

CAPÍTULO 1

PROGRAMACIÓN MULTIPROCESO

Contenidos

Procesos. Estados de un proceso. Gestión de procesos en Linux.

Creación de procesos en Java.

Programación concurrente.

Programación paralela y distribuida. Ejecución de tareas en paralelo con PVM.

Objetivos

Conocer las características de un proceso y su ejecución por el sistema operativo.

Conocer las características y diferencias de la programación concurrente, paralela y distribuida.

Crear procesos en Linux y utilizar clases Java para crear procesos.

Desarrollar programas que ejecuten tareas en paralelo.

RESUMEN DEL CAPÍTULO

En este capítulo estudiaremos los procesos. Aprenderemos a gestionar los procesos que se están ejecutando en el sistema operativo. Utilizaremos clases para lanzar procesos. Conoceremos las características de la programación concurrente, paralela y distribuida y aprenderemos a crear programas paralelos con PVM.

1.1. INTRODUCCIÓN

Todos los ordenadores actuales realizan varias tareas a la vez, por ejemplo ejecutar un programa de procesador de textos, leer información de un disco duro, imprimir un documento por la impresora, visualizar información en pantalla, etc... Cuando un programa se carga en la memoria para su ejecución se convierte en un proceso.

En un sistema operativo **multiproceso** o **multitarea** se puede ejecutar más de un proceso (programa) a la vez, dando la sensación al usuario de que cada proceso es el único que se está ejecutando. La única forma de ejecutar varios procesos simultáneamente es tener varias CPUs (ya sea en una máquina o en varias). En los sistemas operativos con una única CPU se va alternando la ejecución de los procesos, es decir, se quita un proceso de la CPU, se ejecuta otro y se vuelve a colocar el primero sin que se entere de nada; esta operación se realiza tan rápido que parece que cada proceso tiene dedicación exclusiva.

La **programación multiproceso** tiene en cuenta la posibilidad de que múltiples procesos puedan estar ejecutándose simultáneamente sobre el mismo código de programa. Es decir desde una misma aplicación podemos realizar varias tareas de forma simultánea, o lo que es lo mismo, podemos dividir un proceso en varios subprocesos. En este capítulo aprenderemos a ejecutar varios procesos simultáneamente.

1.2. PROCESOS Y SISTEMA OPERATIVO

Se puede definir un **proceso** como un programa en ejecución. Consiste básicamente en el código ejecutable del programa, los datos y la pila del programa, el contador de programa, el puntero de pila y otros registros, y toda la información necesaria para ejecutar el programa.

Todos los programas que se ejecutan en el ordenador se organizan como un conjunto de procesos. El sistema operativo decide parar la ejecución de un proceso, por ejemplo, porque ha consumido su tiempo de CPU, y arrancar la de otro. Cuando se suspende temporalmente la ejecución de un proceso debe reanudarse posteriormente en el mismo estado en que encontraba cuando se paró, esto implica que toda la información referente al proceso debe almacenarse en alguna parte.

El **BCP** es una estructura de datos llamada *Bloque de Control de Proceso* donde se almacena información acerca de un proceso:

- Identificación del proceso. Cada proceso que se inicia es referenciado por un identificador único.
- Estado del proceso.
- Contador de programa.
- Registros de CPU.
- Información de planificación de CPU como la prioridad del proceso.
- Información de gestión de memoria.
- Información contable como la cantidad de tiempo de CPU y tiempo real consumido.
- Información de estado de E/S como la lista de dispositivos asignados, archivos abiertos, etc.

Mediante el comando **ps** (*process status*) de Linux podemos ver parte de la información asociada a cada proceso. El siguiente ejemplo muestra los procesos actualmente vivos en la máquina, se muestran 2 procesos ejecutándose, uno es el shell y el otro es la ejecución de la orden ps:

```

mj@ubuntu-mj:~$ ps
  PID TTY          TIME CMD
 1906 pts/0        00:00:00 bash
 2362 pts/0        00:00:00 ps

```

PID: identificador del proceso:

TTY: terminal asociado del que lee y al que escribe. Si no hay aparece interrogación.

TIME: tiempo de ejecución asociado, es la cantidad total de tiempo de CPU que el proceso ha utilizado desde que nació.

CMD: nombre del proceso.

La orden **ps -f** muestra más información:

```

mj@ubuntu-mj:~$ ps -f
  UID          PID    PPID  C STIME TTY          TIME CMD
  mj             1906    1298  0 18:06 pts/0        00:00:00 bash
  mj             2363    1906  0 18:33 pts/0        00:00:00 ps -f

```

UID: nombre de usuario

PPID: PID del padre de cada proceso.

C: porcentaje de recursos de CPU utilizado por el proceso.

STIME: hora de inicio del proceso.

La orden **ps -AF** muestra todos los procesos activos con todos los detalles; como en los ejemplos anteriores se puede observar que la última línea que aparece es la del comando que se está ejecutando:

```

mj@ubuntu-mj:~$ ps -AF
  UID          PID    PPID  C   SZ   RSS  PSR STIME TTY          TIME CMD
  root             1      0  0   703 1524   0 18:05 ?           00:00:00 /sbin/init
  root             2      0  0     0     0   0 18:05 ?           00:00:00 [kthreadd]
  root             3      2  0     0     0   0 18:05 ?           00:00:00 [migration/0]
  root             4      2  0     0     0   0 18:05 ?           00:00:00 [ksoftirqd/0]
  root             5      2  0     0     0   0 18:05 ?           00:00:00 [watchdog/0]
  root             6      2  0     0     0   0 18:05 ?           00:00:00 [events/0]
  root             7      2  0     0     0   0 18:05 ?           00:00:00 [cpuset]
  root             8      2  0     0     0   0 18:05 ?           00:00:00 [khelper]
  root             9      2  0     0     0   0 18:05 ?           00:00:00 [async/mgr]
  . . . . .
  postfix        1782     968  0  1457 1348   0 18:06 ?           00:00:00 pickup -l -t fi
  root           1786    1510  0   904 1080   0 18:06 ?           00:00:00 hald-addon-stor
  nobody         1846    1324  0 13116 2776   0 18:06 ?           00:00:00 /opt/lampp/bin/
  mj             1847      1  0 10495 9648   0 18:06 ?           00:00:02 /usr/lib/gnome-
  nobody         1849    1324  0 13048 2744   0 18:06 ?           00:00:00 /opt/lampp/bin/
  . . . . .
  mj             2497    2493  0   405   440   0 18:34 ?           00:00:00 heart -pid 2493
  mj             2557    2379  0  4263 10404   0 18:34 ?           00:00:01 /usr/bin/python
  root           2698     324  0   664   424   0 18:39 ?           00:00:00 udevd --daemon
  root           2699     324  0   664   448   0 18:39 ?           00:00:00 udevd --daemon
  mj             2720    1906  0   679  1044   0 18:49 pts/0        00:00:00 ps -AF

```

C: porcentaje de CPU utilizado por el proceso.

SZ: tamaño virtual de la imagen del proceso.

4 Programación de servicios y procesos

RSS: tamaño de la parte residente en memoria en kilobytes.

PSR: procesador que el proceso tiene actualmente asignado.

En Ubuntu escribiendo desde la terminal **sudo gnome-system-monitor** podemos acceder a la interfaz gráfica que nos muestra información sobre los procesos que se están ejecutando, véase Figura 1.1.

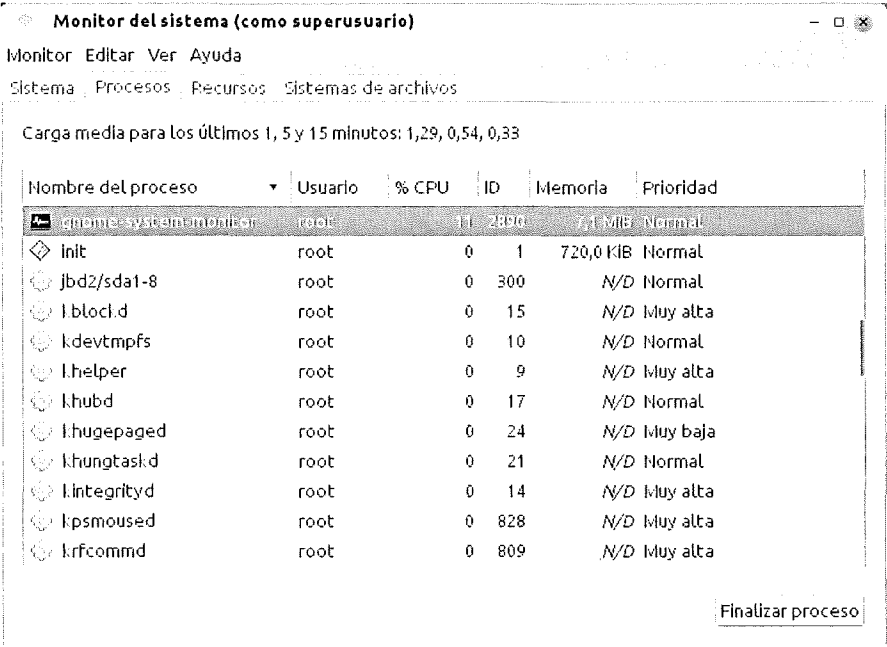


Figura 1.1. Monitor del sistema en Ubuntu.

En sistemas operativos Windows podemos usar desde la línea de comandos del DOS la orden **tasklist** para ver los procesos que se están ejecutando:

```
D:\>tasklist

Nombre de imagen          PID Nombre de sesión Núm. de Uso de memor
=====
System Idle Process      0 Console 0 28 KB
System                   4 Console 0 340 KB
smss.exe                 1528 Console 0 436 KB
csrss.exe                 1640 Console 0 8.052 KB
winlogon.exe             1664 Console 0 1.736 KB
services.exe             1708 Console 0 3.676 KB
lsass.exe                1720 Console 0 1.412 KB
nvsvc32.exe              1900 Console 0 5.488 KB
svchost.exe              1956 Console 0 5.536 KB
svchost.exe              2008 Console 0 4.736 KB
svchost.exe              1624 Console 0 31.096 KB
svchost.exe              248 Console 0 4.036 KB
svchost.exe              956 Console 0 4.056 KB
spoolsv.exe              1276 Console 0 6.932 KB
explorer.exe             1540 Console 0 43.888 KB
RTHDCPL.exe              1600 Console 0 23.004 KB
rundll32.exe             1636 Console 0 4.364 KB
. . . . .
```

La siguiente orden muestra los servicios que se están ejecutando bajo el proceso **svchost.exe**:

```
D:\>tasklist /svc /fi "imagenname eq svchost.exe"
```

Nombre de imagen	PID	Servicios
svchost.exe	1956	DcomLaunch, TermService
svchost.exe	2008	RpcSs
svchost.exe	1624	AudioSrv, Browser, CryptSvc, Dhcp, dmserver, EventSystem, FastUserSwitchingCompatibility, helpsvc, LanmanServer, lanmanworkstation, Netman, Nla, RasMan, Schedule, seclogon, SENS, SharedAccess, ShellHWDetection, TapiSrv, Themes, TrkWks, W32Time, winmgmt, wuauserv, WZCSVC
svchost.exe	248	Dnscache
svchost.exe	956	LmHosts, SSDPSRV
svchost.exe	1608	WebClient
svchost.exe	2872	stisvc
svchost.exe	6108	HTTPFilter

Aunque lo más típico es usar la combinación de teclas [CTRL+ Alt + Supr] para que se muestre la pantalla que da acceso al *Administrador de tareas de Windows*, véase Figura 1.2.

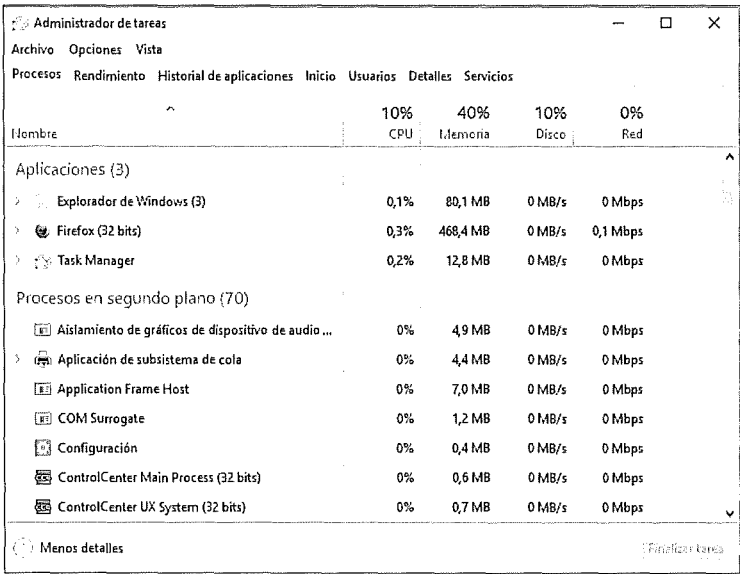


Figura 1.2. Administrador de tareas de Windows.

1.2.1. ESTADOS DE UN PROCESO

Un proceso, aunque es una entidad independiente, puede generar una salida que se use como entrada para otro proceso. Entonces este segundo proceso tendrá que esperar a que el primero termine para obtener los datos a procesar, en este caso debe bloquearse hasta que sus datos de entrada estén disponibles. Un proceso también se puede parar porque el sistema operativo decida asignar el procesador a otro proceso. En definitiva, los estados en los que se puede encontrar un proceso son los siguientes:

- **En ejecución:** el proceso está actualmente ejecutándose, es decir, usando el procesador.
- **Bloqueado:** el proceso no puede hacer nada hasta que no ocurra un evento externo, como por ejemplo la finalización de una operación de E/S.

- **Listo:** el proceso está parado temporalmente y listo para ejecutarse cuando se le de oportunidad.

La Figura 1.3 muestra mediante un diagrama de estados los estados en que se puede encontrar un proceso.

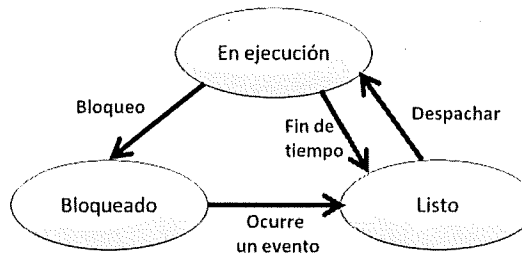


Figura 1.3. Estados de un proceso.

Las transiciones entre los estados son las siguientes:

- **En ejecución - Bloqueado:** un proceso pasa de ejecución a bloqueado cuando espera la ocurrencia de un evento externo.
- **Bloqueado - Listo:** un proceso pasa de bloqueado a listo cuando ocurre el evento externo que se esperaba.
- **Listo - En ejecución:** un proceso pasa de listo a ejecución cuando el sistema le otorga un tiempo de CPU.
- **En ejecución - Listo:** un proceso pasa de ejecución a listo cuando se le acaba el tiempo asignado por el sistema operativo.

1.2.2. CONTROL DE PROCESOS EN LINUX

Seguro que más de una vez hemos necesitado dentro de un programa ejecutar otro programa que realice alguna tarea concreta. Linux ofrece varias funciones para realizar esto: **system()**, **fork()** y **execl()**.

La función **system()** se encuentra en la librería estándar *stdlib.h* por lo que funciona en cualquier sistema operativo que tenga un compilador de C/C++ como por ejemplo Linux, Windows, etc. El formato es el siguiente:

```
int system(const char *cadena)
```

La función recibe como parámetro una cadena de caracteres que indica el comando que se desea procesar. Dicha instrucción es pasada al intérprete de comandos del ambiente en el que se esté trabajando y se ejecuta. Devuelve el valor -1 si ocurre un error y el estado devuelto por el comando en caso contrario.

La ejecución del siguiente ejemplo en C lista el contenido del directorio actual y lo envía a un fichero, abre el editor *gedit* con el fichero generado y ejecuta un comando que no existe en el intérprete de comandos de Linux:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    printf("Ejemplo de uso de system():");
    printf("\n\tListado del directorio actual y envío a un fichero:");
    printf("%d",system("ls > ficsalida"));
    printf("\n\tAbrimos con el gedit el fichero...");
}
```

```

printf("%d",system("gedit ficsalida"));
printf("\n\tEste comando es erróneo: %d",system("ged"));
printf("\nFin de programa....\n");
}

```

Lo compilamos y lo ejecutamos desde Linux (hemos de cerrar gedit para que el programa continúe):

```

mj@ubuntu-mj:~$ gcc ejemploSystem.c -o ejemploSystem
mj@ubuntu-mj:~$ ./ejemploSystem
Ejemplo de uso de system():
    Listado del directorio actual y envío a un fichero:0
sh: 1: gedit: not found
    Abrimos con el gedit el fichero...0
    Este comando es erróneo: 32512
Fin de programa....
mj@ubuntu-mj:~$

```

Esta función no se debe usar desde un programa con privilegios de administrador porque pudiera ser que se emplearan valores extraños para algunas variables de entorno y podrían comprometer la integridad del sistema. En este caso se utiliza **execl()**.

La función **execl()** tiene otras 5 funciones relacionadas (que no se tratarán en el tema). Realiza la ejecución y terminación del proceso. La llamada a la función **execl()** nos va a permitir sustituir el proceso llamante por uno nuevo. Su formato es:

```

#include <unistd.h>

int execl(const char *path_y_programa, const char *arg0, ...,
          char *argn, (char *)NULL);

```

La función recibe el nombre del programa que se va a ejecutar con su trayectoria y luego los argumentos del programa terminando con un puntero nulo. Devuelve -1 si ocurre algún error y en la variable global *errno* se pondrá el código de error adecuado. Por ejemplo, para ejecutar el comando `/bin/ls -l` escribimos lo siguiente:

```

execl("/bin/ls", "ls", "-l", (char *)NULL);

```

A continuación, se muestra un ejemplo de uso de la función:

```

#include <unistd.h>
#include <stdio.h>
void main()
{
    printf("Ejemplo de uso de exec():");
    printf("Los archivos en el directorio son:\n");
    execl("/bin/ls", "ls", "-l", (char *)NULL);
    printf(";;; Esto no se ejecuta !!!\n");
}

```

Compilamos y ejecutamos:

```

mj@ubuntu-mj:~$ gcc ejemploExec.c -o ejemploExec
mj@ubuntu-mj:~$ ./ejemploExec
Ejemplo de uso de exec():Los archivos en el directorio son:
total 176

```

8 Programación de servicios y procesos

```
-rwxr-xr-x 1 mj mj 7259 2012-11-15 17:17 a.out
-rw-r--r-- 1 root root 59105 2011-10-11 16:07 core.img
drwxr-xr-x 2 mj mj 4096 2012-11-14 22:24 Descargas
drwxr-xr-x 2 mj mj 4096 2012-09-19 20:29 Documentos
drwxrwsr-x 9 mj mj 4096 2012-11-15 00:36 eclipse
-rw-r--r-- 1 mj mj 919 2012-11-15 01:39 ejemplo.c
-rwxr-xr-x 1 mj mj 7256 2012-11-15 18:44 ejemploExec
-rwxr-xr-x 1 mj mj 1110 2012-11-15 18:43 ejemploExec.c
-rwxr-xr-x 1 mj mj 7259 2012-11-15 17:45 ejemploSystem
. . . . .
```

Observa que no se muestra el mensaje *!!! Esto no se ejecuta !!!* ya que esta función sustituye el proceso que hace la llamada por el que se ejecuta. El siguiente programa utiliza la función **execl()** para listar los procesos activos en el momento de la ejecución, prueba cambiando el path del comando, los argumentos, y comprueba los mensajes que se visualizan:

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
void main() {
    printf ("Lista de procesos\n");
    if (execl ("/bin/ps", "ps", "-f", (char *)NULL) < 0)
        printf("Error en exec %d\n", errno);
    else
        printf ("Fin de la lista de procesos\n");
}
```

Para crear nuevos procesos se dispone de la función **fork()** sin ningún tipo de parámetros y que se trata en el siguiente apartado.

