upna Universidad Pública de Navarra Nafarroako Unibertsitate Publikoa
Área de Lenguajes y Sistemas Informáticos
Departamento de Estadística, Informática y Matemáticas

Prácticas de Sistemas operativos

240304– Grado en Ingeniería Informática
Segundo curso, semestre de otoño

CURSO ACADÉMICO 2025-2026

José Javier Astrain Escola
Josu Irisarri Erviti

La asignatura 240304 - *Sistemas Operativos* de tercer semestre del Grado en Ingeniería Informática dispone de tres créditos ECTS asignados a prácticas. Dichas prácticas tendrán lugar en el horario y aula que se detalla a continuación:

Grupo teoría	Grupo prácticas	Profesor	Horario	Aula
G1	G1P1	Josu Irisarri Erviti	Jueves, 19:00 a 21:00	310
G1	G1P2	José Javier Astrain Escola	Jueves, 17:00 a 19:00	310
G2	G2P1	José Javier Astrain Escola	Lunes, 17:00 a 19:00	304
G2	G2P2	Josu Irisarri Erviti	Lunes, 19:00 a 21:00	304

Las prácticas se realizarán sobre PCs virtualizados equipados con sistema operativo Linux. Los equipos virtuales estarán disponibles en horario 24x7 para su uso por los alumnos. Para ello se contará tanto con la infraestructura VDI de las aulas de informática del Aulario, como con la infraestructura del Laboratorio Virtual (https://eim-laboratoriovirtual.unavarra.es/laboratorio/).

Durante este curso, se realizarán las siguientes prácticas:

Calendario de prácticas (G1P1 y G1P2, jueves)

Fecha	Sesión	Contenido
04/09/2025	1	Operaciones de E/S, manejo de ficheros.
11/09/2025	2	Procesos e hilos.
18/09/2025	3	Comunicación entre procesos: pipes.
25/09/2025	4	Comunicación entre procesos: señales.
02/10/2025	5	Construcción de una shell.
09/10/2025	6	
16/10/2025	7	**Primer examen parcial**
23/10/2025	8	Comunicación entre procesos: semáforos y memoria compartida.
30/10/2025	9	Comunicación entre procesos: colas de mensajes.
06/11/2025	10	**Construcción de un planificador de procesos mediante colas de múltiples niveles.**
13/11/2025	11	
20/11/2025	12	
27/11/2025	13	**Construcción de un pequeño sistema concurrente.**
04/11/2025	14	
11/12/2025	15	

Calendario de prácticas (G2P1 y G2P2 – lunes)

Fecha	Sesión	Contenido
08/09/2025	1	Operaciones de E/S, manejo de ficheros.
15/09/2025	2	Procesos e hilos.
22/09/2025	3	Comunicación entre procesos: pipes.
29/09/2025	4	Comunicación entre procesos: señales.
06/10/2025	5	Construcción de una shell.
13/10/2025	6	
20/10/2025	7	**Primer examen parcial.**
27/10/2025	8	Comunicación entre procesos: semáforos y memoria compartida.
03/11/2025	9	Comunicación entre procesos: colas de mensajes.
10/11/2025	10	**Construcción de un planificador de procesos mediante colas de múltiples niveles.**
17/11/2025	11	
24/11/2025	12	**Construcción de un pequeño sistema concurrente.**
01/12/2025	13	

Para superar el conjunto de la asignatura es imprescindible haber superado la parte práctica de la asignatura. La parte práctica de la asignatura se evaluará mediante la entrega y evaluación de prácticas, que tiene un peso del 40% de la calificación final de prácticas, y dos exámenes parciales a lo largo del semestre, que tienen un peso del 60% de la calificación final de prácticas. Para superar la parte práctica es necesario superar el examen práctico (NotaExamenParcial1 + NotaExamenParcial2 >=3/6). Cada alumno entregará para su evaluación dos prácticas (planificador y sistema concurrente) a lo largo del semestre. Las entregas de las prácticas se realizarán mediante la herramienta MiAulario. Las fechas de entrega de las prácticas serán las siguientes:

1) Construcción de un planificador de procesos: 28/11/2025.
2) Construcción de un pequeño sistema concurrente: 12/12/2025.

En la **convocatoria ordinaria**, la nota de la parte práctica (**50%** de la calificación final) será la suma de la nota del **trabajo de clase** (40% de la calificación de la parte práctica) y la obtenida en los **exámenes prácticos** (60% total de la calificación de la parte práctica, 20% primer parcial y 40% el segundo parcial). Para superar esta parte **es necesario aprobar la parte correspondiente al examen práctico** (NotaExamenParcial1 + NotaExamenParcial2 >=3/6). Si no se aprueba el examen práctico, la calificación de la

parte práctica será como máximo de 4'5 sobre 10 (suspenso). Aquellos alumnos que no superen el examen práctico de la asignatura mediante la evaluación continua, podrán realizar el examen de recuperación (50% de la calificación final) en el que ya no se tendrán en cuenta las calificaciones de las prácticas ni de los exámenes parciales.

Las prácticas que se entreguen para su evaluación deben contener el trabajo original y personal del alumno que las entregue. En el caso de detectarse plagio o incumplirse los requisitos citados previamente, la parte práctica quedará automáticamente suspendida (calificación de 0,0).

Tutorías

Las tutorías tendrán lugar en los lugares y horarios que a continuación se indican.

Profesor	Lugar	Horario
José Javier Astrain	ZOOM 460 648 5317 Despacho 10900 9029 [Encinas, planta sótano]	Lunes y jueves de 9:00 a 11:00h. Miércoles de 17:00 a 19:00h. https://www.unavarra.es/pdi?uid=2330&dato=tutorias
Josu Irisarri Erviti	Despacho 10900 9001 [Encinas, planta sótano]	Miércoles de 15:00 a 17:00h. Jueves de 10:00 a 12:00h. https://www.unavarra.es/pdi/?uid=812158&dato=tutorias

Práctica 0 : INTRODUCCIÓN AL INTÉRPRETE DE COMANDOS (SHELL)

1.- Introducción

El objetivo de esta práctica es introducir al alumno en el manejo de las herramientas que va a emplear a lo largo del curso para realizar las sesiones prácticas correspondientes a la asignatura *Sistemas Operativos*. Con tal motivo, se introduce el uso del sistema operativo Linux y del lenguaje de programación C.

2.- Conceptos básicos de Linux

En este apartado se listan algunos de los comandos que pueden resultar de interés a la hora de realizar las prácticas.

Utilidades de compresión: tar, gzip, gunzip, unzip, zip.

tar : Permite adjuntar y comprimir archivos. Se recomienda el uso de *tar czfv <destino> <origen>*.

gzip, zip : Permiten comprimir archivos. Se recomienda el uso de *gzip <destino> <origen>*.

gunzip, unzip : Permiten descomprimir archivos en formato ZIP. Se recomienda el uso de *gunzip <destino> <origen>* o *unzip <destino> <origen>*.

Manejo de procesos: ps, top, kill, killall.

ps : Lista los procesos que se están ejecutando actualmente en el sistema. Se recomienda el uso de la construcción *ps axu |more* para una mejor visualización de la información, aunque también existen otras alternativas de interés.

top : Lista los procesos con un mayor consumo de recursos que se están ejecutando actualmente en el sistema. Si se desea emplear una herramienta gráfica, se recomienda el uso de *gtop* o de *ktop*.

kill : Permite matar un proceso. Este comando es útil cuando un proceso ha quedado zombie o presenta un comportamiento no deseado.

Se recomienda el uso de *kill –9 <num_proceso>* en caso de querer eliminar de forma incondicional el proceso, o de *kill –HUP <num_proceso>* en caso de querer reiniciar dicho proceso.

killall : Es similar al anterior, pero se aplica a un conjunto de procesos.

Manejo de ficheros: ls, cd, cp, mkdir, mv, pwd, rm, rmdir, chmod, chown, touch, ln.

ls : Lista el contenido de directorios del sistema. Es lo que el usuario de MS-DOS conoce como DIR. Hay muchas opciones para este comando (*-a, -l, -d, -r,...*), que a su vez se pueden combinar de muchas formas. Dado que esto puede variar según el sistema Unix en el que se esté, es recomendado consultar su página de manual (*man ls*). Sin embargo, de todas éstas, las que podríamos considerar más comunes son:

-l (*long*): Formato de salida largo, con más información que utilizando un listado normal.

-a (*all*): Se muestran también archivos y directorios ocultos.

-R (*recursive*): Lista recursivamente los subdirectorios.

Por ejemplo:

```
MathLand:~/Practicas_MNEDP/Practica1$ ls -l
total 6
drwxr-xr-x  2 lorna    users      1024 Dic 15 04:30 TeX
drwxr-xr-x  2 lorna    users      1024 Dic 15 04:30 Datos_GNUPlot
drwxr-xr-x  2 lorna    users      1024 Dic 28 04:30 Datos_Programa
-rwxr-x---  1 lorna    users    230496 Nov 12 03:08 ascii
-rw-r-----  1 lorna    users      6246 Nov 12 03:07 ascii.cpp
-rw-r-----  1 lorna    users     12920 Nov 19 03:28 graficos.cpp
MathLand:~/Practicas_MNEDP/Practica1$
```

La primera columna indica las ternas de permisos de cada fichero o directorio, más un primer carácter especial que indica, entre otras cosas, si el archivo listado es un directorio, como vemos en el ejemplo, que tenemos una d en los directorios y simplemente - si es un archivo normal. El segundo campo hace referencia al número de enlaces del archivo, mientras que los dos siguientes indican el propietario y el grupo al que pertenece. El quinto campo corresponde al tamaño del fichero en bytes, y los siguientes dan la fecha y hora de la última modificación del archivo. Por fin, la última columna indica el nombre del fichero o directorio.

cd : Con este comando podremos cambiar de directorio de trabajo. La sintaxis básica es la siguiente:

cd [nombre_directorio]

Si usamos el *shell* bash, (que es el que se instala por defecto en los sistemas Linux), simplemente tecleando cd, volvemos a nuestro directorio HOME. Si le pasamos como parámetro una ruta (que puede ser absoluta, es decir, el nombre completo desde el directorio raíz, o relativa a dónde estamos) de un directorio, cambiaremos a ese directorio, siempre y cuando tengamos permiso para entrar. Es muy posible que en nuestro Linux, al hacer cd para entrar en algún directorio, no lo veamos reflejado en el *prompt* (como sucede en MS-DOS). Para poder ver en qué directorio estamos, podemos usar el comando pwd (**p**rint **w**orking **d**irectory).

Algunos ejemplos de cd:

```
MathLand:~# cd Programas/C
MathLand:Programas/C# cd
MathLand:~#
MathLand:~# cd /usr/bin
MathLand:/usr/bin#
```

cp : Con el comando cp copiamos un archivo (origen), en otro lugar del disco (puede ser un archivo o un directorio), indicado en destino.

Su sintaxis es *cp <origen><destino>*. Si el destino es un directorio, los archivos de origen serán copiados dentro de él. Hay que decir que los *shells* de Unix aceptan *wildcards* (comodines, los trataremos en otro capítulo), que son los caracteres especiales * y ?. Así, una orden como

```
MathLand:~# cp *.html Directorio_HTML
```

copiará todos los archivos que finalicen en .html en el directorio Directorio_HTML, si éste existe. Debemos recordar que en Unix el campo "." **no** separa el nombre de la extensión de un fichero, como en MS-DOS, sino que es simplemente un caracter más. No debemos olvidarlo, sobre todo con comandos como rm, si no queremos llevarnos sorpresas desagradables. Para copiar de forma recursiva (es decir, también subdirectorios) podemos usar la opción *-r* .

mkdir : Tal y como su nombre parece querer decir, crea un directorio. La sintaxis será simplemente *mkdir <nombre_directorio>*. Para crear un directorio tenemos que tener en cuenta los permisos del directorio en que nos encontremos trabajando pues, si no tenemos permiso de escritura, no podremos crear el directorio.

mv : Con este comando podemos renombrar un archivo o directorio, o mover un archivo de un directorio a otro. Dependiendo del uso que hagamos, su sintaxis variará. Por ejemplo:

mv <archivo/s> <directorio> moverá los archivos especificados a un directorio, mientras que con *mv <archivo1><archivo2>* renombrará el primer fichero, asignándole el nombre indicado en *<archivo2>*.

Veamos unos ejemplos:

MathLand:~# mv Hola_Mundo.c Copia_Hello.c
(Renombra el archivo Hola_Mundo.c como Copia_Hello.c)
MathLand:~# mv *.c Un_Directorio
(Mueve todos los archivos finalizados en .c al directorio Un_Directorio)

pwd : Imprime en pantalla la ruta completa del directorio de trabajo donde nos encontramos actualmente. No tiene opciones, y es un comando útil para saber en todo momento en qué punto del sistema de archivos de Unix nos encontramos.

rm : Elimina archivos o directorios. Sus tres opciones son *-r* (borrado recursivo, es decir, de subdirectorios), *-f* (no hace preguntas acerca de los modos de los archivos), y *-i*(interactivo, solicita confirmación antes de borrar cada archivo). Su sintaxis es muy sencilla: *rm [-r] [-f] [-i] <archivo>*. Hay que tener mucho cuidado con este comando cuando se usen comodines, sobre todo si no lo ejecutamos en modo interactivo.

rmdir : Borra directorios *únicamente* si están vacios. Su sintaxis básica será *rmdir <directorio>*. Si queremos borrar directorios que no estén vacíos, hemos de utilizar el comando *rm -r <directorio>*.

chmod : Con este comando, cambiamos los permisos de acceso del archivo o del directorio que le especifiquemos como argumento. Podemos ejecutar de dos formas básicas este comando: la primera es *chmod <modo> <archivo>*, siendo *modo* un valor numérico de tres cifras octales que describe los permisos para el archivo. Cada una de estas cifras codifica los permisos para el dueño del archivo, para los usuarios que pertenecen al mismo grupo que el dueño, y para el resto de los usuarios, en este orden. Veamos un ejemplo: supongamos que queremos cambiar los permisos de un archivo llamado Algo.txt (por poner algún nombre), de manera que nosotros podamos leerlo y modificarlo, pero no ejecutarlo, los usuarios de nuestro grupo puedan leerlo y el resto de usuarios no pueda ni leer el fichero. La terna de permisos que describe esta situación es:

rw- r-- ---

Si escribimos un 1 en el lugar del permiso activado y un 0 en el del permiso desactivado, tenemos:

rw- r-- ---
110 100 000

Y si ahora codificamos los tripletes en binario a octal, tenemos que el permiso será 640, por tanto, tendremos que escribir el comando

```
chmod 640 Algo.txt
```

La segunda forma es algo más complicada:

```
chmod <who> +|- <permiso> <archivo>
```

Indicaremos, en el parámetro *who*, la identidad del usuario/s cuyos permisos queremos modificar *(u-user, g-group, o-others)*; a continuación irá un + o un -, dependiendo de si reseteamos el bit correspondiente o lo activamos, y en permiso debemos colocar el permiso a modificar *(r-read, w-write, x-exec)*.

Veamos unos ejemplos de ambos tipos de cambio de permisos:

```
MathLand:~# chmod 123 Ejercicio2.txt
MathLand:~# chmod 765 Practica.tex
MathLand:~# chmod g+r Un_programa.o
MathLand:~# chmod o+rx Otro_programa.p
```

chown : Con este comando, cambiamos los propietarios del archivo o directorio que le especifiquemos como argumento. Se puede cambiar no sólo el propietario del archivo, sino también el grupo al que pertenece.

chown <usuario>.<grupo> <archivo>

Un ejemplo:

```
MathLand:~# chown taxista.transportista coche.sp
```

touch : Actualiza la fecha de modificación de un archivo, o crea un archivo vacío si el fichero pasado como parámetro no existe. Con la opción *-c* no crea este archivo vacío. Su sintaxis es *touch* [*-c*] <*archivo*>.

ln : Enlaza dos archivos. Mediante *ln –s <destino> <origen>* se establece un enlace simbólico entre el archivo de destino y el de origen, de modo que el contenido al que apuntarán ambos ficheros será el mismo.

