorio S1.
4. Lista 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. Consulta qué opción debes 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 estás en ese momento. Si ejecutas (**cd .**) verás que sigues 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 ejecutas (**cd ..**) verás que cambias al directorio de donde venías.
 - Fíjate que estos ficheros ocultos no aparecen en el entorno gráfico: si entras en la carpeta S1 aparece vacía (en el entorno gráfico, también existe una opción de configuración para las carpetas que permite visualizar los ficheros ocultos).



PREGUNTA 2. ¿Qué comando utilizas 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. Mira en el manual qué opción hay que incluir para ver información extendida sobre los ficheros y pruébalo.



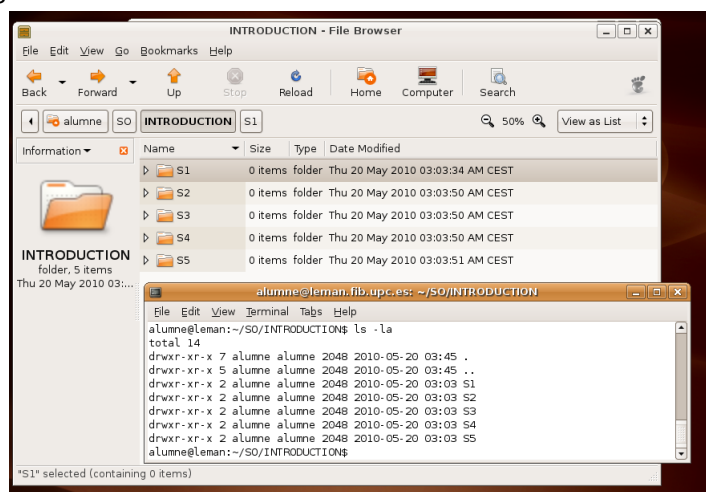
PREGUNTA 3. ¿Qué opción hay que añadir a ls para ver información extendida de los ficheros? ¿Qué campos se ven por defecto con esa opción? (si no encuentras la información en el man pregunta a tu profesor)

6. Cuando utilizamos muy a menudo una configuración específica de un comando 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 de la siguiente manera:

```
#alias ls='ls -la'
```

Ejecuta este comando alias y a continuación ejecuta ls sin argumentos. Comprueba que la salida es la del comando ls con las opciones -la.

7. Podemos ver una información similar a la del comando ls -la en el entorno gráfico. Mira cómo modificar el File Browser para conseguirlo. Deberías 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 has activado para extender la información que muestra el File Browser?

8. Desde la Shell borra algunos de los directorios que has creado, comprueba que realmente no aparecen y vuelve a crearlos. Mira cómo se hace lo mismo en el entorno gráfico.



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

Comandos básicos del sistema para acceder a ficheros

1. Crea 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 verás que tienes que abrir otra Shell porque la que tenías está bloqueada por el editor (esto sucede sólo al abrir el primer fichero, no si ya tienes abierto el editor). Esto es así porque la Shell por defecto espera hasta que el comando actual acabe antes de mostrar el prompt y procesar el siguiente comando. Para evitar tener una Shell abierta para cada comando que queremos ejecutar simultáneamente, podemos pedirle al Shell que ejecute los comandos en **segundo plano** (o en *background*). Cuando usamos este método, la Shell deja en ejecución el comando e inmediatamente muestra el prompt y pasa a esperar la siguiente orden (sin esperar hasta que el comando acabe). 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. Añade al fichero test 3 o 4 **páginas** de texto (cualquier cosa). Prueba los comandos cat y less.
 - Nota: si el fichero que creas no es lo suficientemente grande no verás ninguna diferencia.



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

4. Copia 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? Mira 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. Prueba a borrar alguno de los ficheros que acabas de crear y a cambiarle el nombre a otros. Haz un alias con la opción -i del comando rm (llámalo rm). Comprueba también la opción -i del comando mv.



PREGUNTA 8. ¿Qué hace la opción -i del comando rm? ¿Y la opción -i del mv? Escribe la orden para hacer un alias del comando rm que incluya la opción -i.

6. Otro comando que es muy útil es el comando grep. El comando grep nos permite buscar un texto (explícito o mediante un patrón) en uno o más archivos. Añade una palabra en uno de los ficheros que has copiado y prueba el comando grep para buscar esa palabra. Por ejemplo, añadimos la palabra "hola" a uno de los ficheros y hacemos la prueba:

```
#grep hola test test1 test2 test3 test4
```

7. 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 hacen referencia a tres operaciones o 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.

Modifica los permisos del fichero de test para dejar solamente los de escritura para el propietario del fichero, su grupo y el resto de usuarios, e intenta hacer un cat. Vuelve a modificar los permisos de test dejando solamente los de lectura para el propietario del fichero, su grupo y el resto de usuarios e intenta borrarlo.



PREGUNTA 9. ¿Qué opciones de chmod has utilizado para dejar solamente los permisos de escritura? ¿Qué resultado ha devuelto cat al intentar ver el fichero test? ¿Qué opciones de chmod has utilizado para dejar solamente los permisos de lectura? ¿Has 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. Se explicarán más detalles sobre el contexto o entorno de un programa en el tema de procesos.

En esta sesión introduciremos las **variables de entorno**. Las variables de entorno son similares a las constantes que se pueden 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 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).

8. Ejecuta el comando “env” para ver la lista de variables definidas en el entorno actual y su valor.

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. Para ver el valor de una variable en concreto utiliza el comando echo:

```
#echo $USERNAME
#echo $PWD
```

9. 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?
10. Comprueba el valor de las variables PATH y HOME.



PREGUNTA 10. ¿Cuál es el significado de las variables de entorno PATH, HOME y PWD?



PREGUNTA 11. La variable PATH es una lista de directorios, ¿Qué carácter hace de separador entre un directorio y otro?

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).
```

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



PREGUNTA 12. ¿Qué comando has usado para definir y consultar el valor de las nuevas variables que has definido?

12. Bájate el fichero S1.tar.gz y cópialo en la carpeta S1. Descomprímelo ejecutando el comando `tar xvfz S1.tar.gz` para obtener el programa “ls” que vamos a usar a continuación.
13. Sitúate en la carpeta S1 y ejecuta los siguiente comandos:

```
#ls
#./ls
```

Fíjate que con la primera opción, se ejecuta el comando del sistema en lugar del comando ls que hay en tu directorio. Sin embargo, con la segunda opción has ejecutado el programa ls que te acabas de bajar en lugar de usar el comando ls del sistema.

14. Añade el directorio “.” al principio de la variable PATH mediante el comando (fíjate en el carácter separador de directorios):

```
#export PATH=.:$PATH
```

Muestra el nuevo valor de la variable PATH y comprueba que, aparte del directorio “.”, aún contiene los directorios que tenía originalmente (no queremos perderlos).

Ejecuta ahora el comando:

```
#ls
```



PREGUNTA 13. ¿Qué versión del ls se ha ejecutado?: El ls del sistema o el que te acabas de descargar? Ejecuta el comando “which ls” para comprobarlo.

PREGUNTA 14. ¿El directorio en el que estás, está definido en la variable PATH? ¿Qué implica esto?

15. Modifica la variable PATH para eliminar el directorio “.”. No es posible modificar parcialmente la variable por lo que tienes que redefinirla. Muestra el contenido actual de la variable PATH y redefínela usando todos los directorios que contenía excepto el “.”.

Ejecuta ahora el comando:

```
#ls
```



16. Añade el directorio “.” al final de la variable PATH (fíjate en el carácter separador de directorios):

```
#export PATH=$PATH:.
```

Comprueba que, aparte del directorio “.”, la variable PATH aún contiene los directorios que tenía originalmente (no queremos perderlos).

Ejecuta ahora el comando:

```
#ls
```



PREGUNTA 15. ¿Qué binario ls se ha ejecutado en cada caso de los anteriores: El ls del sistema o el que te acabas de descargar? Ejecuta el comando “which ls” para comprobarlo.

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 es \$HOME/.bashrc. Cada vez que iniciamos una sesión, la Shell se configura ejecutando todos los comandos que encuentre en ese fichero.

17. 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 tus ficheros se pierden y los ficheros de configuración del sistema se reinician mediante una copia remota. Eso significa que si reinicias la sesión empezarás a trabajar con el fichero `.bashrc` original y no se conservarán tus 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`. Prueba a ejecutar el comando anterior. A continuación, pruébalo con otro comando pero con el mismo fichero de salida y comprueba 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 16. ¿Qué diferencia hay entre utilizar `>` 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 puedes utilizar el comando `tar`. Por ejemplo, si quieres generar un fichero que contenga todos los ficheros del directorio `S1`, además del fichero `.bashrc`, puedes ejecutar el siguiente comando desde tu directorio `HOME`:

```
#tar zcvf carpetaS1.tar.gz S1/* .bashrc
```

Finalmente debes guardar este fichero en un lugar seguro como por ejemplo un pendrive o enviártelo por correo electrónico.

Sesión 2: El lenguaje C

Preparación previa

Objetivos

En esta sesión el objetivo 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 en pantalla 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, printf.
- 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.

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. No es necesario leer las páginas de man completas de cada comando sino su funcionamiento básico que podéis ampliar a medida que lo necesitéis.

Para leer en el man	Descripción básica	Opciones
<code>make</code>	Utilidad para automatizar el proceso de compilación/linkaje de un programa o grupo de programas	

gcc	Compilador de C de GNU	-c,-o, -l (i mayúscula), -L,-l (ele minúscula)
sprintf	Conversión de formato almacenándola en un búffer	
atoi	Convierte un string a un número entero	
indent	Indentación de ficheros fuente	

- Consultar el resumen sobre programación en C que está disponible en la página web de la asignatura:
<http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EsquemaProgramaC.pdf>
- Consultar el resumen sobre la comparación entre C y C++ que está disponible en la página web de la asignatura:
<http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/Laboratorios/C++vsC.ppsx>
- Crea la carpeta \$HOME/S2 y sitúate en ella para realizar los ejercicios.
- Bájate el fichero S2.tar.gz, descomprímelo con `tar xzfv S2.tar.gz` para obtener los ficheros de esta sesión.

Durante el curso, utilizaremos directamente las llamadas a sistema para escribir en pantalla. Lo que en C++ era un `cout`, aquí es una combinación de **sprintf** (función de la librería de C) + **write** (llamada a sistema para escribir).

La función **sprintf** de la librería estándar de C se utiliza para formatear un texto y almacenar el resultado en un búffer. El primer parámetro es el búffer, de tipo `char*`, el segundo parámetro es una cadena de caracteres que especifica el texto a guardar así como el formato y tipo de todas las variables o constantes que se quieren incluir en la cadena, y a partir de ahí esas variables o constantes (mirad los ejemplos que os damos en el resumen de C). Todo lo que escribamos en la pantalla debe ser ASCII, por lo que previamente deberemos formatearlo con `sprintf` (excepto en el caso de que ya sea ASCII).

Posteriormente se tiene que utilizar la llamada al sistema **write** para escribir este buffer en un dispositivo. Durante estas primeras sesiones escribiremos en la “salida estándar”, que por defecto es la pantalla, o por la “salida estándar de error”, que por defecto también es la pantalla. En UNIX/Linux, los dispositivos se identifican con un número, que se suele llamar canal. La salida estándar es el canal 1 y la salida estándar de error es el canal 2. El **número de canal** es el primer parámetro de la llamada a sistema `write`. El resto de parámetros son la **dirección** donde empiezan los datos (el buffer que hemos generado con `sprintf`) y el número de bytes que se quieren escribir a partir de esa dirección. Para conocer la longitud del texto utilizaremos **strlen** (función librería de C).

Un ejemplo de utilización de `sprintf` podría ser (utiliza el man para consultar cómo especificar diferentes formatos):

```
char buffer[256];
sprintf(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”.

Solucionando problemas con el makefile

1. Modifica el fichero makefile para que funcione.
 - El makefile es el fichero que utiliza por defecto la herramienta make. Si ejecutas “make” sin parámetros, el fichero por defecto es makefile.
 - Para que el formato de un fichero makefile sea correcto debe seguir un formato como este (TAB significa que necesita un tabulador).

```
Target: dependencia1 dependencia2.. dependenciaN
[TAB]como generar Target a partir de las dependencias
```

Por ejemplo:

```
P1: P1.c
    gcc -o P1 P1.c
```

2. Modifica el makefile para que la regla listaParametros tenga bien definidas sus dependencias. Los ficheros ejecutables dependen de los ficheros fuente a partir de los cuales se generan (o ficheros objeto si separamos la generación del ejecutable en compilación y montaje).
3. Modifica el makefile para que el fichero de salida no sea a.out sino listaParametros.
4. Crea una copia del makefile, llamándola makefile_1, para entregarla.

Solucionando problemas de compilación

- Soluciona todos los errores de compilación que aparezcan. Siempre es recomendable solucionar los errores en el orden en el que aparecen.

El compilador siempre da mensajes indicando donde se ha producido el error, en qué línea de código, y cuál es el error. En este caso concreto tenemos el siguiente código:

```
1. void main(int argc,char *argv[])
2. {
3.     char buf[80];
4.     for (i=0;i<argc;i++){
5.         sprintf(buf,"El argumento %d es %s\n",i,argv[i]);
6.         write(1,buf,strlen(buf));
7.     return 0;
8. }
```

Y los errores son (son errores muy típicos, por eso los hemos puesto):

```
listaParametros.c: In function "main":
listaParametros.c:4: error: "i" undeclared (first use in this
function)
listaParametros.c:4: error: (Each undeclared identifier is
reported only once
listaParametros.c:4: error: for each function it appears in.)
listaParametros.c:5: warning: incompatible implicit declaration
of built-in function "sprintf"
listaParametros.c:6: warning: incompatible implicit declaration
of built-in function "strlen"
listaParametros.c:7: warning: "return" with a value, in
function returning void
listaParametros.c:8: error: syntax error at end of input
```

El primero indica que hay una variable (*i* en este caso) sin declarar en la **línea 4**. La línea 4 es donde se utiliza. Si te fijas, usamos la *i* en el bucle pero no está declarada. En las **líneas 5 y 6** utilizamos unas funciones (*sprintf* y *strlen*) que no hemos declarado. El compilador no las encuentra y no pueda saber si son correctas. Estas funciones están definidas en la librería de C que se añade al generar el binario, pero el compilador necesita la cabecera para ver si se usa correctamente. Consulta el man para ver que ficheros de cabecera (.h) es necesario incluir. La **línea 7** nos indica que tenemos una función (el *main*) que acaba con un *return 0* cuando la habíamos declarado como un *void*. Lo normal es definir el *main* como una función que devuelve un *int*. El último error suele aparecer cuando hay un error que se ha propagado. En este caso, faltaba cerrar una llave (la del *for*).

- Asegúrate de que el directorio "." aparece en la variable *PATH* de forma que se encuentren los ejecutables que estén en el directorio actual.

Analiza y entiende el fichero *listaParametros.c*.

Si no sabes programar en C lee los documentos recomendados antes de hacer estos ejercicios.

- La primera función de un programa escrito en C que se ejecuta es la rutina *main*.
- La rutina *main* tiene dos parámetros que se llaman *argc* y *argv*. El parámetro *argc* es un entero que contiene el número de elementos del array *argv*. Y *argv* es un array de strings que contiene los parámetros de entrada que el programa recibe al ponerse en ejecución. Existe la convención de tratar el nombre del ejecutable como primer parámetro. Por lo tanto si se sigue esa convención *argc* siempre será mayor o igual que uno y el primer elemento de *argv* será el nombre de ese ejecutable (la Shell sigue esa convención para todos los programas que pone en ejecución).
- Compila *listaParametros.c* para obtener el ejecutable *listaParametros*, y ejecútalo con diferentes parámetros (tipo y número), por ejemplo:

```
#!/listaParametros
#!/listaParametros a
#!/listaParametros a b
#!/listaParametros a b c
#!/listaParametros 1 2 3 4 5 6 7
```

¿Qué valores contienen argc y el vector argv en cada uno de los ejemplos? Fíjate, que aunque pasemos números como parámetro, los programas los reciben SIEMPRE como una cadena de caracteres.

El sistema operativo es el encargado de rellenar ambos parámetros (argc y argv) usando los datos introducidos por el usuario (lo veremos en el tema 2 de teoría).

Analiza detenidamente y entiende el fichero punteros.c.

El objetivo es que asimiles lo que significa declarar una variable de tipo puntero. Un puntero se utiliza para guardar simplemente una dirección de memoria. Y en esa dirección de memoria hay un valor del tipo indicado en la declaración del puntero. Aquí tienes un ejemplo con el caso concreto de punteros a enteros.

```
char buffer[128];
int A;          // Esto es un entero
int *PA;        // Esto es un puntero a entero (esta sin inicializar, no se
                // puede utilizar)
PA=&A;          // Inicializamos PA con la direccion de A
*PA=4;          // Ponemos un 4 en la posicion de memoria que apunta PA
if (*PA==A) {
    sprintf(buffer,"Este mensaje deberia salir siempre!\n");
}else{
    sprintf(buffer,"Este mensaje NO deberia salir NUNCA!\n");
}
write(1,buffer,strlen(buffer));
```

- Crea un fichero numeros.c y añade una función que compruebe que un string (en C se define como un "char *" o como un vector de caracteres, ya que no existe el tipo "string")¹ 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 8 cifras (define una constante que llamaremos MAX_SIZE y utilízala), y 0 en cualquier otro caso. La función debe tener el siguiente prototipo:
 - int esNumero(char *str);
- Para probar la función esNumero, añade la función main al fichero numeros.c y haz que ejecute la función esNumero pasándole todos los parámetros que reciba el programa. Escribe un mensaje para cada uno indicando si es un número o no.
- Crea un Makefile para este programa
- Utiliza indent para indentar el fichero numeros.c (#indent numeros.c).
- **PARA ENTREGAR: previo02.tar.gz**
#tar zcfv previo02.tar.gz numeros.c Makefile listaParametros.c makefile_1

¹ La definición y utilización de "strings" es básica. Si no entiendes como funciona acláralo con el profesor de laboratorio


Bibliografía

- Tutorial de programación en C: <http://www.elrincondelc.com/cursoc/cursoc.html>
- Resumen sobre programación en C:
<http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EsquemaProgramaC.pdf>
- Comparación entre C y C++:
<http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/Laboratorios/C++vsC.ppsx>

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.
- Contesta en un fichero de texto llamado entrega.txt todas las preguntas que aparecen en el documento, indicando para cada pregunta su número y tu 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 makefile

Comprobando el número de parámetros

A partir de este ejercicio, haremos que todos nuestros programas sean robustos y usables comprobando que el número de parámetros que reciben sea correcto. En caso de que no lo sea, los programas (1) imprimirán un mensaje que describa cómo usarlos, (2) una línea que describa su funcionalidad, y (3) acabarán la ejecución. Es lo que se conoce como añadir una función `usage()`.