1.2.2.1. Creación y ejecución de procesos

Hasta ahora hemos visto funciones que ejecutaban comandos ya sea del intérprete de comandos o de ficheros en disco, a continuación, veremos una función cuya misión es crear un proceso. Se trata de la función **fork()**. Su sintaxis es:

```
#include <unistd.h>

pid_t fork(void);
```

Al llamar a esta función se crea un nuevo proceso (proceso hijo) que es una copia exacta en código y datos del proceso que ha realizado la llamada (el proceso padre), salvo el PID y la memoria que ocupa. Las variables del proceso hijo son una copia de las del padre, por lo que modificar una variable en uno de los procesos no se refleja en el otro (ya que tienen distintas memorias). Una vez realizada la copia el proceso padre y el hijo continúan la ejecución en el mismo punto y a partir del **fork()**.

El valor devuelto por **fork()** es un valor numérico:

- Devuelve -1 si se produce algún error en la ejecución.
- Devuelve 0 si no se produce ningún error y nos encontramos en el proceso hijo.
- Devuelve el PID asignado al proceso hijo si no se produce ningún error y nos encontramos en el proceso padre.

Antes de hacer un ejemplo con la función **fork()** vamos a ver como obtener el identificador de un proceso o PID. Para ello usamos 2 funciones que devuelven un tipo *pid_t*. Las funciones son las siguientes:


```
pid_t getpid(void);
```

Devuelve el identificador del proceso que realiza la llamada, es decir, del proceso actual.

```
pid_t getppid(void);
```

Devuelve el identificador del proceso padre del proceso actual.

Veamos un simple ejemplo para ver los PID del proceso actual y del proceso padre:

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

void main(void)
{
    pid_t id_pactual, id_padre;

    id_pactual = getpid();
    id_padre = getppid();

    printf("PID de este proceso: %d\n", id_pactual);
    printf("PID del proceso padre: %d\n", id_padre);
}
```

Lo compilamos y ejecutamos. Se visualiza una salida similar a:

```
mj@ubuntu-mj:~$ gcc ejemploPadres.c -o ejemploPadres
mj@ubuntu-mj:~$ ./ejemploPadres
PID de este proceso: 2833
PID del proceso padre: 1923
```

Si ejecutamos el comando **ps** para ver los procesos que se están ejecutando, podemos ver que el PID del shell de Ubuntu (1923) coincide con el padre del proceso ejecutado anteriormente:

```
mj@ubuntu-mj:~$ ps
  PID TTY          TIME CMD
 1923 pts/0        00:00:00 bash
 2834 pts/0        00:00:00 ps
```

A continuación vamos a ver un ejemplo donde el proceso actual (proceso padre) crea un proceso (proceso hijo) con la función **fork()** (Figura 1.4):

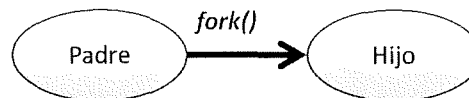


Figura 1.4. Proceso padre crea un hijo.

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

void main() {
    pid_t pid, Hijo_pid;
    pid = fork();

    if (pid == -1 ) //Ha ocurrido un error
    {
        printf("No se ha podido crear el proceso hijo...");
        exit(-1);
    }

    if (pid == 0 ) //Nos encontramos en Proceso hijo
```

10 Programación de servicios y procesos

```
{
    printf("Soy el proceso hijo \n\t
           Mi PID es %d, El PID de mi padre es: %d.\n",
           getpid(), getppid() );
}
else    //Nos encontramos en Proceso padre
{
    Hijo_pid = wait(NULL); //espera la finalización del proceso hijo
    printf("Soy el proceso padre:\n\t
           Mi PID es %d, El PID de mi padre es: %d.\n\t
           Mi hijo: %d terminó.\n",
           getpid(), getppid(), pid);
}
exit(0);
}
```

Lo compilamos y ejecutamos. Se visualiza una salida similar a:

```
mj@ubuntu-mj:~$ gcc ejemplo1Fork.c -o ejemplo1Fork
mj@ubuntu-mj:~$ ./ejemplo1Fork
Soy el proceso hijo
    Mi PID es 3098, El PID de mi padre es: 3097.
Soy el proceso padre:
    Mi PID es 3097, El PID de mi padre es: 1923.
    Mi hijo: 3098 terminó.
mj@ubuntu-mj:~$
```

En el código anterior se utiliza la función **wait()** para que el proceso padre espere la finalización del proceso hijo, el proceso padre quedará bloqueado hasta que termine el hijo. La sintaxis de la orden es la siguiente:

```
pid_t wait(int *status);
```

Devuelve el identificador del proceso hijo cuya ejecución ha finalizado. La sentencia **wait(NULL)** es la forma más básica de esperar a que un hijo termine.

Partiendo del ejemplo anterior, creamos un nuevo proceso en el proceso hijo; así tendremos el proceso padre (ABUELO), el proceso hijo (HIJO) y el proceso hijo del hijo (NIETO) (Figura 1.5):



Figura 1.5. Procesos Abuelo-Hijo-Nieto.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
//ABUELO-HIJO-NIETO
void main() {
    pid_t pid, Hijo_pid, pid2, Hijo2_pid;

    pid = fork(); //Soy el Abuelo, creo a Hijo

    if (pid == -1 ) //Ha ocurrido un error
    {
```

```

    printf("No se ha podido crear el proceso hijo...");
    exit(-1);
}

if (pid == 0 ) //Nos encontramos en Proceso hijo
{
    pid2 = fork();//Soy el Hijo, creo a Nieto
    switch(pid2)
    {
        case -1: // error
            printf("No se ha podido crear el proceso hijo
                en el HIJO...");
            exit(-1);
            break;
        case 0: // proceso hijo
            printf("\t\tSoy el proceso NIETO %d; Mi padre es = %d \n",
                getpid(), getppid());
            break;
        default: // proceso padre
            Hijo2_pid=wait(NULL);
            printf("\tSoy el proceso HIJO %d, Mi padre es: %d.\n",
                getpid(), getppid());
            printf("\tMi hijo: %d terminó.\n", Hijo2_pid);
    }
}

else //Nos encontramos en Proceso padre
{
    Hijo_pid = wait(NULL); //espera la finalización del proceso hijo
    printf("Soy el proceso ABUELO: %d, Mi HIJO: %d terminó.\n",
        getpid(), pid);
}
exit(0);
}

```

La compilación y ejecución muestra la siguiente salida:

```

mj@ubuntu-mj:~$ gcc ejemplo1_2Fork.c -o ejemplo1_2Fork
mj@ubuntu-mj:~$ ./ejemplo1_2Fork
    Soy el proceso NIETO 4486; Mi padre es = 4485
    Soy el proceso HIJO 4485, Mi padre es: 4484.
    Mi hijo: 4486 terminó.
Soy el proceso ABUELO: 4484, Mi HIJO: 4485 terminó.
mj@ubuntu-mj:~$

```

ACTIVIDAD 1.1

Realiza un programa en C que cree un proceso (tendremos 2 procesos uno padre y otro hijo). El programa definirá una variable entera y le dará el valor 6. El proceso padre incrementará dicho valor en 5 y el hijo restará 5. Se deben mostrar los valores en pantalla. A continuación, se muestra un ejemplo de la ejecución:

```

mj@ubuntu-mj:~$ gcc actividad1_1.c -o actividad1_1
mj@ubuntu-mj:~$ ./actividad1_1
Valor inicial de la variable: 6
Variable en Proceso Hijo: 1
Variable en Proceso Padre: 11
mj@ubuntu-mj:~$

```

1.2.2.2. Comunicación entre procesos

Existen varias formas de comunicación entre procesos (*Inter-Process Communication* o IPC) de Linux: pipes, colas de mensajes, semáforos y segmentos de memoria compartida. En este tema trataremos los mecanismos más sencillos, los **pipes** (*tuberías* en castellano).

PIPES SIN NOMBRE

Un *pipe* es una especie de falso fichero que sirve para conectar dos procesos, véase Figura 1.6. Si el proceso A quiere enviar datos al proceso B, los escribe en el *pipe* como si este fuera un fichero de salida. El proceso B puede leer los datos sin más que leer el *pipe* como si se tratara de un fichero de entrada. Así la comunicación entre procesos es parecida a la lectura y escritura en ficheros normales.

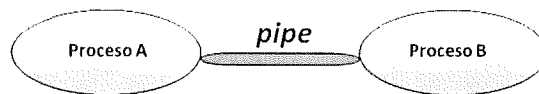


Figura 1.6. Dos procesos conectados por un *pipe*.

Cuando un proceso quiere leer del *pipe* y este está vacío, tendrá que esperar (es decir, se bloqueará) hasta que algún otro proceso ponga datos en él. Igualmente, cuando un proceso intenta escribir en el *pipe* y está lleno se bloqueará hasta que se vacíe. El pipe es bidireccional, pero, cada proceso lo utiliza en una única dirección, es este caso, el kernel gestiona la sincronización.

Para crear un *pipe* se realiza una llamada a la función **pipe()**:

```
#include <unistd.h>

int pipe(int fd[2]);
```

Esta función recibe un solo argumento, que es un array de dos enteros: *fd[0]* contiene el descriptor para lectura y *fd[1]* el de escritura. Si la función tiene éxito devuelve 0 y el array contendrá dos nuevos descriptors de archivos para ser usados por la tubería. Si ocurre algún error devuelve -1.

Para enviar datos al *pipe*, se usa la función **write()**, y para recuperar datos del *pipe*, se usa la función **read()**. La sintaxis es la siguiente:

```
int read( int fd, void *buf, int count );

int write( int fd, void *buf, int count );
```

read() intenta leer *count* bytes del descriptor de fichero definido en *fd*, para guardarlos en el buffer *buf*. Devuelve el número de bytes leídos; si comparamos este valor con la variable *count* podemos saber si ha conseguido leer tantos bytes como se pedían.

write() es muy similar. A *buf* le damos el valor de lo que queramos escribir, definimos su tamaño en *count* y especificamos el fichero en el que escribiremos en *fd*. Veamos a continuación un sencillo ejemplo que usa ficheros, pero antes se exponen las funciones que abren y cierran ficheros. Usamos la función **open()** para abrirlo y **close()** para cerrarlo, la sintaxis es la siguiente:

```
int open( const char *fichero, int modo );

int close(int fd);
```

open() abre el fichero indicado en la cadena *fichero* según el modo de acceso indicado en el entero *modo* (0 para lectura, 1 para escritura, 2 para lectura y escritura, etc). Devuelve -1 si ocurre algún error. Para cerrar el fichero usamos **close()** indicando entre paréntesis el descriptor de fichero a cerrar.

Se parte de la existencia de un fichero vacío de nombre *texto.txt*, el programa abre el fichero para escritura, escribe un saludo y después cierra el fichero. Posteriormente vuelve a abrir el fichero en modo lectura y hace un recorrido leyendo los bytes de uno en uno. Al finalizar la lectura se cierra el fichero. El programa es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
    char saludo[] = "Un saludo!!!\n";
    char buffer[10];
    int fd, bytesleidos;

    fd=open("texto.txt",1);//fichero se abre solo para escritura

    if( fd == -1 )
    {
        printf("ERROR AL ABRIR EL FICHERO...\n");
        exit(-1);
    }

    printf("Escribo el saludo...\n");
    write(fd,saludo, strlen(saludo));
    close(fd); //cierro el fichero

    fd=open("texto.txt",0);//el fichero se abre solo para lectura
    printf("Contenido del Fichero: \n");

    //leo bytes de uno en uno y lo guardo en buffer
    bytesleidos= read(fd, buffer, 1);
    while (bytesleidos!=0){
        printf("%1c", buffer[0]); //pinto el byte leído
        bytesleidos= read(fd, buffer, 1);//leo otro byte
    }
    close(fd);
}
```

La compilación y ejecución muestra la siguiente salida:

```
administrador@ubuntul:~$ gcc ejemWriteRead.c -o ejemWriteRead
administrador@ubuntul:~$ ./ejemWriteRead
Escribo el saludo...
Contenido del Fichero:
Un saludo!!!
administrador@ubuntul:~$
```

Una vez que sabemos cómo leer y escribir en ficheros veamos algunos ejemplos usando *pipes*. En el primer ejemplo se crea un proceso hijo con **fork()**. El proceso hijo envía al proceso padre mediante el uso de *pipes* el mensaje “*Hola papi*” en el descriptor para escritura *fd[1]*, el proceso padre mediante el descriptor *fd[0]* lee los datos enviados por el hijo:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
```

```

{
    int fd[2];
    char buffer[30];
    pid_t pid;

    pipe(fd); //se crea el pipe

    pid=fork(); //se crea el proceso hijo

    switch(pid) {
        case -1 : //ERROR
            printf("NO SE HA PODIDO CREAR HIJO...");
            exit(-1);
            break;
        case 0 : //HIJO
            printf("El HIJO escribe en el pipe...\n");
            write(fd[1], "Hola papi", 10);
            break;
        default : //PADRE
            wait(NULL); //espera que finalice proceso hijo
            printf("El PADRE lee del pipe...\n");
            read(fd[0], buffer, 10);
            printf("\tMensaje leído: %s\n",buffer);
            break;
    }
}

```

La compilación y ejecución muestra la siguiente salida:

```

administrador@ubuntul:~$ gcc ejemploPipe1.c -o ejemploPipe1
administrador@ubuntul:~$ ./ejemploPipe1
El HIJO escribe en el pipe...
El PADRE lee del pipe...
    Mensaje leído: Hola papi
administrador@ubuntul:~$

```

Primero se crea la tubería con **pipe()** y a continuación el proceso hijo. Recordemos que cuando se crea un proceso hijo con **fork()**, recibe una copia de todos los descriptores de ficheros del proceso padre, incluyendo copia de los descriptores de ficheros del pipe (*fd[0]* y *fd[1]*). Esto permite que el proceso hijo mande datos al extremo de escritura del pipe *fd[1]*, y el padre los reciba del extremo de lectura *fd[0]*, véase Figura 1.7.

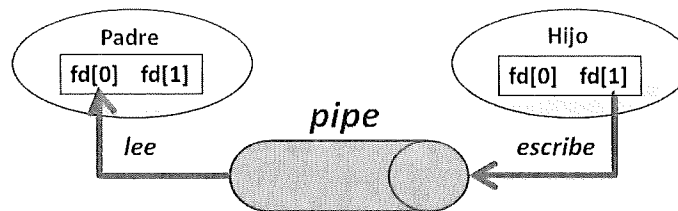


Figura 1.7. Pipe para un proceso hijo que escribe y otro padre que lee.

Los procesos padre e hijo están unidos por el pipe (Figura 1.8), pero la comunicación es en una única dirección, por tanto, se debe decidir en qué dirección se envía la información, del padre al hijo o del hijo al padre; y dado que los descriptores se comparten siempre debemos estar seguros de cerrar el extremo que no nos interesa.

Cuando el flujo de información va del padre hacia el hijo:

- El padre debe cerrar el descriptor de lectura *fd[0]*.
- El hijo debe cerrar el descriptor de escritura *fd[1]*.

Cuando el flujo de información va del hijo hacia padre ocurre lo contrario:

- El padre debe cerrar el descriptor de escritura *fd[1]*.
- El hijo debe cerrar el descriptor de lectura *fd[0]*.

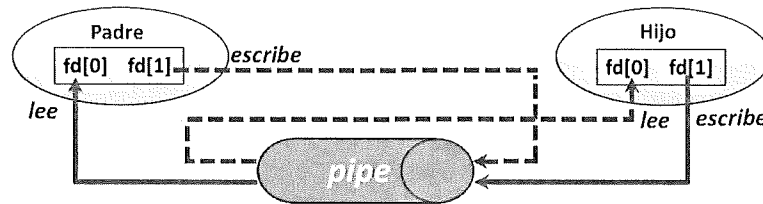


Figura 1.8. Pipe para un proceso padre e hijo que se envían datos.

El siguiente ejemplo crea un pipe en el que el padre envía un mensaje al hijo, el flujo de la información va del padre al hijo, el padre debe cerrar el descriptor *fd[0]* y el hijo *fd[1]*; el padre escribe en *fd[1]* y el hijo lee de *fd[0]*:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
int main(void)
{
    int fd[2];
    pid_t pid;
    char saludoPadre[]="Buenos dias hijo.\0";
    char buffer[80];

    pipe(fd); //creo pipe
    pid=fork(); //creo proceso

    switch(pid) {
        case -1 : //ERROR
            printf("NO SE HA PODIDO CREAR HIJO...");
            exit(-1);

        case 0 : //HIJO RECIBE
            close(fd[1]); //cierra el descriptor de entrada
            read(fd[0], buffer, sizeof(buffer)); //leo el pipe
            printf("\tEl HIJO recibe algo del pipe: %s\n",buffer);
            break;

        default : //PADRE ENVIA
            close(fd[0]);
            write(fd[1],saludoPadre,strlen(saludoPadre)); //escribo en pipe
            printf("El PADRE ENVIA MENSAJE AL HIJO...\n");
            wait(NULL); //espero al proceso hijo
            break;
    }
}
```

```

return 0;
}

```

La compilación y ejecución muestra la siguiente salida:

```

administrador@ubuntu1:~$ gcc ejemploPipe3.c -o ejemploPipe3
administrador@ubuntu1:~$ ./ejemploPipe3
El PADRE ENVIA MENSAJE AL HIJO...
    El HIJO recibe algo del pipe: Buenos dias hijo.
administrador@ubuntu1:~$

```

ACTIVIDAD 1.2

Siguiendo el ejemplo anterior, realiza un programa en C que cree un pipe en el que el hijo envíe un mensaje al padre, es decir la información fluya del hijo al padre. La ejecución debe mostrar la siguiente salida:

```

administrador@ubuntu1:~$ gcc actividad1_2.c -o actividad1_2
administrador@ubuntu1:~$ ./actividad1_2
    El HIJO envía algo al pipe.
El PADRE recibe algo del pipe: Buenos dias padre.
administrador@ubuntu1:~$

```

En el siguiente ejemplo vamos a hacer que padres e hijos puedan enviar y recibir información, como la comunicación es en un único sentido crearemos dos pipes *fd1* y *fd2*. Cada proceso usará un pipe para enviar la información y otro para recibirla. Partimos de los procesos ABUELO, HIJO y NIETO, la comunicación entre ellos se muestra en la Figura 1.9:

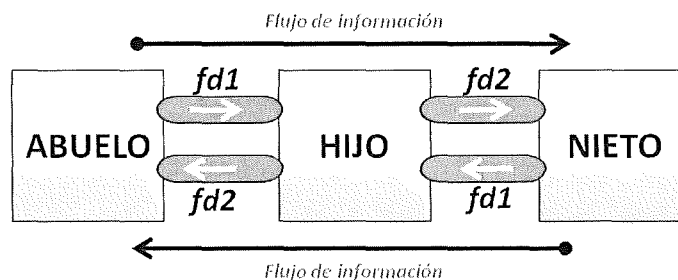


Figura 1.9. Pipe para un proceso padre e hijo que se envían datos.