Tratamiento de ficheros: cat, file, more, less, last, tail, head.

cat : Concatena y muestra el contenido de archivos. La salida de la orden cat será por defecto la pantalla. Veamos un ejemplo:

```
MathLand:~# cat Hola_Mundo.c
#include <stdio.h>
int main(void){
  printf("Con todos ustedes, un clásico.\n\nHola, mundo!!\n");
  return 0;
  }
MathLand:~#
```

Otro uso de cat, mucho más útil, es unir varios archivos de texto en uno solo, redireccionando la salida a un fichero. Más adelante se hablará sobre redirección, pero por el momento vamos a ver un ejemplo que aclarará esta idea:

MathLand:~# cat *.c << Varios_Programas_C

Esto copiará todos los archivos terminados en .c al archivo Varios_Programas_C.

Se pueden concatenar varios ficheros, de tal manera que se añaden los contenidos de uno tras el anterior.

cat otro_fichero > mifichero

file : proporcionainformación sobre el tipo de un archivo especificado como argumento. Si lo usamos con la siguiente opción, puede sernos bastante útil: *-f <farchivo>* nos indica que en *farchivo* están los nombres de los ficheros a examinar. La sintaxis del comando es: *file* [*-f <farchivo>*] archivo. Por ejemplo:

```
MathLand:~# file Que_sera_esto
MathLand:~# Que_sera_esto: ASCII text
```

quiere decirnos que el fichero es, casi seguramente, un fichero ASCII. Este comando no es infalible, pues hay ficheros cuyo tipo no sabe reconocer, pero es útil de cara a saber, como mínimo, si es texto plano o no lo es (y si no lo es, no usaremos un editor de texto para abrirlo...).

more : Visualiza un archivo pantalla a pantalla, no de forma continua, como hace cat. Es como cat, pero con pausas, lo que nos permite leer más tranquilamente un archivo. Al final de cada pantalla nos aparecerá un mensaje indicando --More--. Si en ese momento pulsamos *RETURN*, veremos una o más líneas del archivo; si pulsamos la barra espaciadora, veremos la siguiente pantalla, si pulsamos *b* la anterior, y si pulsamos *q* saldremos de *more*. Su sintaxis es *more <archivo>*.

passwd : El comando *passwd* se utiliza para cambiar la clave de acceso al sistema. Cuando ejecutemos este comando, tendremos que escribir la clave dos veces, y en ambas ha de coincidir. Esto nos evita que se nos asigne una clave no deseada (y, lo peor, no recordada) por culpa de un error al mecanografiarla.

wc (word count) : Cuenta el número de líneas, palabras y caracteres de un fichero.

sort : Ordena las líneas de un fichero.

more - Paginar un fichero

tail : Saca las últimas líneas de un fichero (10 por defecto) .

head : Saca las primeras líneas de un fichero (10 por defecto) .

grep (get regular expression) : Saca sólo las líneas de un fichero que contienen un patrón.

find : Busca un fichero.

Comandos de red: telnet, ftp, ssh, finger, dnslookup, ping, dnsdomainname…

Por las especiales condiciones en las que se ha instalado la red del laboratorio, estos comandos serán explicados en el laboratorio.

Otros comandos: who, whoami, last, man, diff, wich…

Se trata de comandos que pueden consultarse con la orden *man* <*nombre del comando*>.

3.- Conceptos básicos de C

A la hora de programar con C, se recomienda el uso de un buen editor de textos, como puede ser el caso de *nedit, kedit, ged*it o similares. Además, es preciso el uso del compilador gcc y si procede, de un buen depurador de código o *debugger* como ddd.

La compilación de un archivo C se realiza mediante gcc –o <nombre_destino> <fichero.c>, tal y como se detalla en la documentación sobre C que se adjunta en esta memoria.

Se recomienda el uso de *Makefile* para automatizar la tarea de compilación.

4.- Ejercicios prácticos

1. Escribir un fichero de bienvenida personalizado. Es decir, cuando se ejecute debe pedir el nombre del usuario y presentar en pantalla un mensaje de bienvenida dedicado al usuario.

2. Escribir un menú con cuatro opciones para elegir las cuatro operaciones básicas: suma, resta, multiplicación y división. Utilizar la sentencia *switch* para ello. Con cada opción se debe presentar un mensaje donde se indique la operación elegida.

3. Realizar un programa que imprima en pantalla en orden decreciente tres números enteros introducidos por teclado.

4. Un centro númerico es un número que separa una lista de números enteros (comenzando desde el 1) en dos grupos de números cuyas sumas son iguales. El primer centro numérico es el 6, que separa la lista (1, 2, 3, 4, 5) y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, y separa una lista desde 1 a 49 en dos (1 a 34) y (36 a 49). La suma de los números de cada grupo es 595. Escribir un programa que halle los centros numéricos entre 1 y N, donde N es un número, menor de 7000, que se introducirá por teclado.

Utilizar para ello una función que compruebe si un número es centro numérico o no.
Una función que devuelva la suma de los números de cada grupo.

Práctica 1: OPERACIONES DE E/S, MANEJO DE FICHEROS

1.- Introducción

El lenguaje C provee varias funciones para la edición de ficheros. Estas funciones están definidas en la librería stdio.h, y por lo general empiezan con la letra f, de *file*. Además, C provee el tipo FILE, que se usará como apuntador a la información del fichero. La secuencia de tareas que se ha de seguir a la hora de acceder a un fichero es la siguiente:

- Crear un apuntador del tipo FILE * .
- Abrir el archivo utilizando la función fopen y asignándole el resultado de la llamada a nuestro apuntador.
- Hacer las diversas operaciones (lectura, escritura, etc).
- Cerrar el archivo utilizando la función fclose.

2.- Apertura y cierre de ficheros

La función ***fopen*** sirve para abrir y crear ficheros en un sistema de ficheros. El prototipo correspondiente de ***fopen*** es:

```
FILE *fopen(const char *path, const char *mode);
```

El argumento *path* hace referencia al nombre del fichero que se desea abrir, incluyendo la ruta completa dentro del sistema de ficheros.

El argumento *mode* hace referencia a los permisos con los que se dea abrir el fichero.

- "r": abrir un archivo para lectura, el fichero debe existir.
- "w": abrir un archivo para escritura, se crea si no existe o se sobreescribe si existe.
- "a": abrir un archivo para escritura al final del contenido, si no existe se crea.
- "r+": abrir un archivo para lectura y escritura, el fichero debe existir.
- "w+": crear un archivo para lectura y escritura, se crea si no existe o se sobreescribe si existe.
- "a+": abrir/crear un archivo para lectura y escritura al final del contenido.

El valor devuelto es el puntero al fichero (File *) deseado.

La función ***fclose*** sirve para cerrar ficheros en un sistema de ficheros. El prototipo correspondiente de ***fclose*** es:

```
FILE *fclose(FILE *fp);
```

Esta función sirve para poder cerrar un fichero que se ha abierto previamente. El prototipo de esta función ***fclose*** es:

```
int fclose (FILE *stream);
```

Un valor de retorno cero indica que el fichero ha sido correctamente cerrado, si ha habido algún error, el valor de retorno es la constante EOF.

A continuación, se ilustra el proceso de apertura y cierre de un proceso.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
        FILE *fp;
        fp = fopen ( "nombreFichero", "r" );
        fclose ( fp );

        return 0;
}
```

3.- Fin del fichero

La función ***feof*** sirve para determinar si el cursor dentro del archivo ha alcanzado o no el final (***e**nd **o**f **f**ile, EOF*). La función ***feof*** devuelve cero (falso) si no se alcanza el final del fichero, de lo contrario devuelve un valor distinto de cero (verdadero).

El prototipo correspondiente de feof es:

```
int feof(FILE *fichero);
```

4.- Función de vuelta al origen

La función ***rewind*** permite situar el cursor de lectura/escritura al principio del archivo.

El prototipo correspondiente de rewind es:

```
void rewind(FILE *fichero);
```

5.- Lectura de un fichero

Existen varias formas de trabajar con ficheros y diferentes funciones para hacerlo. Las funciones que podríamos usar para leer un archivo son:

- `char fgetc(FILE *archivo)`
- `char *fgets(char *buffer, int tamano, FILE *archivo)`
- `size_t fread(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);`
- `int fscanf(FILE *fichero, const char *formato, argumento, ...);`

5.1.- fgetc

Esta función lee un caracter del archivo señalado con el puntero **archivo*. En caso de que la lectura sea exitosa devuelve el caracter leído y en caso de que no lo sea o de encontrar el final del archivo devuelve EOF.

El prototipo correspondiente de fgetc es:

```
char fgetc(FILE *archivo);
```

Esta función se usa generalmente para recorrer archivos de texto.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        FILE *archivo;
        char caracter;

        archivo = fopen("prueba.txt","r");

        if (archivo == NULL){

                printf("\nError de apertura del archivo. \n\n");
        } else {

            printf("\nEl contenido del archivo de prueba es \n\n");

            while (feof(archivo) == 0) {
                caracter = fgetc(archivo);
                printf("%c",caracter);
            }
        }

        return 0;
}
```

5.2.- fgets

Esta función permite leer cadenas de caracteres. Leerá hasta *tamaño* caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído.

El prototipo correspondiente de **fgets** es:

```
char *fgets(char *buffer, int tamaño, FILE *archivo);
```

El primer parámetro, *buffer*, es el puntero a la zona de memoria en la que se almacenará el contenido leído como un vector de caracteres. El segundo parámetro es *tamaño* que es el limite de caracteres a leer para la funcion fgets. Y por último, el puntero del archivo indica de qué fichero se debe leer. Esta función permite leer una línea en una sola operación.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        FILE *archivo;

        char caracteres[100];

        archivo = fopen("prueba.txt","r");

        if (archivo == NULL)exit(1);

        printf("\nEl contenido del archivo de prueba es \n\n");
        while (feof(archivo) == 0) {
                fgets(caracteres,100,archivo);
                printf("%s",caracteres);
        }
        system("pause");
        return 0;
}
```

5.3.- fread

La función ***fread*** lee *nmiemb* elementos de datos, cada uno de *tam* bytes de largo, del flujo de datos apuntado por *flujo*, almacenándolos en el sitio apuntado por *ptr*.

```
size_t fread(void *ptr, size_t tam, size_t nmiemb, FILE *flujo);
```

fread devuelve el número de elementos (no de caracteres) leídos correctamente. Si ocurre un error o se llega al final del fichero, devuelve un número negativo (o cero). ***fread*** no distingue entre fin-de-fichero y error, así que quien llame a esta función debe emplear feof(3) y ferror(3) para determinar qué ha ocurrido.

5.4.- fscanf

La función ***fscanf*** funciona igual que *scanf* en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

El prototipo correspondiente de ***fscanf*** es:

```
int fscanf(FILE *fichero, const char *formato, argumento, ...);
```

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc, char **argv ) {
        FILE *fp;

        char buffer[100];

        fp = fopen ( "fichero.txt", "r" );

        fscanf(fp, "%s" ,buffer);
        printf("%s\n",buffer);

        fclose (fp);

        return 1;
}
```

6.- Escritura de un fichero

Existen varias funciones que permiten escribir datos en ficheros. Las funciones que podríamos usar para escribir dentro de un archivo son:

- int fputc(int caracter, FILE *archivo)
- int fputs(const char *buffer, FILE *archivo)
- size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);
- int fprintf(FILE *archivo, const char *formato, argumento, ...);

6.1.- fputc

Esta función escribe un carácter en el archivo indicado con el puntero ***archivo**. El valor devuelto es el carácter escrito, si la operación fue completada con éxito, y en caso contrario **EOF**.

El prototipo correspondiente de **fputc** es:

```
int fputc(int carácter, FILE *archivo);
```

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc, char **argv ) {
        FILE *fp;

        char caracter;

        fp = fopen("fichero.txt", "r+");

        printf("\nIntroduce un texto al fichero: ");

        while((caracter = getchar()) != '\n')
         printf("%c\n", fputc(caracter, fp));

        fclose (fp);
```

```
        return 0;
}
```

6.2.- fputs

La función **fputs** escribe una cadena en un fichero. No se añade el carácter de retorno de línea ni el carácter nulo final. El valor de retorno es un *número no negativo* o **EOF** en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura **FILE** del fichero donde se realizará la escritura.

El prototipo correspondiente de **fputs** es:

```
int fputs(const char *buffer, FILE *archivo);
```

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc, char **argv ) {
        FILE *fp;

        char cadena[] = "Mostrando el uso de fputs en un fichero.\n";

        fp = fopen("fichero.txt", "r+");

        fputs(cadena, fp);

        fclose(fp);

        return 0;
}
```

6.3.- fwrite

Esta función está pensada para trabajar con registros de longitud constante de forma análoga a **fread**. Es capaz de escribir hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada. El valor de retorno es el número de registros escritos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria de donde se obtendrán los datos a escribir, el tamaño de cada registro, el número de registros a escribir y un puntero a la estructura **FILE** del fichero al que se hará la escritura.

El prototipo correspondiente de **fwrite** es:

```
size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE
                                *archivo);
```

```
#include <stdio.h>
#include <stdlib.h>

void menu();
```

```
void CrearFichero(FILE *Fichero);
void InsertarDatos(FILE *Fichero);
void VerDatos(FILE *Fichero);

struct sRegistro {
   char Nombre[25];
   int Edad;
   float Sueldo;
} registro;

int main() {
        int opcion;
        int exit = 0;
        FILE *fichero;

        while (!exit) {
                menu();
                printf("\nOpcion: ");
                scanf("%d", &opcion);

                switch(opcion) {
                        case 1:
                                CrearFichero(fichero);
                        break;
                        case 2:
                                InsertarDatos(fichero);
                        break;
                        case 3:
                                VerDatos(fichero);
                        break;
                        case 4:
                                exit = 1;
                        break;
                        default:
                                printf("\nopcion no valida");
                }
        }

        return 0;
}

void menu() {
        printf("\nMenu:");
        printf("\n\t1. Crear fichero");
        printf("\n\t2. Insertar datos");
        printf("\n\t3. Ver datos");
        printf("\n\t4. Salir");
}

void CrearFichero(FILE *Fichero) {
        Fichero = fopen("fichero", "r");

        if(!Fichero) {
                Fichero = fopen("fichero", "w");
                printf("\nArchivo creado!");
        } else printf("\nEl fichero ya existe!");

        fclose(Fichero);
        return;
}

void InsertarDatos(FILE *Fichero) {
```

```
        Fichero = fopen("fichero", "a+");

        if(Fichero == NULL) {
                printf("\nFichero no existe! \nPor favor creelo");
                return;
        }

        printf("\nDigita el nombre: ");
        scanf("%s", registro.Nombre);

        printf("\nDigita la edad: ");
        scanf("%d", &registro.Edad);

        printf("\nDigita el sueldo: ");
        scanf("%f", &registro.Sueldo);

        fwrite(&registro, sizeof(struct sRegistro), 1, Fichero);

        fclose(Fichero);
        return;
}

void VerDatos(FILE *Fichero) {
        int numero = 1;

        Fichero = fopen("fichero", "r");

        if(Fichero == NULL) {
                printf("\nFichero no existe! \nPor favor creelo");
                return;
        }
        fread(&registro, sizeof(struct sRegistro), 1, Fichero);
        printf("\nNumero \tNombre \tEdad \tSueldo");

        while(!feof(Fichero)) {
                printf("\n%d \t%s \t%d \t%.2f",numero,
registro.Nombre, registro.Edad, registro.Sueldo);
                fread(&registro, sizeof(struct sRegistro), 1,
Fichero);
                numero++;
        }

        fclose(Fichero);
        return;
}
```

6.4.- fprintf

La función **fprintf** funciona igual que printf en cuanto a parámetros, pero la salida se dirige a un archivo en lugar de a la pantalla.