1. Implementa **una función** que se encargue de mostrar el mensaje descriptivo de utilización del programa `listaParametros` y acabe la ejecución del programa (llama a esta función `Usage()`). **Modifica el programa `listaParametros`** para que compruebe que por

lo menos hay 1 parámetro y en caso contrario invoque a la función `Usage()`. Recuerda que el nombre del ejecutable se considera un parámetro más y así se refleja en las variables `argc` y `argv`.

Ejemplo de comportamiento esperado:

```
#listaParametros a b
El argumento 0 es listaParametros
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
```

Ejemplo de uso correcto

Ejemplo de uso incorrecto

Procesado de parámetros

2. Crea una copia de `numeros.c` (trabajo previo) llamada `suma.c`.
3. Añade otra función al `suma.c` 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);
```

4. 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 el resultado es indefinido. El comportamiento de esta función es el mismo que el de la función `atoi` de la librería de C. Utiliza la función `char2int` del apartado anterior.

```
int mi_atoi(char *s);
```

5. Modifica `suma.c` para que se comporte de la siguiente manera: Después de comprobar que todos los parámetros 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
```

6. 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 **cabecera** (“*include*”). Son ficheros de texto con extensión “.h” que contienen **prototipos de funciones** (cabeceras), definiciones de **tipos de datos** y definiciones de **constantes**. 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` y su único objetivo es facilitar la legibilidad y modularidad del código. Lo correcto es poner en los ficheros `include` aquellos prototipos de funciones, tipos de datos y constantes que queramos usar en más de un fichero.
 - 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`).
7. Separa las funciones realizadas en los ejercicios anteriores (excepto el main) en un fichero aparte (`mis_funciones.c`) y crea un fichero de cabeceras (`mis_funciones.h`), donde defines las cabeceras de las funciones que ofreces, e inclúyelo en el programa `suma.c`. Añade una pequeña descripción de cada función junto al prototipo (añádalo como comentario). Modifica el `makefile` añadiendo una nueva regla para crear el fichero objeto y modifica la regla del programa `suma` para que ahora se cree utilizando este fichero objeto. Añade las dependencias que creas necesarias.
 8. Modifica la función `Usage` de forma que como mínimo el programa `suma` reciba 2 parámetros.
 9. Crea una copia del `makefile`, llamándola `makefile_5`, para entregarla



PREGUNTA 17. ¿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

10. Mira el código del programa `punteros.c`. Modifica el `makefile` para compilarlo y ejecútalo. Analiza el código para entender todo lo que hace.
11. 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 primer 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
```


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

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

- Consulta el vídeo sobre creación de procesos que tienes en la página web de la asignatura:
<http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemploCreacionProcesosVideo.wmv>
- 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
<code>getpid</code>	Retorna el PID del proceso que la ejecuta	
<code>fork</code>	Crea un proceso nuevo, hijo del que la ejecuta	
<code>exit</code>	Termina el proceso que ejecuta la llamada	
<code>waitpid</code>	Espera la finalización de un proceso hijo	
<code>execlp</code>	Ejecuta un programa en el contexto del mismo proceso	

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.

- ¿Por qué 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`

Ejercicios a realizar en el laboratorio

- Para todos los ejercicios, se asume que realizas todos los pasos involucrados.
- Contesta en un fichero de texto llamado entrega.txt todas las preguntas que aparecen en el documento, indicando para cada pregunta su número y tu 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_v0.c myPS.c myPS2.c myPS3.c parsExec.c.`

Comprobación de errores de las llamadas a sistema

1. A partir de ahora se incluirá siempre, **para TODAS las llamadas a sistema**, la comprobación de errores. Utiliza la llamada perror (man 3 perror) para escribir un mensaje que describa el motivo que ha producido el error. Además, en caso de 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);
}
```

Creación y mutación de procesos: myPS.c

El objetivo de esta sección es practicar con las llamadas a sistema de creación y de mutación de procesos. Los códigos que desarrollarás en esta sesión te servirán como base del trabajo de las siguientes secciones.

Creación y mutación de procesos

El objetivo de esta sección es practicar con las llamadas a sistema de creación y mutación de procesos. Para ello vas a crear dos códigos diferentes: myPS.c y myPS_v0.c. Estos códigos nos servirán como base para los ejercicios de las siguientes secciones.

2. Crea un programa llamado myPS.c que reciba un parámetro (que será un nombre de usuario: root, alumne, etc.) y que cree un proceso hijo. El proceso padre escribirá un mensaje indicando su PID. El proceso hijo escribirá un mensaje con su PID y el parámetro que ha recibido el programa. Después de escribir el mensaje, ambos procesos ejecutarán un bucle “while(1);” para que no terminen (este bucle lo añadimos porque usaremos este código en la siguiente sección sobre la consulta de información de los procesos, y en esa sección nos interesa que los procesos no acaben la ejecución).
3. Crea un makefile, que incluya las reglas “all” y “clean”, para compilar y montar el programa myPS.c.



PREGUNTA 18. ¿Cómo puede saber un proceso el pid de sus hijos? ¿Qué llamada pueden utilizar los procesos para consultar su propio PID?

4. Copia el código de myPS.c en una versión myPS_v0.c. Modifica el Makefile para compilar myPS_v0.c
5. Modifica myPS.c para que el proceso hijo, después de escribir el mensaje, ejecute la función muta_a_PS. Esta función mutará al programa ps. Añade también el código de la función muta_a_PS:

```
/* 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 */
void muta_a_PS(char *username)
{
    execlp("ps", "ps", "-u", username, (char*)NULL);
    error_y_exit("Ha fallado la mutación al ps", 1);
}
```



PREGUNTA 19. ¿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

El objetivo de esta sección aprender a utilizar el `/proc` para consultar alguna información sobre la ejecución de los procesos. Para ello utilizaremos los códigos `myPS.c` y `myPS_v0.c` que has desarrollado en la sección anterior.

6. Para este ejercicio vamos a utilizar dos terminales de la Shell. En una ejecuta `myPS` con un solo `username` como parámetro. En la segunda ventana ves al directorio `/proc` y comprueba que aparecen varios directorios que coinciden con los números de PIDs de los procesos. Entra en el directorio del padre y de su hijo y mira la información extendida (permisos, propietario, etc.) de los ficheros del directorio.

PREGUNTA 20. ¿Qué directorios hay dentro de `/proc/PID_PADRE`? ¿Qué opción de `ls` has usado?

PREGUNTA 21. Para el proceso padre, apunta el contenido de los ficheros `status` y `cmdline`. Compara el contenido del fichero `environ` con el resultado del comando `env` (fíjate por ejemplo en la variable `PATH` y la variable `PWD`) ¿Qué relación hay? Busca en el contenido del fichero `status` el estado en el que se encuentra el proceso y apúntalo en el fichero de respuestas. Anota también el tiempo de CPU que ha consumido en modo usuario (búscalo en el fichero `stat` del proceso).

PREGUNTA 22. En el caso del proceso hijo, ¿a qué ficheros “apuntan” los ficheros `cwd` y `exe`? ¿Cuál crees que es el significado de `cwd` y `exe`?

PREGUNTA 23. En el caso del proceso hijo, ¿puedes mostrar el contenido de los ficheros `environ`, `status` y `cmdline` del proceso hijo? ¿En qué estado se encuentra?

7. Repite el experimento utilizando el programa `myPS_v0.c` en lugar de `myPS.c` y responde de nuevo a las preguntas para el proceso hijo. Observa las diferencias que hay entre ambas versiones del código. Recuerda que en la `v0` el proceso hijo no mutaba.

PREGUNTA 24. En el caso del proceso hijo, ¿a qué ficheros “apuntan” los ficheros `cwd` y `exe`? ¿Cuál crees que es el significado de `cwd` y `exe`? ¿Qué diferencias hay con la versión anterior? ¿A qué se deben?

