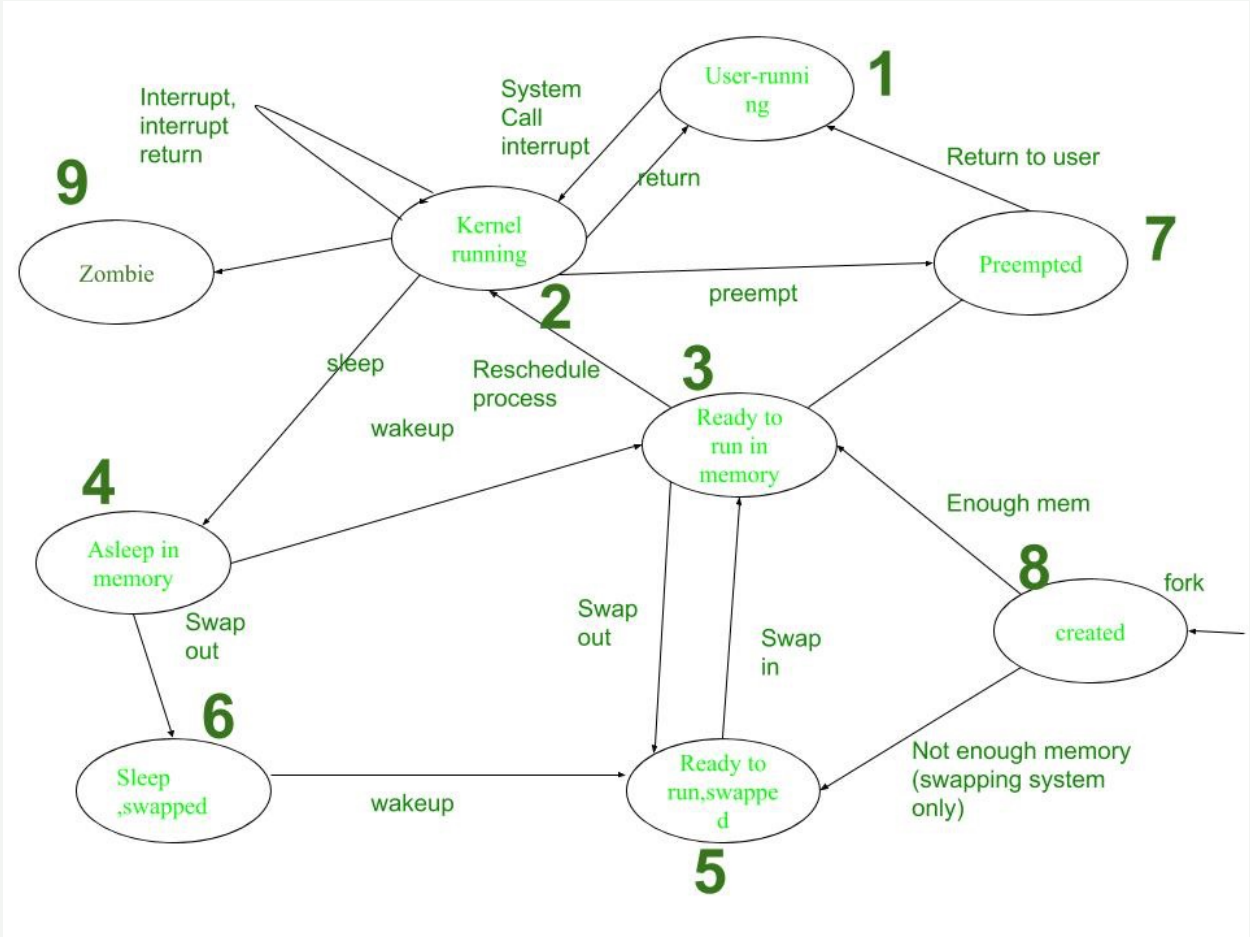
Gestión de procesos

SISTEMAS LINUX



Gestión de procesos. ¿Qué es un proceso?

Un proceso es cualquier programa en ejecución totalmente independiente de otros procesos o, también como la unidad de procesamiento gestionada por el sistema operativo.

Un proceso mantiene por cada proceso estructuras con información del proceso:

- Identificador, características, recursos asignados (memoria, descriptores de ficheros abierto, descriptores de puertos de comunicación, …
- El bloque de control de proceso (BCP) contiene gran parte de los datos anteriores

Tras el arranque del sistema se genera una jerarquía de procesos. Se usan los términos padre e hijo para hacer referencia a estas relaciones.

```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 oct18 ?        00:00:01 /sbin/init
root           2       0  0 oct18 ?        00:00:00 [kthreadd]
root           3       2  0 oct18 ?        00:00:00 [rcu_gp]
root           4       2  0 oct18 ?        00:00:00 [rcu_par_gp]
root           6       2  0 oct18 ?        00:00:00 [kworker/0:0H-events_highpri
```

Todo proceso:

- Tiene asociado un espacio de direcciones que contiene el código, los datos del programa y su pila.
- Se ejecuta “simultáneamente” a otros procesos

Cuando UNIX comienza su ejecución únicamente existe un proceso en el sistema. Este proceso se llama ***init*** y su identificador de proceso (PID) es 1.