El prototipo correspondiente de **fprintf** es:

```
int fprintf(FILE *archivo, const char *formato, argumento, ...);
```

```
#include <stdio.h>
#include <stdlib.h>


int main ( int argc, char **argv ) {
```

```
        FILE *fp;

        char buffer[100] = "Esto es un texto dentro del fichero.";

        fp = fopen ( "fichero.txt", "r+" );

        fprintf(fp, buffer);
        fprintf(fp, "%s\n", "\nEsto es otro texto dentro del
fichero.");

        fclose (fp);

        return 0;
}
```

7.- Otra forma de hacer lo mismo

En lugar de trabajar con punteros a FILE, se puede trabajar a más bajo nivel con descriptores de ficheros. Esta forma de trabajar es compatible con tuberías, sockets… y resulta de especial interés.

7.1.- open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *camino, int flags);
int open(const char *camino, int flags, mode_t modo);
```

La llamada al sistema **open()** se utiliza para convertir una ruta en un descriptor de fichero (un pequeño entero no negativo que se utiliza en las operaciones de E/S posteriores como en **read**, **write**, etc). Cuando la llamada tiene éxito, el descriptor de fichero devuelto será el descriptor de fichero más pequeño no abierto actualmente para el proceso.

El parámetro *flags* es uno de **O_RDONLY**, **O_WRONLY** u **O_RDWR** que, respectivamente, piden que la apertura del fichero sea solamente para lectura, solamente para escritura, o para lectura y escritura, combinándose mediante el operador de bits OR (|), con cero o más macros entre las que destaca **O_CREAT (s**i el fichero no existe, será creado).

7.2.- close

Permite cerrar el fichero cuyo descriptor de fichero se pasa como argumento.

```
int close(int fildes);
```

7.3.- read

read() intenta leer hasta *nbytes* bytes del fichero cuyo descriptor de fichero es *fd* y guardarlos en la zona de memoria que empieza en *buf*.

Si *nbytes* es cero, **read()** devuelve cero y no tiene otro efecto. Si *nbytes* es mayor que SSIZE_MAX, el resultado es indefinido.

ssize_t read(int *fd*, **void *** *buf*, **size_t** *nbytes*);

7.4.- write

write escribe a un descriptor de fichero. **write** escribe hasta *num* bytes en el fichero referenciado por el descriptor de fichero *fd* desde el búfer que comienza en *buf.*

ssize_t write(int *fd*, **const void** **buf*, **size_t** *num*);

```
int main(){

int salida, fd;
char buffer[50];
ssize_t tamano_mensaje;

char mensaje[] = "hola\n";
tamano_mensaje = strlen(mensaje);

char ruta[]="/tmp/prueba";
fd = open(ruta, O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);

salida = write (fd,mensaje,tamano_mensaje);
close (fd);

fd = open(ruta, O_RDONLY);
salida = read (fd,buffer,salida);
salida = write (1,buffer,salida);

close (fd);
return 0;
}
```

Cuestiones a resolver

1.- Construya un programa que abra un fichero cuyo nombre se pase como argumento y escriba en él "Hola mundo" tras esperar 5 segundos.
2.- Construya un programa que abra un fichero cuyo nombre se pase como argumento y lea el contenido del fichero y lo imprima por pantalla.
3.- Construya un programa que cambie las vocales en minúscula a mayúsculas.

Práctica 2: PROCESOS E HILOS

1.- Introducción

Cada proceso tiene un identificador único en Linux que se denomina *process id* o *pid*.

Al proceso que solicita la creación de un nuevo proceso se le denomina **proceso padre**, y al proceso resultante se le denomina **proceso hijo**.

Cuando termina la ejecución de un proceso, éste debe finalizar ordenada y correctamente la ejecución de sus procesos hijo, pues de lo contrario dará lugar a procesos *zombies*. Algo similar ocurre con la terminación de los hilos de un proceso.

Recuerde que los comados ps, top, kill y killall le pueden ser de utilidad en el desarrollo de esta práctica.

2.- Identificadores de proceso

Las llamadas al sistema que se emplean para obtener el identificador de proceso o pid son: **getpid()** que ofrece el pid del proceso actual; y **getppid()** que ofrece el pid del proceso padre.

Si se desea obtener el identificador de usuario o uid, se emplea la llamada al sistema **getuid().**

```
#include <sys/types.h>
#include <unistd.h>

pid_t pid, ppid;         /*   Declaración de variables*/
uid_t  uid;

pid = getpid();         /* Asignación de los valores devueltos por las
                                        llamadas al sistema*/
pid = getppid();
uid = getuid();
```

Nota: pid_t y uid_t son un entero largo con el ID de proceso o usuario, respectivamente.

Ejemplo de uso:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
printf("PID del proceso: %ld.\n", (long)getpid());
printf("PID del proceso padre: %ld.\n", (long)getppid());
printf("UID del usuario propietario: %ld.\n", (long)getuid());

return (0);
}
```

3.- Creación de procesos

La creación de procesos se lleva a cabo con la ayuda de la llamada al sistema *fork*. El nuevo proceso creado recibe una copia exacta del espacio de direcciones del padre.

Los dos procesos (padre e hijo) continúan su ejecución en la instrucción siguiente al fork. La llamada fork crea procesos nuevos haciendo una copia de la imagen en la memoria del padre. El hijo hereda la mayor los atributos del padre, incluyendo el entorno y los privilegios. El hijo también hereda alguno delos recursos del padre, tales como los archivos y dispositivos abiertos.

El valor devuelto por la llamada la sistema fork permite diferenciar el proceso padre del hijo, ya que fork devuelve el valor 0 al hijo y el ID del proceso hijo al padre. Si se produce algún error durante la creación del nuevo proceso, la función devuelve el valor -1.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

El siguiente código genera un árbol de procesos con el padre como nodo raíz y cinco ramas que cuelgan de él.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main () {
  int i, pid;

  padre = 1;

  for (i=0; i < 5; i++)
       pid = fork();

       if (pid == -1) {
          printf("Error en la creación del proceso\n");
          exit (-1);
       }
       if (pid == 0) {                          /* Proceso hijo */

          printf("Este es el proceso hijo %d, cuyo padre es %ld\n", i,
                 (long)getppid());
          exit(999);
       } else {                                 /* Proceso padre */
          printf("Este es el proceso padre con ID %ld\n",
                 (long)getpid());
        }
     }

  return (0);
}
```

3.1. Herencia de descriptores

Cuando fork crea un proceso hijo, éste hereda la mayor parte del entorno y contexto del padre, que incluye el estado de las señales, los parámetros de la planificación de procesos y la tabla de descriptores de archivo.

Hay que tener cuidado con la herencia de los descriptores de archivos porque los procesos padre e hijo comparten el mismo desplazamiento de archivo para los archivos que fueron abiertos por el padre antes del fork.

4. Las llamadas al sistema wait, waitpid y exit

La llamada al sistema wait, no confundir con el comando wait, permite que un proceso espere a que termine la ejecución de uno de sus hijos. Esta espera puede ser por un proceso concreto (waitpid) o por uno cualquiera de sus hijos (wait). En el primer caso se pasa como argumento a la llamada waitpid el pid del proceso cuya finalización se desea esperar, mientras que si basta con la finalización de uno solo de los procesos hijos, se empleará wait.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Si el estado no es NULL, tanto wait () como waitpid () almacenan la información del estado en el puntero indicado como argumento (status). Este entero puede inspeccionarse con las siguientes macros:

1. WIFEXITED(status)
   Devuelve verdadero si el hijo terminó con normalidad (exit o final de la función main).

2. WEXITSTATUS(status)
   Devuelve el estado de salida del hijo. Son los 8 bits menos significativos devueltos por el hijo mediante la llamada al sistema exit o dentro del return final de la función main. Esta macro sólo debe emplearse si WIFEXITED devuelve cierto.

3. WIFSIGNALED(status)
   Devuelve true si el proceso hijo finalizó por una señal.

4. WTERMSIG(status)
   Devuelve el número de la señal que causó la finalización del proceso hijo. Esta macro sólo debe emplearse si WIFSIGNALED devuelve cierto.

5. WIFSTOPPED(status)
   Devuelve true si el proceso hijo fue detenido por una señal.

6. WSTOPSIG(status)
   Devuelve el identificador de la señal que provocó la parada del proceso hijo. Esta macro sólo debe emplearse si WIFSTOPPED devuelve cierto.

7. WIFCONTINUED(status)
   Devuelve true si el proceso hijo se reanudó mediante la entrega de SIGINT.

En el caso de que el hijo haya terminado cuando se solicita la espera (o simplemente no existe ningún proceso hijo) la llamada *wait* devuelve el control de inmediato.

Cuando un proceso hijo termina, la llamada wait devuelve el ID de dicho hijo al padre. En caso contrario devuelve -1.

wait(&status) es equivalente a ***waitpid(-1, &status, 0)***.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main (void) {
  pid_t childpid;
  int status=0, result;

  if ((childpid = fork()) == -1) {
          perror("Error en llamada a fork\n");
          exit(1);
  } else if (childpid == 0) {
          result = getpid() < getppid() ;
          fprintf(stdout, "Soy el proceso hijo (%ld) y voy a devolver a
                  mi padre (%ld) el valor %d despues de esperar 2
                  segundos\n", (long)getpid(), (long)getppid(),result);
          sleep(2);
          exit (result);
  } else {
          while( childpid != wait(&status));
          fprintf(stdout, "Soy el proceso padre (%ld) y mi hijo (%ld)
            me ha devuelto STATUS=%d\n",(long)getpid(),
            (long)childpid,status);
  }
  return (0);
}
```

La llamada al sistema exit finaliza la ejecución de un proceso. Permite asignar el valor que se pasa como argumento a la variable de entorno correspondiente para poder averiguar cuál ha sido la causa de la terminación del proceso.

```
#include <stdlib.h>
void exit (int status);
```

Si el proceso padre del que ejecuta la llamada a exit está ejecutando una llamada a wait, se le notifica la finalización de su proceso hijo y se le envía el byte menos significativo de su *status*. Con esta información el proceso padre puede saber en qué condiciones ha terminado el proceso hijo.

Si el proceso padre no está ejecutando una llamada a wait, el proceso hijo se transforma en un proceso *zombie*.

5. La llamada exec

La llamada fork crea una copia del proceso llamante. Muchas veces es necesario que el proceso hijo ejecute un código totalmente distinto al del padre. La familia de llamadas al sistema *exec* permiten la ejecución de comandos del sistema o la ejecución de programas de usuario. Tienen la particularidad de finalizar el proceso llamante cuando concluye la ejecución del citado comando o programa.

Las llamadas exec reciben como parámetros de entrada el comando a ejecutar y todos sus argumentos. Dado que el número de argumentos puede ser variable, el último parámetro de entrada es NULL, para indicar que la cadena de argumentos ha finalizado.

Las llamadas execv (execv, execvp y execve) pasan la lista de argumentos en un array de punteros a char (**char) y son útiles cuando no se conoce el número de argumentos en tiempo de compilación. Las llamadas execl (execl, execle) pasan la lista de argumentos en los argumentos de la función, indicando el último mediante el puntero 0 (NULL).

```
#include <unistd.h>
int execl (const char *path, const char *arg0, ..., const char *argn,
  char * /*NULL*/);
int execv (const char *path, char *const argv[]);
int execle (const char *path, const char *arg0, ..., const char *argn,
  char * /*NULL*/, char *const envp[]);
int execve (const char *path, char *const argv[], char *const envp[]);
int execlp (const char *file, const char *arg0, ..., const char *argn,
  char * /*NULL*/);
int execvp (const char *file, char *const argv[]);
```

Las seis variaciones de la llamada exec se distinguen por la forma en que se le pasan los argumentos.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main() {
  int status;
  printf ("Lista de procesos\n");
  if (execl ("ps", "ps", "-aux", NULL) < 0) {
          fprintf(stderr, "Error en exec %d\n", errno);
          exit(1);
```

```
  }
  printf ("Fin de la lista de procesos\n");

  exit(0);
}
```

6. Terminación de procesos

Cuando termina un proceso (correcta o incorrectamente), el sistema operativo recupera los recursos asignados al proceso terminado, actualiza las estadísticas apropiadas y notifica a los demás procesos la terminación.

Cuando un proceso termina, sus hijos huérfanos son adoptados por el proceso init, cuyo ID el 1. Si un proceso padre no espera a que sus hijos terminen su ejecución, entonces éstos se convierten en procesos *zombies* y tiene que ser el proceso init el que los libere (lo hace de manera periódica).

7. Hilos

```
#include <stdio.h>
#include <pthread.h>
#include <errno.h>

main (){

  pthread_t tid;
  int misargs[2], tipohilo;
  pthread_attr_t atributos;

  void *mifuncion(void *arg);
       //Creo mi propia estructura de atributos para luego modificarla
  if (pthread_attr_init(&atributos) != 0) {
          perror("En creación de estructura de atributos.");
          exit(-1);
  }
     // Compruebo el campo contentionscope con pthread_attr_getscope()
  if (pthread_attr_getscope(&atributos, &tipohilo) != 0) {
          perror("En la obtención de atributos.");
          exit(-1);
  }

  printf("Atributo de ambito por defecto: %s\n",
        (tipohilo==PTHREAD_SCOPE_SYSTEM) ? "de núcleo" : "de usuario");
  // Cambio el ámbito en mis atributos
  if (pthread_attr_setscope(&atributos, PTHREAD_SCOPE_SYSTEM) != 0) {
          perror("En el cambio de atributos.");
          exit(-1);
  }
  if (pthread_attr_getscope(&atributos, &tipohilo) != 0) {
          perror("En la obtención de atributos.");
          exit(-1);
  }
  printf("Nuevo atributo de ambito: %s\n",
        (tipohilo==PTHREAD_SCOPE_SYSTEM) ? "de núcleo" : "de usuario");
        // Ahora ya se pueden crear los hilos con el nuevo atributo
  printf("Crear hilo...\n");
```

Cuestiones a resolver

1.- Analice y describa el funcionamiento de los cuatro programas de ejemplo (proc_01.c a proc_04.c) suministrados.

2.- Construya un programa que cree cuatro procesos, A, B, C y D, de forma que A sea padre de B, B sea padre de C, y C sea padre de D.

3.- Construya un programa que cree un árbol de procesos de tres niveles de profundidad, de modo que cada rama tenga dos procesos.

4.- Realice un programa ***ejecuta***r que lea de la entrada estándar el nombre de un programa y cree un proceso hijo para ejecutar dicho programa.

5.- Construya un programa, similar al de la cuestión 2, en el que se creen cinco hijos y cada proceso termine ordenadamente 1 segundo después de hacerlo su hijo.

Práctica 3: COMUNICACIÓN ENTRE PROCESOS: PIPES

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos basadas en tuberías o *pipes*, así como métodos de atención a varios canales de comunicación distintos (*select*) y los efectos del empleo de operaciones de lectura y escritura bloqueantes y no bloqueantes. Las *pipes* también se denominan *tuberías sin nombre*, en contraste con las FIFO que son tuberías con nombre.

Se trata de un mecanismo de comunicación entre procesos mediante el paso de mensajes, que permite que el flujo de salida de un proceso se convierta en el flujo de entrada de un segundo proceso. Para mejorar el rendimiento, la mayoría de los sistemas operativos implementan las tuberías usando *buffers*, lo que permite al proceso proveedor generar más datos que lo que el proceso consumidor puede atender inmediatamente.

2.- *pipe*

La llamada al sistema *pipe* crea un par de descriptores de ficheros, que apuntan a una tubería, y los pone en el vector de dos elementos apuntado por *filedescriptor*, donde *filedescriptor[0]* contiene el descriptor de lectura y *filedescriptor[1]* contiene el descriptor de escritura. Aquello que se escribe en *filedescriptor[1]* es leído en *filedescriptor[0]*.