PREGUNTA 25. En el caso del proceso hijo, ¿puedes mostrar el contenido de los ficheros `environ`, `status` y `cmdline` del proceso hijo? ¿En qué estado se encuentra? ¿Qué diferencias hay con la versión anterior? ¿A qué se deben?

Ejecución secuencial de los hijos: myPS2.c

El objetivo de esta sección es practicar con la llamada a sistema `waitpid` y entender cómo influye en la concurrencia de los procesos creados. En particular la vas a utilizar para crear procesos que se ejecuten de manera secuencial.

Esta llamada sirve para que el proceso padre espere a que sus procesos hijos terminen, para que compruebe su estado de finalización y para que el kernel libere las estructuras de datos que los representan internamente (PCBs). 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 esta sección 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.

```
//Ejemplo esquema secuencial
for (i=0;i<num_hijos;i++){
    pid=fork();
    if (pid==0) {
        // código hijo
        exit(0); // Solo si el hijo no muta y queremos que termine
    }
    // Esperamos a que termine antes de crear el siguiente
    waitpid(...); // los parámetros depende de lo que queramos
}
```

8. 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`.
9. Modifica `myPS2.c` para que, en lugar de recibir un parámetro (`username`), pueda recibir `N`. Haz que el programa principal cree un proceso para cada `username` y que se ejecuten de manera secuencial. Puedes eliminar el bucle infinito del final de la ejecución del proceso padre.

Ejecución concurrente de los hijos: `myPS3.c`

En esta sección se continúa trabajando con la llamada a sistema `waitpid`. Ahora la utilizarás para crear un esquema de ejecución concurrente.

```
//Ejemplo esquema concurrente
for (i=0;i<num_hijos;i++){
    pid=fork();
    if (pid==0) {
        // código hijo
        exit(0); // Solo si el hijo no muta y queremos que termine
    }
}
// Esperamos a todos los procesos
while (waitpid(...)>0); // los parámetros depende de lo que queramos
```

Aprovecharemos también para comprobar los posibles efectos que puede tener la concurrencia sobre el resultado de la ejecución.

10. Crea una copia de `myPS2.c`, llamada `myPS3.c`, con la que trabajarás en este ejercicio. Modifica también el `makefile` para poder compilar y montar `myPS3`.
11. Modifica el programa `myPS3.c` para que los hijos se creen de forma concurrente. Para poder consultar la información de los procesos, haz que el padre se quede esperando una tecla después de ejecutar el bucle de `waitpid`. Para esperar una tecla puedes usar la llamada a sistema `read` (`read(1, &c, sizeof(char))`), donde `c` es un `char`.
12. Ejecuta `myPS3` con varios `usernames` y deja al padre bloqueado después del bucle de `waitpids`. En otra ventana comprueba que ningún proceso hijo tiene un directorio en `/proc`. Comprueba también el estado en el que se encuentra el proceso padre.



PREGUNTA 26. Comprueba el fichero status de /proc/PID_PADRE. ¿En qué estado está el proceso padre?

13. 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. Comprueba si el orden en que se ejecutan los ps's es siempre el mismo. Piensa que es posible que varios resultados sean iguales.



PREGUNTA 27. ¿Qué has hecho para guardar la salida de las ejecuciones de myPS3?

Paso de parámetros a los ejecutables a través del execlp

El objetivo de esta sección es entender la relación entre los parámetros recibidos por el main de un ejecutable y los parámetros pasados a la llamada a sistema execlp. Recordad: la rutina main recibe dos parámetros: argc, de tipo entero, y argv de tipo array de strings. El sistema operativo rellena el parámetro argc con el número de elementos que tiene argv, y rellena argv con los parámetros que el usuario indica en la llamada a sistema execlp (que tienen que ser de tipo string). Para seguir la misma convención que utiliza la Shell, se tiene que hacer que el primer parámetro sea el nombre del ejecutable al que se va a mutar.

14. El programa listaParametros que trabajamos en la sesión 2, simplemente muestra por pantalla los parámetros que recibe. Escribe un programa parsExec.c que cree 4 procesos hijos. Una vez creados, el proceso padre sólo tiene que esperar hasta que todos los hijos acaben. Cada proceso hijo sólo tiene que mutar al ejecutable listaParametros, cada uno pasando unos parámetros diferentes, de manera que, al ejecutar parsExec, en la pantalla se tienen que ver los siguientes mensajes (el orden de los mensajes da igual y puede ser diferente para cada ejecución). Cada grupo de mensajes corresponde con la salida de un proceso de los que mutan:

```
El argumento 0 es listaParametros
El argumento 1 es a
El argumento 2 es b
```

```
Usage: listaParametros arg1 [arg2..argn]
Este programa escribe por su salida la lista de argumentos que recibe
```

```
El argumento 0 es listaParametros
El argumento 1 es 25
El argumento 2 es 4
```

```
El argumento 0 es listaParametros
El argumento 1 es 1024
El argumento 2 es hola
El argumento 3 es adios
```

15. Modifica el makefile para poder compilar y montar parsExec.c.

Sesión 4: Comunicación de 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, qué 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. Las funciones de tratamiento de signal deben tener la siguiente cabecera:

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

Cuando el proceso recibe un signal, el sistema pasa a ejecutar el tratamiento asociado a ese signal para ese proceso. En el caso de que el tratamiento sea una función definida por el usuario, la función recibe como parámetro el número de signal que ha provocado su ejecución. Esto nos permite asociar una misma función a diferentes tipos de signal y hacer un tratamiento diferenciado dentro de esa función.

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 del man relacionadas con el tema que se indican en la siguiente tabla.

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	Bloquea el proceso que la ejecuta hasta que recibe un signal (los signals cuyo tratamiento es ser ignorado no desbloquean el proceso)	
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 encontrarás hay una pequeña descripción de lo que hacen y cómo compilarlos. Intenta entenderlos y comprender cómo 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.

Los signals que principalmente vamos a usar son SIGALRM (alarma, temporizador), SIGCHLD (fin de un proceso hijo), o SIGUSR1/SIGUSR2 (sin significado predefinido, para que el programador pueda usarlos con el significado que le convenga). 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).

Sobre alarm1: recepción de diferentes tipos de signals y envío de eventos desde la consola

1. Ejecuta alarm1 en una consola y observa su comportamiento. ¿Qué ocurre cuando pasan 5 segundos?
2. Ejecuta de nuevo alarm1 y antes de que acabe el temporizador envíale un signal de tipo SIGKILL. Para ello, en otro terminal ejecuta el comando ps para obtener el pid del proceso y a continuación utiliza el comando kill con la opción “-KILL” para enviar al proceso el signal SIGKILL. ¿El comportamiento es el mismo que si esperas que llegue el SIGALRM? ¿Recibes un mensaje diferente en el terminal?

Recuerda que el Shell es el encargado de crear el proceso que ejecuta los comandos que introduces y recoger el estado de finalización de ese proceso. El pseudo-código del Shell para ejecutar un comando en primer plano es el siguiente:

```

while (1) {
    comando = leer_comando();
    pid_h = fork();
    if (pid_h == 0)
        execlp(comando,...);
    }
    waitpid (pid_h, &status, 0);
}

```

3. ¿Qué proceso se encarga de mostrar los mensajes que aparecen en pantalla cuando muere el proceso alarm1? ¿Con qué llamada a sistema el shell recoge el estado de finalización del proceso que ejecuta alarm1 cuando éste termina?
4. ¿Es necesario el exit que hay en el código de alarm1? ¿Se ejecuta en algún caso?

Sobre alarm2: cualquier signal se puede enviar desde el Shell.

1. Ejecuta alarm2 en un terminal. Abre otro, averigua el pid del proceso alarm2 y utiliza el comando kill para enviarle el signal “-ALRM” varias veces. ¿Qué comportamiento observas? ¿El control de tiempo ha funcionado como pretendía el código?
2. ¿Se puede modificar el tratamiento asociado a cualquier signal?
3. Mira en el man (man alarm) el valor de retorno de la llamada a sistema alarm y piensa cómo podríamos arreglar el código para detectar cuándo un SIGALRM nos llega sin estar relacionado con el temporizador.

Sobre alarm3: la tabla de programación de signals se hereda.

1. ¿Quién recibe los SIGALRM: el padre, el hijo o los dos? ¿Cómo lo has comprobado? Modifica la función “funcion_alarma” para que en el mensaje que se escribe aparezca también el pid del proceso, de forma que podamos ver fácilmente qué proceso recibe los signals.

Sobre alarm4: los SIGALRM son recibidos únicamente por el proceso que los ha generado

1. ¿Cuántos SIGALRM 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.

Sobre ejemplo_waitpid: envío de signals mediante llamada a sistema, gestión del estado de finalización de los hijos y desactivación del temporizador.