- El ABUELO usará el *fd1* para enviar información al HIJO y recibirá la información de este a través del *fd2*.
- El HIJO usará el *fd2* para enviar información al NIETO y recibirá la información de este a través del *fd1*.
- El NIETO usará el *fd1* para enviar información al HIJO (su padre) y recibirá la información de este a través del *fd2*.

El código del programa es el siguiente:

```

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
//ABUELO-HIJO-NIETO
void main() {
    pid_t pid, Hijo_pid, pid2, Hijo2_pid;

```



```

int fd1[2];
int fd2[2];

char saludoAbuelo[] = "Saludos del Abuelo.\0";
char saludoPadre[] = "Saludos del Padre..\0";
char saludoHijo[] = "Saludos del Hijo...\0";
char saludoNieto[] = "Saludos del Nieto..\0";

char buffer[80] = "";

pipe(fd1); //pipe para comunicación de padre a hijo
pipe(fd2); //pipe para comunicación de hijo a padre

pid = fork(); //Soy el Abuelo, creo a Hijo

if (pid == -1 ) //Ha ocurrido un error
{
    printf("No se ha podido crear el proceso hijo...");
    exit(-1);
}

if (pid == 0 ) //Nos encontramos en Proceso hijo
{
    pid2 = fork(); //Soy el Hijo, creo a Nieto
    switch(pid2)
    {
        case -1: // error
            printf("No se ha podido crear el proceso hijo en el HIJO.");
            exit(-1);
            break;
        case 0: // proceso hijo (nieto)
            //NIETO RECIBE
            close(fd2[1]); //cierra el descriptor de entrada
            read(fd2[0], buffer, sizeof(buffer)); //leo el pipe
            printf("\t\tNIETO RECIBE mensaje de su padre:
                %s\n", buffer);

            //NIETO ENVIA
            printf("\t\tNIETO ENVIA MENSAJE a su padre...\n");
            close(fd1[0]);
            write(fd1[1], saludoNieto, strlen(saludoNieto));
            break;
        default: // proceso padre (hijo)
            //HIJO RECIBE
            close(fd1[1]); //cierra el descriptor de entrada
            read(fd1[0], buffer, sizeof(buffer)); //leo el pipe
            printf("\t\tHIJO recibe mensaje de ABUELO: %s\n", buffer);

            //HIJO ENVIA a su hijo
            close(fd2[0]);
            write(fd2[1], saludoPadre, strlen(saludoPadre));
            Hijo2_pid = wait(NULL); //espero al hijo

            //RECIBE de su hijo
            close(fd1[1]); //cierra el descriptor de entrada
            read(fd1[0], buffer, sizeof(buffer)); //leo el pipe

```

```

        printf("\tHIJO RECIBE mensaje de su hijo: %s\n",buffer);

        //HIJO ENVIA a su PADRE
        printf("\tHIJO ENVIA MENSAJE a su padre...\n",buffer);
        close(fd2[0]);
        write(fd2[1], saludoHijo, strlen(saludoHijo));
    }
}
else //Nos encontramos en Proceso padre (abuelo)
{
    //PADRE ENVIA
    printf("ABUELO ENVIA MENSAJE AL HIJO...\n");
    close(fd1[0]);
    write(fd1[1], saludoAbuelo, strlen(saludoAbuelo)); //escribo
    Hijo_pid = wait(NULL); //espera la finalización del hijo

    //PADRE RECIBE
    close(fd2[1]); //cierra el descriptor de entrada
    read(fd2[0], buffer, sizeof(buffer)); //leo el pipe
    printf("El ABUELO RECIBE MENSAJE del HIJO: %s\n", buffer);
}
exit(0);
}

```

La compilación y ejecución muestra la siguiente salida:

```

administrador@ubuntu1:~$ gcc ejemploForkPipe.c -o ejemploForkPipe
administrador@ubuntu1:~$ ./ejemploForkPipe
ABUELO ENVIA MENSAJE AL HIJO...
    HIJO recibe mensaje de ABUELO: Saludos del Abuelo.
        NIETO RECIBE mensaje de su padre: Saludos del Padre..
            NIETO ENVIA MENSAJE a su padre...
                HIJO RECIBE mensaje de su hijo: Saludos del Nieto..
                    HIJO ENVIA MENSAJE a su padre...
El ABUELO RECIBE MENSAJE del HIJO: Saludos del Hijo...
administrador@ubuntu1:~$

```

PIPES CON NOMBRE O FIFOS (First In First Out)

Los pipes vistos anteriormente establecían un canal de comunicación entre procesos emparentados (padre-hijo). Los FIFOS permiten comunicar procesos que no tienen que estar emparentados.

Un FIFO es como un fichero con nombre que existe en el sistema de ficheros y que pueden abrir, leer y escribir múltiples procesos. Los datos escritos se leen como en una cola, primero en entrar (*FIRST IN*), primero en salir (*FIRST OUT*); y una vez leídos no pueden ser leídos de nuevo. Los FIFOS tienen algunas diferencias con los ficheros:

- Una operación de escritura en un FIFO queda en espera hasta que el proceso pertinente abra el FIFO para iniciar la lectura.
- Sólo se permite la escritura de información cuando un proceso vaya a recoger dicha información.

Hay varias formas de crear un FIFO: ejecutando el comando **mkfifo** desde la línea de comandos de Linux o desde un programa C usando la función **mkfifo()**.

Para usar **mkfifo** desde la línea de comandos de Linux seguimos el siguiente formato:

```
mkfifo [opciones] nombreFichero p
```

Donde:

nombreFichero es el nombre del FIFO. Las *opciones* pueden ser:

-m modo, *--mode=modo*: establece los permisos de los ficheros creados según el valor de *modo* (su comportamiento es similar al del comando *chmod*).

--help: muestra en la salida estándar ayuda sobre el modo de empleo del comando, y luego finaliza.

--version: muestra en la salida estándar información sobre la versión, y luego finaliza.

El siguiente ejemplo crea un FIFO llamado FIFO1 desde la línea de comandos y luego se muestra la información del fichero creado. Se puede observar el indicador “p” que aparece en la lista del directorio y el símbolo de pipe | detrás del nombre:

```
administrador@ubuntu1:~$ mkfifo FIFO1
administrador@ubuntu1:~$ l -l FIFO1
prw-rw-r-- 1 administrador administrador 0 jul  4 15:35 FIFO1|
```

Desde el explorador de archivos el FIFO creado tiene el siguiente aspecto:



FIFO1 0 bytes tubería

A continuación, veamos cómo funciona el FIFO. Ejecuto desde la línea de comandos la orden **cat** con el nombre FIFO1:

```
administrador@ubuntu1:~$ cat FIFO1
```

Observamos que se queda a la espera. Abro una nueva terminal y ejecuto desde la línea de comandos la orden **/** para enviar la información del directorio al FIFO1:

```
administrador@ubuntu1:~$ l > FIFO1
```

Veremos que el **cat** que anteriormente estaba a la espera se ejecuta ya que ha recibido la información.

Para crear un FIFO en C, utilizamos la función **mkfifo()**. Su formato es el siguiente:

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

Donde:

pathname: es el nombre del dispositivo creado.

modo: especifica tanto los permisos de uso y el tipo de nodo que se creará. Debe ser una combinación (utilizando OR bit a bit) de uno de los tipos de fichero que se enumeran a continuación y los permisos para el nuevo nodo. El tipo de nodo debe ser uno de los siguientes:

- S_IFREG o 0: para especificar un fichero normal (que será creado vacío).
- S_IFCHR: para especificar un fichero especial de caracteres.
- S_IFBLK: un fichero especial de bloques.
- S_IFIFO: para crear un FIFO.

Si *pathname* ya existe, o es un enlace simbólico, esta llamada fallará devolviendo el error EEXIST.

La función **mkfifo()** devuelve 0 si ha funcionado correctamente, -1 si ha ocurrido un error.

A continuación, se muestra un ejemplo de uso de FIFOS. El programa *fifocrea.c* crea un FIFO de nombre *FIFO2* y lee la información del FIFO; mientras no hay información quedará en espera. El programa *fifoescribe.c* escribe información en el FIFO. La Figura 1.10 muestra la ejecución, primero se ejecuta *fifocrea* desde un terminal y después ejecutamos varias veces *fifoescribe* desde otro terminal.

```

administrador@ubuntu1: ~
Archivo Editar Ver Buscar Terminal Ayuda
administrador@ubuntu1:~$ gcc fifocrea.c -o fifocrea
administrador@ubuntu1:~$ ./fifocrea
OBTENIENDO Información...Un saludo!!!
OBTENIENDO Información...Un saludo!!!
OBTENIENDO Información...Un saludo!!!

administrador@ubuntu1: ~
Archivo Editar Ver Buscar Terminal Ayuda
administrador@ubuntu1:~$ gcc fifoescribe.c -o fifoescribe
administrador@ubuntu1:~$ ./fifoescribe
Mandando información al FIFO...
administrador@ubuntu1:~$ ./fifoescribe
Mandando información al FIFO...
administrador@ubuntu1:~$ ./fifoescribe
Mandando información al FIFO...
administrador@ubuntu1:~$

```

Figura 1.10. Ejecución de una FIFO.

El código es el siguiente:

```

//fifocrea.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(void)
{
    int fp;
    int p, bytesleidos;
    char saludo[] = "Un saludo!!!\n", buffer[10];

    p=mkfifo("FIFO2", S_IFIFO|0666); //permiso de lectura y escritura

    if (p== -1) {
        printf("HA OCURRIDO UN ERROR...\n");
        exit(0);
    }
}

```

```

    }

    while(1) {
        fp = open("FIFO2", 0);
        bytesleidos= read(fp, buffer, 1);
        printf("OBTENIENDO Información...");
        while (bytesleidos!=0){
            printf("%1c", buffer[0]); //pinto el byte leído
            bytesleidos= read(fp, buffer, 1); //leo otro byte
        }
        close(fp);
    }
    return(0);
}

```

```

//fifoescribe.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    int fp;
    char saludo[] = "Un saludo!!!\n";
    fp = open("FIFO2", 1);

    if(fp == -1) {
        printf("ERROR AL ABRIR EL FICHERO...");
        exit(1);
    }
    printf("Mandando información al FIFO...\n");
    write(fp, saludo, strlen(saludo));
    close(fp);
    return 0;
}

```

Con `mkfifo("FIFO2", S_IFIFO|0666)` se crea un FIFO de nombre *FIFO2* con permisos de lectura y escritura.

1.2.2.3. Sincronización entre procesos

En el epígrafe anterior se han tratado los mecanismos mas sencillos de comunicación entre procesos. Pero para que los procesos interactúen unos con otros necesitan cierto nivel de sincronización, es decir, es necesario que haya un funcionamiento coordinado entre los procesos a la hora de ejecutar alguna tarea. Podemos utilizar **señales** para llevar a cabo la sincronización entre dos procesos.

A continuación, se muestran una serie de funciones útiles que utilizaremos para que un proceso padre y otro hijo se comuniquen de forma síncrona usando señales.

Una señal es como un aviso que un proceso manda a otro proceso. La función `signal()` es el gestor de señales por excelencia que especifica la acción que debe realizarse cuando un proceso recibe una señal. Su formato es el siguiente:

```

#include <signal.h>

void (*signal(int Señal, void (*Func)(int)))(int);

```

Recibe dos parámetros:

Señal: contiene el número de señal que queremos capturar. En nuestro ejemplo pondremos SIGUSR1 que es una señal definida por el usuario para ser usada en programas de aplicación. Otra señal interesante es SIGKILL que se usa para terminar con un proceso.

Func: contiene la función a la que queremos que se llame. Esta función es conocida como el manejador de la señal (*signal handler*). En el ejemplo que se verá a continuación se definen dos manejadores de señal, uno para el proceso padre (*void gestion_padre(int signal)*) y otro para el hijo (*void gestion_hijo(int signal)*).

La función devuelve un puntero al manejador previamente instalado para esa señal. Un ejemplo de uso de la función: **signal(SIGUSR1, gestion_padre)**; significa que cuando el proceso (en este caso el proceso padre) recibe una señal SIGUSR1 se realizará una llamada a la función *gestion_padre()*.

Para enviar una señal usaremos la función **kill()**:

```
#include <signal.h>

int kill(int Pid, int Señal);
```

Recibe dos parámetros: el PID del proceso que recibirá la señal y la señal. Por ejemplo y suponiendo que *pid_padre* es el PID de un proceso padre: **kill(pid_padre, SIGUSR1)**; envía una señal SIGUSR1 al proceso padre.

Cuando queremos que un proceso espere a que le llegue una señal, usamos la función **pause()**. Para capturar esa señal, el proceso debe haber establecido un tratamiento de la misma con la función **signal()**. Este es su formato:

```
int pause(void);
```

Por último la función **sleep()** suspende al proceso que realiza la llamada la cantidad de segundos indicada o hasta que se reciba una señal.

```
#include <unistd.h>

unsigned int sleep (unsigned int seconds);
```

En el siguiente ejemplo se crea un proceso hijo y el proceso padre le va a enviar dos señales SIGUSR1. Se define la función *manejador()* para gestionar la señal, visualizará un mensaje cuando el proceso hijo la reciba. En el proceso hijo se realiza la llamada a **signal()** donde se decide lo que se hará en el caso de recibir una señal, en este caso pinta un mensaje. Después hacemos un bucle infinito que no hace nada para simular que el proceso hijo está haciendo algo.

En el proceso padre se hacen las llamadas a **kill()** para enviar las señales. Con la función **sleep()** hacemos que los procesos esperen un segundo antes de continuar.

El código es el siguiente:

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
/*-----*/
/* gestión de señales en proceso HIJO */
void manejador( int signal )
{
    printf("Hijo recibe señal..%d\n", signal);
}
```

```

/*-----*/
int main()
{
    int pid_hijo;
    pid_hijo = fork(); //creamos hijo

    switch(pid_hijo)
    {
        case -1:
            printf( "Error al crear el proceso hijo...\n");
            exit( -1 );
        case 0: //HIJO
            signal(SIGUSR1, manejador); //MANEJADOR DE SEÑAL EN HIJO
            while(1) {
            };
            break;
        default: //PADRE envia 2 señales
            sleep(1);
            kill(pid_hijo, SIGUSR1); //ENVIA SEÑAL AL HIJO
            sleep(1);
            kill(pid_hijo, SIGUSR1); //ENVIA SEÑAL AL HIJO
            sleep(1);
            break;
    }
    return 0;
}

```

La compilación y ejecución muestra la siguiente salida:

```

administrador@ubuntu1:~$ gcc sincronizar-1.c -o sincronizar-1
administrador@ubuntu1:~$ ./sincronizar-1
Hijo recibe señal..10
Hijo recibe señal..10
administrador@ubuntu1:~$

```

En el ejemplo que se muestra a continuación un proceso padre y otro hijo se ejecutan de forma síncrona. Se han definido dos funciones para gestionar la señal uno para el padre y otro para el hijo, con las acciones que se realizarán cuando los procesos reciban una señal; en este caso se visualizará un mensaje.

En primer lugar el proceso padre crea el proceso hijo. Dentro de cada proceso se realiza una llamada a **signal()** donde se decide lo que se hará en el caso de recibir una señal. En el proceso padre tenemos las siguientes instrucciones donde se observa que entra en bucle infinito esperando a recibir una señal. Cuando recibe la señal se ejecutaría la función *gestion_padre()*. Con **kill()** envía la señal de respuesta al proceso hijo mediante su PID, y el proceso se vuelve a repetir:

```

signal( SIGUSR1, gestion_padre );
while(1) {
    pause(); //padre espera hasta recibir una señal del hijo
    sleep(1);
    kill(pid_hijo, SIGUSR1); //ENVIA SEÑAL AL HIJO
}

```

En el proceso hijo también tenemos un trozo de código parecido, por la colocación de la función `pause()` se puede deducir que es el proceso hijo el que inicia la comunicación con el padre mediante la llamada `kill()`. Primero envía la señal al padre y después espera a que le llegue una señal de respuesta, cuando recibe la señal ejecutaría la función `gestion_hijo()`:

```
signal( SIGUSR1, gestion_hijo );
while(1) { //bucle infinito
    sleep(1);
    kill(pid_padre, SIGUSR1); //ENVIA SEÑAL AL PADRE
    pause(); //hijo espera hasta que llegue una señal de respuesta
}
```

El código completo es el siguiente:

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
/*-----*/
/* gestión de señales en proceso padre */
void gestion_padre( int signal )
{
    printf("Padre recibe señal..%d\n", signal);
}

/* gestión de señales en proceso hijo */
void gestion_hijo( int signal )
{
    printf("Hijo recibe señal..%d\n", signal);
}
/*-----*/

int main()
{
    int pid_padre, pid_hijo;

    pid_padre = getpid();
    pid_hijo = fork(); //se crea el hijo

    switch(pid_hijo)
    {
        case -1:
            printf( "Error al crear el proceso hijo...\n");
            exit( -1 );

        case 0: //HIJO
            //tratamiento de la señal en proceso hijo
            signal( SIGUSR1, gestion_hijo );
            while(1) { //bucle infinito
                sleep(1);
                kill(pid_padre, SIGUSR1); //ENVIA SEÑAL AL PADRE
                pause(); //hijo espera hasta que llegue una señal de respuesta
            }
            break;

        default: //PADRE
```

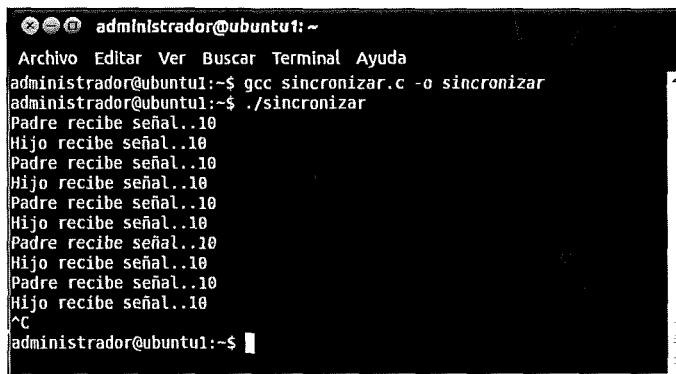


```

//tratamiento de la señal en proceso padre
signal( SIGUSR1, gestion_padre );
while(1) {
    pause();//padre espera hasta recibir una señal del hijo
    sleep(1);
    kill(pid_hijo, SIGUSR1);//ENVIA SEÑAL AL HIJO
}
break;
}
return 0;
}

```

La Figura 1.11 muestra un momento de la ejecución de los procesos.



```

administrador@ubuntu: ~
Archivo Editar Ver Buscar Terminal Ayuda
administrador@ubuntu:~$ gcc sincronizar.c -o sincronizar
administrador@ubuntu:~$ ./sincronizar
Padre recibe señal..10
Hijo recibe señal..10
Padre recibe señal..10
Hijo recibe señal..10
Padre recibe señal..10
Hijo recibe señal..10
Padre recibe señal..10
Hijo recibe señal..10
Padre recibe señal..10
Hijo recibe señal..10
^C
administrador@ubuntu:~$

```

Figura 1.11. Ejecución de procesos sincronizados.