Gestión de procesos. Vida de un proceso

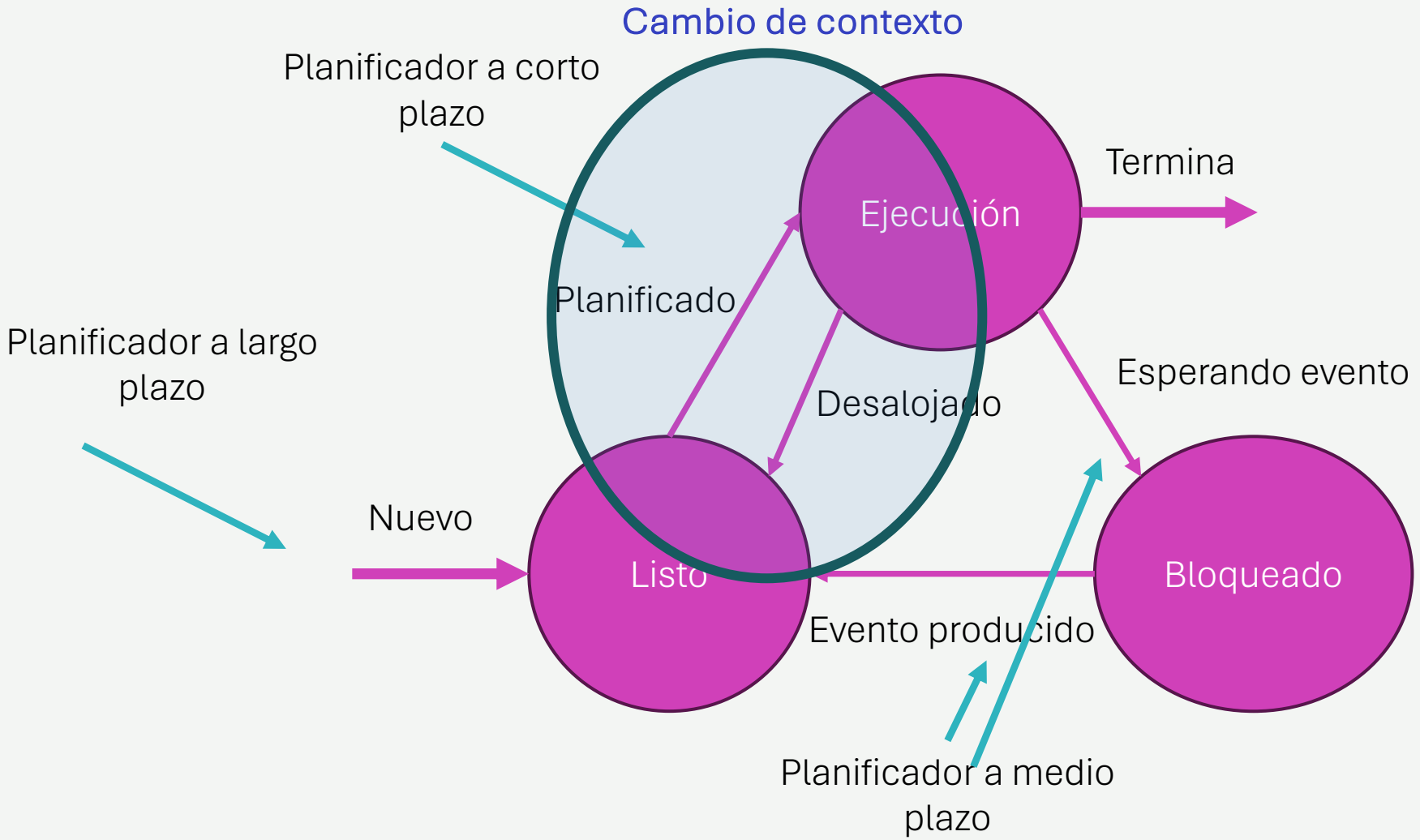
La vida de un proceso consiste:

- **Creación**: se utilizan llamadas al sistema **fork** o **exec**
  - Asignación de espacio en memoria
  - Encontrar un BCP libre donde guardar su información
  - Cargar en memoria código, datos y pila
- **Ejecución**: un proceso no se ejecuta de una vez, la ejecución incluye interrupciones (hardware o software) y activaciones del proceso

  Cuando un proceso está en ejecución parte de su información está en los registros de la máquina.
  - Una interrupción implica salvaguardar la información del proceso
  - La activación implica restaurar ese contexto en el mimo punto en que fue interrumpido
- **Muerte** o terminación: bien porque ha terminado su ejecución o porque se ha decidido que ya no es necesario. En cualquier caso, los recursos que tenía asignados deben ser liberados. La llamada al sistema **kill** permite el envío de señales de finalización a un proceso dado.

Gestión de procesos. Vida de un proceso

Sistema de estado básico de un proceso



Las notificaciones de eventos en Linux/Unix se realizan mediante el mecanismo de señales

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Los principales servicios y sus correspondientes prototipos en lenguaje C son: Identificador de procesos, entorno de procesos, creación de procesos y finalización de procesos

- **Identificación de procesos**:
  - Cada proceso se identifica mediante un valor entero denominado **identificador de proceso** del tipo **pid_t** y tiene un identificador de usuario y grupo que son del tipo **uid_t**.
  - Servicios proporcionados
    - **pid_t getpid(void):** Devuelve el identificador del proceso que realiza la solicitud.
    - **pid_t getppid(void):** Devuelve el identificador del proceso padre que realiza la solicitud.
    - **uid_t getuid(void):** Devuelve el identificador del usuario real (invocador) del proceso que hace la solicitud
    - **uid_t geteuid(void):** Devuelve el identificador del usuario efectivo (ejecuta) del proceso que hace la solicitud
    - **uid_t getgid(void):** Devuelve el identificador del grupo real del proceso que hace la solicitud
    - **uid_t getegid(void):** Devuelve el identificador del grupo efectivo del proceso que hace la solicitud

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo: Creación de un programa en C que visualice los identificadores del proceso que realiza la petición y el identificador de su proceso padre

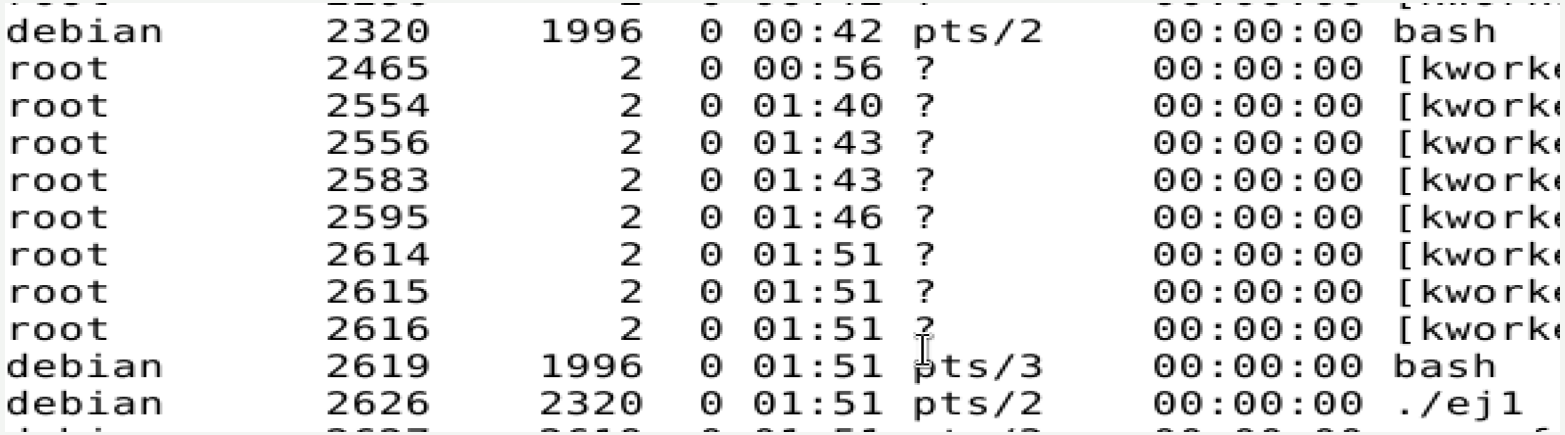
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
 pid_t idproc, idproc_padre;
 idproc=getpid();
 idproc_padre=getppid();
 printf("Identificador del proceso: %d \n", idproc);
 printf("Identificador del proceso padre: %d \n", idproc_padre);
 return 0;
}
```