1. Analiza el código de este programa y ejecútalo. Observa que dentro del código de los procesos hijos se utiliza un temporizador para fijar su tiempo de ejecución a un segundo.
2. Modifica el código para que el proceso padre al salir del waitpid muestre en salida estándar un mensaje describiendo la causa de la muerte del hijo (ha acabado con un exit o ha muerto porque ha recibido un signal).
3. Completa el programa para limitar el tiempo máximo de espera en el waitpid en cada iteración del bucle: si al cabo de 2 segundos ningún hijo ha acabado la ejecución, el padre deberá enviarle un SIGKILL a todos sus hijos vivos. En caso de que algún hijo acabe a tiempo,

el padre desactivará el temporizador y mostrará un mensaje indicando cuánto tiempo faltaba para que se enviara el SIGALRM. Para desactivar el temporizador se puede utilizar la llamada a sistema alarm pasándole como parámetro un 0. NOTA: ten en cuenta que para hacer esto es necesario guardar todos los pids de los hijos creados e ir registrando los hijos que ya han muerto.

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.
- Contesta en un fichero de texto llamado entrega.txt todas las preguntas que aparecen en el documento, indicando para cada pregunta su número y tu 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 czfv sesion04.tar.gz entrega.txt makefile ejemplo_alarm2.c ejemplo_alarm3.c  
ejemplo_signal.c ejemplo_signal2.c eventos.c eventos2.c
```

Sobre alarm2: Detectamos qué signal ha llegado

1. 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. En el trabajo previo tenéis más detalles.

Sobre alarm3: Signals y creación/mutación de procesos

Estos ejercicios están orientados a que observéis lo que hemos explicado en clase de teoría relacionado con la gestión de signals y la creación/mutación de procesos: Al crear un proceso el

hijo HEREDA la tabla de signals de su padre. Al mutar un proceso, su tabla de signals se pone por defecto.



2. Modifica el código para que la reprogramación del signal SIGALRM (llamada a sistema signal) sólo la haga el hijo. **OBSERVA** como antes de esta modificación, tanto padre como hijo (por medio de la herencia) tenían capturado el SIGALRM. Después del cambio, la modificación del hijo es privada, por lo que el padre tiene asociado a SIGALRM la acción por defecto.

PREGUNTA 28. ¿Qué le pasa al padre ahora cuando le llega el evento SIGALRM?

3. Modifica el código para que después de programar el temporizador el proceso hijo mute para ejecutar otro programa. Este programa tiene que durar más de 10 segundos para que reciba el signal SIGALRM antes de acabar (por ejemplo, puedes implementar un programa que sólo ejecute un bucle infinito). **OBSERVA** como al mutar la acción asociada a SIGALRM se pone por defecto, ya que la tabla de signals se resetea.



PREGUNTA 29. ¿Qué pasa con la tabla de tratamientos de signals si hacemos un `execvp` (y cambiamos el código)? ¿Se mantiene la reprogramación de signals? ¿Se pone por defecto?

Sobre `ejemplo_waitpid`: Esperamos que acaben los hijos. Impacto de la implementación de los signals



PREGUNTA 30. El programa `ejemplo_waitpid`, ¿Es secuencial o concurrente?

La llamada a sistema `waitpid` normalmente se utiliza de forma bloqueante. El proceso que la ejecuta no continua hasta que algún proceso hijo termina. Si deseamos controlar la finalización de los procesos hijos sin bloquear al padre, podemos capturar el signal `SIGCHLD` y esperar que vaya llegando. En este caso, al ser un caso un poco forzado, deberemos realizar una espera activa en el proceso padre ya que de otra forma terminará su ejecución antes de que terminen los procesos hijos.

4. Crea una copia de este programa llamada `ejemplo_signal.c`. Modifica este programa para que el proceso padre no se bloquee en el `main` a la espera de los hijos, sino que ejecute la llamada `waitpid` en la función de atención al `SIGCHLD` (notificación de que un hijo ha acabado). Esto le permite continuar con su ejecución normalmente. Recuerda que en este caso debes hacer que el padre realice una espera activa para no acabar su ejecución. Lo normal sería utilizar esta opción en cas que el padre tenga algo para ejecutar. Puedes utilizar una variable global que te diga si aún quedan hijos vivos.

CAUTION! Mientras se está ejecutando el tratamiento de un signal, el sistema bloquea la notificación de nuevos signals del mismo tipo, y el proceso los recibirá al salir de la rutina. Pero, el sistema operativo sólo es capaz de recordar un signal pendiente de cada tipo. Es decir, si mientras se está ejecutando el tratamiento de un signal el proceso recibe más, al salir de la rutina sólo se tratará uno de cada tipo y el resto se perderán. Por este motivo, la rutina de tratamiento de `SIGCHLD` debe asegurar que al salir de ella se ha esperado a todos los hijos que hayan muerto durante el tratamiento del signal

pero sin bloquearte (recuerda que dentro de una rutina de tratamiento de signal jamás debes bloquearte). Mira el significado de la opción WNOHANG de la llamada a sistema waitpid (man waitpid) y utilízala para que waitpid sea no bloqueante.

5. Crea una copia de ejemplo_signal.c con el nombre ejemplo_signal2.c. Modifica este programa para que el padre, una vez creados todos los procesos hijos, les envíe a todos ellos el signal SIGUSR1 (cuya acción por defecto es terminar) y luego continúe incrementando la variable contador. Además, después de ejecutar la llamada waitpid, el padre mostrará el PID del hijo 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.
 - Crea una función que, utilizando las macros anteriores, escriba el PID del proceso que ha terminado y un mensaje indicando si ha terminado por un exit o un signal y en cada caso el exit_status o signal_code. Tenéis un ejemplo de cómo usar estas macros en las transparencias del T2.

PROTECCIÓN ENTRE PROCESOS

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

6. 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 signals al proceso que está ejecutando alarm2 desde la sesión iniciada con el usuario so1.



PREGUNTA 31. ¿El usuario so1 puedes enviar al signals al proceso lanzado por el usuario alumne?, ¿qué error da?

GESTIÓN DE SIGNALS

El objetivo de esta sección es que seas capaz de implementar por completo un programa que gestione varios tipos de signals utilizando un tratamiento diferente al de por defecto.

7. Haz un programa, llamado eventos.c, que ejecute un bucle infinito y que tenga una variable global para almacenar el tiempo que lleva el proceso ejecutándose (en segundos). Esta variable se gestiona mediante signals de la siguiente manera:
 - Cada segundo se incrementa
 - Si el proceso recibe un SIGUSR1 la variable se pondrá a cero
 - Si el proceso recibe un SIGUSR2 escribirá por pantalla el valor actual del contador

Haz que todos los signals del programa sean atendidos por la misma función. Envía los signals desde la línea de comandos y comprueba que funciona correctamente.

COMPORTAMIENTO POR DEFECTO

La reprogramación de un signal en Linux 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.

8. Crea una copia de eventos.c con el nombre eventos2.c. Modifica el código de eventos2.c para que la segunda vez que reciba el mismo signal se ejecute el comportamiento por defecto de ese signal. PISTA: consulta en la página del manual del signal la constante SIG_DFL.



PREGUNTA 32. ¿Qué mensaje muestra el Shell cuando se envía por segunda vez el mismo signal?

RIESGOS DE LA CONCURRENCIA

Cuando se programan aplicaciones con varios procesos concurrentes no se puede asumir nada sobre el orden de ejecución de las instrucciones de los diferentes procesos. Esto también aplica al envío y recepción de signals: no podemos asumir que un proceso recibirá un signal en un momento determinado. Esto hay que tenerlo en cuenta al utilizar la llamada pause.

El programa signal_perdido.c muestra esta problemática de forma artificial. Este programa crea un proceso que pretende esperar en una llamada pause la recepción de un signal. El proceso padre es el encargado de enviarle este signal. El momento en el que se envía el signal depende del parámetro del programa: se hace inmediatamente (valor de parámetro 0) o se retarda un tiempo (valor de parámetro 1).

9. Ejecuta el programa signal_perdido pasándole como parámetro 1. Anota en el fichero entrega.txt qué resultado observas. A continuación ejecuta de nuevo el programa pasándole como parámetro 0. Anota de nuevo en respuestas.txt el resultado que observas.



PREGUNTA 33. Explica a qué se debe el resultado de las ejecuciones de signal_perdido con parámetro 1 y con parámetro 0

Sesión 5: 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 qué región se ubica cada elemento de un proceso.
- Entender el efecto de malloc/free/sbrk sobre el espacio de direcciones, en particular sobre el heap.
- Saber cómo reservar y liberar memoria dinámica y sus implicaciones en el sistema.
- Ser capaces de detectar y corregir errores en el uso de memoria de un código.

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 (variables globales y funciones)	

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 S5.tar.gz que contiene todos los ficheros fuente que utilizaréis en esta sesión. Créate un directorio en tu máquina, copia en él el fichero S5.tar.gz y desempaquéalo (tar xzf S5.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>
#include<string.h>
char buff[100];
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);
    sprintf(buff,"suma de %d y %d es %d\n",i,j,s);
    write(1,buff, strlen(buff));
}
```