Para detener el proceso podemos pulsar las teclas [CTRL+C] o bien mediante el comando **ps** podemos ver el PID de los procesos padre e hijo que se están ejecutando:

```

administrador@ubuntu:~$ ps -fe | grep sincronizar
1000      1678  1549  0 22:20 pts/0    00:00:00 ./sincronizar
1000      1679  1678  0 22:20 pts/0    00:00:00 ./sincronizar
1000      1687  1572  0 22:21 pts/1    00:00:00 grep --color=auto
                                           sincronizar

administrador@ubuntu:~$ kill 1679
administrador@ubuntu:~$ kill 1678
administrador@ubuntu:~$

```

Primero eliminaremos el proceso hijo (PID 1679) y después el padre (PID 1678). Al eliminar el hijo el padre quedará esperando.

ACTIVIDAD 1.3

Realiza un programa C en donde un hijo envíe 3 señales SIGUSR1 a su padre y después envíe una señal SIGKILL para que el proceso padre termine.

1.2.3. CREACIÓN DE PROCESOS CON JAVA

Java dispone en el paquete **java.lang** de varias clases para la gestión de procesos. Una de ellas es la clase **ProcessBuilder**. Cada instancia **ProcessBuilder** gestiona una colección de atributos del proceso. El método **start()** crea una nueva instancia de **Process** con esos atributos y puede ser invocado varias veces desde la misma instancia para crear nuevos subprocesos con atributos idénticos o relacionados.

El ejemplo es equivalente al siguiente:

```
public class Ejemplo1 {
    public static void main(String[] args) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("NOTEPAD");
        Process p = pb.start();
    }
} //Ejemplo1
```

Para los comandos de Windows que no tienen ejecutable (como por ejemplo DIR o ATTRIB) es necesario utilizar el comando CMD.EXE. Entonces para hacer un DIR desde un programa Java tendríamos que construir un objeto **ProcessBuilder** con los siguientes argumentos: "CMD", "/C" y "DIR".

Sabías que...

CMD Inicia una nueva instancia del intérprete de comandos de Windows. Para ver la sintaxis del comando escribimos desde el indicador del DOS: HELP CMD.

Para ejecutar un comando escribimos:

CMD /C comando: Ejecuta el comando especificado y luego finaliza.

CMD /K comando: Ejecuta el comando especificado, pero sigue activo.

El siguiente ejemplo ejecuta el comando DIR. Usaremos el método **getInputStream()** de la clase **Process** para leer el stream de salida del proceso, es decir, para leer lo que el comando DIR envía a la consola. Definiremos así el stream:

```
InputStream is = p.getInputStream();
```

Para leer la salida usamos el método **read()** de **InputStream** que nos devolverá carácter a carácter la salida generada por el comando. El programa Java es el siguiente:

```
import java.io.*;
public class Ejemplo2 {
    public static void main(String[] args) throws IOException {

        //Ejecutamos el proceso DIR
        Process p = new ProcessBuilder("CMD", "/C", "DIR").start();

        //Mostramos carácter a carácter la salida generada por DIR
        try {
            InputStream is = p.getInputStream();
            int c;
            while ((c = is.read()) != -1)
                System.out.print((char) c);
            is.close();

        } catch (Exception e) {
            e.printStackTrace();
        }

        //COMPROBACIÓN DE ERROR - 0 bien - 1 mal
```



```

//obtener la salida devuelta por el proceso
try {
    InputStream is = p.getInputStream();
    int c;
    while ((c = is.read()) != -1)
        System.out.print((char) c);
    is.close();
} catch (Exception e) {
    e.printStackTrace();
}
}
} // Ejemplo3

```

La salida mostrará los ficheros y carpetas del directorio definido en la variable *directorio*. Si ambos ficheros están en la misma carpeta o directorio, no será necesario establecer el directorio de trabajo para el objeto **ProcessBuilder**. Si el *Ejemplo2* a ejecutar se encontrase en la carpeta D:\PSP, tendríamos que definir el objeto *directorio* de la siguiente manera: *File directorio = new File("D:\PSP")*.

ACTIVIDAD 1.4

Crea un programa Java llamado *LeerNombre.java* que reciba desde los argumentos de *main()* un nombre y lo visualice en pantalla. Utiliza *System.exit(1)* para una finalización correcta del programa y *System.exit(-1)* para el caso que no se hayan introducido los argumentos correctos en *main()*.

A continuación, haz un programa parecido a *Ejemplo3.java* para ejecutar *LeerNombre.java*. Utiliza el método **waitFor()** para comprobar el valor de salida del proceso que se ejecuta. Prueba la ejecución del programa dando valor a los argumentos de *main()* y sin darle valor. ¿Qué valor devuelve **waitFor()** en un caso y en otro?

Realiza el Ejercicio 4.

La clase **Process** posee el método **getErrorStream()** que nos va a permitir obtener un stream para poder leer los posibles errores que se produzcan al lanzar el proceso. En el *Ejemplo2.java* si cambiamos los argumentos y escribimos algo incorrecto, por ejemplo lo siguiente:

```
Process p = new ProcessBuilder("CMD", "/C", "DIRR").start();
```

Al ejecutarlo aparecerá como valor de salida 1 indicando que el proceso no ha finalizado correctamente. Pero si añadimos el siguiente código al ejemplo:

```

try {
    InputStream er = p.getErrorStream();
    BufferedReader brer =
        new BufferedReader(new InputStreamReader(er));
    String liner = null;
    while ((liner = brer.readLine()) != null)
        System.out.println("ERROR >" + liner);
} catch (IOException ioe) {
    ioe.printStackTrace();
}
}

```

Se obtendrá la siguiente salida indicando el error que se ha producido:

```
ERROR >"DIRR" no se reconoce como un comando interno o externo,
ERROR >programa o archivo por lotes ejecutable.
Valor de Salida: 1
```

ACTIVIDAD 1.5

Partiendo del *Ejemplo3.java*, muestra los errores que se producen al ejecutar un programa Java que no exista.

Realiza los ejercicios 5 y 6

ENVIAR DATOS AL STREAM DE ENTRADA DEL PROCESO

Supongamos ahora que queremos ejecutar un proceso que necesita información de entrada. Por ejemplo, si ejecutamos DATE desde la línea de comandos y pulsamos la tecla [Intro] nos pide escribir una nueva fecha:

```
D:\CAPIT1>DATE
La fecha actual es: 14/06/2018
Escriba la nueva fecha: (dd-mm-aa) 15-06-18
```

La clase **Process** posee el método **getOutputStream()** que nos permite escribir en el stream de entrada del proceso, así podemos enviarle datos. El siguiente ejemplo ejecuta el comando DATE y le da los valores 15-06-18. Con el método **write()** se envían los bytes al stream, el método **getBytes()** codifica la cadena en una secuencia de bytes que utilizan juego de caracteres por defecto de la plataforma:

```
import java.io.*;
public class Ejemplo4 {
    public static void main(String[] args) throws IOException {

        Process p = new ProcessBuilder("CMD", "/C", "DATE").start();

        // escritura -- envia entrada a DATE
        OutputStream os = p.getOutputStream();
        os.write("15-06-18".getBytes());
        os.flush(); // vacía el buffer de salida

        // lectura -- obtiene la salida de DATE
        InputStream is = p.getInputStream();
        int c;
        while ((c = is.read()) != -1)
            System.out.print((char) c);
        is.close();

        // COMPROBACION DE ERROR - 0 bien - 1 mal
        int exitVal;
        try {
            exitVal = p.waitFor();
            System.out.println("Valor de Salida: " + exitVal);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
} //Ejemplo4
```

La ejecución muestra la siguiente salida:

```
La fecha actual es: 14/06/2018
Escriba la nueva fecha: (dd-mm-aa) 15-06-18
Valor de Salida: 0
```

Supongamos que tenemos un programa Java que lee una cadena desde la entrada estándar y la visualiza:

```
import java.io.*;
public class EjemploLectura{
    public static void main (String [] args)
    {
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader (in);
        String texto;
        try {
            System.out.println("Introduce una cadena....");
            texto= br.readLine();
            System.out.println("Cadena escrita: "+texto);
            in.close();
        }catch (Exception e) { e.printStackTrace();}
    }
} //EjemploLectura
```

Con el método `getOutputStream()` podemos enviar datos a la entrada estándar del programa *EjemploLectura.java*. Por ejemplo si queremos enviar la cadena “Hola Manuel” cambiaríamos varias cosas en el *Ejemplo4.java*:

```
File directorio = new File(".\\bin");
ProcessBuilder pb = new ProcessBuilder("java", "EjemploLectura");
pb.directory(directorio);

// se ejecuta el proceso
Process p = pb.start();

// escritura - se envia la entrada
OutputStream os = p.getOutputStream();
os.write("Hola Manuel\n".getBytes());
os.flush(); // vacía el buffer de salida
```

Cada línea que mandemos a *EjemploLectura* debe terminar con `"\n"`, igual que cuando escribimos desde el terminal la lectura termina cuando pulsamos la tecla [Intro]. Suponiendo que hemos guardado estos cambios en *Ejemplo5.java*, la ejecución muestra la siguiente salida:

```
Introduce una cadena....
Cadena escrita: Hola Manuel
Valor de Salida: 0
```

ACTIVIDAD 1.6

Escribe un programa Java que lea dos números desde la entrada estándar y visualice su suma. Controlar que lo introducido por teclado sean dos números. Haz otro programa Java para ejecutar el anterior. Realiza el ejercicio 7.

El siguiente ejemplo usa varios métodos de la clase **ProcessBuilder**: **environment()** que devuelve las variables de entorno del proceso; el método **command()** sin parámetros, que devuelve los argumentos del proceso definido en el objeto **ProcessBuilder**; y con parámetros donde se define un nuevo proceso y sus argumentos. Después se ejecutará este último proceso:

```
import java.io.*;
import java.util.*;

public class Ejemplo6 {
    public static void main(String args[]) {
        ProcessBuilder test = new ProcessBuilder();
        Map entorno = test.environment();
        System.out.println("Variables de entorno:");
        System.out.println(entorno);

        test = new ProcessBuilder("java", "LeerNombre", "Maria Jesús");

        // devuelve el nombre del proceso y sus argumentos
        List l = test.command();
        Iterator iter = l.iterator();
        System.out.println("\nArgumentos del comando:");
        while (iter.hasNext())
            System.out.println(iter.next());

        test = test.command("CMD", "/C", "DIR");
        try {
            Process p = test.start(); //se ejecuta DIR
            InputStream is = p.getInputStream();

            System.out.println();
            // mostramos en pantalla caracter a caracter
            int c;
            while ((c = is.read()) != -1)
                System.out.print((char) c);
            is.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
} // Ejemplo6
```

La compilación y ejecución muestra la siguiente salida:

```
Variables de entorno:
{configsetroot=C:\WINDOWS\ConfigSetRoot, USERDOMAIN_ROAMINGPROFILE=PC-
ASUS, PROCESSOR_LEVEL=6, FP_NO_HOST_CHECK=NO, SESSIONNAME=Console,
ALLUSERSPROFILE=C:\ProgramData, PROCESSOR_ARCHITECTURE=AMD64,
PSModulePath=C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\;C:\Pro
gram Files\VisualSVN Server\PowerShellModules, SystemDrive=C:,
JRE_HOME=C:\Program Files\Java\jre-10.0.1, USERNAME=mjesus,
. . . . .
. . . . . NUMBER_OF_PROCESSORS=4}

Argumentos del comando:
java
LeerNombre
```

Maria Jesús

El volumen de la unidad D es Data
El número de serie del volumen es: 9013-7A66

Directorio de D:\CLASE\PSP_2018\CAPITULO1

```
14/06/2018  00:14    <DIR>          .
14/06/2018  00:14    <DIR>          ..
14/06/2018  00:14                396 .classpath
14/06/2018  00:14                385 .project
14/06/2018  00:14    <DIR>          .settings
14/06/2018  00:15    <DIR>          bin
14/06/2018  00:15    <DIR>          src
                2 archivos          781 bytes
                5 dirs  134.146.215.936 bytes libres
```

REDIRECCIONANDO LA ENTRADA Y LA SALIDA

Los métodos `redirectOutput()` y `redirectError()` nos permiten redirigir la salida estándar y de error a un fichero. El siguiente ejemplo ejecuta el comando `DIR` y envía la salida al fichero *salida.txt*, si ocurre algún error se envía a *error.txt*:

```
import java.io.File;
import java.io.IOException;

public class Ejemplo7 {
    public static void main(String args[]) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");

        File fOut = new File("salida.txt");
        File fErr = new File("error.txt");

        pb.redirectOutput(fOut);
        pb.redirectError(fErr);
        pb.start();
    }
} // Ejemplo7
```

También podemos ejecutar varios comandos del sistema operativo dentro de un fichero BAT. El siguiente ejemplo ejecuta los comandos MS-DOS que se encuentran en el fichero *fichero.bat*. Se utiliza el método `redirectInput()` para indicar que la entrada al proceso se encuentra en un fichero, es decir la entrada para el comando `CMD` será el *fichero.bat*. La salida del proceso se envía al fichero *salida.txt* y la salida de error al fichero *error.txt*:

```
import java.io.File;
import java.io.IOException;

public class Ejemplo8 {
    public static void main(String args[]) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD");

        File fBat = new File("fichero.bat");
        File fOut = new File("salida.txt");
        File fErr = new File("error.txt");
```



```

        pb.redirectInput(fBat);
        pb.redirectOutput(fOut);
        pb.redirectError(fErr);
        pb.start();
    }
} // Ejemplo8

```

Suponiendo que los comandos MS-DOS del *fichero.bat* son estos (este fichero se debe crear en el proyecto Eclipse):

```

MKDIR NUEVO
CD NUEVO
ECHO CREO FICHERO > Mifichero.txt
DIR
DIRR
ECHO FIN COMANDOS

```

Donde se crea una carpeta, nos dirigimos a dicha carpeta, se crea el fichero *Mifichero.txt*, se hace un DIR del directorio actual, el siguiente comando DIRR es erróneo y se visualiza FIN COMANDOS. Al ejecutarlo desde el entorno Eclipse el contenido del fichero de salida *salida.txt* es el siguiente:

```

Microsoft Windows [Versión 10.0.17134.48]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

D:\CLASE\PSP_2018\CAPITULO1>MKDIR NUEVO

D:\CLASE\PSP_2018\CAPITULO1>CD NUEVO

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>ECHO CREO FICHERO > Mifichero.txt

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>DIR
El volumen de la unidad D es Data
El número de serie del volumen es: 9013-7A66

Directorio de D:\CLASE\PSP_2018\CAPITULO1\NUEVO
14/06/2018  00:34    <DIR>          .
14/06/2018  00:34    <DIR>          ..
14/06/2018  00:34                15 Mifichero.txt
                   1 archivos                15 bytes
                   2 dirs  134.146.215.936 bytes libres

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>DIRR

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>ECHO FIN COMANDOS
FIN COMANDOS

D:\CLASE\PSP_2018\CAPITULO1\NUEVO>

```

Y el del fichero de error *error.txt*:

"DIRR" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

ACTIVIDAD 1.7

Modifica el *Ejemplo5.java* para que la salida del proceso y la salida de error se almacenen en un fichero de texto, y la entrada la tome desde otro fichero de texto.

Para llevar a cabo el redireccionamiento, tanto de entrada como de salida del proceso que se ejecuta, también podemos usar la clase **ProcessBuilder.Redirect**. El redireccionamiento puede ser uno de los siguientes:

- El valor especial **Redirect.INHERIT**, indica que la la fuente de entrada y salida del proceso será la misma que la del proceso actual.
- **Redirect.from (File)**, indica redirección para leer de un fichero, la entrada al proceso se encuentra en el objeto **File**.
- **Redirect.to(File)**, indica redirección para escribir en un fichero, el proceso escribirá en el objeto **File** especificado.
- **Redirect.appendTo (File)**, indica redirección para añadir a un fichero, la salida del proceso se añadirá al objeto **File** especificado.

El ejemplo anterior usando esta clase quedaría de esta manera:

```
pb.redirectInput(ProcessBuilder.Redirect.from(fBat));
pb.redirectOutput(ProcessBuilder.Redirect.to(fOut));
pb.redirectError(ProcessBuilder.Redirect.to(fErr));
```

El siguiente ejemplo muestra en la consola la salida del comando DIR,

```
import java.io.IOException;
public class Ejemplo9 {
    public static void main(String args[]) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
        pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
        Process p = pb.start();
    }
} // Ejemplo9
```

ACTIVIDAD 1.8

Usando **ProcessBuilder.Redirect**, modifica el *Ejemplo5.java* para que la salida del proceso se muestre en la consola, la entrada la tome desde un fichero de texto, y la salida la lleve a un fichero de texto. Realiza los ejercicios 7, 8 y 9.