```
debian@debian:~/so2324$ ./ej1
Identificador proceso: 2612
Identificador proceso padre: 2320
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Abrir una pestaña al terminal y en el código, añadir antes de **return 0** una sentencia **sleep(15),** compilar, ejecutar el programa, id a la pestaña nueva y ejecutad el comando **ps –ef.** Así, podéis comprobar la jerarquía mostrada del proceso en ejecución

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
 pid_t idproc, idproc_padre;
 idproc=getpid();
 idproc_padre=getppid();
 printf("Identificador del proceso: %d \n", idproc);
 printf("Identificador del proceso padre: %d \n", idproc_padre);
 sleep(15);
 return 0; }
```



```
debian      2320    1996  0 00:42 pts/2    00:00:00 bash
root        2465       2  0 00:56 ?        00:00:00 [kworke
root        2554       2  0 01:40 ?        00:00:00 [kworke
root        2556       2  0 01:43 ?        00:00:00 [kworke
root        2583       2  0 01:43 ?        00:00:00 [kworke
root        2595       2  0 01:46 ?        00:00:00 [kworke
root        2614       2  0 01:51 ?        00:00:00 [kworke
root        2615       2  0 01:51 ?        00:00:00 [kworke
root        2616       2  0 01:51 ?        00:00:00 [kworke
debian      2619    1996  0 01:51 pts/3    00:00:00 bash
debian      2626    2320  0 01:51 pts/2    00:00:00 ./ej1
```

Continuación: Dentro del mismo programa visualiza los identificadores de usuario real y efectivo, así como los identificadores de grupo real y efectivo.

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

- **Entorno de procesos**

Se encuentra definido por una lista de variables, denominadas **variables de entorno** y que se envían al ejecutar el programa en la variable global **environ,** cuya definición es: **extern global **environ;**

environ contiene un conjunto de cadenas de caracteres de la forma ***nombre=valor***, nombre es es nombre de la variable y valor el valor asignado.

Ejemplo: programa que muestra las variables de entorno del programa ejecutado

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

extern char **environ;
int main(int argc, char *argv[])
{
    for (int i=0;environ[i]!=NULL; i++)
    {
    printf("Variable entorno[%d]: %s \n",i,environ[i]);
    }
    return 0;
}
```

La llamada al sistema **getenv** obtiene el valor de una variable de entorno dada **char * getenv (const char *nombre)**

Ejemplo de uso: char *ruta;

path = getenv(“PATH”);

Si devuelve NULL, la variable de entorno no estaría definida.

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

- **Creación de procesos**

La creación de procesos en Linux/UNIX se realiza invocando a la llamada al sistema **fork**.

Formato: **pid_t fork();**

Fork clona el proceso que realiza la solicitud que se convierte en el padre del nuevo proceso creado. ***Fork se invoca una vez, pero retorna dos veces, una vez al proceso padre y otra al proceso hijo.***

Atributos que hereda el proceso hijo del padre

- Entorno
- Directorio actual y directorio raíz
- Señales capturadas
- SUID y SGID
- Estado de privilegios y prioridades
- Librerías compartidas
- Máscara y límites de medida de ficheros.....

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

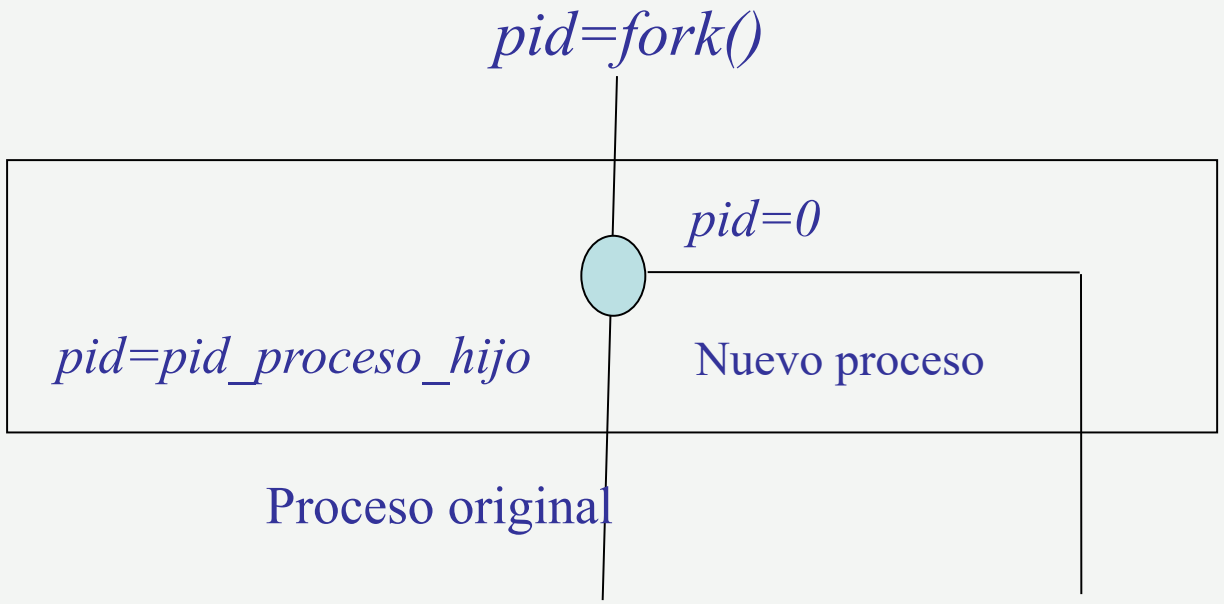
El proceso nuevo o hijo no es totalmente idéntico al proceso padre. Las principales diferencias son:

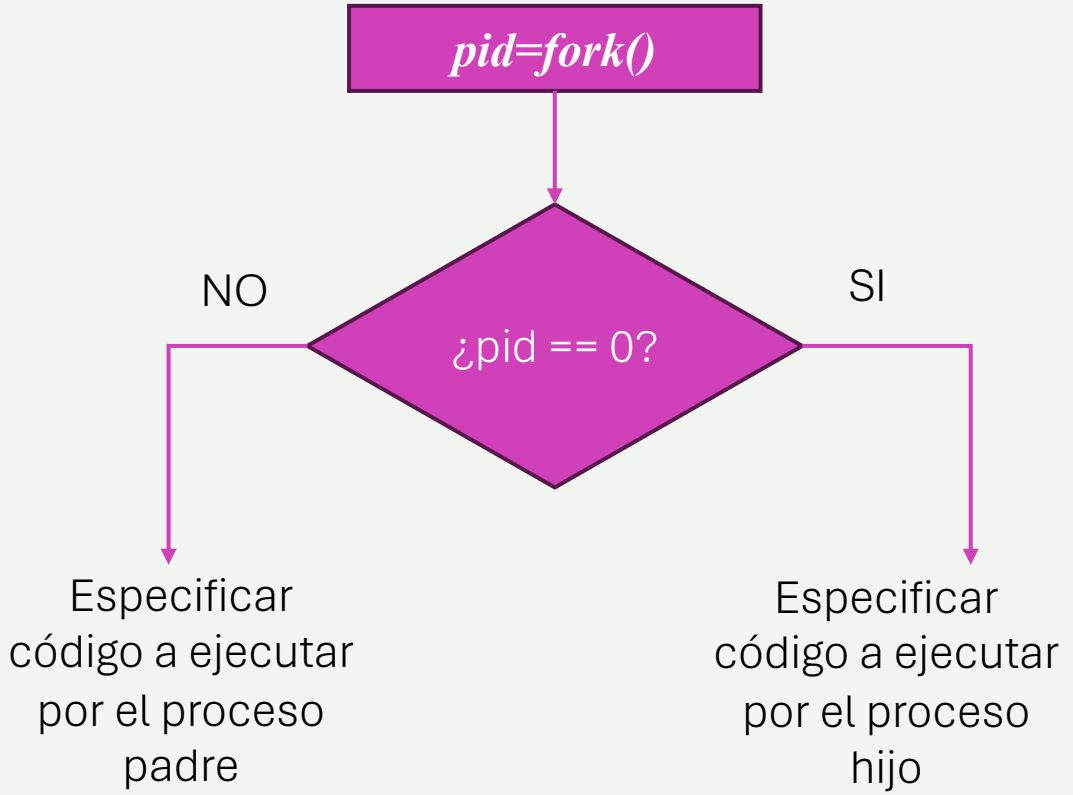
- Identificador del proceso hijo (PID) es diferente y único
- Identificador del proceso padre (PPID) es diferente (PPID del hijo coincide con el PID del padre)
- Copia propia de los descriptores de ficheros del padre
- Bloqueos de proceso, texto y datos no se heredan
- Las subrutinas times se ponen a 0
- Las alarmas pendientes toman su valor inicial
- Se eliminan las señales pendientes para el proceso hijo

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Diagrama de invocación a fork

```
pid_t pid;
```





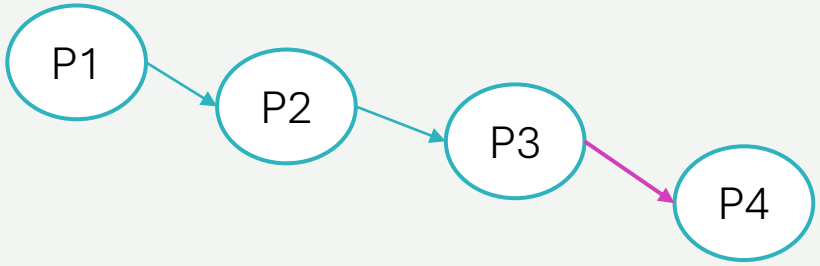
Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo: Creación de un proceso hijo, controlando si se ha podido crear o no el proceso hijo y en cada proceso visualizar el identificador de proceso y el identificador del proceso padre.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]){
 pid_t pid;
 pid=fork();
 switch (pid){
   case -1:
    printf("Error al crear el proceso\n");
    break;
case 0:
 printf("Proceso hijo con id. de proceso: %d  e id. del proceso padre
%d \n",getpid(),getppid());
 break;
default:
    printf("Proceso padre (invocador) con id. de proceso: %d  e id. del
      proceso padre %d \n",getpid(),getppid());
    break;
 }
}
```

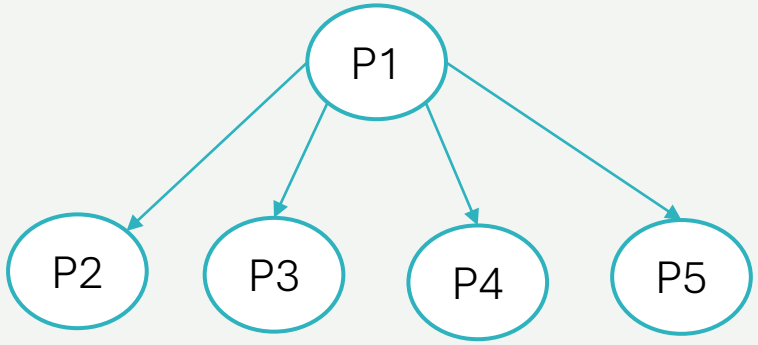
Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejercicio: Crear una cadena de 4 procesos, donde cada proceso sea el proceso hijo del anteriormente creado



El proceso principal que se genera con la ejecución del programa a implementar sólo genera el primer proceso, a partir de ahí cada proceso hijo crea el siguiente proceso

Ejercicio: Crear la siguiente secuencia de procesos, donde el proceso correspondiente al programa desarrollado es padre de todos y cada uno de los siguientes.



Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Familia de llamadas exec(). La familia de llamadas al sistema *exec()* reemplazan la imagen del proceso que las invoca (código, datos y pila) por la del programa ejecutable indicado en el argumento *ruta*.

- **int execl (ruta, arg0, arg1,..., argn, NULL);** **char *ruta, *arg0, *arg1, ..., *argn ;**

ruta = ruta donde se encuentra el ejecutable/binario

arg0 = nombre del binario

arg1, …, argn = parámetros que recibe el binario

- **int execv(ruta, argv);** **char *ruta, *argv[] ;**
- **int execlp(ruta, arg0, ..., argn, NULL);** **char *ruta, *arg0, ..., *argn ;**
- **int execvp(ruta, argv);** **char *ruta, *argv[] ;**

argv=lista con todos y cada uno de los parámetros que recibe el programa, argv[0]=nombre del programa

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo con execlp, el último parámetro de execlp debe ser un puntero a NULL

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

int main(int argc, char *argv[]) {
 pid_t pid;
 int estado;
 pid=fork();
 switch (pid){
  case -1:
   printf("Error al crear el proceso\n");
   break;
 case 0:
   execlp("ls","ls","-l",NULL);
   perror("Al ejecutar exec");
   return 1;
  default:
   printf("Proceso padre (invocador) con id. de proceso:
%d  e id. del proceso padre %d \n",getpid(),getppid());
   break;
  }
  return 0;
}
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo exec 2. Programa que ejecuta otro programa pasado como parámetro utilizando la llamada **execvp**

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

int main(int argc, char *argv[])
{
 pid_t pid;
 int estado;
 pid=fork();
 switch (pid){
   case -1:
    printf("Error al crear el proceso\n");
    break;
   case 0:
    execvp(argv[1],&argv[1]);
    perror("Al ejecutar exec");
    return 1;
   default:
    printf("Proceso padre (invocador) con id. de proceso: %d  e id. del proceso padre %d \
n",getpid(),getppid());
    break;
   }
   return 0;
}
```

```
$ ./ej5 ps -eo user,pid,ppid,cmd
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

- **Finalización de procesos**

Un proceso puede terminar de forma normal o no. Las tres formas de terminar son:

- Finalizando con una sentencia **return** (p.e. return 2)
- Ejecutando la función **exit** (p.e. exit(0))
- Mediante la llamada al servicio _exit

exit es una función que llama al servicio **_exit.** Además, cuando una función main termina con **return x**, el efecto es el mismo que cuando hace **exit(x).**

Todo proceso debe finalizar correctamente utilizando para ello la llamada a la función **exit(x)**, donde **x** se utiliza para dar información del motivo por el que el proceso ha finalizado.

Mediante la rutina **atexit** es posible configurar que se ejecute una función cuando se realiza una llamada a **exit**

Algunas de las importantes operaciones de exit son:

- Cerrar todos los descriptores de ficheros
- Si el proceso padre está esperando por las sentencias wait/waitpid, será notificado de la finalización
- Se liberan todos los recursos utilizados por el proceso que finaliza
- Se liberan todas las áreas de entrada salid

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

- **Ejemplo de uso de exit y atexit**

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

void finalizar()
{
  printf("Finalizando programa con pid: %d cuyo padre es %d \n",getpid(),getppid());
}
int main(void){
        if (atexit(finalizar)!=0)
        {
                printf("Error al configurar rutina atexit\n");
                exit(-1);
        }
  printf("Se está ejecutando el programa con pid: %d cuyo padre es %d \n",getpid(),getppid
());
  printf("Antes de llamar a exit\n");
  exit(1);
}
debian@debian:~/so2324$ ./ej6
Se está ejecutando el programa con pid: 3710 cuyo padre es 2320
Antes de llamar a exit
Finalizando programa con pid: 3710 cuyo padre es 2320
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

- **¿Cómo esperar a que un proceso finalice y comprobar la causa por la que ha finalizado?**

Servicio wait y waitpid. Ambos permiten bloquear al proceso padre hasta que un proceso hijo haya finalizado y permiten obtener información del estado de finalización del mismo. Se encuentra en la librería **sys/wait.h**

Formato: **pid_t wait(int *status)**

pid_t waitpid(pid_t pid, int *status, int options)

- **wait**: pone a la espera al proceso que lo invoca hasta que finalice el primer proceso hijo que lo haga. Devuelve el identificador del proceso hijo que ha finalizado y en **status** el motivo por el que ha finalizado.
- **waitpid**: funciona igual, si en el parámetro pid=-1 y options=0, si pid>0 se espera a la finalización del proceso indicado, si pid<-1 se espera al proceso con pid =|número indicado|. Options puede ser WNOHANG o WUNTRACED
  - **WNOHANG:** evita la suspensión del padre mientras esté esperando a algún hijo.
  - **WUNTRACED:** el padre obtiene información adicional si el hijo recibe alguna de las señales SIGTTIN, SIGTTOU, SIGSSTP o SIGTSTOP

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Las siguientes macros permiten consultar el valor de la variable **status**:

- **WIFSTOPPED (estado)** devuelve !=0, si *estado* es de un hijo parado.
- **int WSTOPSIG (estado)** devuelve el número de señal que ha causado la parada.
- **WIFEXITED (estado)** devuelve !=0, si *estado* es de salida normal.
- **int WEXITSTATUS (estado)** devuelve los 8 bits altos del estado de salida.
- **WIFSIGNALED (estado)** devuelve !=0, si *estado* es de salida anormal.
- **WTERMSIG (estado)** devuelve el número de señal que ha causado la salida

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo

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

int main(int argc, char *argv[])  {
 pid_t pid;
 int estado;
 long suma;
 pid=fork();
 switch (pid){
   case -1:
    printf("Error al crear el proceso\n");
    break;
case 0:
    srand(getpid());
    printf("Proceso hijo con id. de proceso: %d
  e id. del proceso padre %d \n",getpid(),getppid());
   for (int i=0;i<1000;i++) {
    suma=suma+rand()%(1000+1);
    usleep(200);
   }
   printf("Hijo calculo = %ld \n",suma);
   exit(5);
 default:
  printf("Proceso padre (invocador) con id. de proceso: %d  e id. del
proceso padre %d \n",getpid(),getppid());
  wait(&estado);
   if (estado==0) printf("Proceso hijo finalizado normalmente\n");
    else  printf("Proceso hijo finalizado anormalmente\n");
          if (WIFEXITED(estado))
            printf("Proceso hijo finalizado por llamada a exit(%d)
                    \n",WEXITSTATUS(estado));
          if (WIFSIGNALED(estado))
            printf("Proceso hijo finalizado por recepción de señal %d \n",
                    WTERMSIG(estado));
          break;
    }
    return 0;
}
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Trabajando con señales

- Las señales permiten que un proceso ejecute código de una manera asíncrona, es decir el orden de ejecución dentro del programa no es lo determinante, una acción se planifica para que se realice cuando se recibe una determinada señal
- Podemos definir las Señales como recursos para manejar condiciones excepcionales. Cada señal corresponde a una condición distinta. Una señal puede ser generada por una interrupción desde teclado, un error en un proceso o por varios eventos asíncronos (cronómetros o señales de control de trabajos del shell).
- Existen señales POSIX.1b, señales System V, y varias variantes de señales BSD.

