Programación de Procesos - C

En este apartado vamos a ver cómo es posible escribir programas en lenguaje C++ para trabajar con Procesos. Dada la similitud con Java, no es necesario un conocimiento exhaustivo del lenguaje C.

Linux y Geany

Para ejecutar estos programas necesitamos tener instalado un **Linux (Ubuntu)** en una máquina virtual. Una vez instalado, empleamos un IDE básico llamado **Geany** para editar nuestros programas. Este IDE puede ser instalado desde la tienda que tiene el propio Ubuntu, o directamente a mano desde la consola. Dadas las limitaciones del Geany, será necesario que **guardes a mano** el fichero de código cada vez que lo compiles o ejecutes. A demás, se requerirá que **añadas a mano** la extensión de los ficheros al crearlos (.cpp) o no se ejecutará.

No es posible ejecutar este código en una máquina Windows debido a las peculiaridades del Sistema Operativo. Si lo haces, aunque utilices otro IDE con C++, no serás capaz de ver los problemas de **concurrencia** o la comunicación con **Pipes**.

Pasos para instalar

1) Instala Virtual Box. No necesitas más de 10GB de disco. Indica que el disco duro virtual va a ser para una distribución de Linux (Ubuntu). Recuerda hacerla bootable. Finalmente, asegúrate de que la máquina virtual tiene acceso a internet.

2) Arranca la máquina virtual e instala la versión de Ubuntu. Recuerda poner que el usuario es admin/admin y que no pida claves al arrancar.

3) Una vez instalado, en el escritorio, accede a la tienda e instala Geany (el primero).

4) Accede a la consola. Nos hace falta instalar el compilador de C++ para Linux. Entra en modo administrador y teclea: **apt-get install g++**

5) Ya puedes usar el IDE Geany. Para ejecutar un programa, tienes que pulsar en secuencia los botones Compilar – Construir – Ejecutar.

Llamadas al sistema para control de Procesos (Linux)

C++ nos ofrece una serie de llamadas al sistema para trabajar con Procesos. Son **funciones** que se encuentran escritas en librerías, al igual que en Java. Desde nuestros programas podremos hacer uso de ellas de forma similar.

Identificación de procesos	
int getpid ();	Devuelve el identificador del proceso (PID)
int getppid ();	Devuelve el identificador del proceso (PID) del Proceso padre
int getuid ();	Devuelve el identificador de usuario (UID) del dueño del Proceso

Estas llamadas permiten obtener información del Proceso que se genera al ejecutar nuestros programas en el **Geany**.

Creación de procesos	
int fork ();	Crea un proceso nuevo, una copia exacta del proceso que llama a fork. El proceso que invoca se le llama **padre** y el proceso nuevo **hijo**. El hijo hereda todo del padre, excepto el PID. Fork devuelve un 0 al hijo, el PID del hijo al padre, y un -1 si hay un error.
int execl ();	Cambia el programa que se está ejecutando, lo elimina de memoria, y lo cambia por otro. No devuelve nada si todo va bien, o un -1 si hay un error. Execl es una familia extensa de funciones: execle, execv…

Para generar Procesos nuevos.

Terminación de procesos	
void exit (int i)	Terminación controlada del proceso.
void wait (int i)	Bloquea al Proceso hasta que alguno de sus hijos finalice. Devuelve el **PID** del proceso hijo que ha terminado. Si no tiene hijos devuelve -1, sin bloquear al proceso.

Para finalizar un Proceso.

Control de señales	
int kill(int pid, int señal);	Envía la **señal** indicada al proceso PID. Devuelve: 0 si todo ok, 1 si error.
unsigned int alarm (unsigned int s);	Envía una **señal** SIGALRM al propio proceso pasados los segundos indicados. Devuelve los segundos que quedan para que expire la anterior señal SIGALRM.

void pause ();	Bloquea al proceso hasta que este recibe una **señal** (cualquiera).
void signal (int s, void funcion());	Cuando se reciba cierta **señal** en concreto, se ejecutará la función indicada (Captura la señal). Si en vez de una función decimos: - signal(señal, **SIG_IGN**): Se ignora la señal indicada. - signal(señal, **SIG_DFL**): Se ejecuta el tratamiento por defecto de la señal indicada.