- 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. Busca en el man (y apunta en previo.txt) los tipos de símbolos que nos muestra nm. Por ejemplo, los símbolos etiquetados con una “D” significa que están en la sección de datos (variables globales). **¿Es posible saber la dirección de las variables “i” y “j” con el comando nm? ¿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 asígnele 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 asígnele 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” qué comando has utilizado.**
- El programa mem2_previo lee de teclado el número de elementos de un vector de enteros, inicializa el vector con ese número de elementos y a continuación los suma. Modifica este programa (llámalo mem2_previo_v2.c) para que en lugar de usar un vector estático, use memoria dinámica. Para ello declara la variable “vector” como un puntero a entero. Después de leer por teclado el número de elementos que debe tener el vector, el programa debe utilizar la llamada a sistema sbrk para reservar una región de memoria en la que quepan esos elementos y asignar la dirección de esta región a la variable vector.
- Compila de forma estática tanto mem2_previo como mem2_previo_v2. Para ambos programas haz lo siguiente: ejecuta el programa y antes de pulsar Return para que acabe, desde otro terminal, accede al directorio del /proc/PID_del_proceso que contiene la información sobre el proceso y observa el fichero maps. Este fichero contiene una línea para cada región de memoria 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. Anota en el fichero previo.txt el tamaño total de la region de heap y datos para los siguientes números de elementos del vector: 10 y 40000. **La región del heap esta etiquetada como [heap] pero la región de datos está etiquetada con el nombre del ejecutable. Deberás deducir cual es (hay varias) por los permisos de la región.** ¿Existe alguna diferencia entre los distintos valores de las ejecuciones de ambos programas?
- El fichero mem3_previo.c contiene un código que tiene un error en el uso de un puntero. Ejecuta el programa y comprueba qué error aparece. Modifica el código (en el fichero mem3_previo_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.
- **PARA ENTREGAR: previo05.tar.gz**
 - **#tar zcfv previo05.tar.gz previo.txt mem1_previo_v2.c mem2_previo_v2.c mem3_previo_v2.c**

Bibliografía

- 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.

- Contesta en un fichero de texto llamado entrega.txt todas las preguntas que aparecen en el documento, indicando para cada pregunta su número y tu 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 Makefile mem1_v2.c mem1_v3.c mem2_v2.c`

Espacio de direcciones de un proceso y compilación estática y dinámica

El objetivo de esta sección es doble: por un lado, entender cómo se organizan los datos y el código de un programa en el espacio de direcciones del proceso; por otro lado, entender cómo influye tanto en el ejecutable como en el espacio de direcciones el tipo de compilación utilizado (estática o dinámica). Para ello, usaremos comandos que permiten analizar los ejecutables y observaremos la información sobre el espacio de direcciones guardada en el `/proc`.

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 34. ¿Qué variables aparecen en la salida del `nm` de `mem1_dynamic`? ¿Qué dirección ocupa cada una? Indica a que región pertenece cada variable según la salida del `nm` y el tipo de variable (local o global).

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:
 - a. Utilizando el comando `nm` para ver los símbolos definidos dentro de cada ejecutable.
 - b. Utilizando el comando `objdump` con la opción `-d` para ver el código traducido.
 - c. Comparando los tamaños de los ejecutables resultantes.



PREGUNTA 35. Para los dos ejecutables, compilado estático y dinámico, observa su tamaño, la salida de los comandos `nm` y `objdump` ¿En qué se diferencian los dos ejecutables?

3. Ejecuta en background (en segundo plano) las dos versiones del ejecutable y compara el contenido del fichero `maps` del `/proc` para cada uno de los procesos.



PREGUNTA 36. 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?



PREGUNTA 37. ¿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.

Memoria dinámica

El objetivo de esta sección es entender cómo afecta al espacio de direcciones de un proceso la memoria dinámica reservada en tiempo de ejecución, dependiendo del interfaz que se use: sbrk o malloc. Recuerda que sbrk es la llamada a sistema, y que simplemente aumenta o reduce el tamaño del heap en X bytes (lo que pida el usuario). Malloc/free son las funciones de la librería de C, que ofrecen una gestión mucho más sofisticada de la memoria dinámica. La librería de C reserva una zona grande de heap y la utiliza para ir gestionando las peticiones del usuario sin tener que modificar el tamaño del heap. Para ello, utiliza estructuras de datos (también almacenadas en el heap) que sirven para anotar que trozos están ocupados y que trozos están libres.

4. Ejecuta en background el programa mem1 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. Analiza los resultados en función de lo explicado en clase sobre el funcionamiento de malloc.



PREGUNTA 38. Para cada ejecución apunta el número de mallocs hechos y la dirección inicial y final del heap que muestra el fichero maps del /proc. ¿Cambia el tamaño según el parámetro de entrada? ¿Por qué?

5. Crea una copia del fichero mem1.c llamada mem1_v2.c.
6. 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 39. ¿Cuál es el tamaño del heap en este caso? Explica los resultados.

7. 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 ejecútalo en background, usando los mismos parámetros que en el apartado 4 (3, 5 y 100). Observa el fichero maps del /proc y el tamaño del heap que alberga las regiones reservadas con sbrk.



PREGUNTA 40. 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? ¿Por qué?

8. 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).

Accesos incorrectos

El fichero mem2.c contiene un código que tiene un error en el uso de un puntero (diferente del error que había en el fichero mem3_previo.c visto en el estudio previo). Ejecuta el programa y comprueba que realmente no funciona.

9. Modifica el código (llámalo mem2_v2.c) para que cuando el proceso reciba el signal de tipo SIGSEGV se imprima en la salida estándar un mensaje indicando:
 - a. La dirección de la variable p.
 - b. El valor de la variable (dirección a la que apunta el puntero).
 - c. La dirección dónde finaliza el heap del proceso (esta dirección ya no es válida).

NOTA: Imprime sólo los 3 valores que os pedimos, si copias directamente el write del código puedes estar repitiendo el mismo error que provoca el signal.



PREGUNTA 41. ¿Qué error contiene el código del programa? ¿Por qué el programa no falla en las primeras iteraciones? Propón una alternativa a este código que evite que se genere la excepción, detectando, antes de que pase, que vamos a hacer un acceso fuera del espacio de direcciones.

Sesión 6: Análisis de rendimiento

Preparación previa

1. Objetivos

Los objetivos de esta sesión son: 1) entender las implicaciones de la gestión de procesos y de memoria en el rendimiento de un sistema multiusuario, y 2) medir el tiempo de ejecución de un programa según la carga del sistema y el cambio en las prioridades de los procesos.

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.
- Relacionar el rendimiento del sistema con el uso de memoria que hacen los procesos.
- Ser capaz de obtener información sobre los procesos en ejecución a través del `/proc`.
- 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, y poner especial atención a las opciones que aparecen en la columna “Opciones” de la siguiente tabla.

Para leer en el <code>man</code>	Descripción básica	Opciones
<code>nice</code>	Ejecuta un programa modificándole la prioridad de planificación	
<code>uptime</code>	Muestra cuánto tiempo lleva encendido 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>/proc/[pid]/stat</code>
<code>time</code>	Ejecuta un programa y mide el tiempo que tarda en ejecutarse	
<code>vmstat</code>	Muestra estadísticas sobre el uso de la memoria	<code>delay</code>
<code>top</code>	Muestra información sobre el sistema y los procesos en ejecución	comando interactivo: <code>f</code>

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). La prioridad se define mediante un número. Valores numéricos bajos de la prioridad implica que el proceso es más prioritario.

Los comandos **top**, **uptime** y **w** nos muestran información del sistema. Estos comandos son 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 tienen, 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 sobre 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 imagen anterior 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).

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.