1.3. PROGRAMACIÓN CONCURRENTES

El diccionario *WordReference.com* (<http://www.wordreference.com/definicion/>) nos muestra varias acepciones de la palabra concurrencia. Nos quedamos con la tercera: “*Acaecimiento o concurso de varios sucesos en un mismo tiempo*”. Si sustituimos sucesos por procesos ya tenemos una aproximación de lo que es la concurrencia en informática: la existencia simultánea de varios procesos en ejecución.

1.3.1. PROGRAMA Y PROCESO

Al principio del tema se definió un **proceso** como un programa en ejecución. Y ¿qué es un programa?, podemos definir **programa** como un conjunto de instrucciones que se aplican a un conjunto de datos de entrada para obtener una salida. Un proceso es algo activo que cuenta con una serie de recursos asociados, en cambio un programa es algo pasivo, para que pueda hacer algo hay que ejecutarlo.

Pero un programa al ponerse en ejecución puede dar lugar a más de un proceso, cada uno ejecutando una parte del programa. Por ejemplo, el navegador web, por un lado está controlando las acciones del usuario con la interfaz, por otro hace las peticiones al servidor web. Entonces cada vez que se ejecuta este programa crea 2 procesos.

En la Figura 1.12 existe un programa almacenado en disco y 3 instancias del mismo ejecutándose, por ejemplo, por 3 usuarios diferentes. Cada instancia del programa es un proceso, por tanto, existen 3 procesos independientes ejecutándose al mismo tiempo sobre el sistema operativo, tenemos entonces 3 procesos concurrentes.

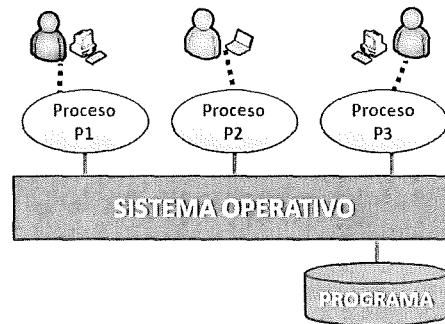


Figura 1.12. Un programa con 3 instancias ejecutándose.

Dos procesos serán concurrentes cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última. Es decir, existe un solapamiento o intercalado en la ejecución de sus instrucciones. No hay que confundir el solapamiento con la ejecución simultánea de las instrucciones, en este caso estaríamos en una situación de **programación paralela**, aunque a veces el hardware subyacente (más de un procesador) sí permitirá la ejecución simultánea.

Supongamos ahora que el programa anterior al ejecutarse da lugar a 2 procesos más, cada uno ejecutando una parte del programa, entonces la Figura 1.12 se convierte en la 1.13. Ya que un programa puede estar compuesto por diversos procesos, una definición más acertada de proceso es la de una actividad asíncrona susceptible de ser asignada a un procesador¹.

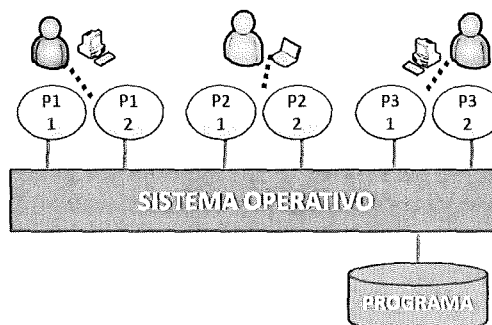


Figura 1.13. Un programa dando lugar a más de un proceso.

Cuando varios procesos se ejecutan concurrentemente puede haber procesos que colaboren para un determinado fin (por ejemplo, P1.1 y P1.2), y otros que compitan por los recursos del sistema (por ejemplo P2.1 y P3.1). Estas tareas de colaboración y competencia por los recursos exigen **mecanismos de comunicación y sincronización entre procesos**.

¹ Programación concurrente. José Tomás Palma Méndez y otros. Ed Paraninfo. ISBN: 9788497321846

1.3.2. CARACTERÍSTICAS

La **programación concurrente** es la disciplina que se encarga del estudio de las notaciones que permiten especificar la ejecución concurrente de las acciones de un programa, así como las técnicas para resolver los problemas inherentes a la ejecución concurrente (comunicación y sincronización).

BENEFICIOS

La programación concurrente aporta una serie de beneficios:

Mejor aprovechamiento de la CPU. Un proceso puede aprovechar ciclos de CPU mientras otro realiza una operación de entrada/salida.

Velocidad de ejecución. Al subdividir un programa en procesos, éstos se pueden “repartir” entre procesadores o gestionar en un único procesador según importancia.

Solución a problemas de naturaleza concurrente. Existen algunos problemas cuya solución es más fácil utilizando esta metodología:

- **Sistemas de control:** son sistemas en los que hay captura de datos, normalmente a través de sensores, análisis y actuación en función del análisis. Un ejemplo son los sistemas de tiempo real.
- **Tecnologías web:** los servidores web son capaces de atender múltiples peticiones de usuarios concurrentemente, también los servidores de chat, correo, los propios navegadores web, etc.
- **Aplicaciones basadas en GUI:** el usuario puede interactuar con la aplicación mientras la aplicación está realizando otra tarea. Por ejemplo, el navegador web puede estar descargando un archivo mientras el usuario navega por las páginas.
- **Simulación:** programas que modelan sistemas físicos con autonomía.
- **Sistemas Gestores de Bases de Datos:** Los usuarios interactúan con el sistema, cada usuario puede ser visto como un proceso.

CONCURRENCIA Y HARDWARE

En un sistema **monoprocesador** (de un solo procesador) se puede tener una ejecución concurrente gestionando el tiempo de procesador para cada proceso. El S.O. va alternando el tiempo entre los distintos procesos, cuando uno necesita realizar una operación de entrada salida, lo abandona y otro lo ocupa; de esta forma se aprovechan los ciclos del procesador. En la Figura 1.14 se muestra como el tiempo de procesador es repartido entre 3 procesos, en cada momento sólo hay un proceso. Esta forma de gestionar los procesos en un sistema monoprocesador recibe el nombre de **multiprogramación**.

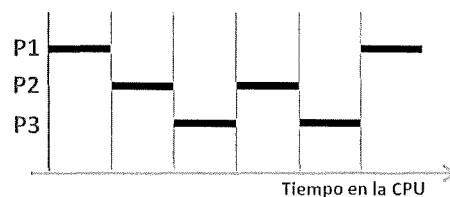


Figura 1.14. Concurrencia.

En un sistema **monoprocesador** todos los procesos comparten la misma memoria. La forma de comunicar y sincronizar procesos se realiza mediante variables compartidas.

En un sistema **multiprocesador** (existe más de un procesador) podemos tener un proceso en cada procesador. Esto permite que exista paralelismo real entre los procesos, véase Figura 1.15.

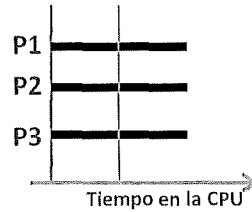


Figura 1.15. Paralelismo.

Estos sistemas se pueden clasificar en:

- Fuertemente acoplados: cuando poseen una memoria compartida por todos los procesadores, véase Figura 1.16.
- Débilmente acoplados: cuando los procesadores poseen memorias locales y no existe la compartición de memoria, véase Figura 1.17.

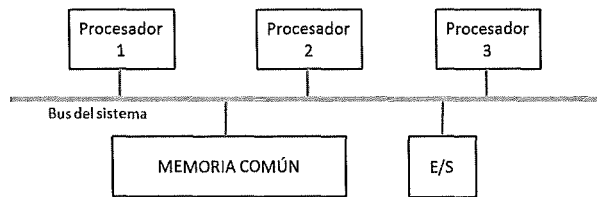


Figura 1.16. Fuertemente acoplados.

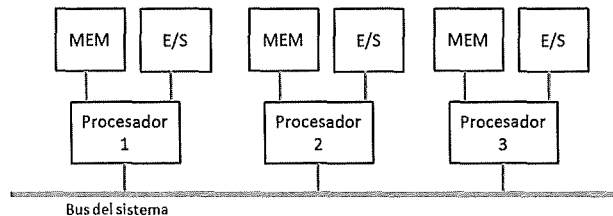


Figura 1.17. Débilmente acoplados.

Se denomina **multiproceso** a la gestión de varios procesos dentro de un sistema multiprocesador, donde cada procesador puede acceder a una memoria común.

1.3.3. PROGRAMAS CONCURRENTES

Un **programa concurrente** define un conjunto de acciones que pueden ser ejecutadas simultáneamente. Supongamos que tenemos estas dos instrucciones en un programa, está claro que el orden de la ejecución de las mismas influirá en el resultado final:

$x=x+1;$	La primera instrucción se debe ejecutar antes de la segunda.
$y=x+1;$	

En cambio, si tenemos estas otras, el orden de ejecución es indiferente:

$x=1;$	El orden no interviene en el resultado final.
$y=2;$	
$z=3;$	

CONDICIONES DE BERNSTEIN

Bernstein definió unas condiciones para que dos conjuntos de instrucciones se puedan ejecutar concurrentemente. En primer lugar es necesario formar 2 conjuntos de instrucciones:

- **Conjunto de lectura:** formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución.
- **Conjunto de escritura:** formado por instrucciones que cuenta con variables a las que se accede en modo escritura durante su ejecución.

Por ejemplo, sean las siguientes instrucciones:

Instrucción 1:	$x := y+1$
Instrucción 2;	$y := x+2$
Instrucción 3:	$z := a+b$

Los conjuntos de lectura y escritura estarían formados por las variables siguientes:

	Conjunto lectura - L	Conjunto escritura - E
Instrucción 1- I1:	y	x
Instrucción 2- I2:	x	y
Instrucción 3- I3:	a,b	z

Se pueden expresar de la siguiente manera:

$L(I1)=\{y\}$	$E(I1)=\{x\}$
$L(I2)=\{x\}$	$E(I2)=\{y\}$
$L(I3)=\{a,b\}$	$E(I3)=\{z\}$

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- La intersección entre las variables leídas por un conjunto de instrucciones I_i y las variables escritas por otro conjunto I_j debe ser vacío, es decir, no debe haber variables comunes:

$$L(I_i) \cap E(I_j) = \emptyset$$

- La intersección entre las variables de escritura de un conjunto de instrucciones I_i y las variables leídas por otro conjunto I_j debe ser nulo, es decir, no debe haber variables comunes:

$$E(I_i) \cap L(I_j) = \emptyset$$

- Por último, la intersección entre las variables de escritura de un conjunto de instrucciones I_i y las variables de escritura de un conjunto I_j debe ser vacío, no debe haber variables comunes:

$$E(I_i) \cap E(I_j) = \emptyset$$

En el ejemplo anterior tenemos las siguientes condiciones, donde se observa que las instrucciones I1 e I2 no se pueden ejecutar concurrentemente porque no cumplen las 3 condiciones:

Conjunto I1 e I2	Conjunto I2 e I3	Conjunto I1 e I3
$L(I1) \cap E(I2) \neq \emptyset$	$L(I2) \cap E(I3) = \emptyset$	$L(I1) \cap E(I3) = \emptyset$
$E(I1) \cap L(I2) \neq \emptyset$	$E(I2) \cap L(I3) = \emptyset$	$E(I1) \cap L(I3) = \emptyset$
$E(I1) \cap E(I2) = \emptyset$	$E(I2) \cap E(I3) = \emptyset$	$E(I1) \cap E(I3) = \emptyset$

En los programas secuenciales hay un orden fijo de ejecución de las instrucciones, siempre se sabe por dónde va a ir el programa. En cambio, en los programas concurrentes hay un orden parcial. Al haber solapamiento de instrucciones no se sabe cuál va a ser el orden de ejecución, puede ocurrir que ante unos mismos datos de entrada el flujo de ejecución no sea el mismo. Esto da lugar a que los programas concurrentes tengan un comportamiento indeterminista donde repetidas ejecuciones sobre un mismo conjunto de datos puedan dar diferentes resultados.

1.3.4. PROBLEMAS INHERENTES A LA PROGRAMACIÓN CONCURRENTES

A la hora de crear un programa concurrente podemos encontrarnos con dos problemas:

- **Exclusión mutua.** En programación concurrente es muy típico que varios procesos accedan a la vez a una variable compartida para actualizarla. Esto se debe evitar, ya que puede producir inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro la puede estar leyendo. Por ello es necesario conseguir la exclusión mutua de los procesos respecto a la variable compartida. Para ello se propuso la **región crítica**. Cuando dos o más procesos comparten una variable, el acceso a dicha variable debe efectuarse siempre dentro de la región crítica asociada a la variable. Sólo uno de los procesos podrá acceder para actualizarla y los demás deberán esperar, el tiempo de estancia es finito.
- **Condición de sincronización.** Hace referencia a la necesidad de coordinar los procesos con el fin de sincronizar sus actividades. Puede ocurrir que un proceso P1 llegue a un estado X que no pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución. La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

Algunas herramientas para manejar la concurrencia son: la región crítica, los semáforos, región crítica condicional, buzones, sucesos, monitores y sincronización por rendez-vous.

ACTIVIDAD 1.9

Responde a las siguientes cuestiones:

Escribe alguna característica de un programa concurrente.

¿Cuál es la ventaja de la concurrencia en los sistemas monoprocesador?

¿Cuáles son las diferencias entre multiprogramación y multiproceso?

¿Cuáles son los dos problemas principales inherentes a la programación concurrente?

1.3.5. PROGRAMACIÓN CONCURRENTES CON JAVA

Al igual que el sistema operativo puede ejecutar varios procesos concurrentemente, dentro de un proceso podemos encontrarnos con varios hilos de ejecución. Un hilo es como una secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente, véase Figura 1.18. Los hilos comparten el contexto del proceso, pero cada hilo mantiene una parte local.

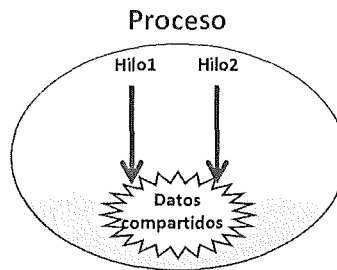


Figura 1.18. Hilos en un proceso.

Entre procesos e hilos hay algunas diferencias:

- Los hilos comparten el espacio de memoria del proceso, muchos comparten datos y espacios de direcciones; a diferencia de los procesos que generalmente poseen espacios de memoria de trabajo independientes e interactúan a través de mecanismos de comunicación dados por el sistema.
- Hilos y procesos pueden encontrarse en diferentes estados, pero los cambios de estado en los procesos son más costosos ya que los hilos pertenecen al mismo proceso. A los hilos también se les llama procesos ligeros.
- Se tarda menos tiempo en crear o en terminar un hilo que un proceso.
- En la comunicación entre procesos debe intervenir el núcleo del sistema, entre hilos no se necesita que intervenga el núcleo.

Para programar concurrentemente podemos dividir nuestro programa en hilos. Java proporciona la construcción de programas concurrentes mediante la clase **Thread** (hilo o hebra). Esta clase permite ejecutar código en un hilo de ejecución independiente.

En Java existen dos formas de utilizar o crear un hilo:

- Creando una clase que herede de la clase **Thread** y sobrecargando el método **run()**.
- Implementando la interface **Runnable**, y declarando el método **run()**. Se utiliza este modo cuando una clase ya deriva de otra. Por ejemplo, un applet deriva de la clase **Applet** por lo que no puede derivar también de **Thread**, en este caso tiene que implementar la interface **Runnable**.

El siguiente ejemplo crea un hilo de nombre *HiloSimple* heredando de la clase **Thread**. En el método **run()** se indican las líneas de código que se ejecutarán simultáneamente con las otras partes del programa. Cuando se termina la ejecución de ese método, el hilo de ejecución termina también:

```
public class HiloSimple extends Thread {
    public void run() {
        for (int i=0; i<5;i++)
            System.out.println("En el Hilo... ");
    }
} //
```

Para usar el hilo creo la clase *UsaHilo*:

```
public class UsaHilo {
    public static void main(String[] args) {
        HiloSimple hs = new HiloSimple();
        hs.start();
        for (int i=0; i<5;i++)
```



```

        System.out.println("Fuera del hilo..");
    }
}

```

Desde esta clase se arranca el hilo: primero se invoca al operador *new* para crear el hilo y luego al método **start()** que invoca al método **run()**. La compilación y ejecución muestra la siguiente salida, en la que se puede observar que se intercala las instrucciones del hilo y de fuera del hilo. La salida puede variar cada vez que se ejecutemos el programa:

```

D:\CAPIT1>javac HiloSimple.java
D:\CAPIT1>javac UsaHilo.java
D:\CAPIT1>java UsaHilo
Fuera del hilo..
Fuera del hilo..
Fuera del hilo..
En el Hilo...
En el Hilo...
En el Hilo...
En el Hilo...
En el Hilo...
Fuera del hilo..
Fuera del hilo..

```

Las 2 clases anteriores implementando la interfaz **Runnable** quedarían así:

```

public class HiloSimple2 implements Runnable{
    public void run() {
        for (int i=0; i<5;i++)
            System.out.println("En el Hilo...");
    }
}

public class UsaHilo2 {
    public static void main(String[] args) {
        HiloSimple2 hs = new HiloSimple2();
        Thread t = new Thread(hs);
        t.start();
        for (int i=0; i<5;i++)
            System.out.println("Fuera del hilo..");
    }
}

```

En el siguiente capítulo se tratarán más ampliamente los hilos con Java.

1.4. PROGRAMACIÓN PARALELA Y DISTRIBUIDA

1.4.1. PROGRAMACIÓN PARALELA

Un **programa paralelo** es un tipo de programa concurrente diseñado para ejecutarse en un sistema multiprocesador. El procesamiento paralelo permite que muchos elementos de proceso independientes trabajen simultáneamente para resolver un problema. Estos elementos pueden ser un número arbitrario de equipos conectados por una red, un único equipo con varios procesadores o una combinación de ambos. El problema a resolver se divide en partes independientes de tal forma que cada elemento pueda ejecutar la parte de programa que le corresponda a la vez que los demás.

Recordemos que en un sistema **multiprocesador**, donde existe más de un procesador, podemos tener un proceso en cada procesador y todos juntos trabajan para resolver un problema. Cada procesador realiza una parte del problema y necesita intercambiar información con el resto. Según cómo se realice este intercambio podemos tener modelos distintos de programación paralela:

- Modelo de **memoria compartida**: los procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito en memoria por un procesador puede ser leído directamente por cualquier otro.
- Modelo de **paso de mensajes**: cada procesador dispone de su propia memoria independiente del resto y accesible sólo por él. Para realizar el intercambio de información es necesario que cada procesador realice la petición de datos al procesador que los tiene, y éste haga el envío. El entorno de programación PVM que veremos más adelante utiliza este modelo.

El intercambio de información entre procesadores depende del sistema de almacenamiento que se disponga. Según este criterio las arquitecturas paralelas se clasifican en: **Sistemas de memoria compartida o multiprocesadores**: los procesadores comparten físicamente la memoria; y **Sistemas de memoria distribuida o multicomputadores**: cada procesador dispone de su propia memoria.