```
debian@debian:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Trabajando con señales

- **Armar una señal**: es la configuración por la que un proceso se pone a la escucha de una señal y define la acción a realizar en el caso de recibirla.

¿Cómo arma una señal un proceso? **signal(num_sig, funcion)**:

Está en la librería **signal.h**.

La función a ejecutar como segundo parámetro se denomina también manejador de señal. Si no se especifica, se puede poner:

- **SIG_IGN:** Al recibir la señal el proceso no hará nada y seguirá su ejecución.
- **SIG_DFL:** Al recibir la señal, se ejecutará la acción por defecto.

/* Espera que se pulse CTRL_C, que envía la señal SIGINT al proceso en ejecución */

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

void manejador_senial_sigint(int);
int main (void) {
  // armar señal SIGINT
  signal (SIGINT, manejador_senial_sigint);

while (1) {
   printf (”Mensaje cuando se pulsa Ctrl-C \n");
   sleep(2);
 }
}
void manejador_senial_sigint(int senial) {
   printf ("Senial recibida: %d \n", senial);
  exit(0);
}
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

¿Cómo envía un proceso una señal a otro proceso?

kill(id_proceso, num_sig)

Las señales pueden sincronizar procesos, como el ejercicio que tenéis a la derecha, donde proceso padre e hijo tienen asignadas unas tareas y en un momento dado, el padre envía la señal SIGINT al hijo y esto desencadena una serie de acciones en el hijo, que responda con el envío de una señal al padre y finalice. El padre al recibir la comunicación del hijo también finaliza.