1. Consulta el man del *proc* (man 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 totales del proceso (minor page faults + major page faults en Linux).
 - a. **NOTA:** En Linux, es posible que un fallo de página no implique un acceso a disco gracias a alguna de las optimizaciones de memoria. En este caso se considera un fallo de página "soft" o minor page fault, ya que es menos costoso que aquellos que implican ir al disco.
2. Ejecuta el comando **vmstat** de manera que la salida se muestre 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".
 - a. **NOTA:** Swap-in y swap-out es la cantidad de memoria que se ha traído/enviado desde/hacia disco.
3. Consulta la página del manual del comando **time** (man time). El comando **/usr/bin/time** sirve para ejecutar un programa midiendo su tiempo de ejecución. Este programa muestra por defecto 3 valores de tiempo: tiempo empleado en modo usuario (*user*), tiempo empleado en modo sistema (*system*) y tiempo total (*elapsed*). Durante la sesión usaremos el tiempo total de ejecución.
 - a. **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.
4. **Familiarízate con el entorno que usarás en la sesión:** bájate el fichero S6.tar.gz y descomprímelo (tar zxvf S6.tar.gz). En esta sesión te proporcionamos varios **scripts** para facilitar la ejecución de los programas. Un script no es más que un fichero de 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. En el paquete S6.tar.gz tienes el fichero fibonacci.c que contiene un programa de cálculo y que lo utilizaremos para ver el impacto que tiene la carga del sistema y las prioridades en la ejecución del programa. Utiliza el makefile para compilar el programa y ejecútalo con diferentes parámetros para ver cómo funciona.
6. **Utiliza el comando /usr/bin/time** para medir el tiempo que tarda en ejecutar Fibonacci si se le pasan los parámetros 10, 20, 30, 40 y 50 y apunta ese tiempo en el fichero previo.txt.
7. En el paquete tienes 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&
```

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.

Finalmente, también os damos el fichero RendimientoProcesos.ods que es una hoja de cálculo de OpenOffice que iréis rellenando durante la sesión.

- **PARA ENTREGAR: previo06.tar.gz**
#tar zcfv previo06.tar.gz previo.txt

4. Bibliografía

- Transparencias del Tema 2 y 3 (Procesos y Memoria) de SO-grau
- Capítulo 5, 8 y 9 (Scheduling, Main Memory y 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

- Contesta en un fichero de texto llamado entrega.txt todas las preguntas que aparecen en el documento, indicando para cada pregunta su número y tu 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 RendimientoProcesos.ods
```

RENDIMIENTO DE LA GESTIÓN DE PROCESOS

¿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 42. Apunta en el fichero entrega.txt el número de unidades de cálculo (cores o CPUS) 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 secuencialmente el siguiente comando (cuando acaba uno lanza el siguiente). Recuerda que como los procesos se ejecutan en background, hay que esperar que salgan los mensajes, no es suficiente con esperar a volver al prompt:
 - # ./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 43. Apunta el tiempo medio, máximo y mínimo para una instancia en la Tabla 1 de la hoja de cálculo adjunta.

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 cómo 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 44. Anota para cada experimento los tiempos medio, máximo y mínimo en la Tabla 1 de la hoja de cálculo adjunta.

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 con el usuario inicial (“alumne”), ejecuta 5 veces de forma secuencial el comando ./FIB 1.



PREGUNTA 45. ¿Cómo se ve afectado el tiempo de ejecución del proceso?



PREGUNTA 46. ¿Qué %CPU ha asignado el sistema a cada proceso?



PREGUNTA 47. La asignación, ¿ha sido por proceso o por usuario? ¿Crees que es posible que 1 usuario saturé el sistema con este criterio?

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 cómo 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 1`. 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 del trabajo previo.
- 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 48. ¿Cómo se ve afectado el tiempo de ejecución de FIB respecto al número de instancias de BAJA_PRIO_FIB?



PREGUNTA 49. ¿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`. Mide el tiempo medio de ejecución y comprueba que este tiempo es similar entre ellos. Ahora ejecuta el comando `FIB 5` varias veces y 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.

RENDIMIENTO DE LA GESTIÓN DE MEMORIA

El área de swap

Analiza el código que contiene el fichero `mem1.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, usaremos F para referirnos a este valor en el resto del documento). Consulta en el man la opción necesaria para que el comando free nos de la información en bytes.



PREGUNTA 50. ¿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 mem1 con los parámetros que se indican en la siguiente tabla (las ejecuciones tienen que ser secuenciales, es decir no lances el siguiente mem1 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 observa 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 totales que ha provocado su ejecución (minor page faults + 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 51. Rellena la siguiente tabla y añádela al fichero RendimientoProcesos.ods. ¿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. Para configurarlo, cuando estés ejecutando top aprieta la tecla "h" y te saldrá un menú con las diferentes opciones de top que te permitirán ordenar las columnas, elegir determinadas columnas, etc. En la otra Shell ejecuta el programa mem1 utilizando los siguientes parámetros (las ejecuciones tienen que ser secuenciales, es decir no lances el siguiente mem1 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 52. Rellena la siguiente tabla y añádela al fichero RendimientoProcesos.ods (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 7: 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 interrumpen 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
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 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 desempaquetalo.
- Contesta a las siguientes preguntas en el fichero “previo.txt”.

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:

1. 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?**
2. 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 ‘|’ en la shell). 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 (que ejecutan comando1 y comando2 respectivamente) y los conecte mediante una pipe sin nombre. Este pipeline 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.

Por ejemplo, en el directorio donde has descomprimido el fichero de la sesión, ejecuta el pipeline:

```
#ls -l |grep es
```

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

Enlazar los dos comandos mediante la pipe es similar a realizar la combinación siguiente:

```
# ls -l > salida_ls  
# grep es < salida_ls  
# rm salida_ls
```

3. Ejecuta un pipeline 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. Para ello utiliza los comandos `ps` y `grep` combinados con una pipe. **Anota el comando en el fichero "previo.txt".**

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.

Por ejemplo, si queremos leer un número de la consola (que es un dispositivo que solo acepta ASCII, tanto de entrada como de salida), primero habrá que leer los caracteres y luego convertirlo a número. Aquí tenéis un ejemplo suponiendo que el usuario escribe el número y luego aprieta Ctrl-D.

```
char buffer[64];  
int num,i=0;  
// Cuando el usuario apriete CtrlD el read devolverá 0  
// Como no conocemos cuantas cifras tiene el número, hay que leerlo con un bucle  
while (read(0,&buffer[i],1)>0) i++;  
buffer[i]='\0';  
num=atoi(buffer);
```

Por ejemplo, si queremos escribir el entero `10562` usando su representación interna en la máquina, estaremos escribiendo el número de bytes que ocupa un entero (4 bytes si usamos el tipo `int` en una máquina de 32 bits).

```
int num=10562;  
write(1,&num,sizeof(int)); // Si el canal 1 es la consola, veremos basura, ya que solo acepta ascii.
```

Para recuperarlo e interpretar correctamente su valor deberemos leer ese mismo número de bytes y guardarlo en una variable de tipo entero.


```
//Ejemplo de lectura (sin control de errores), suponiendo que en el fichero datos.txt hay enteros
haríamos...
int fd, num;
fd=open("datos.txt",O_RDONLY);
read(fd,&num,sizeof(int)); // En este caso el número de bytes a leer es fijo
```

Si por el contrario queremos escribir el mismo número como una cadena de caracteres (por ejemplo para mostrarlo por pantalla), el primer paso es convertirlo a una cadena de caracteres, en la que cada carácter representa un dígito. Esto implica que estaremos usando tantos bytes como dígitos tenga el número (en este ejemplo, 5 bytes).

```
char buff[64];
int num=10562;
sprintf(buff,"%d",num);
write(1,buff,strlen(buff));
```

4. El fichero es7.c contiene un programa que escribe en la salida estándar un entero usando la representación interna de la máquina. Compíllalo 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.

5. En el caso del dispositivo lógico **terminal (o consola)**, 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. Compíllalo y ejecútalo. **¿Cuántos bytes se escriben en cada caso? ¿Qué diferencias observas en lo que aparece en pantalla?**

Asociación de dispositivo lógico y dispositivo físico

6. 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 tienes 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: previo07.tar.gz

- i. `#tar zcfv previo07.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

- Contesta en un fichero de texto llamado entrega.txt todas las preguntas que aparecen en el documento, indicando para cada pregunta su número y tu 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 es1_v2.c es6_v2.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. El comando *strace* ejecuta el programa que se le pasa como parámetro y muestra información sobre la secuencia de llamadas a sistema que realiza. Con la opción *-e* se le especifica que muestre información sobre una única llamada a sistema y con la opción *-o* se le especifica que guarde esta información en un fichero.

Queremos comparar el número de llamadas a sistema *read* que ejecutan las dos versiones del programa (es1 y es1_v2). Para ello ejecuta los siguientes comandos:

```
# strace -o salida_v2 -e read ./es1_v2 < es2.c
```

```
# strace -o salida_v1 -e read ./es1 < es2.c
```



PREGUNTA 53. Apunta en el fichero “entrega.txt” los comandos que has utilizado en cada apartado. Además entrega el fichero “es1_v2.c” ¿Qué diferencias observas en las dos ejecuciones de *strace*? ¿Cuántas llamadas a sistema *read* ejecuta cada versión? ¿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 54. 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 55. 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 56. 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 y es1.c, y asegúrate 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 57. Escribe los valores que ha mostrado el script showCpuTime.sh para cada uno de los procesos y comenta las diferencias entre ellos en cuanto al consumo de cpu. ¿A qué se deben? Identifica las líneas de código de marcan la diferencia entre ellos

Modificamos la gestión de signals por parte del kernel: siginterrupt

En Linux, cuando un proceso está bloqueado en una entrada/salida y se recibe un signal, el proceso se desbloquea, se gestiona el signal y, si es necesario, se continúa con la operación de entrada/salida de forma transparente al usuario. Sin embargo, ese no es el funcionamiento estándar de UNIX, donde la entrada/salida no continúa automáticamente.