Dentro de los sistemas de memoria distribuida o multicomputadores nos encontramos con los **Clusters**. Son sistemas de procesamiento paralelo y distribuido donde se utilizan múltiples ordenadores, cada uno con su propio procesador, enlazados por una red de interconexión más o menos rápida, de tal forma que el conjunto de ordenadores es visto como un único ordenador, más potente que los comunes de escritorio.

Tradicionalmente, el paralelismo se ha utilizado en centros de supercomputación para resolver problemas de elevado coste computacional en un tiempo razonable, pero en la última década su interés se ha extendido por la difusión de los procesadores con múltiples núcleos (combina dos o más procesadores independientes en un solo circuito integrado). Estos procesadores permiten que un dispositivo computacional exhiba una cierta forma del paralelismo a nivel de thread (thread-level parallelism) (TLP) sin incluir múltiples microprocesadores en paquetes físicos separados. Esta forma de TLP se conoce a menudo como multiprocesamiento a nivel de chip (chip-level multiprocessing) o CMP².

VENTAJAS E INCONVENIENTES

Ventajas del procesamiento paralelo:

- Proporciona ejecución simultánea de tareas.
- Disminuye el tiempo total de ejecución de una aplicación.
- Resolución de problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales, por ejemplo, los recursos que están en una red distribuida, una WAN o la propia red internet.
- Disminución de costos, en vez de gastar en un supercomputador muy caro se pueden utilizar otros recursos más baratos disponibles remotamente.

Pero no todo son ventajas, algunos inconvenientes son:

² https://es.wikipedia.org/wiki/Procesador_multinúcleo

- Los compiladores y entornos de programación para sistemas paralelos son más difíciles de desarrollar.
- Los programas paralelos son más difíciles de escribir.
- El consumo de energía de los elementos que forman el sistema.
- Mayor complejidad en el acceso a los datos.
- La comunicación y la sincronización entre las diferentes subtarefas.

La computación paralela resuelve problemas como: predicciones y estudios meteorológicos, estudio del genoma humano, modelado de la biosfera, predicciones sísmicas, simulación de moléculas... En algunos casos se dispone de tal cantidad de datos que serían muy lento o imposible tratar con máquinas convencionales.

ACTIVIDAD 1.10

Entra en la siguiente URL https://computing.llnl.gov/tutorials/parallel_comp/ y responde a las siguientes cuestiones:

Cita algunas características de la computación serie.

Cita algunas características de la computación en paralelo.

Ámbitos en los que se usa la computación en paralelo.

¿Cómo hace uso de la computación paralela el proyecto SETI @ home?

1.4.2. PROGRAMACIÓN DISTRIBUIDA

Uno de los motivos principales para construir un sistema distribuido es compartir recursos. Probablemente, el sistema distribuido más conocido por todos es Internet que permite a los usuarios donde quiera que estén hacer uso de la World Wide Web, el correo electrónico y la transferencia de ficheros. Entre las aplicaciones más recientes de la computación distribuida se encuentra el *Cloud Computing* que es la computación en la nube o servicios en la nube, que ofrece servicios de computación a través de Internet.

Se define un sistema distribuido como aquel en el que los componentes hardware o software, localizados en computadores unidos mediante una red, comunican y coordinan sus acciones mediante el paso de mensajes. Esta definición tiene las siguientes consecuencias³:

- **Concurrencia:** lo normal en una red de ordenadores es la ejecución de programas concurrentes.
- **Inexistencia de reloj global:** cuando los programas necesitan cooperar coordinan sus acciones mediante el paso de mensajes. No hay una temporalización, los relojes de los host no están sincronizados.
- **Fallos independientes:** cada componente del sistema puede fallar independientemente, permitiendo que los demás continúen su ejecución.

La programación distribuida es un paradigma de programación enfocado en desarrollar sistemas distribuidos, abiertos, escalables, transparentes y tolerantes a fallos. Este paradigma es el resultado natural del uso de las computadoras y las redes. Casi cualquier lenguaje de

³ Sistemas Distribuidos: Conceptos y Diseño. George Coulouris y otros. Ed: Addison-Wesley.

programación que tenga acceso al máximo al hardware del sistema puede manejar la programación distribuida, considerando una buena cantidad de tiempo y código⁴.

Una arquitectura típica para el desarrollo de sistemas distribuidos es la arquitectura **cliente-servidor**. Los clientes son elementos activos que demandan servicios a los servidores realizando peticiones y esperando la respuesta, los servidores son elementos pasivos que realizan las tareas bajo requerimientos de los clientes.

Por ejemplo, un cliente web solicita una página, el servidor web envía al cliente la página solicitada. Véase Figura 1.19. La comunicación entre servidores y clientes se realiza a través de la red.

Existen varios modelos de programación para la comunicación entre los procesos de un sistema distribuido:

- **Sockets.** Proporcionan los puntos extremos para la comunicación entre procesos. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación. En el capítulo 3 se tratarán los sockets en Java.
- **Llamada de procedimientos remotos o RPC (*Remote Procedure Call*).** Permite a un programa cliente llamar a un procedimiento de otro programa en ejecución en un proceso servidor. El proceso servidor define en su interfaz de servicio los procedimientos disponibles para ser llamados remotamente.
- **Invocación remota de objetos.** El modelo de programación basado en objetos ha sido extendido para permitir que los objetos de diferentes procesos se comuniquen uno con otro por medio de una *invocación a un método remoto* o **RMI (*Remote Method Invocation*)**. Un objeto que vive en un proceso puede invocar métodos de un objeto que reside en otro proceso. **Java RMI** extiende el modelo de objetos de Java para proporcionar soporte de objetos distribuidos en lenguaje Java.

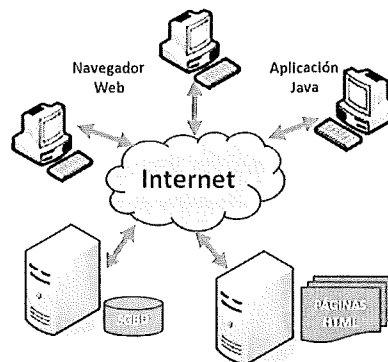


Figura 1.19. Cliente-servidor sobre web.

VENTAJAS E INCONVENIENTES

Ventajas que aportan los sistemas distribuidos:

- Se pueden compartir recursos y datos.
- Capacidad de crecimiento incremental.

⁴ http://es.wikipedia.org/wiki/Programación_distribuida

- Mayor flexibilidad al poderse distribuir la carga de trabajo entre diferentes ordenadores.
- Alta disponibilidad.
- Soporte de aplicaciones inherentemente distribuidas.
- Carácter abierto y heterogéneo.

Pero no todo son ventajas, algunos inconvenientes son:

- Aumento de la complejidad, se necesita nuevo tipo de software.
- Problemas con las redes de comunicación: pérdida de mensajes, saturación del tráfico.
- Problemas de seguridad como por ejemplo ataques de denegación de servicio en la que se “bombardea” un servicio con peticiones inútiles de forma que un usuario interesado en usar el servicio no pueda usarlo.

ACTIVIDAD 1.11

Busca en Internet aplicaciones de los sistemas distribuidos.

PROGRAMACION CONCURRENTE, PARALELA Y DISTRIBUIDA

Programación Concurrente: Tenemos varios elementos de proceso (hilos, procesos) que trabajan de forma conjunta en la resolución de un problema. Se suele llevar a cabo en un único procesador o núcleo.

Programación Paralela: Es programación concurrente cuando se utiliza para acelerar la resolución de los problemas, normalmente usando varios procesadores o núcleos.

Programación Distribuida: Es programación paralela cuando los sistemas están distribuidos a través de una red (una red de procesadores); se usa paso de mensajes.

1.4.3. PVM. INSTALACIÓN Y CONFIGURACIÓN

PVM (*Parallel Virtual Machine* - Máquina virtual en paralelo) es un conjunto de herramientas software que permiten emular un marco de computación concurrente, distribuido y de propósito general, utilizando para ello grupos de ordenadores conectados, de manera que ni los ordenadores ni las redes que los conectan tienen las mismas características arquitectónicas.

Permite conectar entre sí ordenadores Unix y Windows (WIN95, NT 3.5, NT 4.0) para ser usados como un único gran ordenador paralelo de alto rendimiento. Así, grandes problemas de cómputo se pueden resolver de manera más rentable aprovechando la potencia y memoria de muchos equipos conectados.

Cientos de sitios en todo el mundo están usando PVM para resolver importantes problemas científicos, industriales y médicos, además de su uso como una herramienta educativa para enseñar programación paralela.

El modelo de computación de PVM se basa en considerar que una aplicación es una colección de tareas que se comunican y sincronizan mediante el paradigma de paso de mensajes. El sistema de PVM se compone de 3 partes⁵:

⁵ Procesamiento paralelo teoría y programación. Sebastián Dormido Canto y otros, Ed: Sanz y Torres.

- **El demonio**, llamado **pvm3d** y a veces abreviado **PVMD**, que reside en todos los equipos que componen la máquina virtual. Uno de los equipos de la máquina virtual actúa como equipo anfitrión o maestro y los demás como esclavos, el maestro es el que inicia las tareas en paralelo. Figura 1.20.

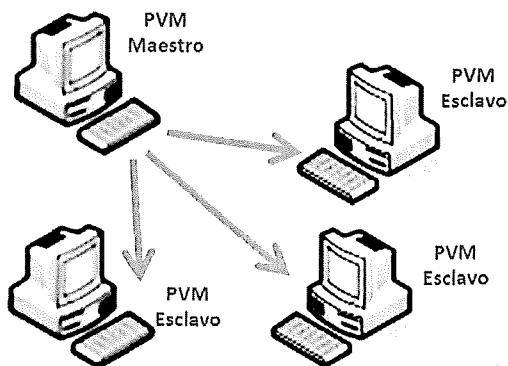


Figura 1.20. Máquina virtual PVM con varios equipos.

- **La biblioteca de desarrollo (API)**, que contiene un repertorio de funciones que son necesarias para operar con las tareas de una aplicación, transmitir mensajes entre ellas y alterar la configuración de la máquina virtual.
- **La consola de PVM**, que actúa a modo de intérprete de comandos proporcionando una interfaz entre el usuario y el demonio.

VENTAJAS E INCONVENIENTES

Entre las ventajas podemos destacar las siguientes:

- Es una de las librerías de paso de mensajes más fáciles de usar.
- Fácil de instalar.
- Fácil de configurar. La aplicación decide dónde y cuando ejecutar o terminar las tareas, qué máquinas se añaden o se eliminan desde la máquina virtual paralela, qué tareas se pueden comunicar y/o sincronizar con otras.
- Se puede incorporar cualquier ordenador al esquema de la máquina virtual.

Inconvenientes:

- Al ser un esquema heterogéneo de ordenadores, el rendimiento puede verse mermado o incrementado, dependiendo de la capacidad de procesamiento de los ordenadores que formen parte del esquema de la máquina virtual. Lo que era una ventaja tiene su desventaja.
- Es algo deficiente en cuanto al paso de mensajes se refiere.
- PVM no es un estándar.

INSTALACIÓN

En este apartado veremos cómo instalar PVM en una máquina que tiene instalado Ubuntu. La instalación se debe realizar en todos los equipos que formen parte de PVM. Se recomienda tener privilegios de administrador para llevar a cabo la instalación. Podemos instalarlo de dos formas:

- Desde la línea de comandos de Ubuntu escribiendo las siguientes órdenes (incluye el paquete de ejemplos):

```
$ sudo apt-get install pvm pvm-dev pvm-examples
```

- Usando el entorno gráfico desde la opción de menú: **Sistema** → **Administración** → **Centro de software de Ubuntu**. Buscamos el paquete **pvm**, **pvm-dev** y **pvm-examples**, al seleccionarlos nos pedirá confirmar la instalación de otros paquetes. Pulsamos el botón *Instalar*. Véase Figura 1.21.

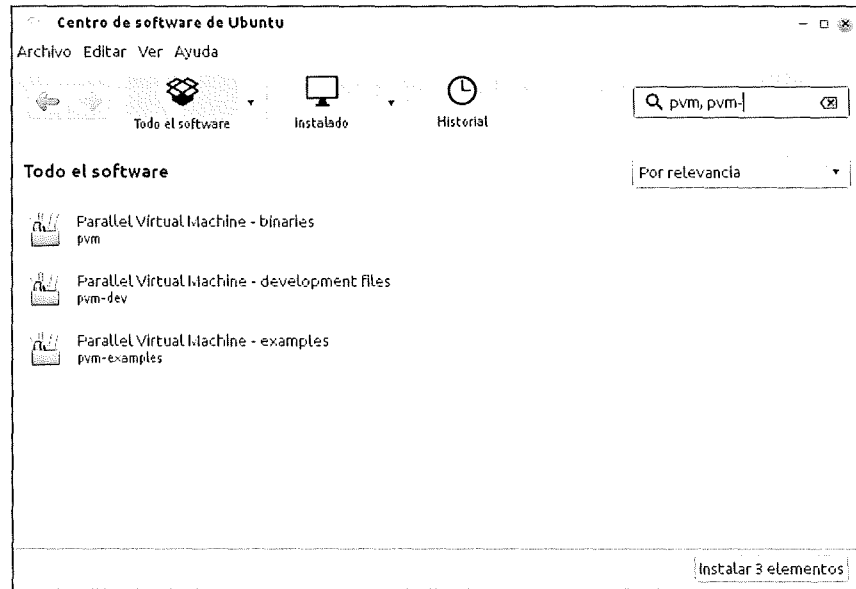


Figura 1.21. Instalación de PVM.

Una vez instalados los paquetes se habrá creado la carpeta **pvm3** en **/usr/lib/**. También se habrán creado 3 subcarpetas: **bin** que contiene un único fichero, **conf** que contienen dos ficheros llamados *LINUX.def* y *LINUX.m4* (que hacen referencia a la arquitectura del equipo) y **lib** donde se encuentran los ejecutables, Figura 1.22.

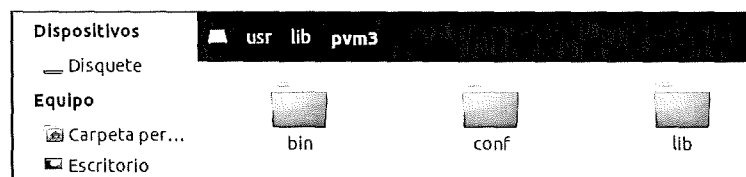


Figura 1.22. Carpetas /usr/lib/pvm3.

A continuación, configuramos PVM para trabajar con diferentes equipos. Supongamos que tengo 2 equipos con Ubuntu12.04, uno se llama *ubuntu12-maestro* y el otro *ubuntu12-esclavo*. El primer equipo actuará como maestro. Estos equipos tienen que tener instalado PVM. En ambos existe una cuenta de usuario con nombre *administrador* y clave *admin1234*.

Creo el fichero **.rhosts** (o lo edito si ya existe) en el directorio personal del equipo maestro y del esclavo, en este caso en **/home/administrador**. Escribo una entrada por cada equipo al que me voy a conectar y el nombre de usuario para inicio de sesión en ese equipo, por ejemplo, desde el equipo maestro para conectarme al equipo de nombre *ubuntu12-esclavo* con usuario de nombre *administrador* escribo: *ubuntu12-esclavo administrador*. Desde el equipo esclavo añado la entrada: *ubuntu12-maestro administrador* para conectarme al equipo maestro con usuario *administrador*.

En ambos equipos se añaden las siguientes variables de entorno en el fichero **.bashrc** que está en la carpeta */home/administrador*:

```
export PVM_ROOT=/usr/lib/pvm3
export PVM_ARCH=$PVM_ROOT/lib/pvmgetarch
export PVM_RSH=/usr/bin/ssh
```

Con esto se indica donde está PVM instalado y qué arquitectura usamos (se define en el fichero *pvmgetarch*). En los equipos esclavos creamos la carpeta **LINUX** para guardar los ejecutables en */usr/lib/pvm3/bin*. Nos debe quedar: */usr/lib/pvm3/bin/LINUX*.

Realizamos la primera prueba donde se usa la función **pvm_mytid()** para saber el identificador de la tarea o TID. Creamos el programa C de nombre *hola.c* en nuestra carpeta personal:

```
#include <stdio.h>
#include <pvm3.h>
int main()
{
    int mytid;

    mytid = pvm_mytid();
    printf("Mi TID es %x\n", mytid);
    pvm_exit();
    return 0;
}
```

La compilamos con la opción **-lpvm3**:

```
administrador@ubuntu12-maestro:~$ gcc hola.c -o hola -lpvm3
```

Después ejecutamos PVM y lanzamos la tarea con **spawn** de la siguiente manera:

```
administrador@ubuntu12-maestro:~$ pvm
pvm> spawn -> hola
spawn -> hola
[1]
0 successful
No such file
```

Se visualiza un error, *No such file*, indicando que no existe el fichero. Es porque falta indicar en una variable de entorno donde están los ejecutables. Salimos de PVM con **halt** y definimos la variable **PVM_PATH**, y volvemos a entrar para lanzar la tarea:

```
pvm> halt
halt
Terminado
administrador@ubuntu12-maestro:~$ PVM_PATH=/home/administrador
administrador@ubuntu12-maestro:~$ export PVM_PATH
administrador@ubuntu12-maestro:~$ pvm
pvm> spawn -> hola
spawn -> hola
[2]
1 successful
```



```
t40003
pvm> [2:t40003] Mi TID es 40003
[2:t40003] EOF
[2] finished

pvm>
```

De momento vemos que funciona. Antes de añadir máquinas o host a nuestro PVM hemos de instalar **SSH** (*Secure SHell*, intérprete de órdenes segura) si no lo tenemos instalado; ya que nos permite acceder a máquinas remotas a través de una red. Lo instalamos en el maestro y en el esclavo.

Podemos instalarlo desde el **Centro de software de Ubuntu**, buscamos *ssh* y pulsamos el botón *Instalar*. Véase Figura 1.23. Se deben instalar también los paquetes afectados.