¿Cómo esperar a que suceda un evento? **pause()**

El proceso que la llama espera a que le llegue una señal, cuando llega ejecuta el manejador de señal y el proceso continúa hasta que encuentre la siguiente llamada a pause. (La utilizaremos en otro ejemplo)

```
/* Un proceso que crea un proceso hijo. El proceso hijo no finaliza hasta que el proceso
padre no le envía la señal sigint. Cuando el proceso hijo recibe SIGINT, envia al padre la
señal SIGUSR1  */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void manejador_senial_sigint(int);
void manejador_senial_sigusr1(int);

int main(int argc, char* argv[]) {
 int pid;
 signal(SIGUSR1,manejador_senial_sigusr1);
 printf("Proceso padre ejecutándose, crea un proceso hijo\n");
 pid=fork();
 switch(pid) {
  case -1: printf("Error al crear proceso\n");
           break;
  case 0: signal(SIGINT,manejador_senial_sigint);
          printf("Proceso hijo: ejecutándose hasta recibir SIGINT del padre\n");
          while(1) {
            printf("Hijo %d: mensaje hasta recibir SIGINT\n",getpid());
            usleep(500);
           }
          exit(0);
  default:printf("Proceso padre: continúa hasta recibir SIGUSR1 del hijo \n");
          for (int i=0;i<5;i++) {
            printf("Padre %d: ejecuta tarea \n",getpid());
            usleep(500);
           }
          kill(pid,SIGINT);
          while(1) {
            printf("Padre %d: mensaje hasta recibir SIGUSR1 \n",getpid());
            usleep(300);
           }
          exit(0);
 }
}
void manejador_senial_sigint(int senial){
 int p=getpid();
 printf("Proceso hijo: %d recibe señal SIGINT (%d) \n",p,senial);
 printf("Proceso hijo: %d envía SIGUSR1 al padre\n",p);
 kill(getppid(),SIGUSR1);
 exit(0);
}
void manejador_senial_sigusr1(int senial){
 printf("Proceso padre: %d recibe SIGUSR1 (%d) enviada por el hijo\n",getpid(),senial);
 exit(0);
}
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

¿Otro forma de armar señales optimizada?

int sigaction(int sig, const struct sigaction *a, struct sigaction *old) */* librería signal.h*/*

```
struct sigaction {
        void (*sa_handler)(int); /* manejador de señal, SIG_IGN o SIG_DFL */
        sigset_t sa_mask;        /* señales adicionales a bloquear cuando se maneje ésta*/
        int sa_flags;            /* Indica si la señal se pondrá en la cola para que no se pierda */
    }
```

- Configura el manejador de señal en el campo **sa_handler**
- La señal a armar se especifica en el parámetro **sig**
- La configuración del manejador va en la estructura **a**
- **old** es una salvaguarda de una configuración anterior, que se podría recuperar en caso de error en una nueva configuración

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

void manejador_senial_sigint(int);

int main(int argc, char* argv[])
{
 struct sigaction a;

 a.sa_handler=manejador_senial_sigint;
 sigemptyset(&a.sa_mask);
 a.sa_flags=0;
 sigaction(SIGINT,&a,NULL);
 while(1)
  {
   printf("Mensaje hasta que se pulse CTRL_C\n");
   sleep(3);
  }
}
void manejador_senial_sigint(int senial)
{
 printf("Señal recibida: %d \n",senial);
 exit(0);
}
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Comunicación entre procesos

- La comunicación entre procesos se produce mediante varias herramientas:
  - Ficheros
  - **Pipes**
  - **FIFOS**
  - **Variables en memoria compartida**
  - Paso de mensajes
  - Sockets

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Comunicación entre procesos. PIPES

Crea un canal de comunicación entre procesos emparentados. Librerías **unistd.h, fcntl.h, time.h**

El canal de comunicación tiene un extremo de escritura (emisor escribe) y un extremo de lectura (receptor lee)

- Declaración del canal de comunicación: **int descriptores[2];**
  - descriptores[0] -> se reserva para el descriptor de fichero asociado al extremo de lectura
  - descriptores[1] -> se reserva para el descriptor de fichero asociado al extremo de escritura

- Creación de una tubería o PIPE: **pipe (descriptores)**
  - Parámetros recibidos por la llamada al sistema **pipe**: vector de dos posiciones donde guardará los descriptores de lectura y escritura de la tubería.
  - Devuelve: **0** si se ha completado correctamente; **-1** en caso de error.

- Escribir en una tubería o PIPE: **write (descriptores[1], datos, tamaño)**
- Leer de una tubería o PIPE: **read (descriptores[0], datos, tamaño)**

Parámetros:

- **descriptores[x]:** extremo asociado a la posición de la operación a realizar (0->escritura, 1->lectura)
- **datos:** variables con los datos a escribir/leer de la tubería
- **tamaño:** nº de bytes a escribir **(sizeof(datos)** -> devuelve el nº de bytes, es decir el tamaño)

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Notas:

- Si se intenta leer de una tubería que está vacía, el proceso que intenta leer quedará bloqueado.
- Si se intenta escribir en una tubería y la tubería está llena, el proceso que intenta escribir quedará bloqueado.
- Si un proceso sólo va a utilizar uno de los descriptores de la tubería, el otro extremo puede cerrarlo.
- Si un proceso crea otro, el proceso hijo tiene una copia de la tubería, aunque el proceso padre cierre uno de los extremos, para él hijo sigue abierto y utilizable.
- Las lecturas y escrituras han de ser atómicas, tamaño y número de las operaciones a realizar debe ser analizado
- La sincronización de los procesos que se comunican por tuberías se realiza mediante el envío de señales:

El proceso emisor escribe en la tubería y envía una señal al proceso receptor para que lea de la tubería al recibir la señal.

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo comunicando valores numéricos:

```c
/*Crea un proceso y le envía 5 valores aleatorios, sincronizando con señales, el hijo al
 * recibirlo lo muestra. El hijo finaliza cuando el padre le envía la señal SIGTERM. */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <time.h>
#include <sys/wait.h>
#include <sys/types.h>

void manejador_senial_sigterm(int);
void manejador_senial_sigint(int);
int descriptores[2];

int main(int argc, char* argv[])
{
 // Las variables declaradas antes de crear un proceso, la heredan los hijos el valor actual
 struct sigaction a;
 pid_t p;
 int aleatorio,estado;

 pipe(descriptores);  // creación de la tuberia
 p=fork();          // creación proceso nuevo
 switch(p) {
  case -1: printf("Error al crear el proceso\n");
           break;
  case 0:
        // armado señales en el hijo. Armar señales es lo primero que debe hacer un proceso
        a.sa_handler=manejador_senial_sigterm;  // armado señal SIGTERM para finalizar
        sigemptyset(&a.sa_mask);
        a.sa_flags=0;
        sigaction(SIGTERM,&a,NULL);

        a.sa_handler=manejador_senial_sigint; // armado señal SIGINT para leer de la tuberia
        sigemptyset(&a.sa_mask);
        a.sa_flags=0;
        sigaction(SIGINT,&a,NULL);