```
#include <stdio.h>
int pipe(int filedescriptor[2]);
```

Las tuberías permiten una comunicación entre procesos muy sencilla mediante el paso de mensajes, ya que el tratamiento de las tuberías es el mismo que el de los ficheros. Las primitivas *write* y *read* permiten escribir y leer, respectivamente, de una tubería. Sus principales limitaciones es que sólo pueden ser empleadas para comunicar procesos relacionados, es decir, con procesos que tengan algún ancestro en común; y que únicamente permiten la comunicación unidireccional.

Para comunicar dos procesos mediante una tubería, es necesario crear la tubería antes que los dos procesos para que éstos puedan compartir la misma tubería "heredando" sus descriptores. Por ello, el proceso creará primero una tubería y luego ejecutará la llamada *fork* para crear los nuevos procesos a intercomunicar. Como los descriptores de fichero se heredan de padres a hijos, los procesos creados tras el *fork* conocerán los descriptores de la tubería, y por tanto, podrán comunicarse entre sí.

Para evitar inconsistencias cada uno de los procesos intercomunicados por una tubería cierra el descriptor de fichero correspondiente a la operación de lectura o

escritura que no va a realizar. Después se realizan las lecturas y escrituras que se deseen y al concluir, se cierra la tubería.

```
#include <stdio.h>
int tubería[2], pid;
…
pipe(tuberia);
pid = fork();
if (pid == 0) {
          char *cadena;
          cadena = "Hola mundo";
          close(tubería[0]);
          write(tubería[1], cadena, strlen(cadena)+1);
          close(tuberia[1]);
          exit(1);
} else {
          char *cadena;
          int bytesLeidos;
      cadena = (char *) malloc(100 * strlen(char));
          close(tubería[1]);
          bytesLeidos = read(tubería[0], cadena, 100);
          printf("Mensaje leído: %s\n", cadena);
          close(tuberia[0]);
          exit(1);
}
…
```

La tubería usa un *buffer* gestionado por el sistema operativo, cuyo tamaño puede variarse. En general, las llamadas read y write son bloqueantes por defecto, de tal modo que se bloquean en lectura o en escritura hasta la finalización de la operación. Si una tubería está vacía y se intenta una operación de lectura, esta operación permanecerá bloqueda (interrumpiendo la ejecución del resto del programa) hasta que la tubería tenga algo para leer y se consume la operación de lectura. Del mismo modo, si un proceso intenta escribir en la tubería y ésta está llena, la operación de escritura se bloqueará hasta que se libere espacio en la tubería y se pueda completar la escritura. La gestión de la tubería es llevada a cabo por el sistema operativo siguiendo una política FIFO (*first input, first output*).

Si se intenta escribir cuando el extremo lector se ha cerrado se genera la señal SIGPIPE (ver sección 6 de la página 79), y cuando se cierra el extremo escritor, se recibe un EOF tras la recepción de los últimos datos.

2.1.- fcntl

Permite modificar las propiedades de los descriptores de fichero mediante el argumento *cmd*. El argumento *fd* indica sobre qué descriptor de fichero se desea actuar, *cmd* el comando que se desea efectuar, y *arg* el argumento del comando indicado.

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

F_GETFL: Lee las banderas de un descriptor de fichero.

F_SETFL: De las banderas de un descriptor, establece la parte que se corresponde con las banderas de situación de un fichero al valor especificado por *arg*. Los restantes bits (modo de acceso, banderas de creación de un fichero) de *arg* se ignoran. En Linux, esta orden sólo puede cambiar las banderas O_APPEND, O_NONBLOCK, O_ASYNC y O_DIRECT.

```
#include <unistd.h>
#include <fcntl.h>
...
int miTuberia[2];

fcntl(miTuberia[0], F_SETFL, O_NONBLOCK); // Lectura no bloqueante
fcntl(miTuberia[1], F_SETFL, O_NONBLOCK); // Escritura no bloqueante
```

3.- *fifo*: tuberías con nombre.

Las tuberías con nombre, también denominadas FIFO, permiten comunicación entre dos procesos cualesquiera, entre los que no existe relación de parentesco. Es decir, no tiene un ancestro común que pueda crear la tubería y estos procesos heredarla. Son un tipo especial de fichero, y por tanto, son persistentes.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t modo);
```

El argumento *pathname* define la ruta de la tubería con nombre que se va a crear y el argumento *modo* define la máscara de permisos del fichero.

La apertura, cierre, eliminación, lectura y escritura en una *fifo* es equivalente a las operaciones sobre ficheros (open, close, unlink, read y write).

Apertura bloqueante

- open("fifo_ejemplo", O_WRONLY)
  El proceso escritor se bloquea hasta que no haya otro proceso que abra la tubería para leer de ella.
- open("fifo_ejemplo",O_RD ONLY)
  El proceso lector se bloquea hasta que no haya otro proceso que abra la tubería para escribir en ella.

Apertura no bloqueante

En este caso se emplea en la apertura el modificador O_NONBLOCK. Si se especifica, un open de sólo lectura retorna inmediatamente. Un open de sólo escritura retorna un error si ningún proceso tiene la *fifo* abierta para lectura.

En algunas ocasiones puede que sólo exista un lector y un escritor, pero no es necesario que sea siempre así, puesto que pueden existir múltiples lectores y escritores. En este caso es necesario implementar mecanismos para coordinar el uso compartido de la *fifo* .

La constante PIPE_BUF define el número máximo de caracteres que se pueden escribir en una tubería atómicamente.

Desde la línea de comandos (s*hell*) se pueden crear tuberías con nombre con la orden ***mkfifo***.

Es muy importante eliminar la fifo cuando se termine de usarla, puesto que se trata de un elemento persistente que no se elimina del sistema al concluir la ejecución de los procesos. Para ello se puede utilizar la orden ***remove***.

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

#define NOMBREFIFO "mififo"
#define TAM_BUF 100
#define TRUE 1

int main(void)
{
  int fp;
  char buffer[TAM_BUF];
  int nbytes;

  mkfifo(NOMBREFIFO,S_IFIFO|0660,0);

  while(TRUE)
  {
    fp=open(NOMBREFIFO,O_RDONLY);
    nbytes=read(fp,buffer,TAM_BUF-1);
    buffer[nbytes]='\0';
    printf("Cadena recibida: %s \n",buffer);
    close(fp);
  }
  return 0;
}
```

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

#define NOMBREFIFO "mififo"

int main(int argc, char *argv[])
{
  int fp;
  int result=1;

  if(argc!=2) printf("uso: %s cadena \n", argv[0]);
```

```
  else if ((fp=open(NOMBREFIFO,O_WRONLY))==-1)
  perror("fopen");
  else {
    write(fp,argv[1],strlen(argv[1]));
    close(fp);
    result=0;
  }

  return result;
}
```

4.- *select()*

La función `select()` indica cuál de los descriptores de fichero especificados está listo para lectura, listo para escritura, o ha producido un error sin atender. Si esta condición es falsa para todos los descriptores de fichero especificados, *select()* se bloquea hasta que venza un *timeout* o se cumpla la condición para al menos uno de los descriptores de fichero especificados.

```
int select(int nfds, fd_set *readfds, fdset *writefds, fd_set
*errorfds, struct timeval *timeout);
```

Los parámetros de la función **select()** son los siguientes:

- **int nfds:** con el valor del descriptor de fichero más alto que queremos tratar más uno. Cada vez que se abre un fichero, la llamada *open* devuelve un descriptor de fichero que es entero. Estos descriptores suelen tener valores consecutivos a partir del 3, ya que el valor 0 suele estar reservado para la entrada estándar (*stdin*), el 1 para la salida estándar (*stdout*), el 2 para la salida estándar de error (*stderr*). y a partir del 3 se nos irán asignando cada vez que abramos algún "fichero". Aquí debemos dar el valor más alto del descriptor que queramos pasar a la función más uno.
- **fd_set *readfds:** es un puntero a los descriptores de fichero de los que se pretende averiguar si hay algún dato disponible para leer o que queremos que se nos avise cuando lo haya.
- **fd_set *writefds:** es un puntero a los descriptores de fichero de los que que se pretende averiguar si se puede escribir en ellos sin peligro. Si en el otro lado han cerrado la tubería e intentamos escribir, se nos enviará una señal SIGPIPE que hará que nuestro programa se caiga (salvo que tratemos la señal).
- **fd_set *errorfds:** es un puntero a los descriptores de fichero de los que se pretende averiguar si ha ocurrido alguna excepción.
- **struct timeval *timeout:** es el tiempo que queremos esperar como máximo. Si el valor es **NULL**, la llamada a **select()** es bloqueante y no se interrumpe hasta que suceda algo en alguno de los descriptores.

Cuando la función retorna, modifica los contenidos de los **fd_set** para indicar qué descriptores de fichero tienen algo. Por ello es importante inicializarlos completamente antes de volver a llamar a la función **select()**.

Para rellenar los *fd_set* y ver su contenido tenemos una serie de macros:

- **FD_ZERO (fd_set *):** vacía el puntero, de forma que indica que no nos interesa ningún descriptor de fichero.
- **FD_SET (int, fd_set *):** mete el descriptor que le pasamos en **int** al puntero **fd_set**. De esta forma se indica que tenemos interés en ese descriptor. Llamando primero a **FD_ZERO()** para inicializar el contenido del puntero y luego a **FD_SET()** tantas veces como descriptores tengamos, ya tenemos la variable dipuesta para llamar a **select()**.
- **FD_ISSET (int, fd_set *):** indica si ha habido algo en el descriptor **int** dentro de **fd_set**. Cuando **select()** sale, debemos ir interrogando a todos los descriptores uno por uno con esta macro.
- **FD_CLR (int, fd_set *)** elimina el descriptor dentro del **fd_set**.

```
fd_set descriptoresLectura ;

FD_ZERO (&descriptoresLectura);
FD_SET (socketServidor, &descriptoresLectura);
for (i=0; i<numeroClientes; i++)
    FD_SET (socketCliente[i], &descriptoresLectura);
...
select (maximo+1, &descriptoresLectura, NULL, NULL, NULL);
...
```

5.- Redireccionamiento de una Pipe a la E/S Estándar

Por defecto, la salida estándar (*stdout*) y la salida estándar de error (*stderr*) están direccionados hacia la pantalla del terminal, y la entrada estándar (*stdin*) corresponde al teclado. Pero en ocasiones puede ser que no nos interese que la información salga en pantalla, sino que nos interesa filtrarla o redireccionarla a un archivo para guardar la información o para un tratamiento posterior, o que la entrada a un programa sea un fichero o el resultado de la ejecución de otro. Con este fin los sistemas operativos permiten la utilización de tuberías y redirecciones.

El descriptor de fichero de valor 0 suele estar reservado para la *stdin*, el 1 para la *stdout*, el 2 para la *stderr*.

El redireccionamiento se emplea para conectar procesos a través de la Entrada/Salida estándar. Esta comunicación puede ser controlada por un proceso sin modificar el código de programa. Para hacerlo hay que cerrar el descriptor de STDIN o STDOUT y llamar a **dup** o **dup2**.

```
if ((pid = fork()) < 0) {
          error sys("error en fork");

} else if (pid == 0)
          ...       /* código del padre */

} else { /* hijo */
     close (STDIN_FILENO);          * se cierra entrada estándar */
     if (dup2(fd[0],STDIN_FILENO) < 0)
               error_sys("error en dup2");
     close(fd[0]);

     /* ahora la entrada estándar se realiza desde la salida de la
pipe */
     ...
     execvp(...);
     exit(1);
}
```

Cuestiones a resolver

1.- Construya un programa que cree dos procesos que se comuniquen entre sí mediante tuberías e intercambien diez mensajes entre sí. Para ello, el primer proceso construirá un mensaje que contendrá un contador de secuencia del mensaje y su identificador de proceso (PID). Este mensaje se enviará al otro proceso, que leerá el mensaje, aumentará el número de secuencia e introducirá su identificador de proceso. El programa concluirá cuando se hayan intercambiado los citados diez mensajes y los dos procesos concluirán ordenadamente imprimiendo por pantalla un mensaje de despedida.

```
struct mensaje {
     int secuencia, pidEmisor;
} mensaje;
```

2.- Construya dos programas independientes que actúen como se describe en la cuestión anterior, pero que hagan uso de una tubería con nombre (FIFO) para su comunicación.

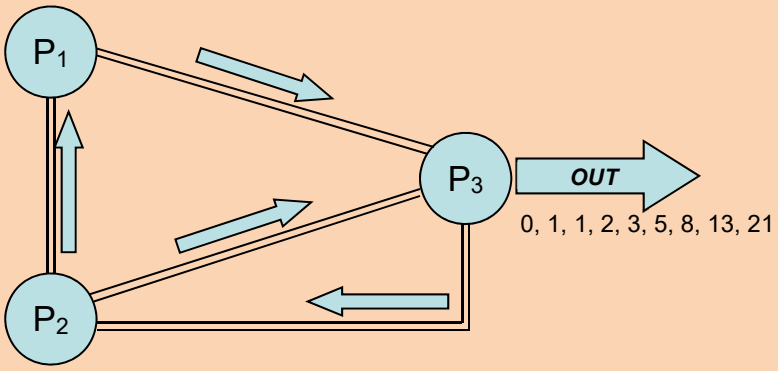
3.- Construya un programa similar al de la cuestión 1, pero empleando lecturas no bloqueantes y la función *select*.

4.- En matemáticas, la sucesión de Fibonacci es la sucesión infinita de números naturales 0,1,1,2,3,5,8,13,21…donde cada elemento es la suma de los dos **anteriores. A cada** elemento de esta sucesión se le llama número de Fibonacci.

Definición: Los números de Fibonacci f_0,f_1,f_2,f_3… quedan definidos por las ecuaciones $f_0 = 0$, $f_1=1$ y $f_n = f_{n-1} + f_{n-2}$, para n = 2, 3, 4… Es importante notar que la secuencia no tiene fin.

$$f_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f_{n-1} + f_{n-2} & \text{si } n > 1 \end{cases}$$

Escriba un programa en C que calcule la sucesión de Fibonacci empleando para ello tres procesos y sus correspondientes mecanismos de comunicación tal y como se indica en la siguiente figura. El proceso P3 irá imprimiendo en pantalla los valores obtenidos para la sucesión separados entre comas (ej.: 0, 1, 1, 2, 3, 5, 8, 13, 21) al paso de un segundo.



Práctica 4: COMUNICACIÓN ENTRE PROCESOS: SEÑALES

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos basadas en *señales*.

Una señal (*signal*) es un a forma limitada de comunicación entre procesos empleada en Unix y otros sistemas operativos compatibles con POSIX (*Portable Operating System Interface*). Es una notificación asíncrona enviada a un proceso para notificarle un evento. Cuando se le manda una señal a un proceso, el sistema operativo modifica su ejecución normal. Si se había establecido anteriormente un procedimiento (*handler*) para tratar esa señal se ejecuta éste, si no se estableció nada previamente se ejecuta la acción por defecto para esa señal.

El usuario ya está familiarizado con su uso, ya que al escribir Ctrl-C en la shell donde se ejecuta un proceso el sistema le envía una señal SIGINT, que por defecto causa la terminación del proceso. Del mismo modo, Ctrl-Z hace que el sistema envíe una señal SIGSTOP que suspende la ejecución del proceso. Además, la llamada al sistema *kill(2)* enviará la señal especificada al proceso y el comando *kill(1)* envía la señal SIGKILL al proceso indicado como argumento. Excepciones como la división por cero o la violación de segmento también generan señales.