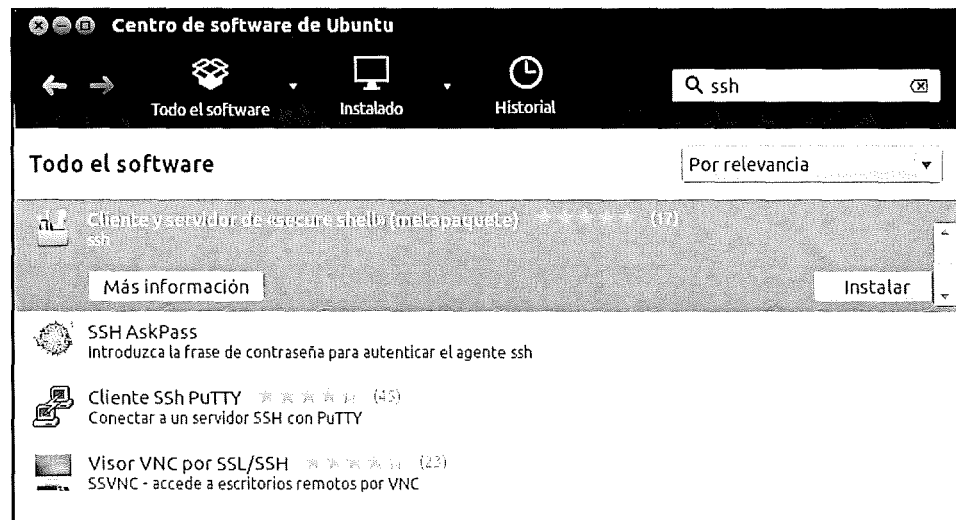


Figura 1.23. Instalación de SSH.

Para añadir un host desde PVM escribimos: *add nombrehost*. Por ejemplo, para añadir la máquina *ubuntu12-esclavo* escribo:

```
pvm> add ubuntu12-esclavo
add ubuntu12-esclavo
0 successful

          HOST      DTID
ubuntu12-esclavo No such host

pvm>
```

Pero puede ocurrir que no encuentre el host (*No such host*). Entonces añado la IP del host al que me voy a conectar en el fichero */etc/hosts*, también añado la IP del maestro (las añadimos al principio del fichero), por ejemplo:

```
192.168.176.130  ubuntu12-maestro
192.168.176.134  ubuntu12-esclavo
```

Esto se hace en todas las máquinas que formen parte de PVM. Una vez realizado el cambio entro en PVM para añadir la máquina *ubuntu12-esclavo*:

52 Programación de servicios y procesos

```
pvm> add ubuntu12-esclavo
add ubuntu12-esclavo
The authenticity of host 'ubuntu12-esclavo (192.168.176.134)' can't be
established.
ECDSA key fingerprint is
cd:4e:ac:ad:21:77:96:27:ac:c7:6d:18:ed:1b:86:c7.
Are you sure you want to continue connecting (yes/no)? yes
administrador@ubuntu12-esclavo's password:
1 successful

          HOST      DTID
ubuntu12-esclavo  80000

pvm>
```

Se visualiza un mensaje indicando que la autenticidad del host *ubuntu12-esclavo* no puede ser establecida, y nos muestra la huella digital (fingerprint) de la clave ECDSA. Nos pregunta si queremos continuar, al decir *yes* agrega el host *ubuntu12-esclavo* a la lista de host conocidos o confiables y entonces pide la contraseña del usuario con nombre *administrador* de la máquina *ubuntu12-esclavo*. A continuación, se muestra el host y el DTID que se le ha asignado. Cada vez que nos conectemos a *ubuntu12-esclavo* nos pedirá la clave. Luego veremos cómo quitarlo.

Ahora escribimos desde PVM la orden **conf** para que nos muestre los hosts que están en PVM; vemos dos, el maestro y el esclavo:

```
pvm> conf
conf
2 hosts, 1 data format

          HOST      DTID      ARCH      SPEED      DSIG
ubuntu12-maestro  40000      LINUX      1000  0x00408841
ubuntu12-esclavo  80000      LINUX      1000  0x00408841

pvm>
```

Con esta orden se puede comprobar el nombre y número de los hosts que componen la máquina virtual, así como la arquitectura (columna ARCH), número base para identificadores de tareas (DTID), velocidad relativa (SPEED) y DSIG.

Hasta aquí ya tenemos preparado el entorno para empezar a ejecutar los programas. Antes veamos como hacer para que no nos vuelva a pedir la clave del usuario *administrador* cada vez que se añada el host *ubuntu12-esclavo*.

Desde la línea de comandos (fuera de PVM) y en la máquina que hace de maestro hacemos lo siguiente: nos vamos a la carpeta **.ssh**, y ejecutamos **ssh-keygen -t dsa** para generar un par de claves pública/privada; cada vez que nos pida la clave pulsamos la tecla [Intro], es decir dejamos vacío el *passphrase*:

```
administrador@ubuntu12-maestro:~$ cd ~/.ssh
administrador@ubuntu12-maestro:~/.ssh$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/administrador/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/administrador/.ssh/id_dsa.
Your public key has been saved in /home/administrador/.ssh/id_dsa.pub.
The key fingerprint is:
8d:1e:4b:98:dd:df:e1:a9:15:e1:31:a8:24:3b:27:eb administrador@ubuntu12-
maestro
The key's randomart image is:
```

```

+--[ DSA 1024]-----+
|                       |
|               .       |
|           . . . +     |
|       + B . . +      |
|   o S =  +           |
|   o B . o +          |
|       +   . =        |
|       .       o      |
|       E   .          |
+-----+

```

```
administrador@ubuntu12-maestro:~/.ssh$
```

Esto genera 2 ficheros: *id_dsa.pub* (public key) e *id_dsa* (private key). Se lo pasamos al usuario *administrador* en *ubuntu12-esclavo* mediante el comando **ssh-copy-id**, nos pedirá la clave del usuario:

```

administrador@ubuntu12-maestro:~/.ssh$ ssh-copy-id
                                administrador@ubuntu12-esclavo
administrador@ubuntu12-esclavo's password:
Now try logging into the machine, with "ssh 'administrador@ubuntu12-
esclavo'", and check in:

```

```
~/.ssh/authorized_keys
```

to make sure we haven't added extra keys that you weren't expecting.

```
administrador@ubuntu12-maestro:~/.ssh$
```

Probamos que no nos pide la clave al conectarnos a *ubuntu12-esclavo* con el comando *ssh*:

```

administrador@ubuntu12-maestro:~/.ssh$ ssh administrador@ubuntu12-
                                esclavo
Welcome to Ubuntu 12.04.3 LTS (GNU/Linux 3.5.0-40-generic i686)

```

```
* Documentation:  https://help.ubuntu.com/
```

```

New release '14.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

```

```

Last login: Wed Jul  5 11:57:49 2017 from localhost
administrador@ubuntu12-esclavo:~$

```

Para salir usamos el comando **exit**:

```

administrador@ubuntu12-esclavo:~$ exit
logout
Connection to ubuntu12-esclavo closed.
administrador@ubuntu12-maestro:~/.ssh$

```

NOTA: La dirección IP de los host debe ser fija. Hemos de tenerlo en cuenta cada vez que empecemos a trabajar con PVM. Si es dinámica puede que un día funcione y otro no; entonces hemos de revisar las IPs y el fichero */etc/hosts*.

Si no tenemos máquinas físicas podemos usar entornos como VMware con varios sistemas Linux instalados.

En la siguiente tabla se muestran algunos de los comandos más importantes del intérprete de comandos PVM:

Comandos	
add máquina	Incorpora la máquina indicada a PVM
delete máquina	Elimina la máquina del entorno PVM, no se puede eliminar la máquina desde la que estamos ejecutando los comandos
conf	Muestra la configuración actual de PVM.
ps	Listado de procesos de PVM.
halt	Apaga y sale de PVM.
help	Lista los comandos de PVM.
id	Visualiza el TID de la consola.
jobs	Genera un listado de los trabajos en ejecución
kill	Mata un proceso de la PVM.
quit	Sale de la máquina paralela virtual sin apagarla.
spawn	Arranca una aplicación bajo PVM.
version	Visualiza la versión de PVM
reset	Inicializa PVM.

Con esto ya podemos empezar a crear programas paralelos.

1.4.4. EJECUCIÓN DE TAREAS EN PARALELO CON PVM

PVM está basado en el paso de mensajes. Inicialmente para cada tarea PVM se crea un buffer activo de envío y otro de recepción, no siendo necesario, en la mayoría de los casos, la creación de nuevos buffers. Todas las operaciones de empaquetamiento y desempaquetamiento se realizarán en el buffer activo. El envío de un mensaje requiere 3 pasos:

- Inicialización de un buffer de envío. Usaremos la función **pvm_initsend()**.
- Empaquetamiento del mensaje en el buffer, para ello usaremos las funciones **pvm_pkXXX()**.
- Envío del mensaje a una o varias tareas, función **pvm_send()**.

La recepción requiere dos pasos:

- Recepción del mensaje, función **pvm_recv()**;
- Desempaquetamiento de los datos enviados en el mensaje, funciones **pvm_upkXXX()**.

Veamos a continuación estas funciones y otras que nos serán útiles para lanzar tareas en paralelo:

int tid = pvm_mytid(void): devuelve el TID del proceso o tarea que invoca a la función. Un valor menor que 0 indica error.

int info = pvm_exit(void): sale del ambiente PVM.

int tid = pvm_parent(void): devuelve el TID del proceso que creó la tarea que invoca a la función. Si el proceso no fue creado con **pvm_spawn()**, entonces *tid = PvmNoParent*.

int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids): se utiliza para la creación de procesos. Genera *ntask* copias del proceso cuyo nombre es el string *task*. El parámetro *argv* es un puntero a un array de argumentos para *task*. El argumento

flag puede tener varios valores, es 0 cuando PVM elige en qué máquina ejecutar los procesos (también se puede poner *PvmTaskDefault* en lugar de 0), *flag* = 1 cuando el proceso se ejecuta en la máquina indicada en el parámetro *where* (se puede usar *PvmTaskHost* en lugar de 1), con *flag* = 2 (*PvmTaskArch*) el parámetro *where* especifica el tipo de arquitectura en que se va a crear el proceso, etc. *where* es una cadena de caracteres que indica donde crear el proceso. La función devuelve en *numt* el número de tareas que se crearon satisfactoriamente y en el vector *tids* los TIDs de las tareas creadas con éxito. Ejemplo:

```
cc = pvm_spawn("esclavo1", (char**)0, 0, "", 1, &tids);
```

Se lanza el proceso *esclavo1*, el parámetro *argv* es nulo es decir no se envían argumentos al proceso, *flag* es 0 y *where* es nulo es decir "", en este caso PVM elige la máquina donde crear el proceso; el parámetro *ntask* es 1 indica que se crea 1 copia del proceso y por último *&tids* contiene el TID de la tarea creada.

int bufid = pvm_initsend(int encoding): esta función se utiliza para limpiar el buffer de envío que se encuentre activo. Siempre debe usarse antes de proceder al empaquetado de un mensaje. Inicializa el buffer de envío dándole un esquema de codificación identificado por *encoding*. Los valores posibles son: *PvmDataDefault* es el modo por defecto igual a XDR encoding, *PvmDataRaw* no se realiza codificación, etc. Ejemplo:

```
pvm_initsend(PvmDataDefault);
```

int info = pvm_send(int tid, int msgtag): envía a la tarea identificada por *tid* el mensaje previamente empaquetado en el buffer activo. *msgtag* es la etiqueta que se da al mensaje que está en el buffer activo.

int info = pvm_pkstr(char *cp): empaqueta una cadena de caracteres, recibe un puntero a la cadena a empaquetar. Devuelve un código de estado de la operación, un valor menor que 0 indica error. El siguiente ejemplo limpia el buffer de envío (con **pvm_initsend()**), empaqueta una cadena de caracteres (**pvm_pkstr()**) y se la envía (**pvm_send()**) al proceso padre identificado por *ptid*:

```
char buf[100];
int ptid = pvm_parent();
strcpy(buf, "Hola Mundo");
pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);
```

int pvm_recv(int tid, int msgtag): espera a recibir un mensaje etiquetado como *msgtag* enviado por la tarea *tid*. Cuando se coloca el valor -1 en *tid* y/o *msgtag* se aceptarán mensajes con cualquier etiqueta y/o de cualquier tarea. Una vez que se recibe el mensaje se crea un nuevo buffer de recepción. Devuelve el identificador del buffer de recepción que ha sido creado del mensaje activo.

int info = pvm_buinfo(int bufid, int *bytes, int *msgtag, int *tid): devuelve información sobre el mensaje almacenado en el buffer *bufid*. *bytes* es la longitud del mensaje en bytes, *msgtag* es la etiqueta del mensaje, *tid* el identificador de la tarea. El siguiente ejemplo recibe un mensaje de cualquier tarea lanzada y obtiene información sobre él:

```
bufid = pvm_recv(-1, -1);
pvm_buinfo(bufid, &longitud, &tipo, &tarea_origen);
```

int info = pvm_upkstr(char *cp): desempaqueta los datos recibidos (cadena de caracteres). El siguiente ejemplo espera a recibir un mensaje, lo desempaqueta del buffer activo y lo visualiza:

```
bufid = pvm_rcv(-1, -1);
pvm_upkstr(buf);
printf("Mensaje: %s\n", buf);
```

int info = pvm_pkint(int *ip, int nitem, int stride): empaqueta enteros. *ip* es un puntero al primer elemento del array a empaquetar. *nitem* es el número total de enteros a empaquetar. *stride* establece la distancia entre 2 elementos consecutivos, un valor de 1 indica que se empaqueta el array de forma continua, un valor de 2 indica que cada 2 elementos se empaquetan y así sucesivamente.

int info = pvm_upkint(int *ip, int nitem, int stride): desempaqueta enteros. Los parámetros significan lo mismo que en la función de empaquetado.

A continuación, vamos a probar algunos ejemplos que vienen con la instalación de PVM en el paquete **pvm-examples** y que se instalan en la carpeta **/usr/share/doc/pvm-examples**. Localizamos un fichero de nombre **examples.tar.gz**. Lo descomprimos y extraemos los ejemplos **hello.c** y **hello_other.c**. El primero es un proceso (maestro) que lanza una copia del proceso **hello_other** (esclavo) a las máquinas que forman parte del entorno PVM, recibe un mensaje del proceso llamado. En la Figura 1.24 se muestra el proceso maestro **hello.c**.

```
#include <stdio.h>
#include "pvm3.h" ← Librería de PVM
#include <stdlib.h>
main()
{
    int cc, tid;
    char buf[100];

    printf("I'm tid:\n", pvm_mytid());
    cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

    if (cc == 1) {
        cc = pvm_rcv(-1, -1);
        pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
        pvm_upkstr(buf);
        printf("From tid: %s\n", tid, buf);
    } else
        printf("can't start hello_other\n");

    pvm_exit(); ← Salir de PVM
    exit(0);
}
```

Figura 1.24. Programa **hello.c**.

Nos fijamos en los métodos **pvm_mytid()** que obtiene el TID de este proceso, **pvm_spawn()** para lanzar la tarea, **pvm_rcv()** que espera a recibir un mensaje de cualquier tarea lanzada, **pvm_bufinfo()** que obtiene información del mensaje y **pvm_upkstr()** que desempaqueta el mensaje recibido. El segundo proceso **hello_other.c** se muestra en la Figura 1.25, lo que hace es enviar un mensaje al proceso padre en el que se incluye el nombre del host desde el que se envía.

Nos fijamos en los métodos `pvm_parent()` que obtiene el TID del proceso padre, `pvm_initsend()` que limpia el buffer de envío, `pvm_pkstr()` que empaqueta el mensaje a enviar, en este caso una cadena de caracteres almacenada en `buf`; y `pvm_send()` que envía el mensaje al proceso padre. La función `gethostname()` (no es una función PVM) devuelve el nombre de la máquina.

```

#include "pvm3.h" ← Librería de PVM
#include <stdlib.h>
#include <string.h>
main()
{
    char buf[100];
    int ptid = pvm_parent();
    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);

    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, 1);

    pvm_exit();
    exit(0);
}

```

Diagrama de anotaciones:

- Una flecha horizontal apunta desde el comentario "Librería de PVM" a `"pvm3.h"`.
- Una flecha vertical apunta desde el comentario "TID del proceso padre" a `pvm_parent()`.
- Una flecha horizontal apunta desde el comentario "Prepara el mensaje a enviar" a `strcpy(buf, "hello, world from ");` y `gethostname(buf + strlen(buf), 64);`.
- Una flecha horizontal apunta desde el comentario "Limpia el buffer" a `pvm_initsend(PvmDataDefault);`.
- Una flecha horizontal apunta desde el comentario "Empaquetado del mensaje" a `pvm_pkstr(buf);`.
- Una flecha horizontal apunta desde el comentario "Envío del mensaje" a `pvm_send(ptid, 1);`.
- Una flecha horizontal apunta desde el comentario "Salir de PVM" a `pvm_exit();`.

Figura 1.25. Programa *hello_other.c*.

Para probar estos programas copiamos los dos ficheros en nuestra carpeta personal, en el ordenador que hace de maestro (*ubuntu12-maestro*), y los compilamos (hemos de incluir en los ficheros los includes que se muestran en la imagen, en caso de que no aparezcan):

```
administrador@ubuntu12-maestro:~$ gcc hello.c -o hello -lpvm3
```

Copiamos en la carpeta `/usr/lib/pvm3/bin/LINUX` de la máquina que hace de esclavo (*ubuntu12-esclavo*) el programa ejecutable *hello_other*. También podemos compilar el programa desde el ordenador esclavo:

```
administrador@ubuntu12-esclavo:~$ gcc hello_other.c -o
                                hello_other -lpvm3
administrador@ubuntu12-esclavo:~$ sudo cp hello_other
                                /usr/lib/pvm3/bin/LINUX/
```

A continuación en el maestro definimos la variable `PVM_PATH` y entramos en PVM. Primero añadimos el host esclavo y después lanzamos el proceso *hello* con la orden `spawn -> hello`:

```
administrador@ubuntu12-maestro:~$ PVM_PATH=/home/administrador
administrador@ubuntu12-maestro:~$ export PVM_PATH
administrador@ubuntu12-maestro:~$ pvm
```

```
pvm> add ubuntu12-esclavo
add ubuntu12-esclavo
1 successful
```

```

                HOST      DTID
    ubuntu12-esclavo  80000
pvm> conf
conf
2 hosts, 1 data format
                HOST      DTID      ARCH      SPEED      DSIG
    ubuntu12-maestro  40000      LINUX      1000  0x00408841
    ubuntu12-esclavo  80000      LINUX      1000  0x00408841
spawn -> hello
[2]
1 successful
t40002
pvm> [2:t80002] EOF
[2:t40002] i'm t40002
[2:t40002] from t80002: hello, world from ubuntu12-esclavo
[2:t40002] EOF
[2] finished

pvm>

```

Los números que aparecen entre corchetes identifican las tareas de los hosts y tienen relación con el DTID del host.

NOTA: Podemos probar el programa maestro y el esclavo en la misma máquina, es decir sin añadir ningún host. Veremos que los identificadores de tareas hacen referencia al host donde se ejecutan.