        printf("Proceso hijo: %d ejecutandose \n",getpid());
        sleep(1);
        while(1)
          {
            printf("Hijo:realizando tareas hasta recibir datos del padre\n");
            pause();
          }
        break;
   default :
        srand(getpid()); // inicializar semillas para generar números aleatorios
        printf("Proceso padre: %d va a comenzar a enviar datos al hijo: %d \n",getpid(),p);
        for (int i=0; i<5; i++)
          {
           aleatorio=rand()%1000;                       // genera número aleatorio entre 0 y 999
           printf("Enviando el dato %d al proceso nuevo %d \n",aleatorio, p);
           write(descriptores[1],&aleatorio,sizeof(aleatorio)); // lo escribe en la tubería
           kill(p,SIGINT);                              // Envío de la señal para sincronizar lectura
           usleep(300);
          }
        printf("Proceso padre: %d ha terminado de enviar datos y envía señao de finalización al hijo: %d
\n",getpid(),p);
        sleep(1);
        kill(p,SIGTERM);
        wait(&estado);
        exit(0);
   }
}
void manejador_senial_sigint(int senial)
{
 int dato;
 pid_t pid;
 pid=getpid();
 read(descriptores[0],&dato,sizeof(dato));
 printf("Proceso hijo: %d leyendo el dato %d enviado por el padre\n",pid,dato);
}

void manejador_senial_sigterm(int senial)
{
 int dato;
 pid_t pid,ppid;
 pid=getpid();
 ppid=getppid();
 printf("Proceso hijo: %d recibida señal de finalización de mi padre: %d \n",pid,ppid);
 exit(0);
}
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo comunicando cadena:

```c
/* Crear un proceso hijo; el padre le envía 5 mensajes de texto al hijo, que al recibirlo lo muestra
 * en pantalla. El hijo no finaliza hasta que el proceso padre no le envía la señal SIGTERM. */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <time.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <string.h>

void manejador_senial_sigterm(int);
void manejador_senial_sigint(int);
int descriptores[2];

int main(int argc, char* argv[])
{  // Variables declaradas antes de crear un proceso, los hijos heredan el valor actual
 struct sigaction a;
 pid_t p;
 int aleatorio,estado;
 char mensaje[60];

 pipe(descriptores);  // creación de la tuberia
 p=fork(); // creación proceso nuevo
 switch(p) {
  case -1: printf("Error al crear el proceso\n");
           break;
  case 0:
        // armado de las señales en el hijo. Armar señales es lo primero que debe hacer un proceso
        a.sa_handler=manejador_senial_sigterm; // armado señal SIGTERM para finalizar
        sigemptyset(&a.sa_mask);
        a.sa_flags=0;
        sigaction(SIGTERM,&a,NULL);

        a.sa_handler=manejador_senial_sigint; // armado señal SIGINT para leer de la tuberia
        sigemptyset(&a.sa_mask);
        a.sa_flags=0;
        sigaction(SIGINT,&a,NULL);

        printf("Proceso hijo: %d ejecutandose \n",getpid());
        sleep(1);
        while(1)
          {
            printf("Hijo:realizando tareas hasta recibir datos del padre\n");
            pause();
          }
        break;
   default: srand(getpid()); // inicializar semillas para generar números aleatorios
            printf("Proceso padre: %d va a comenzar a enviar datos al hijo: %d \n",getpid(),p);
           for (int i=0; i<5; i++)
            {
             aleatorio=rand()%100;                    // genera número aleatorio entre 0 y 999
             sprintf(mensaje,"Texto que contiene: %d aleatoriamente generado",aleatorio);
             printf("Mensaje a enviar: (%s) al proceso nuevo %d \n",mensaje, p);
             write(descriptores[1],mensaje,60); // lo escribe en la tubería
             kill(p,SIGINT);                          // Envío de la señal para sincronizar lectura
             usleep(300);
            }
          printf("Proceso padre: %d ha terminado de enviar datos y envía señao de finalización al hijo: %d \n
",getpid(),p);
          sleep(1);
          kill(p,SIGTERM);
          wait(&estado);  // Espera a que finalice el hijo
          exit(0);
   }
}
void manejador_senial_sigint(int senial)
{
 char dato[60];
 pid_t pid;
 pid=getpid();
 read(descriptores[0],dato,60);
 printf("Proceso hijo: %d leyendo el mensaje: (%s) enviado por el padre\n",pid,dato);
}
void manejador_senial_sigterm(int senial)
{
 int dato;
pid_t pid,ppid;
 pid=getpid();
 ppid=getppid();
 printf("Proceso hijo: %d recibida señal de finalización de mi padre: %d \n",pid,ppid);
 exit(0);
}
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Comunicación entre procesos. Memoria Compartida

- Permite que un conjunto de direcciones en memoria principal sea compartido por múltiples procesos. Es la forma de IPC más rápida.
- Es necesario sincronizar el acceso para mantener la consistencia de los datos (normalmente con semáforos).

La memoria compartida se accede a ella como si fuera un vector de ”n” posiciones y haciendo uso del puntero que devuelve la llamada al sistema **shmat().**

Para trabajar con memoria se necesitan las librerías específicas **sys/ipc.h, sys/shm.h**

¿Qué llamadas me permiten trabajar con memoria compartida?

- ***key_t ftok(const char *path, int id);***
- ***int shmget(key_t key, int size, int flag);***
- ***int shmat(int shmid, void *addr, int flag);***
- ***int shmdt(const void *shmaddr);***
- ***int shmctl(int shmid, int cmd, struct shmid_ds *buf);***

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

¿Qué hace ftok ? ***key_t ftok(const char *path, int id);***

- **ftok** devuelve una **clave** en base al archivo/directorio proporcionado como primer parámetro y un identificador que es un valor numérico cualquiera. Este archivo o directorio debe ser accesible.
- El valor **clave** devuelto se utilizará posteriormente como **primer parámetro** en la llamada a **shmget()**.

Ej. **key_t clave;**

clave = ftok(”/bin/man”,136)

Tenemos la clave ¿Cuál es el siguiente paso? ***int shmget(key_t key, int size, int flag);***

- **shmget** se encarga de resevar el conjunto de direcciones a utilizar y obtener un identificador único, que devuelve.
- Parámetros:
  - **key** es la clave obtenida con ftok;
  - **size** es tamaño de la memoria a reservar. Dependerá de lo que se vaya a guardar, por ejemplo:
    - Para 10 valores enteros -> **10*sizeof(int)**
    - Para 20 valores float -> **20*sizeof(float)**
    - Para 10 posiciones y cada posición es una estructura llamada datos con varios campos ->**10 * sizeof(datos)**
  - En nuestro caso 0777|IPC_CREAT para crear y dar permisos de acceso
- Ej. **int shmid;**

  shmid=shmget(key,10*sizeof(10),0777|IPC_CREAT)

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Después del identificador usamos **shmat() ->*int shmat(int shmid, void *addr, int flag);***

- **shmat** se encarga de obtener el puntero a la primera dirección de memoria reservada. Este puntero se utiliza para acceder a memoria como si de un vector se tratara
- Parámetros:
  - **shmid** es el identificador de memoria obtenido mediante shmget();
  - **addr** valor recomendado **NULL** ((char*)0) para que tome las posiciones que estén disponibles automáticamente.
  - **flag** son los permisos de acceso, en nuestro caso **0**

- Ej.