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 el signal. Utiliza la llamada a sistema siginterrupt para que la gestión sea como en UNIX. En este caso, el read devuelve error al recibir este signal. 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 (diferente de interrupción por signal), o read interrumpido por signal. Consulta en el man los diferentes valores de **errno** para estos casos.

Haz la siguiente secuencia de ejecuciones para comprobar el buen funcionamiento de tu código:

- a) Ejecuta el programa y pulsa return para desbloquear el *read*.
- b) A continuación ejecuta el programa pero mientras está esperando en el *read* envíale el signal SIGINT pulsando ^C.



PREGUNTA 58. Anota en el fichero entrega.txt el resultado de ambas ejecuciones. Entrega el código programado en el fichero es6_v2.c



PREGUNTA 59. ¿Qué pasaría si no añadiéramos siginterrupt al código? Repite las ejecuciones anteriores eliminando el siginterrupt y anota el resultado en el fichero entrega.txt.

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, substituye 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` (son dispositivos de tipo carácter).



PREGUNTA 60. ¿Estaba 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 `myDriver2` (**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 61. 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 62. Apunta la línea de comandos que has utilizado para crear el dispositivo.

5. Ejecuta el siguiente comando:

```
#!/es1 < myDevice
```



PREGUNTA 63. Anota en el fichero “`entrega.txt`” el resultado de la ejecución y explica el resultado obtenido.

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`. Ejecuta de nuevo el comando del apartado 5.



PREGUNTA 64. Anota el resultado de la ejecución. Explica el motivo de las diferencias observadas comparando la salida de este ejercicio con la de la apartado 5.

7. Elimina `myDevice` y ejecuta el siguiente script:

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

Este script desinstala `myDriver1` y `myDriver2`.

Sesión 8: 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. En este caso el objetivo es saber enviar/recibir datos con una comunicación previamente creada.

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 en el fichero "previo.txt".
- Escribe en el fichero "previo.txt" el fragmento de código que deberíamos añadir a un programa cualquiera 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 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 desempaquéalo (tar xzfv S8.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: previo8.tar.gz**
 - **#tar zcfv previo8.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: sesion8.tar.gz**
 - **#tar zcfv sesion8.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 a continuación 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 65. ¿Qué contiene el fichero al final de la ejecución? ¿Contiene la salida del padre y del hijo, o sólo la del padre? ¿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 66. ¿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 67. 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 (man 2 open) para ver cómo 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. No es necesario modificar el código que establece la conexión, solo introducir la comunicación de datos.

Sesión 9: 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

- Repasad 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_CREAT, 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 qué 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 inodo, 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 inodo tiene cada fichero? ¿Alguno de los links, sl_pr o hl_pr, tiene el mismo inodo 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 y después 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". El programa utiliza el fichero salida.txt a modo de ejemplo pero debe ser genérico, independientemente del tamaño del fichero de entrada siempre escribirá una X "entre el último y el penúltimo carácter".

- **PARA ENTREGAR: previo9.tar.gz**
 - `#tar zcfv previo9.tar.gz previo.txt crea_fichero.c insertax.c`

Bibliografía

- Transparencias del Tema 4 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: sesion9.tar.gz**
`#tar zcfv sesion9.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 68. Contesta a las siguientes preguntas en el fichero “entrega.txt”

- ¿Cual/es de los comandos anteriores han dado error al intentar ejecutarlos? Explica por qué.
- Explica 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. Utilizamos el fichero file a modo de ejemplo pero el programa tiene que ser genérico: para cualquier fichero de entrada, después de ejecutar el programa append.c el fichero tendrá el contenido original duplicado. **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 69. 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 70. 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”. Haz esta prueba a modo de ejemplo pero el programa ha de ser genérico: Dada una posición concreta del fichero se inserta el carácter “X”.



PREGUNTA 71. Entrega el fichero insertarx2.c.

Sesión 10: Concurrency and Parallelism

Preparación previa

Objetivos

- Entender las diferencias entre procesos y threads
- Entender las problemáticas asociadas al uso de memoria compartida

Habilidades

- Ser capaz de identificar las diferencias entre procesos y threads.
- Ser capaz de utilizar comandos y llamadas al sistema básicas para trabajar con threads.
- Ser capaz de identificar y solucionar condiciones de carrera.

Conocimientos previos

- Llamadas al sistema de gestión de procesos.

Guía para el trabajo previo

- 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 man	Descripción básica
<code>pthread_create</code>	crea un nuevo pthread
<code>pthread_join</code>	espera a que finalice el pthread indicado como parámetro
<code>pthread_mutex_init</code>	inicializa una variable para controlar el acceso a una región crítica
<code>pthread_mutex_lock</code>	controla la entrada a una región crítica
<code>pthread_mutex_unlock</code>	controla la salida de una región crítica
<code>pthread_exit</code>	finaliza el pthread actual

El fichero `fork2pthread.c` contiene un programa que crea un proceso hijo mediante la llamada a sistema `fork`. Copia este fichero sobre el fichero `fork2pthread_v2.c` y haz todas las modificaciones necesarias para que utilice pthreads en lugar de procesos tradicionales.

- **PARA ENTREGAR:** `previo10.tar.gz`
 - `#tar zcfv previo10.tar.gz fork2pthread_v2.c`

Bibliografía

- Transparencias del Tema 5 (Concurrencia y Paralelismo) 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 threads_racecondition_v2.c sumavector.c**



Diferencias entre pthreads y procesos

1. **Tiempo de creación.** Los ficheros createProcesses.c y createThreads.c son dos versiones diferentes del mismo código: el primero utiliza procesos tradicionales y el segundo pthreads.
 - a. Analiza el código de los dos programas y asegúrate de que entiendes el interfaz.
 - b. Compila y ejecuta ambos programas y anota en el fichero entrega.txt el tiempo de ejecución de cada uno.



PREGUNTA 72. ¿Qué diferencia observas en el tiempo de ejecución de cada uno de los programas? ¿A qué se debe?

2. **Compartición del espacio de direcciones.** Los ficheros fork_compartMem.c y thread_compartMem.c son dos versiones diferentes del mismo código.
 - a. Analiza el código de los dos programas y asegúrate de que entiendes su funcionalidad.
 - b. Compila y ejecuta ambos programas.



PREGUNTA 73. En el programa fork_compartMem.c, ¿qué valor ve el proceso hijo al principio de su ejecución en las variables a y pidPadre? ¿Y al final? ¿qué valores ve el proceso padre al final de su ejecución? ¿Cómo explicas este comportamiento?



PREGUNTA 74. En el programa thread_compartMem.c, ¿qué valor ve el pthread hijo al principio de su ejecución en las variables a y pidPadre? ¿Y al final? ¿qué valores ve el proceso padre al final de su ejecución? ¿Cómo explicas este comportamiento?

3. En el fichero thread_compartMem2.c hay un error de compilación.



PREGUNTA 75. ¿A qué se debe este error?

- a. Elimina la línea que provoca el error, recompíllalo y ejecuta con parámetro espera a 1.



PREGUNTA 76. En la línea marcada como “PUNTO 1” ¿en qué posición de memoria se está guardando el valor 1111? ¿Qué variable ocupa esa posición de memoria? ¿Qué mensajes muestran el padre y el pthread? ¿Sería posible que un pthread acceda a las variables locales de otro pthread? ¿En qué condiciones?

- b. Ejecuta de nuevo el programa ahora con parámetro espera a 0



PREGUNTA 77. ¿Qué mensajes muestran ahora el padre y el pthread? ¿A qué se debe esta diferencia en el comportamiento?

Condición de carrera

El fichero `threads_racecondition.c` contiene un programa que crea varios pthreads y cuya ejecución sufre una condición de carrera. El resultado esperado de este código es que cada pthread incremente en la misma cantidad la variable `Data`. El proceso principal imprime al final de la ejecución el valor de esa variable.

4. Analiza el código y asegúrate de entender su funcionamiento
5. Compila el código y ejecútalo 10 veces.



PREGUNTA 78. ¿El proceso principal imprime siempre el mismo valor? ¿A qué es debido?
PREGUNTA 79. Anota en el fichero `entrega.txt` qué líneas de código forman la región crítica que provoca la condición de carrera.

6. Copia el fichero sobre el fichero `threads_racecondition_v2.c` y modifícalo con el código necesario para proteger la región crítica que has detectado. Para ello utiliza las funciones que ofrece la librería de pthreads.

Paralelización

En el fichero `sumavector.c` tienes un programa que suma los elementos de un vector. Modifica este código para que esta suma se haga concurrentemente dividiendo el trabajo entre pthreads.

La estructura de datos de tipo `infojob` contiene los campos para describir el trabajo a realizar por cada pthread (la posición inicial del vector en la que empezar a sumar y la posición final) y para almacenar el resultado parcial de la suma al final de la ejecución del pthread. El proceso principal deberá inicializar estas estructuras, crear los pthreads pasando los parámetros adecuados, esperar a que los pthreads acaben y calcular la suma total a partir de los resultados parciales de los pthreads.