En el siguiente ejemplo el maestro envía una cadena al esclavo, y el esclavo se la devuelve al maestro en mayúsculas. Habrá dos tipos de mensajes, uno cuando el maestro envía al esclavo y el otro cuando el esclavo envía al maestro. Al primer mensaje se le etiqueta con la variable *etqenvio*, con valor 1. Al segundo se le etiqueta con la variable *etqrecibe* con valor 2. El código del proceso maestro **cadmaestro.c** es:

```

#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
#include <string.h>
main()
{
    int tareas, cc, tid;
    char buf[100];
    int etqenvio=1;
    int etqrecibe=2;
    strcpy(buf, "mensaje en minúscula");

    tareas = pvm_spawn("cadesclavo", (char**)0, 0, "", 1, &tid);

    //ENVIO CADENA
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(tid, etqenvio); //envio con etiq 1
    printf("\tENVÍO A t%x: %s\n", tid, buf);

    //RECIBO CADENA

```



```

cc = pvm_rcv(tid, etgrecibe); //recibo con etiqa 2
pvm_upkstr(buf); //desempaqueta un entero
printf("\tRECIBO DE t%x: %s\n", tid, buf);

pvm_exit();
exit(0);
}

```

El proceso esclavo recibe del maestro el mensaje etiquetado con 1 y envía el mensaje con etiqueta 2. El código del proceso esclavo **cadescravo.c** es el siguiente:

```

#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
#include <string.h>
main()
{
    char buf2[100];
    char buf[100];
    int i, parent_tid = pvm_parent();

    //SE RECIBEN LOS DATOS DEL MAESTRO
    pvm_rcv(parent_tid, 1); //recibo con etiqueta 1
    pvm_upkstr(buf);

    for(i = 0; i< strlen(buf) ; i++)
        buf2[i] = toupper(buf[i]);

    buf2[strlen(buf)]='\0';

    //SE ENVIA EL RESULTADO AL MAESTRO
    pvm_init send(PvmDataDefault);
    pvm_pkstr(buf2);
    pvm_send(parent_tid, 2); //envio con etiqueta 2

    pvm_exit();
    exit(0);
}

```

Compilamos y ejecutamos, no olvidemos copiar el ejecutable del proceso esclavo *cadescravo* en la máquina que hace de esclavo; o bien compilarlo en la máquina que hace de esclavo y copiarlo en la carpeta **/usr/lib/pvm3/bin/LINUX/**:

```

administrador@ubuntul2-maestro:~$gcc cadmaestro.c -o cadmaestro -lpvm3

administrador@ubuntul2-esclavo:~$ gcc cadescravo.c -o cadescravo -lpvm3
administrador@ubuntul2-esclavo:~$ sudo cp cadescravo
    /usr/lib/pvm3/bin/LINUX/
administrador@ubuntul2-esclavo:~$

```

Desde la máquina que hace de maestro ejecutamos:

```

pvm> spawn -> cadmaestro
spawn -> cadmaestro
[3]
1 successful

```

```

t40003
pvm> [3:t80003] EOF
[3:t40003] ENVÍO A t80003: mensaje en minúscula
[3:t40003] RECIBO DE t80003: MENSAJE EN MINÚSCULA
[3:t40003] EOF
[3] finished

```

ACTIVIDAD 1.12

Para enviar enteros se usa la función `pvm_pkint()`. Por ejemplo para enviar un único entero almacenado en la variable *num* a un proceso escribo `pvm_pkint(&num, 1, 1)`. Escribo un proceso maestro que envíe a un proceso esclavo un número y el esclavo devuelva el cubo del número que recibe.

En los ejemplos anteriores se generaba una copia del proceso esclavo (*hello_other*), el ejecutable se localizaba en *ubuntu12-esclavo*. A continuación copiamos el ejecutable en */usr/lib/pvm3/bin/LINUX* de la máquina que hace de maestro (*ubuntu12-maestro*). Si no hemos creado la carpeta *LINUX* tendremos que crearla y después copiar *hello_other*:

```

administrador@ubuntu12-maestro:~$ gcc hello_other.c -o hello_other
                                -lpvm3
administrador@ubuntu12-maestro:~$ sudo mkdir /usr/lib/pvm3/bin/LINUX/
administrador@ubuntu12-maestro:~$ sudo cp hello_other
                                /usr/lib/pvm3/bin/LINUX/
administrador@ubuntu12-maestro:~$

```

Cambiamos el programa *hello.c*, le llamamos *hello2.c*. En este caso se lanzarán 2 tareas (o procesos) con `pvm_spawn()`, una en cada host, por tanto se generarán dos copias del proceso esclavo: `tareas = pvm_spawn("hello_other", (char**)0, 0, "", 2, tid);`

Después se hace un bucle que recogerá los mensajes enviados por cada uno de los procesos lanzados:

```

#include <stdio.h>
#include "pvm3.h"
#include <stdlib.h>
main()
{
    int tareas, cc, tid[2], i;
    char buf[100];

    printf("PROCESO MAESTRO: t%x\n", pvm_mytid());
    tareas = pvm_spawn("hello_other", (char**)0, 0, "", 2, tid);
    printf("NUMERO DE TAREAS lanzadas :%d \n", tareas);
    //RECIBO MENSAJES
    for(i=0; i< tareas; i++) {
        cc = pvm_recv(tid[i], -1); //se recibe mensaje
        pvm_upkstr(buf); //se desempaqueta
        printf("MENSAJE DE t%x: %s\n", tid[i], buf);
    }
    pvm_exit();
    exit(0);
}

```

A la hora de visualizar el mensaje recibido se puede ver de qué máquina viene, uno de *ubuntu12-esclavo* y el otro de *ubuntu12-maestro*. Lo compilamos y luego lo probamos:

```

administrador@ubuntu12-maestro:~$ gcc hello2.c -o hello2 -lpvm3

pvm> spawn -> hello2
spawn -> hello2
[5]
1 successful
t40005
pvm> [5:t80005] EOF
[5:t40005] PROCESO MAESTRO: t40005
[5:t40005] NUMERO DE TAREAS lanzadas :2
[5:t40005] MENSAJE DE t80005: hello, world from ubuntu12-esclavo
[5:t40005] MENSAJE DE t40006: hello, world from ubuntu12-maestro
[5:t40005] EOF
[5:t40006] EOF
[5] finished

pvm>

```

En el siguiente ejemplo los procesos esclavos realizarán la suma de un array. El proceso maestro envía a los esclavos los elementos del array a sumar. Estos se repartirán entre las dos máquinas que forman PVM de tal forma que una máquina suma los 5 primeros elementos y la otra los 5 siguientes. Habrá dos tipos de mensajes, uno cuando el maestro envía al esclavo y el otro cuando el esclavo envía al maestro (como se vio en un ejemplo anterior), cada uno tendrá una etiqueta diferente. El código del proceso maestro **summaestro.c** es el siguiente:

```

#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
int main()
{
    int tabla[10];    /* Tabla con los números a sumar */
    int tresult[2];   /* resultados de la suma por los esclavos*/
    int tareas, cc, tid[2], i, sum =0 ;
    int etqenvio=1, etqrecibe=2; /*etiquetas de envio y recepcion*/

    printf("PROCESO MAESTRO: t%x\n", pvm_mytid());

    //SE LLENA LA TABLA CON NUMEROS
    for(i = 0; i < 10; i++) tabla[i] = i;

    //SE CREAN LOS ESCLAVOS
    tareas = pvm_spawn("sumesclavo", (char**)0, 0, "", 2, tid);
    printf("TAREAS: %d\n", tareas);

    //SE ENVIAN LOS DATOS
    for(i = 0; i < 2; i++) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(tabla + i*5, 5, 1); //5 num para cada esclavo
        pvm_send(tid[i], etqenvio);    //envía con etiq 1
    }

    //SE RECIBEN LOS DATOS
    for(i = 0; i < 2; i++) {
        cc = pvm_rcv(tid[i], etqrecibe); //recibe con etiq 2
        pvm_upkint(tresult + i, 1, 1); //desempaqueta un entero
        printf("- RECIBO DE t%x: %d\n", tid[i], tresult[i]);
    }
}

```

```

//SE OBTIENE LA SUMA
for(i = 0; i < 2; i++) sum = sum + tresult[i];
printf("LA SUMA ES = %d\n", sum);

pvm_exit();
exit(0);
}

```

El código del proceso esclavo **sumesclavo.c** es el siguiente:

```

#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>
int main()
{
    int mytid, parent_tid;
    int tabla[5]; //tabla para enviar al maestro
    int sum = 0, i;

    mytid = pvm_mytid();
    parent_tid = pvm_parent();

    //SE RECIBEN LOS DATOS DEL MAESTRO
    pvm_recv(parent_tid, 1); //recibo con etiqueta 1 los 5 numeros
    pvm_upkint(tabla, 5, 1);

    //SE CALCULA LA SUMA
    for(i = 0; i < 5; i++) sum = sum + tabla[i];

    //SE ENVIA EL RESULTADO AL MAESTRO
    pvm_init send(PvmDataDefault);
    pvm_pkint(&sum, 1, 1);
    printf("\tESCLAVO t%x Suma = %d ", mytid, sum);
    pvm_send(parent_tid, 2); //envio con la etiqueta 2

    pvm_exit();
    exit(0);
}

```

Lo compilamos y ejecutamos, no debemos olvidar dejar el proceso esclavo en la carpeta **/usr/lib/pvm3/bin/LINUX** de ambas máquinas:

```

administrador@ubuntul2-maestro:~$ gcc summaestro.c -o summaestro -lpvm3
administrador@ubuntul2-maestro:~$ gcc sumesclavo.c -o sumesclavo -lpvm3
administrador@ubuntul2-maestro:~$ sudo cp sumesclavo
                               /usr/lib/pvm3/bin/LINUX/

[sudo] password for administrador:
administrador@ubuntul2-maestro:~$

administrador@ubuntul2-esclavo:~$ gcc sumesclavo.c -o sumesclavo -lpvm3
administrador@ubuntul2-esclavo:~$ sudo cp sumesclavo
                               /usr/lib/pvm3/bin/LINUX/

[sudo] password for administrador:
administrador@ubuntul2-esclavo:~$

```

Desde la máquina que hace de maestro ejecutamos:

```
pvm> spawn -> summaestro
spawn -> summaestro
[6]
1 successful
t40009
pvm> [6:t40009] PROCESO MAESTRO: t40009
[6:t40009] TAREAS: 2
[6:t40009] - RECIBO DE t80009: 10
[6:t40009] - RECIBO DE t4000a: 35
[6:t40009] LA SUMA ES = 45
[6:t40009] EOF
[6:t80009] ESCLAVO t80009 Suma = 10
[6:t80009] EOF
[6:t4000a] ESCLAVO t4000a Suma = 35
[6:t4000a] EOF
[6] finished

pvm>
```

En la siguiente tabla se resumen las funciones de PVM:

Funciones	
Para el control de procesos	pvm_spawn(), pvm_exit(), pvm_kill(), pvm_start pvmd()
De información	pvm_myid(), pvm_parent(), pvm_tidtohost(), pvm_config(), pvm_tasks(), pvm_perror(), pvm_pstat(), pvm_mstat(), pvm_setopt(), pvm_getopt(), pvm_buinfo()
De configuración dinámica de la máquina virtual	pvm_addhosts(), pvm_delhosts()
Para señalización	pvm_sendsig(), pvm_notify()
Para control de buffers	pvm_initsend(), pvm_mkbuf(), pvm_freebuf(), pvm_getsbuf(), pvm_getrbuf(), pvm_setsbuf(), pvm_setrbuf()
Para operaciones colectivas	pvm_joiningroup(), pvm_lvgroup(), pvm_gettid(), pvm_getinst(), pvm_getsize(), pvm_barrier(), pvm_bcast(), pvm_reduce(), pvm_scatter(), pvm_gather()
Para envío y recepción de mensajes	pvm_send(), pvm_psend(), pvm_mcast(), pvm_recv(), pvm_nrecv(), pvm_trecv(), pvm_probe(), pvm_buinfo()
Para empaquetado y desempaqueado de datos (XXX tipo de dato)	En general pvm_pkXXX(), pvm_upkXXX() Para cadenas: pvm_pkstr(), pvm_upkstr()

Más información de las funciones PVM se puede obtener desde esta URL:
<http://www.csm.ornl.gov/pvm/man/manpages.html>.

COMPRUEBA TU APRENDIZAJE

1º) Haz un programa C que genere una estructura de procesos con un padre y 3 hijos (Figura 1.26). Visualiza por cada hijo su PID y el del padre. Visualiza también el PID del padre de todos. Un ejemplo de ejecución se muestra a continuación:

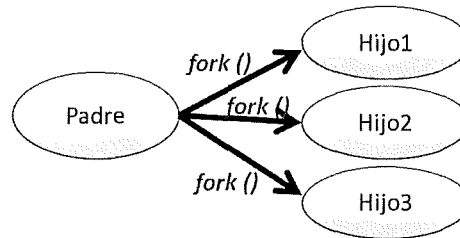


Figura 1.26. Ejercicio 1.

```

mj@ubuntu-mj:~$ gcc ejercicio1_1.c -o ejercicio1_1
mj@ubuntu-mj:~$ ./ejercicio1_1
Soy el hijo= 3, Mi padre es= 5086, Mi PID= 5089
Soy el hijo= 2, Mi padre es= 5086, Mi PID= 5088
Soy el hijo= 1, Mi padre es= 5086, Mi PID= 5087
Proceso PADRE = 5086
  
```

2º) Haz un programa C y crea los pipes necesarios para que la comunicación entre un padre y un hijo fluya en ambos sentidos. Un ejemplo de ejecución se muestra a continuación:

```

administrador@ubuntu1:~$ gcc ejercicio1_2.c -o ejercicio1_2
administrador@ubuntu1:~$ ./ejercicio1_2
PADRE ENVIA MENSAJE.
    HIJO RECIBE MENSAJE de PADRE: Saludos del Padre..
    HIJO ENVIA MENSAJE a su PADRE.
PADRE RECIBE MENSAJE del HIJO: Saludos del Hijo...
administrador@ubuntu1:~$
  
```

3º) Busca información sobre los comandos *top* y *free* de Linux y utilízalos para obtener información de los procesos.

4º) Realiza un programa Java que admita argumentos desde *main()* y devuelva con *System.exit()* los siguientes valores:

- Si el número de argumentos es < 1 debe devolver 1.
- Si el argumento es una cadena debe devolver 2.
- Si el argumento es un número entero menor que 0 debe devolver 3.
- En cualquier otra situación debe devolver 0.

Realiza un segundo programa Java que ejecute al anterior. Este segundo programa deberá mostrar en pantalla lo que pasa dependiendo del valor devuelto al ejecutar el programa anterior

Prueba los programas desde el entorno Eclipse. A continuación, crea una carpeta en el disco duro y almacena los dos programas. Realiza los cambios necesarios para compilarlos y ejecutarlos desde la línea de comandos del DOS.

5º) Crea un programa Java que visualice 5 veces la cadena que se le envía desde los argumentos de *main()*. Si no se le envía ninguna cadena que muestre un mensaje indicándolo y que finalice el programa con *System.exit(1)*.

A continuación, crea un segundo programa Java que introduzca por teclado una cadena y ejecute el programa anterior para visualizar 5 veces esa cadena.

6º) Partiendo del ejercicio anterior, realiza los cambios necesarios para que la cadena introducida por teclado se almacene en un fichero de texto, no mostrándola en pantalla.

Guarda los programas en otra carpeta y compílalos y ejecútalos desde la línea de comandos del DOS.

7º) Crea un programa Java que lea cadenas desde la entrada estándar hasta escribir un *. A continuación, crea otro programa que ejecute el anterior.

8º) Realiza un programa Java que lea una cadena desde la entrada estándar y visualice en pantalla si la cadena es o no palíndromo o si la cadena está vacía (la longitud es 0).

Realiza un segundo programa Java que ejecute el anterior, debe leer la cadena desde teclado y mostrar la salida por pantalla. Transforma este ejercicio para que la cadena se obtenga de un fichero de texto, y se envíe la salida de error a un fichero.

9º) Modifica el Ejercicio 7 para que al ejecutar el programa la entrada al proceso se obtenga a partir de un fichero de texto.

10º) A partir del siguiente conjunto de instrucciones indica las que se pueden ejecutar concurrentemente y las que no:

Instrucción 1:	$a := x + y;$
Instrucción 2:	$b := z - 1;$
Instrucción 3:	$c := a - b;$
Instrucción 4:	$w := c + 1;$

11º) ¿Cuáles de las siguientes afirmaciones son falsas?:

- Un programa concurrente se suele concebir como un conjunto de procesos que colaboran y compiten entre sí.
- Las sentencias de un programa concurrente se ejecutan de acuerdo con un orden estricto.
- Un programa paralelo es un tipo de programa concurrente diseñado para ejecutarse en un sistema multiprocesador.
- En los programas concurrentes no existen múltiples líneas de flujo de control.
- En los programas concurrentes las sentencias que constituyen el programa no se ejecutan siguiendo una ordenación que corresponde a una secuencia temporal lineal.

12º) Indica una breve descripción de lo que hacen estas funciones *pwm_spawn()*:

- `numt = pwm_spawn(argv[0], (char**)0, PvmTaskDefault, "", n, tids);`
- `numt = pwm_spawn(argv[0], (char**)0, 0, "", n, tids);`
- `numt = pwm_spawn("esclavo", (char**)0, PvmTaskHost, "ulises.dia.uned.es", 1, &tids);`

13º) Modifica el programa **summaestro.c** para que un proceso esclavo sume los elementos del array que están en posición par y el otro sume los que están en posición impar.

14º) Se trata de realizar la multiplicación de 2 arrays de números enteros de dimensión 10. El resultado de la multiplicación es un nuevo array donde el elemento de posición 0 es igual a la multiplicación de los elementos de posición 0 de los dos arrays, el de posición 1 multiplica las posiciones 1 de los dos arrays, y así sucesivamente. Realizar un programa maestro que inicialice los arrays con números y envíe a dos procesos esclavos la mitad de cada array. Los procesos esclavos deben devolver la multiplicación de las partes de los arrays que les ha correspondido.

15º) Busca información sobre la función **pvm_config()** y utilízala en un programa C para mostrar información sobre la configuración de la máquina virtual como por ejemplo el número de hosts que forman la máquina virtual, el tipo de arquitectura, el nombre de cada host, etc.

ACTIVIDADES DE AMPLIACIÓN

Crea un entorno PVM en clase con varias máquinas que tengan instalado el sistema operativo Linux y prueba los ejercicios que se han visto en clase. El ordenador del profesor puede hacer de maestro y debe tener los procesos maestros, los de los alumnos tendrán los procesos esclavos. Para el caso de la suma del array defínelo con suficiente tamaño para que todos los esclavos sumen un número considerable de elementos. Por ejemplo, si tenemos 5 máquinas esclavos definimos un array de 500 elementos, cada máquina que sume $500/5$ elementos del array.