2.- Señales

Aunque son muchas las señales que se pueden manejar, la Tabla 1 recoge algunas de las más relevantes. Si se desea obtener más información sobre las señales disponibles y el modo de trabajar con ellas, se recomienda visitar la sección 7 del manual (*man 7 signal*).

Los manipuladores de señales, también conocidos como manejadores o *handlers*, se establecen mediante la llamada al sistema *signal(2)*. En el fondo, lo que indican es la tarea a realizar cuando se reciba la señal indicada. Si hay un manejador para una señal dada se invoca y, si no lo hay, se usa el manipulador por defecto. El proceso puede especificar también dos comportamientos por defecto sin necesidad de crear un manejador: ignorar la señal (SIG_IGN) y usar el manipulador por defecto (SIG_DFL). Las señales SIGKILL y SIGSTOP no pueden ser capturadas, bloqueadas o ignoradas.

Téngase en cuenta que las señales son asíncronas y puede ocurrir que llegue otra señal (incluso del mismo tipo) al proceso mientras transcurre la ejecución de la función que manipula la señal. Puede usarse la función *sigprocmask* para desbloquear la entrega de señales.

Las señales pueden interrumpir una llamada al sistema en proceso.

Señal	Explicación
SIGALRM	Señal de alarma, salta al expirar el timer. Reprogramable.
SIGBUS	Error en el bus *access to undefined portion of memory object* (SUS).
SIGCHLD	Proceso hijo terminado, detenido (*o que continúa). Tratamiento por defecto: ignorar. Reprogramable.
SIGCONT	Continúa si estaba parado.Tratamiento por defecto: continuar. Reprogramable.
SIGFPE	Excepción de coma flotante -- *erroneous arithmetic operation* (SUS).
SIGHUP	Hangup, al salir de la sesión se envía a los processos en Background. Tratamiento por defecto: exit. Reprogramable.
SIGILL	Instrucción ilegal.
SIGINT	Interrupción, se genera al pulsar Ctrl-C durante la ejecución. Tratamiento por defecto: exit. Reprogramable.
SIGKILL	Destrucción inmediata del proceso.Tratamiento: exit. No reprogramable, no ignorable.
SIGPIPE	Se genera al escribir sobre la pipe sin lector. Tratamiento por defecto: exit. Reprogramable.
SIGQUIT	Terminar.
SIGSEGV	*Segmentation violation*. Salta con dirección de memoria ilegal. Tratamiento por defecto: exit + volcado de memoria. Reprogramable.
SIGSTOP	Detiene el proceso. Se genera al pulsar Ctrl-Z durante la ejecución. No reprogramable, no ignorable.
SIGTERM	Terminación. Tratamiento por defecto:exit. Reprogramable.
SIGTSTP	Parada de terminal.
SIGTTIN	Proceso en segundo plano intentando leer (*in*).
SIGTTOU	Proceso en segundo plano intentando escribir (*out*).
SIGUSR1	*User defined 1*. Signal definido por el usuario.Tratamiento por defecto: exit. Reprogramable.
SIGUSR2	*User defined 2*. Signal definido por el usuario.Tratamiento por defecto: exit. Reprogramable

Tabla 1: Algunas señales relevantes.

Cada señal tiene asociado un valor que la identifica de forma unívoca, pero suele ser más inteligible su nombre. Todas las señales empiezan por el prefijo SIG y se escriben en mayúsculas puesto que son constantes que vinculan una etiqueta al valor correspondiente de la señal. La Tabla 2 ilustra esta relación y muestra también cuál es la acción por defecto que lleva a cabo el sistema cuando recibe una determinada señal.

Term	La acción por defecto es terminar el proceso.
Ign	La acción por defecto es ignorar la señal.
Core	La acción por defecto es terminar el proceso y realizar un volcado de memoria.
Stop	La acción por defecto es detener el proceso.

Señal	Valor	Acción	Explicación
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	10	Term	Term User-defined signal 1
SIGUSR2	12	Ign	User-defined signal 2
SIGCHLD	17	Cont	Child stopped or terminated
SIGCONT	18	Stop	Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at tty
SIGTTIN	21	Stop	tty input for background process
SIGTTOU	22	Stop	tty output for background process

Tabla 2: Algunas señales junto con sus valores y acciones asociadas.

3.- Envío de señales

Los procesos pueden enviar señales tanto a otros procesos como a sí mismos usando *kill(2)* por ejemplo *kill(pid,SIGUSR1)* siendo *pid* el identificador del proceso al cual deseamos enviar la señal *SIGUSR1*.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

La llamada al sistema *kill(2)* se puede usar para enviar cualquier señal a un proceso o grupo de procesos. Si el argumento *pid* es positivo, entonces la señal *sig* se envía al proceso *pid*. La llamada devuelve 0 si se ha ejecutado con éxito, o un valor negativo en caso de fallo. Si el valor del argumento *pid* es 0, entonces el sistema envía la señal *sig* a cada uno de los procesos que forman parte del grupo de procesos del proceso actual. Si el valor de *pid* es −1, entonces se envía la señal *sig* a cada proceso, excepto al proceso 1 (*init*). Si el valor de *pid* es menor que −1, entonces se envía la señal *sig* a cada proceso en el grupo de procesos *-pid*. Si el valor de *sig* es 0, entonces no se envía ninguna señal pero se realiza la comprobación de errores.

Si la llamada concluye con éxito, la llamada devuelve el valor 0, mientras que si se produce un error, devuelve −1, y actualiza la variable *errno* apropiadamente.

Como ya se ha indicado, es imposible enviar una señal a la tarea número uno, el proceso *init*, para el que no existe un manejador de señales con el fin de que no se pueda *colgar* el sistema accidentalmente.

También se puede emplear el comando del sistema operativo *kill(1)* para enviar señales. El modo de operar es similar al indicado anteriormente, pero se lleva a cabo desde la línea de comandos (*shell*) y permite el envío simultáneo de una señal a varios procesos.

```
kill -s signal_name pid ...
kill -l [exit_status]
kill [-signal_name] pid ...
kill [-signal_number] pid ...
```

4.- Captura de señales

Todas las señales tienen una función de tratamiento por defecto. Cuando un proceso recibe una señal, suspenderá su ejecución y acudirá a la función destinada al tratamiento de esa señal y luego continuará su ejecución normal (si la señal no es de terminación). Todo esto de una forma transparente al usuario.

Es posible modificar casi todas las funciones de tratamiento de señales establecidas por defecto, pudiendo de este modo lograr que al recibir una señal el programa haga algo distinto a lo establecido por defecto.

4.1- signal

La llamada al sistema *signal()* instala un nuevo manejador de señales para la señal con número *signum*. El manejador de señales queda establecido a *sighandler* que puede ser una función especificada por el usuario o bien SIG_IGN o SIG_DFL. Cuando llega una señal con número *signum*: si el manejador correspondiente está establecido a SIG_IGN, la señal es ignorada; si el manejador está establecido a SIG_DFL, se realiza la acción por defecto asociada a la señal (ver signal(7)); si el manejador está establecido a una función *sighandler* lo primero que se hace es o bien restablecer el manejador a SIG_DFL o un bloqueo de la señal que depende de la implementación, invocando después a *sighandler* con el argumento *signum*.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

4.2- sigaction

La llamada al sistema *sigaction* se emplea para cambiar la acción tomada por un proceso cuando recibe una determinada señal. El argumento *signum* indica la señal y

puede ser cualquiera válida salvo SIGKILL o SIGSTOP. Si el argumento *act* no es nulo, la nueva acción para la señal *signum* se instala como *act*. Si *oldact* no es nulo, la acción anterior se guarda en *oldact*.

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

La estructura sigaction se define del siguiente modo:

```
struct sigaction {
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t sa_mask;
int sa_flags;
void (*sa_restorer)(void);
}
```

El elemento *sa_handler* especifica la acción que se va a asociar con *signum* y puede ser SIG_DFL para la acción predeterminada, SIG_IGN para no tener en cuenta la señal, o un puntero a una función manejadora para la señal. El elemento *sa_mask* da una máscara de señales que deberían bloquearse durante la ejecución del manejador de señal. Además, la señal que lance el manejador será bloqueada, a menos que se activen las opciones SA_NODEFER o SA_NOMASK. El elemento *sa_restorer* está obsoleto y no debería utilizarse. El elemento *sa_flags* especifica un conjunto de opciones que modifican el comportamiento del proceso de manejo de señal. Se forma por la aplicación del operador de bits OR (|) a cero o más de las siguientes constantes:

- **SA_NOCLDSTOP**

  Si *signum* es SIGCHLD, no se recibe notificación cuando los procesos hijos se paren (esto es, cuando los procesos hijos reciban una de las señales SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU).
- **SA_ONESHOT** o **SA_RESETHAND**

  Se restaura la acción para la señal al estado predeterminado una vez que el manejador de señal haya sido invocado.
- **SA_ONSTACK**

  Llama al manejador de señal en una pila de señales alternativa proporcionada por *sigaltstack(2)*. Si esta pila alternativa no está disponible, se utilizará la pila por defecto.
- **SA_NOMASK** o **SA_NODEFER**

  No se impide que se reciba la señal desde su propio manejador.

Los manejadores de señal se definen una vez en el programa y la llamada al sistema signal se invoca una sola vez, no es preciso volver a invocarla cada vez que se produce

una señal. Por el contrario, las señales se pueden enviar tantas veces como se deseen con la ayuda de la llamada al sistema *kill(2).*

Cuestiones a resolver

1.- Construya un programa que cree dos procesos que se comuniquen entre sí mediante señales de modo que intercambien diez mensajes entre sí. Para ello, el primer proceso enviará la señal SIGUSR1 al segundo proceso, y éste le devolverá la señal SIGUSR2. El programa concluirá cuando se hayan intercambiado los citados diez mensajes (diez señales) y los dos procesos concluirán ordenadamente imprimiendo por pantalla un mensaje de despedida.

2.- Construya un programa que cree dos procesos. Uno de ellos simulará ser un contador, mientras que el segundo será quien lo gobierne. El proceso contador, que incrementará el valor de la variable *contador* cada segundo, se pondrá en marcha y se parará cada vez que reciba la señal SIGUSR1 del otro proceso. Además, reiniciará su contador cuando reciba la señal SIGUSR2. Cada vez que reciba la señal SIGUSR1 imprimirá por pantalla el valor del contador.

El otro proceso (gestor) enviará la señal SIGUSR1 al proceso contador cuando se introduzca un 1 por teclado y la señal SIGUSR2 al proceso contador cuando se introduzca un 2 por teclado.

La ejecución concluirá ordenadamente cuando se finalice el proceso gestor (Ctrl-C o el valor 0 por teclado) e implicará la finalización del proceso contador y después del proceso gestor.

Práctica 5: SHELL

1.- Introducción

El objetivo de esta práctica es construir una pequeña shell haciendo uso de una librería, de la llamada al sistema *execvp*, de las redirecciones de entrada y salida (*dup2*) y de comunicación mediante señales. Esta práctica pretende poner en contexto algunos de los conceptos introducidos en las prácticas precedentes, de modo que el alumno sea capaz de entender el funcionamiento de las herramientas de gestión y comunicación de procesos que se la han presentado anteriormente y que sea capaz de desarrollar una pequeña aplicación que emplee los conocimientos adquiridos.

El objetivo de esta práctica es construir una *shell* o intérprete de empleando las llamadas al sistema de manejo de procesos (fork, wait, exec, dup2…) que permita ejecutar cualquier comando del sistema operativo o ejecutar otro programa, que admita tuberías (|), así como redireccionamiento de entrada y salida (<, >).

2.- Especificaciones

En este apartado se especifican las funcionalidades que debe cumplir la aplicación indicada.

- **Admitir pipes** "|". Ejemplo: *more kk.txt | grep hola*, imprimirá por pantalla aquellas líneas del fichero *kk.tx*t que contengan la palabra *hola*. Para ello se crearán dos procesos, uno de los cuales ejecutará el comando *more* y el otro el comando *grep*. Ambos procesos se intercomunicarán con la ayuda de una tubería, de tal modo que la salida del primer proceso se escriba en la tubería y la entrada del segundo proceso se lea de dicha tubería.
- **Admitir redirección a fichero** "**> fichero**". Ejemplo: *ls –al > kk.txt*, listará el contenido del directorio local en el fichero *kk.txt*.
- **Admitir anexión a fichero** "**>> fichero**". Idéntico al anterior salvo por el modo de volcar a fichero. En este caso no se sobreescribe el contenido del fichero sino que se anexa al final de éste.
- **Admitir entrada desde fichero** "**< fichero**". Ejemplo: *wc -l < kk.txt*, contará el número de líneas del fichero *kk.txt*.
- **Admitir redirección de entrada y salida simultánea**.
- ***Prompt* personalizado:** *minishell\>*
- **La conclusión de la ejecución de la *Shell* se alcanzará al introducir el comando *exit* o pulsar *Ctrl+C*.**

3.- Librería

Para construir la *shell* se dispondrá de una pequeña librería con una función (*fragmenta*) que permite trocear cadenas. Esta función tomará una cadena de texto de la línea de comandos y la fragmentará de tal modo que se ajuste a la estructura de datos que posteriormente empleará para rellenar la estructura de datos que precisa la llamada al sistema *execvp* para ejecutar los comandos contenidos en la cadena de texto leída. La librería se completará con otra función (*borrarg*) que permitirá liberar la memoria reservada por la función de fragmentación.

La librería se denominará *fragmenta.o* y presentará las siguientes características:

FRAGMENTA (3) FRAGMENTA (3)

NOMBRE

fragmenta, borrarg – Utilidades para trocear cadenas.

SINOPSIS

```
#include "fragmenta.h"
char **fragmenta(const char *cadena);
void borrarg(char **arg);
```

DESCRIPCIÓN

- ***fragmenta():*** crea una array de *char** con tantos elementos como el número de fragmentos que encuentre en cadena más uno, el último vale siempre *NULL* y es el único con tal valor, con lo que sirve para determinar el final del array. Cada elemento de este array es un puntero a una zona de memoria donde se encuentra uno de los fragmentos de cadena y en el mismo orden. Los fragmentos de cadena vienen definidos por estar separados por uno o más espacios, pudiendo terminar en un fin de línea. Si la cadena a fragmentar posee múltiples espacios en blanco, ninguno de éstos se almacenará en la estructura de datos resultante.
- ***borrarg():*** libera la memoria asociada con el puntero *arg*, así como las zonas de memoria apuntadas por cada uno de los *char** apuntados por *arg* y colocados uno tras otro hasta uno que valga NULL.

VALOR DEVUELTO

fragmenta() devuelve el puntero al array creado o NULL si no puede realizar su función.

El contenido de la cabecera de la librería es:

```
/* Contenido de fragmenta.h */
char **fragmenta(const char*);
void borrarg(char **arg);
/* Fin de fragmenta.h */
```

4.- *Shell*

Para construir la *shell*, será preciso crear nuevos procesos mediante la orden *fork* y duplicar los descriptores mediante la orden *dup2*. Otras llamadas al sistema de interés son *wait, pipe, signal, open* y *close*.

La descripción de la *shell* será:

MSH (3) MSH (3)

NOMBRE

`minishell - Mini Shell`

SINOPSIS

`msh`

DESCRIPCION

Las funcionalidades de esta shell son:

- Ejecución de comandos con un número indeterminado de argumentos.
  Ejemplo:
  minishell\> cp -r sources backup
- Redirección de la salida estándar a fichero mediante>
  Ejemplo:
  minishell\> ls –al > listado
- Redirección de la entrada estándar de fichero mediante <.
  Ejemplo:
  minishell\> wc -l < listado
- Redirección simultánea de entrada y salida estándar en cualquier orden.