```
int * puntero_memoria;
puntero_memoria = shmat (shmid, (char *)0,0)
// Ejemplo de inicialización
for (int i=0;i<10;i++)
   puntero_memoria=0;
```

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Todo proceso que herede o defina el puntero a una memoria compartida debe de liberarlo, antes de finalizar.

El proceso que creo la memoria compartida debe liberar el puntero y eliminar la memoria compartida

- Liberar el puntero **shmdt() ->*int shmdt(const void *shmaddr);***
  - Parámetros:
    - **puntero a la memoria compartida** devuelto por shmat
  - Devuelve 0 si se ha ejecutado correctamente y -1 en caso de error

  Ej.

  shmdt((int *) puntero_memoria)
- Eliminar memoria compartida **shmctl() ->*int shmctl(int shmid, int cmd, struct shmid_ds *buf);***
  - Parámetros
    - **shmid** es el identificador de memoria devuelto por shmget
    - **cmd** especifica la operación a realizar, IPC_RMID para eliminar la memoria compartida
    - **buf** es un puntero a una estructura del sistema, que nosotros no vamos a utilizar, pondremos **0**
  - Devuelve 0 si se ha ejecutado correctamente y -1 en caso de error

  Ej.

  shmctl(shmid, IPC_RMID, 0)

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo memoria compartida

```c
/* Comunicación mediante memoria compartida. Crear un proceso, que guardará 10 valores aleatorios
 * en memoria compartida, a los que el proceso padre también podrá acceder */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <sys/shm.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
  int pid,estado,i;
  int valor=5;               // Variable a modificar por el hijo
  int valor_leido=0;
  srand(time(NULL));

  key_t key;       // Clave a obtener con ftok
  long int shmid;    // Identificador de memoria a obtener con shmget
  int *puntero=NULL; // Puntero memoria compartida a obtener con shmat

  printf("Soy el proceso %d y voy a crear un proceso nuevo\n",getpid());

  // Crear memoria compartida para comunicar proceso padre e hijo, en este caso
  key=ftok("/bin/cat",121);                                    // Obtención de clave unica
  shmid=shmget(key,sizeof(int)*10,0777|IPC_CREAT); // Reserva de espacio 10 valores enteros y
                                                   // devuelve un identificador
  puntero=(int *)shmat(shmid,(char *)0,0);         // Obtiene el puntero a la 1ª posición
                                                  // cada posición puede ser accedida como puntero[i]
  pid=fork();  // Crear proceso nuevo
  switch (pid) {
   case -1: printf("Error\n");
            break;
case 0: //Código que ejecuta sólo el proceso hijo
      printf("Proceso hijo: %d, con padre : %d y genero información para el\n",getpid(),getppid());
      for (i=0;i<10;i++)
        {
         puntero[i]=rand()%100;
         printf("Generado %d: %d \n",i, puntero[i]);
        }
      shmdt((char *) puntero); // Antes de finaliza el hijo libera el puntero a memoria compartida
      exit(220);
      break;
default: // Código que ejecuta solo el proceso padre
        wait(&estado);    // Espera a que finalice el primer proceso hijo que lo haga
        if (estado==0)
           printf("Proceso hijo finalizado normalmente\n");  // Comprueba el motivo de finalización
         else {
           if (WIFEXITED(estado)) printf("Hijo finaliza con$ EXIT(%d)\n",WEXITSTATUS(estado));
           if (WIFSIGNALED(estado)) printf("Hijo finaliza por señal %d recibida\n",WTERMSIG(estado));
         }
        printf("Padre: %d, leo información enviada por mi hijo \n",getpid());
        for (i=0;i<10;i++)
           printf("Memoria[%d]: %d\n",i, puntero[i]);
        printf("Padre: %d finalizo y devuelvo el control al sistema\n",getpid());
        break;
    }
 shmdt((char*)puntero); //Antes de terminar el padre libera el puntero a memoria compartida
 shmctl(shmid,IPC_RMID,0); //Elimina la zona de memoria compartida
 exit(0); //Finaliza el programa
}
```

Comunicación entre procesos. FIFO

Una FIFO es similar a una tubería con pipe, con la diferencia que el canal de comunicación fifo tiene nombre y se utiliza para comunicar procesos que no están directamente emparentados. El caso típico es el de productor-consumidor o escritor-lector. Librerías **sys/types.h, sys/stat.h**

- Creación del canal de comunicación fifo: **int mkfifo(const char *pathname, mode_t mode);**
  - Parámetros
    - **pathname** -> nombre del canal de comunicación a crear
    - **mode** -> permisos del canal de comunicación. Los permisos se pueden cambiar dentro del programa con la llamada al sistema chmod.
  - Devuelve: 0, si todo ha ido bien; -1, en caso de error.
- Abrir una tubería FIFO: **int open (pathname, modo)**

  Devuelve el descriptor del fichero que es un valor entero; Ej. **int descriptor**
  - Modo lectura: **descriptor=open("tuberiafifo",O_RDONLY);** // Apertura por parte del lector/consumidor
  - Modo escritura: **descriptor=open("tuberiafifo",O_WRONLY);** // Apertura por parte del escritor/producto

- Escribir en una tubería o PIPE: **write (descriptor, datos, tamaño)**
- Leer de una tubería o PIPE: **read (descriptor, datos, tamaño)**
- Cierre de una FIFO: **close(descriptor)**
- Borrado de una FIFO: **unlink(pathname)**

Gestión de procesos. Servicios Linux/UNIX para la gestión de procesos

Ejemplo escritor/lector con tuberías con nombre

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

int main()
{
 int des,i;
 char cadena[100];
 sprintf(cadena,"Saludo desde el proceso escritor con pid= %d\n",getpid());
 des=-1;
 do
 {
  des=open("tuberia",O_WRONLY);
  printf("Se abre la tubería\n");
  if (des==-1)
  {
  printf("Error: intentando abrir la tubería\n");
  }
 }while (des==-1);

 for (i=0; i<3;i++)
 {
 write(des,cadena,100);
 printf("He escrito en la tuberia\n");
 sleep(1);
 }
 close(des);
 printf("Acaba el proceso escritor\n ");
}
```

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

int main()
{
 int des;
 char cadena[100];

 mkfifo("tuberia",0);
 chmod("tuberia",460);
do
 {
  des=open("tuberia",O_RDONLY);
  printf("Espero a poder abrir la tubería \n");
  if (des==-1) sleep(1);
 }while (des==-1);

 while (read(des,cadena,100)>0)
  printf("Leido: %s \n",cadena);

 close(des);
 printf("Finaliza lector\n");
 unlink("tuberia");
}
```

```
gcc -o lector lector.c
gcc -o escritor escritor.c
./escritor&
./lector&
```