5.- Entrega de la práctica

En la carpeta de prácticas de MiAulario se dispone de la biblioteca fragmenta, sus fuentes y el *Makefile* correspondiente. En el fichero comprimido a entregar al concluir la práctica debe encontrarse el *Makefile* provisto, así como todos los ficheros .c y .h necesarios para crear *minishell*. La acción por defecto del *Makefile* (la cual debe funcionar con sólo hacer make en ese directorio) debe ser crear el ejecutable de la shell. Igualmente debe responder correctamente a *make fragmenta* y a *make prueba*.

Para la corrección de la práctica se borrarán todos los ejecutables, se hará un *touch* a todos los ficheros fuente y se recompilará mediante el *Makefile*.

6.- Aplicaciones, funciones y llamadas al sistema útiles

`fork(2), execvp(3), wait(2), open(2), close(2), dup2(2), pipe(2), signal(2) y kill(2).`

Práctica 6: COMUNICACIÓN ENTRE PROCESOS: SEMÁFOROS Y MEMORIA COMPARTIDA

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos mediante semáforos y memoria compartida.

La memoria compartida permite el acceso de dos o más procesos a una misma zona de memoria. Esto permite compartir datos entre dichos procesos, pero también implica garantizar el acceso en exclusiva para evitar inconsistencias en los datos mediante mecanismos como señales o semáforos. Si dos procesos tratan de acceder simultáneamente a esta área, se deberá regular su acceso para garantizar la consistencia de los datos. Este recurso IPC es el más rápido, ya que una vez conectados no necesitamos realizar más llamadas al sistema, ni interactuar con el núcleo; se trabaja directamente con el puntero que referencia a la memoria compartida.

Los semáforos son una herramienta especialmente destinada a la sincronización entre procesos. Un semáforo permite el acceso a un recurso a uno de los procesos y se lo deniega a los demás mientras el primero no concluya su tarea. Los semáforos son una interesante herramienta para gobernar el acceso a un recurso como es la memoria compartida puesto que realizan todas sus operaciones de forma atómica. Un semáforo debe garantizar la exclusión mutua (sólo va a haber un proceso en la sección crítica), que un proceso que no está en su sección crítica no pueda bloquear a otros procesos y que el proceso que está en la sección crítica no bloqueará para siempre al resto.

2.- Memoria compartida

Las utilidades de memoria compartida permiten crear segmentos de memoria a los que pueden acceder múltiples procesos, pudiendo definirse restricciones de acceso (sólo lectura, escritura). Para trabajar con un segmento de memoria compartida, es necesario crear un vínculo entre la memoria local del proceso y el segmento compartido. El proceso que vincula un segmento de memoria compartida cree estar trabajando con ella como si fuera cierta área de memoria local.

2.1.- Creación de zonas de memoria compartida

La creación de una zona de memoria compartida se lleva a cabo con la ayuda de la llamada al sistema *shmget*.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

La llamada *shmget()* devuelve el identificador del segmento de memoria compartida asociado con el valor del argumento *key* si la operación se realiza correctamente y -1 si

se produce algún error (y *errno* recoge el número de error encontrado). El valor de error EEXIST indica que la memoria compartida ya existía y no se está creando una nueva. El argumento *key* es la clave para crear la memoria compartida (identificador de memoria) y es la misma para todos los procesos que quieran acceder a esta zona compartida de memoria. Si *key* es IPC_PRIVATE, se creará un nuevo identificador (si existen disponibilidades) que no será devuelto por posteriores invocaciones a *shmget* mientras no se libere mediante la función *shmctl*. El identificador creado podrá utilizarse por el proceso invocador y sus descendientes y por cualquier proceso que posea los permisos adecuados. Esta clave (*key*) debe ser única dentro de la máquina. Si no es así, la estructura para la comunicación entre procesos no se crea y devuelve un error. La clave puede obtenerse de tres formas:

- El servidor crea una nueva estructura especificando una clave de IPC_PRIVATE. El procedimiento creador devuelve un identificador para la nueva estructura. El problema es que ésta debe ser comunicada al proceso cliente de alguna manera.
- Otra forma es que el servidor y cliente se pongan de acuerdo en una clave. El problema es que ésta puede coincidir con otra ya existente.
- El servidor y el cliente pueden convenir un *path* y un identificador de proyecto y llamar a la función *ftok*, que convierte estos dos valores en una clave.

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char * path, int projectID);  /* key_t es un long int */
```

Se crea entonces un nuevo segmento de memoria compartida, del tamaño indicado por el argumento *size*. El argumento *shmflg* contiene las opciones de configuración, y se efectúa un OR entre ellas. Algunas de las opciones son:

- IPC_CREAT: sirve para crear un nuevo segmento. Si no se usa este indicador, *shmget()* encontrará el segmento asociado con key y comprobará que el usuario tenga permiso para acceder al segmento.
- IPC_EXCL: sirve para asegurarse de que no existía anteriormente. Si emplea junto con IPC_CREAT, asegura el fallo si el segmento ya existe.
- Permisos: SHM_R para leer y SHM_W para escribir.

Ejemplo: se crea una zona de memoria compartida del tamaño de una variable entera.

```
int shmid;
shmid = shmget(IPC PRIVATE, sizeof(int), IPC CREAT | IPC EXCL | S IRUSR | S IWUSR);
if (shmid == -1)
perror("Error al crear la memoria compartida.");
```

2.2.- Vínculo de zonas de memoria compartida

La vinculación a una zona de memoria compartida existente se lleva a cabo con la ayuda de la llamada al sistema *shmat*.

```
#include <sys/shm.h>
```

```
char *shmat( int shmid, void *shmaddr, int shmflg );
```

La llamada al sistema *shmat* asocia el segmento de memoria compartida especificado por el argumento *shmid* al segmento de datos del proceso invocador. Si el segmento de memoria compartida aún no se ha asociado al proceso invocador, entonces *shmaddr* debe tener el valor 0 y el segmento se asocia a una posición en memoria seleccionada por el sistema operativo. Dicha localización será la misma en todos los procesos que acceden al objeto de memoria compartida. Si el segmento de memoria compartida ya había sido asociado por el proceso invocador, *shmaddr* podrá tener un valor distinto de cero, en ese caso deberá tomar la dirección asociada actual del segmento referenciado por *shmid*. Un segmento se asocia en modo *sólo lectura* si *shmflg* & *SHM_RDONLY* es verdadero; si no es así, se podrá acceder en modo lectura y escritura. No es posible la asociación en modo *sólo escritura*. Si la función se ejecuta con éxito, entonces devolverá la dirección de comienzo del segmento compartido, si ocurre un error devolverá -1 y la variable global *errno* tomará el código del error producido.

Ejemplo: se recupera la dirección de la zona de memoria compartida obtenida en el ejemplo anterior y coloca en ella el valor 10. Como la dirección de memoria corresponde con un valor entero, se almacena en un puntero a entero.

```
int *entero;
entero = (int *)shmat(shmid, NULL, 0);
if (entero == (int *)-1) {
     perror("Obteniendo dirección de memoria compartida");
     return -1;
}
(*entero)=10;
```

2.3.- Desvinculación de zonas de memoria compartida

La desvinculación de una zona de memoria compartida existente se lleva a cabo con la ayuda de la llamada al sistema *shmdt*.

```
#include <sys/shm.h>

char *shmdt(void *shmaddr);
```

La llamada al sistema *shmdt* desvincula el segmento de datos del proceso invocador del segmento de memoria compartida ubicado en la localización de memoria especificada por *shmaddr*. Si la función se ejecuta sin error, entonces devolverá 0, en caso contrario devolverá -1 y *errno* tomará el código del error que se haya producido.

Ejemplo: se desvincula la memoria compartida vinculada en el ejemplo anterior.

```
r = shmdt(entero);
if (r == -1) perror("Error desvinculando memoria compartida");
```

2.4.- Operaciones de control de zonas de memoria compartida

La llamada al sistema *shmctl* realiza operaciones de control en una región de memoria compartida dada.

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buff);
```

La llamada al sistema *shmctl* permite realizar un conjunto de operaciones de control sobre una zona de memoria compartida identificada por *shmid*. El argumento *cmd* se usa para codificar la operación solicitada (comando). Los valores admisibles para este parámetro son:

- IPC_STAT: lee la estructura de control asociada a *shmid* y la deposita en la estructura apuntada por *buff*.
- IPC_SET: actualiza los campos *shm_perm.uid*, *shm_perm.gid* y *shm_perm.mode* de la estructura de control asociada a *shmid* tomando los valores de la estructura apuntada por *buff*.
- IPC_RMID: elimina el identificador de memoria compartida especificado por *shmid* del sistema, destruyendo el segmento de memoria compartida y las estructuras de control asociadas. Si el segmento está siendo utilizado por más de un proceso, entonces la clave asociada toma el valor IPC_PRIVATE y el segmento de memoria es eliminado, el segmento desaparecerá cuando el último proceso que lo utiliza notifique su desconexión del segmento. Esta operación sólo la podrán utilizar aquellos procesos que posean privilegios de acceso a recurso apropiados para llevar a cabo esta comprobación el sistema considerará el identificador de usuario efectivo del proceso y lo comparará con los campos *shm_perm_id* y *shm_perm_cuid* de la estructura de control asociada a la región de memoria compartida.
- SHM_LOCK: bloquea la zona de memoria compartida especificada por *shmid*. Este comando sólo puede ejecutado por procesos con privilegios de acceso apropiados.
- SHM_UNLOCK: desbloquea la región de memoria compartida especificada por *shmid*. Esta operación, al igual que la anterior, sólo la podrán ejecutar aquellos procesos con privilegios de acceso apropiados.

La función *shmctl* devuelve el valor 0 si se ejecuta con éxito, o -1 si se produce un error, tomando además la variable global *errno* el valor del código del error que se haya producido. El borrado sólo es efectivo si ningún proceso tiene vinculada la memoria

compartida. Si aún existen vínculos a la memoria compartida, la operación de borrado se pospone hasta que desaparezca el último vínculo.

Ejemplo: se elimina la zona de memoria utilizada en los ejemplos anteriores.

```
r = shmctl(shmid, IPC_RMID, NULL);
if (r == -1) perror("Error desvinculando memoria compartida");
```

3.- Semáforos

Un semáforo es un mecanismo, inventado por E. Dijkstra, que permite restringir o permitir el acceso a recursos compartidos en un entorno de multiprocesamiento en el que se ejecutarán varios procesos concurrentemente.

Para utilizar los semáforos en un programa, primero se debe obtener una clave de semáforo que lo identificara unívocamente. En general, se trata de una clave de recurso compartido, ya que la función que nos sirve para obtener dicha clave también vale para memoria compartida y colas de mensajes. Para ello se utiliza, como ya se ha visto antes, la función *key_t ftok(char *, int)* a la que se suministra como primer parámetro el nombre y *path* de un fichero cualquiera que exista y como segundo un entero cualquiera. Todos los procesos que quieran compartir el semáforo, deben suministrar el mismo fichero y el mismo entero. Posteriormente, se obtiene un array de semáforos. La función ***int semget (key_t key, int nsems, int semflag)*** permite obtener dicho array de semáforos. Se le pasa como primer parámetro la clave obtenida en el paso anterior, el segundo parámetro es el número de semáforos que queremos y el tercer parámetro son *flags*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Estos flags gobiernan los permisos de acceso a los semáforos, que son similares a los de los ficheros (lectura y escritura para el usuario, grupo y otros). También lleva unos modificadores para la obtención del semáforo. Por ejemplo, **0640 | IPC_CREATE** que indica permiso de lectura y escritura para el propietario, de lectura para el grupo y que los semáforos se creen si no lo están al llamar a *semget()*. Es importante el 0 ubicado delante del 640, ya que indica al compilador de C que cuanto sigue debe ser interpretado como un número en octal. La función *semget()* devuelve un identificador del array de semáforos.

3.1.- Inicialización del semáforo

Uno de los procesos deberá inicializar el semáforo. La función a utilizar para ello es ***int semctl(int semid, int semnum, int cmd, int ...)***. El primer parámetro (*semid*) es el identificador del array de semáforos obtenido anteriormente con *semget()*, el segundo

parámetro (*semnum*) es el índice del semáforo que queremos inicializar dentro del array de semáforos obtenido. Téngase en cuenta que el primer índice válido es el cero. El tercer parámetro (*cmd*) indica el comando que se desea aplicar al semáforo. En función de su valor, los siguientes parámetros serán una cosa u otra. El cuarto parámetro es opcional y depende de la operación que se quiera realizar (*cmd*). Si es necesario, este parámetro es una unión de tipo ***union semun***, que para el caso de usar la operación **SETVAL**, el campo *val* de dicha unión tomará el valor 1 si queremos el semáforo en "verde" o un 0 si lo queremos en "rojo".

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

En concreto, para inicializar el semáforo, el valor del tercer parámetro es **SETVAL**. Poner el valor de *semval* a *arg.val* para el *semnum*-ésimo semáforo del conjunto, actualizando también el miembro *sem_ctime* de la estructura *semid_ds* asociada al conjunto. Los procesos que se encuentran en la cola de espera son despertados si *semval* se pone a 0 o se incrementa. El proceso que realiza la llamada ha de tener privilegios de escritura en el conjunto de semáforos.

```
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};

…

semun arg;
arg.val = 1; //valor inicial para el semáforo: VERDE

semctl( id_semaforo, 0, SETVAL, arg ); //configurar el semaforo 0 del array a 1
```

El comando GETVAL de *semctl*() devuelve el valor actual del semáforo.

3.2.- Empleo del semáforo

El proceso que quiera acceder a un recurso común decrementa el semáforo mediante la función ***int semop(int semid, struct sembuf *sops, unsigned nsops)***. El primer parámetro (*semid*) es el identificador del array de semáforos obtenido con *semget()*. El segundo parámetro (**sops*) es un array de operaciones sobre el semáforo. Para

decrementarlo, bastará con un array de una única posición. El tercer parámetro es el número de elementos en el citado array (*nsops*).

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);
```

La estructura del segundo parámetro contiene tres campos:

- ***unsigned short sem_num*** es el índice del array del semáforo sobre el que se deseas actuar.
- ***short sem_op*** es el valor en el que se desea decrementar el semáforo. En nuestro caso, -1.
- ***short sem_flg*** son flags que afectan a la operación. En nuestro caso, para no complicarnos la vida, pondremos 0. Algunas opciones son IPC_NOWAIT y SEM_UNDO. Si una operación ejecuta SEM_UNDO, será deshechada cuando el proceso finalice.

Al realizar esta operación, si el semáforo se vuelve negativo, el proceso se quedará "bloqueado" hasta que alguien incremente el semáforo y alcance, como mínimo, el valor 0. Cuando el proceso termine de usar el recurso común, debe incrementar el semáforo. La función a utilizar es la misma, pero poniendo un valor positivo en el campo sem_op de la estructura *strcut sembuf*. Así un valor positivo en sem_op equivale a una operación de tipo *signal* sobre el semáforo y un valor negativo equivale a una función *wait* sobre dicho semáforo. Si *sem_op* es un entero positivo, la operación añade este valor al valor del semáforo (*semval*) y si es un entero negativo, la operación resta este valor al valor del semáforo.

El conjunto de operaciones contenido en **sops* se realiza de forma atómica. El comportamiento de la llamada al sistema en caso de que no todas las operaciones puedan realizarse inmediatamente depende de la presencia de la bandera IPC_NOWAIT en los campos *sem_flg* individuales.

Si la operación se lleva a cabo correctamente, la llamada al sistema devuelve el valor 0, y en cualquier otro caso devuelve -1 con *errno* indicando el error producido.

Los semáforos binarios sólo pueden tomar dos valores (0 ó 1). Cuando el semáforo está a 1 permite el acceso del proceso a la sección crítica, mientras que con 0 bloquea el acceso a ella. Este tipo de semáforos es especialmente útil para garantizar la exclusión mutua a la hora de realizar una tarea crítica en la memoria compartida. Por ejemplo, para controlar la escritura de variables en memoria compartida, de forma que sólo se permita que un proceso esté en la sección crítica mientras que se están modificando los datos. Los semáforos n-arios pueden tomar valores desde 0 hasta N. Su funcionamiento es similar al de los semáforos binarios, ya que cuando el semáforo está a 0, está cerrado y no permite el acceso a la sección crítica. La diferencia está en que puede tomar cualquier otro valor positivo además de 1. De hecho, este tipo de semáforos son muy útiles para permitir que un determinado número de procesos trabajen

concurrentemente en alguna tarea no crítica en la memoria compartida. Por ejemplo, varios procesos pueden estar leyendo simultáneamente de la memoria compartida, mientras que ningún otro proceso intente modificar datos.

```
#include <sys/ipc.h>
#include <sys/sem.h>

void main() {
    key_t clave;
    int id_semaforo;

    clave = ftok("/bin/bash", 'X');

    id_semaforo = semget(clave, 10, 0640 | IPC_CREAT);

    sembuf accion;

    // SIGNAL sobre el semáforo
    accion.sem_num = 0; // índice del semáforo
    accion.sem_op = 1;  // operación signal (enteros positivos)
    accion.sem_flg = 0;

    semop(id_semaforo, &accion, 1);

    // WAIT sobre el semáforo
    accion.sem_num = 0; // índice del semáforo
    accion.sem_op = -1; // operación wait (enteros negativos)
    accion.sem_flg = 0;

    semop(id_semaforo, &accion, 1);
}
```

3.3.- Eliminación de semáforos

La eliminación del conjunto de semáforos y sus estructuras de datos se lleva a cabo con el concurso de la llamada al sistema *semctl()* y el comando IPC_RMID. El identificador de usuario efectivo del proceso invocador debe ser el del super-usuario, o coincidir con el del creador o propietario del conjunto de semáforos. El argumento semnum se ignora.

```
#include <sys/ipc.h>
#include <sys/sem.h>

void main() {
    key_t clave;
    int id_semaforo;

    clave = ftok("./", 'kk');
    id_semaforo = semget(clave, 10, 0640 | IPC_CREAT);

    semctl( id_semaforo, 0, IPC_RMID);   // Eliminación del grupo semafórico
}
```

Los semáforos son entidades que sobreviven a la muerte de sus procesos creadores, como ocurre con los ficheros y fifos, así que se debe prestar una especial atención a liberar los recursos una vez que éstos no vayan a ser utilizados.

Cuestiones a resolver

1.- Construya un programa que cree una zona de memoria compartida de enteros del tamaño especificado como argumento (*tamano*), que tendrá como clave identificadora la que se pase como argumento (*clave*) y que estará en ejecución hasta que se pulse Ctrl+C. Al recibir esta señal, se liberará la memoria compartida y se finalizará ordenadamente la ejecución del proceso.

```
memoria clave tamano
```

2.- Construya un programa que permita acceder a la memoria compartida identificada por el argumento clave y actualice el valor del i-ésimo entero representado por *posicion* al valor indicado por el argumento *valor*. Téngase en cuenta que el acceso puede entrar en conflicto con otras operaciones de lectura y escritura en curso.

```
escribir clave posicion valor
```

3.- Construya un programa que permita acceder a la memoria compartida identificada por el argumento clave y obtenga el valor del i-ésimo entero representado por *posicion* y lo imprima por pantalla. Téngase en cuenta que el acceso puede entrar en conflicto con otras operaciones de lectura y escritura en curso.

```
leer clave posicion valor
```

4.- Construya un programa que inicialice la memoria compartida con el valor indicado por el argumento *valor*. Téngase en cuenta que el acceso puede entrar en conflicto con otras operaciones de lectura y escritura en curso.

```
inicializar clave valor tamano
```

Lectores / escritores

Semáforos: mutex, sem_escritura;
Int num_lecturas

Write_lock
wait(sem_escritura);

Write_unlock
signal(sem_escritura);

Read_lock
wait(mutex);
num_lecturas++;
If (num_lecturas == 1) wait(sem_escritura);
signal(mutex);

Read_unlock
wait(mutex);
num_lecturas--;
If (num_lecturas == 0) signal(sem_escritura);
signal(mutex);

Práctica 7: COMUNICACIÓN ENTRE PROCESOS: COLAS DE MENSAJES

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos mediante paso de mensajes empleando para ello colas de mensajes provistas por el sistema operativo Linux.

En la comunicación entre procesos mediante colas de mensajes, los procesos introducen mensajes en la cola y se van almacenando en ella. Cuando un proceso extrae un mensaje de la cola, extrae el primer mensaje que se introdujo y dicho mensaje se borra de la cola.

Las colas de mensajes permiten la definición de distintos tipos de mensajes, de tal forma que cada tipo de mensaje contiene una información distinta y va identificado por un entero largo, y lo que es más interesante, permite a los procesos retirar mensajes de la cola selectivamente según su tipo.

2.- Creación de colas de mensajes

A continuación se describe la forma de construir una cola de mensajes. En primer lugar se precisa una clave, de tipo **key_t**, que sea común para todos los procesos que quieran compartir la cola de mensajes. Para ello se emplea **ftok**. A dicha función se le pasa un fichero que exista y sea accesible y un entero. Si todos los procesos utilizan el mismo fichero y el mismo entero, obtendrán la misma clave.

Una vez obtenida la clave, se crea la cola de mensajes. Para ello está la función ***int msgget(key_t, int)***. Dicha función crea la cola y devuelve su identificador. Si la cola correspondiente a la *clave* ya existe, simplemente devuelve su identificador. El primer parámetro de *msgget* es la *clave* obtenida anteriormente y el segundo parámetro son unos *flags*. Aunque hay más posibilidades, lo imprescindible es:

- 9 bits menos significativos, son permisos de lectura/escritura/ejecución para propietario/grupo/otros, al igual que los ficheros. Para obtener una cola con todos los permisos para todo el mundo, debemos poner como parte de los flags el número **0777**. Es importante el cero delante, para que el número se interprete en octal y queden los bits en su sitio. El de ejecución se ignora.
- **IPC_CREAT**. Junto con los bits anteriores, este bit indica si se debe crear la cola en caso de que no exista. Si está puesto, la cola se creará si no lo está ya y se devolverá el identificador. Si no está puesto, se intentará obtener el identificador y se obtendrá un error si no está ya creada.

En resumen, los flags deberían ser algo así como 0666 | IPC_CREAT.

3.- Manejo de mensajes en las colas de mensajes

Para encolar un mensaje se utiliza la función ***msgsnd(int,struct msgbuf *,int,int)***. El primer parámetro entero es el identificador de la cola obtenido con **msgget()**.

El segundo parámetro es el mensaje en sí. El mensaje debe ser obligatoriamente una estructura cuyo primer campo sea un **long**. En dicho long se almacena el tipo de mensaje. El resto de los campos pueden ser cualquier cosa que se desee enviar (otra estructura, campos sueltos, etc). Al pasar el mensaje como parámetro, se pasa un puntero al mensaje y se le hace un *cast* a **struct msgbuf *.** No hay ningún problema en este *cast* siempre y cuando el primer campo del mensaje sea un **long**.

El tercer parámetro es el tamaño en bytes del mensaje exceptuando el **long**, es decir, el tamaño en bytes de los campos con la información. El cuarto parámetro son flags. Aunque hay varias opciones, la más habitual es poner un 0 o bien **IPC_NOWAIT**. En el primer caso la llamada a la función queda bloqueada hasta que se pueda enviar el mensaje. En el segundo caso, si el mensaje no se puede enviar, se vuelve inmediatamente con un error. El motivo habitual para que el mensaje no se pueda enviar es que la cola de mensajes esté llena.

Para desencolar un mensaje de la cola se emplea ***msgrcv(int, struct msgbuf *, int, int, int)***. El primer parámetro es el identificador de la cola obtenido con **msgget()**. El segundo parámetro es un puntero a la estructura donde se desea recoger el mensaje. Puede ser, como en la función anterior, cualquier estructura cuyo primer campo sea un long para el tipo de mensaje. El tercer parámetro entero es el tamaño de la estructura exceptuando el **long**. El cuarto parámetro entero es el tipo de mensaje que se quiere retirar. Se puede indicar un entero positivo para un tipo concreto o un 0 para cualquier tipo de mensaje. El quinto parámetro son los *flags*, que habitualmente puede ser 0 o bien **IPC_NOWAIT**. En el primer caso, la llamada a la función se queda bloqueada hasta que haya un mensaje del tipo indicado. En el segundo caso, se vuelve inmediatamente con un error si no hay mensaje de dicho tipo en la cola.

4.- Liberación de las colas de mensajes

Una vez terminada de usar la cola, se debe liberar. Para ello se utiliza la función **msgctl(int, int, struct msqid_ds *)**. Es una función genérica para el control de la cola de mensajes que permite varios comandos. En esta práctica sólo se explica cómo utilizarla para liberar la cola. El primer parámetro es el identificador de la cola de mensajes, obtenido con **msgget()**. El segundo parámetro es el comando que se desea ejecutar sobre la cola, en este caso **IPC_RMID**. El tercer parámetro son datos necesarios para el comando que se quiera ejecutar. En este caso no se necesitan datos y se pasará un **NULL**.

5.- Ejemplo de uso

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>


typedef struct mensaje_t {
   int idMensaje;
   double datoNumerico;
   char contenido[10];
} mensaje t;

struct msgbuf {
    long mtype;
    mensaje t mensaje;
};

int main(int argc, char **argv) {
   key_t clave;
   int idColaMensajes;
   struct msgbuf msgBuffer;

   clave = ftok ("/etc", 22);
   if (clave == (key_t) -1) exit(-1);

   idColaMensajes = msgget (clave, 0600 | IPC CREAT);
   if (idColaMensajes == -1) exit (-2);

   msgBuffer.mtype = 2;
   msgBuffer.mensaje.idMensaje = 1;
   msgBuffer.mensaje.datoNumerico = 11.3;
   strcpy (msgBuffer.mensaje.contenido, "Mensaje");

   msgsnd (idColaMensajes, &msgBuffer, sizeof(struct msgbuf)-sizeof(long), IPC NOWAIT);

   msgrcv (idColaMensajes, &msgBuffer, sizeof(struct msgbuf) -sizeof(long),, 2, 0);

   printf("                                                                    \n");
   printf("ID del mensaje: %ld\n", msgBuffer mensaje.idMensaje);
   printf("Dato del mensaje: %d\n", msgBuffer mensaje.datoNumerico);
   printf("Contenido del mensaje: %s\n", msgBuffer mensaje.contenido);
   printf("____________________________________________________________________\n");

   msgctl (idColaMensajes, IPC RMID, 0);
}
```

Cuestiones a resolver

1.- Construya dos programas que simulen el comportamiento productor/consumidor, de tal modo que uno de ellos produzca mensajes que serán consumidos por el otro programa (hasta una total de diez). Para ello se empleará una cola de mensajes siguiendo los ejemplos anteriormente descritos y el tiempo empleado por cada proceso para producir o consumir un mensaje se pasará como argumento.

```
productor clave_cola periodo
consumidor clave_cola periodo
```

2.- Construya un programa que cree cinco instancias del programa productor del ejercicio anterior con igual periodo de producción y con distintos periodos de producción y evalúe qué es lo que ocurre en ambos casos.

Práctica 8: PLANIFICACIÓN DE PROCESOS

1.- Introducción

El objetivo de esta práctica es profundizar en el conocimiento y manejo de la característica multiproceso de LINUX construyendo un planificador (*scheduler*) de procesos a alto nivel basado en señales, que implementa un sistema de colas de tres niveles NO APROPIATIVOS en el que los procesos de nivel 1 siguen una política ROUND ROBIN con turnos de 4 segundos, los procesos de nivel 2 siguen una política de planificación de PRIORIDADES no propiativas (menor valor, mayor prioridad) y los procesos de nivel 3 siguen una política FCFS (*first come first served*). Siendo los procesos de nivel 1 los de mayor prioridad, que deberán atenderse a la mayor celeridad, y los procesos de nivel 3 los de menos prioridad, que serán los últimos en atenderse.

2.- Planificador de procesos a nivel de usuario procsched

El planificador (*procsched*) se construirá con la ayuda de una cola de mensajes. Se dispondrá de un proceso que estará encargado de recibir las solicitudes de ejecución de procesos. Este proceso construirá una cola de mensajes (y la eliminará ordenadamente cuando proceda), construirá la estructura de datos correspondiente para cada proceso y la encolará en la cola de mensajes que proceda, creará un segundo proceso encargado de la planificación, y liberará todos los recursos (procesos y colas de mensajes) cuando se indique su finalización por parte del usuario.

El planificador admitirá la entrada de solicitudes tanto por teclado, como por un archivo de configuración. Cuando las solicitudes se obtengan por teclado, el final del fichero (EOF) se logra pulsando Ctrl+D.

PROCSCHED(1) PROCSCHED(1)

NOMBRE

procsched - *Scheduler* de procesos a nivel de usuario

SINOPSIS

procsched configfile]

DESCRIPCIÓN

El programa *procsched* crea las colas de mensajes y el planificador.
El programa *procsched* lanza los procesos que le han sido configurados y va encolando las peticiones de ejecución de los procesos en las colas de mensajes conforme recibe las solicitudes. El planificador va ejecutando los procesos encolados en las colas de mensajes atendiendo a la prioridad de cada nivel (la máxima prioridad corresponde al nivel más bajo).

El fichero de configuración configfile es opcional, de no estar presente se leerá la información de configuración de la entrada estándar, siguiendo el mismo formato que el fichero.
Formato del fichero de configuración: Consta de una línea por cada programa a ejecutar. Cada línea es de la siguiente forma:

nivel prioridad nombreprograma argumento1 argumento2 argumento3 …

Donde:
- nivel: indica el nivel de ejecución de cada proceso. Se considerarán 3 niveles NO APROPIATIVOS. Los procesos de nivel 1 siguen una política de planificación ROUND ROBIN con turnos de 4 segundos, los procesos de nivel 2 siguen una política de PRIORIDADES no apropiativas y los procesos de nivel 3 siguen una política FCFS (first come first served).
- prioridad: indica la prioridad de los procesos que sean de nivel 2. Para el resto de procesos, el valor de este argumento no se tendrá en cuenta, sea el valor que sea, se ignorará.
- nombreprograma argumento1 argumento2 argumento3 …: es el programa a ejecutar junto con sus argumentos, que pueden ser un número indeterminado y diferente para cada programa.

El programa *procsched* no finalizará hasta que el usuario pulse CTRL+C, momento en el que se terminará adecuadamente, liberando todos los recursos empleados, y se ofrecerá por pantalla el número de procesos que han concluido con éxito (diferenciando cuántos han seguido una planificación por prioridades, cuántos por FCFS y cuántos por round robin), el número de cambios de contexto que se han producido y el número total de procesos finalizados.

VALOR DEVUELTO

El planificador *procsched* indicará a su finalización el número de procesos que han concluido con éxito (diferenciando cuántos han seguido una planificación por prioridades, cuántos por FCFS y cuántos por round robin), el número de cambios de contexto que se han producido y el número total de procesos finalizados.

3.- Entrega de la práctica

En el fichero comprimido a entregar al concluir la práctica debe encontrarse el *Makefile* provisto, así como todos los ficheros .c y .h necesarios para crear *procsched*. La acción por defecto del *Makefile* (la cual debe funcionar con sólo hacer make en ese directorio) debe ser crear el ejecutable *procsched*. Para la corrección de la práctica se borrarán todos los ejecutables, se hará un *touch* a todos los ficheros fuente y se recompilará mediante el *Makefile*.

Se sugiere emplear la librería *fragmenta.o* ya empleada en la práctica anterior.

4.- Aplicaciones, funciones y llamadas al sistema útiles

kill(1), kill(2), sigaction(2), sleep(3), raise(3), signal(2), signal(7), sleep(2), nanosleep(2), pause(2), alarm(2), execvp(3), fork(2), wait(2).

Práctica 9: CONSTRUCCIÓN DE UN PEQUEÑO SISTEMA CONCURRENTE

1.- Introducción

El objetivo de esta práctica es construir de un pequeño sistema concurrente que permita poner en práctica todos los conceptos aprendidos en las prácticas anteriores.

2.- Desarrollo

Se desea construir un simulador de un sistema hidráulico mediante el empleo de procesos, mecanismos de comunicación entre procesos y llamadas al sistema. Para ello se requieren seis programas (ficheros independientes) que interactuarán entre sí como se indica a continuación. NO deberán emplearse esperas activas en NINGÚN CASO. Los programas NO podrán emplear variables globales.

```
struct mensaje {
  long tipo;
  int pid;
  char texto[100];
};
```

```
struct fluido{
  int contador, caudal;
};
```

El simulador representa el flujo de fluido a lo largo de todo un sistema hidráulico. Un **Gestor** prepara toda la infraestructura del sistema. **LlenaDeposito** se encarga de rellenar el depósito del proveedor. **Surtidor** extrae recursos del depósito del proveedor, incrementando así la capacidad del surtidor, y suministra fluido al Caudalímetro, alertando si su capacidad es insuficiente. El **Caudalímetro** reenvía el fluido al Sumidero, alertando si el caudal de fluido supera cierto umbral. Una vez el fluido llega al **Sumidero**, este decrementa la capacidad del surtidor e informa a Monitor de ello. **Monitor** recibe e imprime las alertas del sistema hidráulico e informa, bajo petición del Sumidero, de la capacidad actual del surtidor.

gestor *gestor clave periodo volumen umbral*

Gestor se encarga, mediante el uso del argumento clave de crear y destruir todos los recursos e infraestructura para el correcto funcionamiento del simulador. Gestor creará una cola de mensajes en la que almacenar los mensajes de alerta, una memoria compartida que representa la capacidad del surtidor (número de litros, un entero, inicializado a 0) y un grupo semafórico con dos semáforos para controlar, respectivamente, el volumen de líquido disponible en el depósito del proveedor y el acceso en exclusión mutua a la memoria compartida (capacidad del surtidor).
Además, gestor creará un proceso *surtidor* proveyéndole los argumentos clave, periodo y volumen; un proceso *caudalímetro*, proveyéndole los argumentos clave y umbral; un

proceso *monitor*, proveyéndole el argumento clave; y un proceso *sumidero*, proveyéndole los argumentos clave y pid_monitor; interconectando adecuadamente mediante dos tuberías (pipes) las salidas y entradas estándar de surtidor-caudalímetro-sumidero, tal y como se muestra en la figura. Tras esto, gestor permanecerá a la espera (nunca será una espera activa) de la recepción de la señal SIGINT (^C). Cuando reciba esta señal, procederá a eliminar, de forma ordenada, la cola de mensajes, la memoria compartida, el grupo semafórico, y los procesos correspondientes a *surtidor*, *caudalímetro*, *monitor* y *sumidero*.

surtidor ***surtidor clave periodo volumen***

Surtidor consumirá volumen litros del depósito del proveedor, operación que repetirá cada periodo segundos. Para ello, periódicamente, esperará el periodo indicado y, siempre que sea posible, descontará del primer semáforo (depósito del proveedor) el volumen de líquido indicado; escribirá en su salida estándar un *struct fluido*, en el que hará constar el volumen de litros traspasados (*caudal*), así como el número de descarga (*contador*, un contador que se incrementará en una unidad con cada traspaso). A continuación, incrementará la capacidad del surtidor. Para ello, bloqueará el segundo semáforo, que garantiza el acceso en exclusión mutua a la memoria compartida (capacidad surtidor), incrementará en volumen litros el valor de la memoria compartida y liberará el semáforo al terminar.

Si no fuera posible suministrar el volumen indicado, generará un mensaje de alerta que enviará a la cola de mensajes empleada para gestionar las alertas. El formato del mensaje será el de *struct mensaje*, e incluirá el PID del proceso que genera el mensaje, el tipo de mensaje y el texto descriptivo de la incidencia detectada. Para surtidor el tipo de mensaje siempre será 1, y el mensaje "Problema de suministro en el surtidor PID, caudal insuficiente", donde PID será el identificador de proceso del surtidor.

Caudalimetro ***caudalimetro clave umbral***

Caudalimetro realizará continuamente lecturas en la entrada estándar de tipo *struct fluido*, esperará un segundo, imprimirá por pantalla un mensaje indicando el *caudal* de líquido que está circulando por la tubería en ese instante, escribirá en la salida estándar el *struct fluido* recibido y comprobará si el caudal del *struct fluido* supera el umbral especificado por argumento umbral. De ser así, construirá un mensaje de tipo *struct mensaje* y lo encolará en la cola de mensajes empleada para gestionar las alertas. El tipo de mensaje será 2, y el texto "Problema de caudal excesivo en PID", donde PID será el identificador de proceso del caudalímetro.

llenaDeposito ***llenaDeposito clave tiempo volumen***

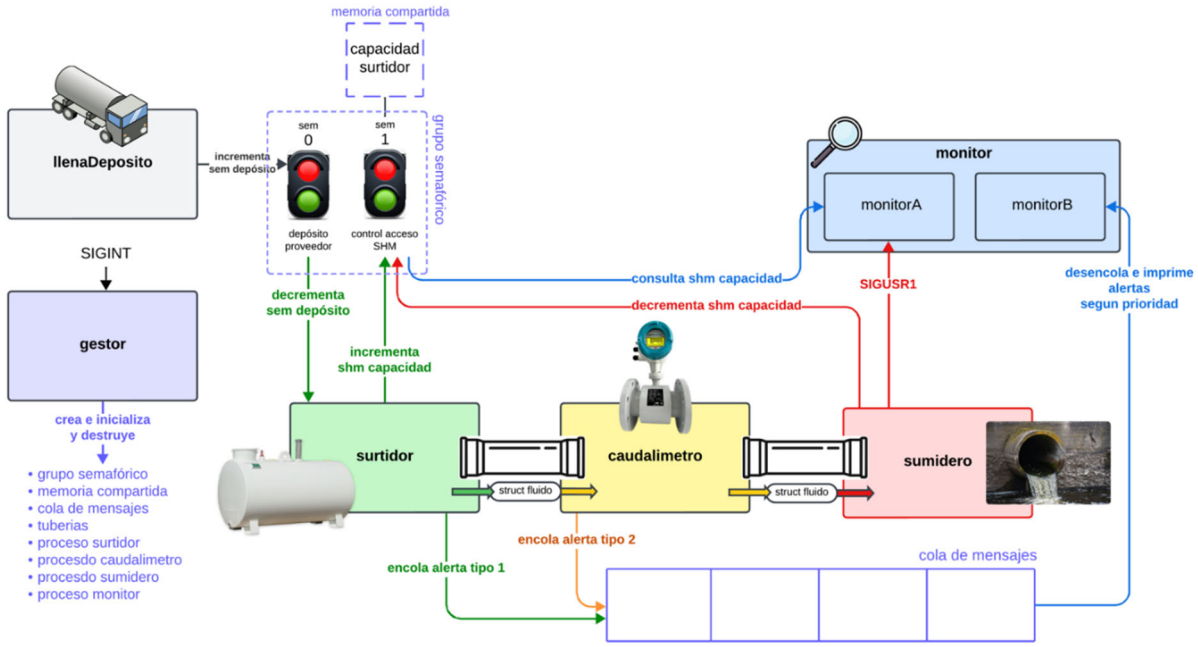
Este programa, esperará el periodo de tiempo indicado por el argumento tiempo, y llenará el deposito del proveedor con tantos litros como indique el argumento volumen. Para ello, realizará la espera indicada, accederá al primer semáforo, que controla el nivel de líquido disponible en el depósito del proveedor, y lo incrementará en tantos litros como indique el argumento volumen. Finalmente, concluirá su ejecución.

sumidero ***sumidero clave pid_monitor***

Sumidero realizará continuamente lecturas en la entrada estándar de tipo *struct fluido*. Para cada lectura esperará dos segundos, bloqueará el semáforo que garantiza el acceso en exclusión mutua de la memoria compartida (capacidad surtidor) y decrementará el valor de la memoria compartida en tantas unidades como indica el *caudal* del *struct fluido*. Posteriormente liberará el semáforo, enviará la señal SIGUSR1 al proceso cuyo identificador de proceso indique al argumento pid_monitor y continuará su ejecución esperando la llegada de nuevos mensajes.

Monitor ***monitor clave***

Monitor realizará dos tareas en paralelo, para lo que precisará crear un nuevo proceso. El primer proceso (padre), esperará continuamente la llegada de la señal SIGUSR1. Cada vez que la reciba, bloqueará el semáforo que garantiza el acceso en exclusión mutua a la memoria compartida (capacidad surtidor), imprimirá en pantalla el valor de la memoria compartida ("Presentes XXX litros en el surtidor"), liberará el semáforo al terminar y volverá a esperar la llegada de la señal.
El segundo proceso (hijo) leerá e imprimirá continuamente los mensajes que se vayan recibiendo en la cola de mensajes de alerta, atendiendo a su prioridad, siendo los mensajes de tipo 1 más prioritarios que los de tipo 2.



3.- Entrega de la práctica

En el fichero comprimido a entregar al concluir la práctica debe encontrarse, además de los citados cinco programas, el *Makefile* que permita la compilación conjunta de los programas, así como todos los ficheros .c y .h necesarios para su correcto

funcionamiento. La acción por defecto del *Makefile* (la cual debe funcionar con sólo hacer make en ese directorio) debe ser crear los ejecutables de los cinco programas anteriormente descritos. Para la corrección de la práctica se borrarán todos los ejecutables, se hará un *touch* a todos los ficheros fuente y se recompilará mediante el fichero *Makefile*.

Se sugiere emplear todos aquellos recursos disponibles de las prácticas anteriores.

Se valorarán especialmente la concisión, claridad, simplicidad y eficiencia en el código.

Anexo I: Ejemplos de interés

Ejemplo de utilización de memoria compartida en UNIX

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define ARRAY_SIZE   4000
#define MALLOC_SIZE 10000
#define SHM_SIZE    10000
#define SHM_MODE    (SHM_R | SHM_W)     /* read/write */

char array[ARRAY_SIZE];  /* datos sin inicializar = bss */

int main() {
   int shmid;
   char *ptr, *shmptr;

   printf("array[] desde %x hasta %x\n", &array[0], &array[ARRAY_SIZE]);
   printf("stack sobre %x\n", &;shmid);

   if ((ptr=malloc(MALLOC_SIZE)) == NULL)
      fprintf(stderr, "error de malloc()\n");
   printf("malloc desde %x hasta %x\n", ptr, ptr+MALLOC_SIZE);

   if ((shmid=shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE))<0)
      fprintf(stderr, "error de shmget()\n");
   if ((shmptr=shmat(shmid, 0, 0)) == (void *) -1)
      fprintf(stderr, "error de shmat()\n");
   printf("shared memory desde %x hasta %x\n", shmptr, shmptr+SHM_SIZE);
   /* proceso padre */
   if (0!=fork()) {
      while (*shmptr != 'x') ;
      printf("he comprobado 'x' en memoria compartida\npulse una tecla");
      getchar(); /* comprobar el semaforo con 'ipcs' */
      /* eliminar memoria compartida */
      if (shmctl(shmid, IPC_RMID, 0) < 0)
         fprintf(stderr,"error de shmctl()\n");

      exit(0);
   /* proceso hijo */
   } else {
      *shmptr = 'x';
      exit(0);
   }
}
```

rshmem.h : Fichero de cabecera con definiciones y declaraciones para usar memoria compartida.

```
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define ARRAY_SIZE   4000
#define MALLOC_SIZE 10000
#define SHM_SIZE    10000
#define SHM_MODE    (SHM_R | SHM_W)     /* read/write */

#define TRUE      1
#define FALSE     0

#ifdef RUTINAS_SHMEM
   static int shmid; /* handler de memoria compartida */
   char *memoria;    /* puntero a zona de memoria compartida */
#else
   extern char *memoria;
#endif

/* Prototipos de funciones de memoria compartida */

void origenTiempo();
void tiempoPasa();
int crearMemoria() ;
int eliminarMemoria() ;

#define TP tiempoPasa();
```

rshmem.c : Fichero con funciones de creación de memoria compartida y varias de utilidad.

```
#define RUTINAS_SHMEM

#include "rshmem.h"

/* Crea memoria compartida.
*    - el manejador de memoria es interno
*    - manda mensajes de error por salida de error estándar.
*/
int crearMemoria() {
   char *funcName = "crearMemoria";
   if ((shmid=shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE))<0) {
      fprintf(stderr, "%s: error de shmget()\n", funcName);
   } else if ((memoria=shmat(shmid, 0, 0)) == (void *) -1) {
      fprintf(stderr, "%s: error de shmat()\n", funcName);
   } else {
      return TRUE;
   }
   return FALSE;
}

/* Destruye la memoria compartida creada por crearMemoria()
*/
int eliminarMemoria() {
   char *funcName = "eliminarMemoria";
   if (shmctl(shmid, IPC_RMID, 0) < 0) {
      fprintf(stderr,"%s: error de shmctl()\n", funcName);
      return FALSE ;
   } else
      return TRUE ;
}


/* Coloca una semilla en el temporizador del bucle de
*  tiempoPasa()
*/
void origenTiempo(){
   srand((unsigned int) time(NULL)) ;
}

/* Rutina que hace pasar un poco de tiempo con un bucle
*  sencillo
*/
void tiempoPasa() {
   unsigned int i;
   int a=3;

   /* Los parametros "50" y "2" dependen mucho de la velocidad
   *  de la computadora y de la configuracion del SO. Espero que
   *  funcionen bien
   */
   for (i=rand()/50; i>0; i--)
      a = a%3 + i;
}
```

Ejemplo de dos procesos con condición de carrera

```
#include "rshmem.h"

void incrementa(int *mem, int k) {
   int i;
   i=*mem; TP
   i=i+k;  TP
   *mem=i;
}

int main() {
   int  *recurso;
   char *marcaFin;

   /* crear zona de memoria compartida */
   if (!crearMemoria())
      fprintf(stderr, "error de crearMemoria\n");

   recurso  = (int *) memoria ;
   marcaFin =  memoria + sizeof(int) ;
   *recurso =   0 ;
   *marcaFin = 'p' ;
   if (0!=fork()) {    /* proceso padre */
      int i;
      for (i=0; i< 1000; i++)
         incrementa(recurso, -5);
      while (*marcaFin != 'x') ; /* espera al hijo */

      printf("El recurso valia 0 y ahora vale %d\n", *recurso);
      if (!eliminarMemoria())   /* eliminar memoria compartida */
         fprintf(stderr, "error de eliminarMemoria\n");
      exit(0);
   } else {            /* proceso hijo */
      int i;
      for (i=0; i< 1000; i++)
         incrementa(recurso, 5);
      /* termina */
      *marcaFin = 'x';
      exit(0);
   }
}
```

Ejemplo de destrucción de memoria compartida en UNIX

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define ARRAY_SIZE 4000
#define MALLOC_SIZE 10000
#define SHM_SIZE 10000
#define SHM_MODE (SHM_R | SHM_W) /* read/write */

int main() {
   int  shmid;
   char shmidStr[128];

   printf("Que zona de memoria desea destruir? ");

   shmid = atoi(gets(shmidStr));
   if (shmctl(shmid, IPC_RMID, 0))<0)
      fprintf(stderr, "error de shmctl()\n");

   exit(0);
}
```