Existe un comportamiento predefinido para ciertas **señales**. Puedes consultarlas a continuación. SIGKILL y SIGSTOP **nunca pueden** ser capturadas, bloqueadas o ignoradas.

Señal	Valor	Acción	Comentario
SIGINT	2	A	Interrupción procedente del teclado(Crtl-C)(salida)
SIGKILL	9	AEF	Señal de matar
SIGALRM	14	A	Señal de alarma de alarm(2)
SIGTERM	15	A	Señal de terminación
SIGUSR1	10	A	Señal definida por usuario 1
SIGUSR2	12	A	Señal definida por usuario 2
SIGCONT	18		Continuar si estaba parado
SIGSTOP	19	DEF	Parar proceso

A - La acción por defecto es terminar el proceso
D - La acción por defecto es parar la ejecución del proceso
E - La señal no puede ser capturada por el programa (manipulada)
F - La señal no puede ser ignorada

Operaciones sobre Procesos

En los siguientes apartados vamos a ver cómo se realizan programas que interactúan con los Procesos. Se van a emplear las **llamadas al sistema** vistas en el apartado anterior. Escribe los programas en el Geany y ejecútalos para comprobar el resultado.

Ejemplo 1 - Identificación de Procesos

Es posible obtener información sobre los procesos mediante código. Por ejemplo, podemos obtener los PID del propio programa en ejecución. Ejecuta este programa:

prueba1.cpp

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4
5   int main () {
6       int pid = getpid();
7       int pid_padre = getppid();
8       int id_usu = getuid();
9
10      printf ("Pid Proceso: %d\n", pid);
11      printf ("Pid Padre: %d\n", pid_padre);
12      printf ("Id usuario: %d\n", id_usu);
13  }
14
```

El programa debería mostrarte el PID del Proceso, de su Proceso Padre (del que 'cuelga' este Proceso) y el Usuario que ha lanzado el Proceso (admin).

Ejemplo 2 - Creación de Procesos

Fork permite crear un **proceso nuevo**. En cuanto se ejecuta la llamada al sistema, se generará otro proceso que será una copia exacta del proceso que ha llamado a fork. El proceso que invoca la llamada es el **Proceso padre**, y el proceso nuevo creado es el **Proceso hijo**.

A tener en cuenta:

- El Hijo comienza su ejecución desde la siguiente instrucción a partir de fork.
- El hijo lo hereda todo del padre, excepto el PID.

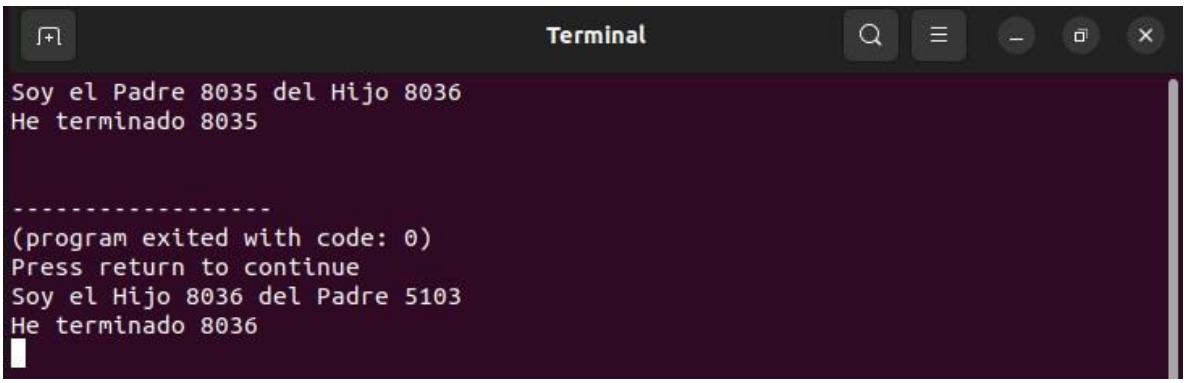
La llamada a fork devuelve al Hijo un 0, al padre el PID del Hijo creado, y un -1 si ha ocurrido alguna clase de error.

Ejecuta este programa:

prueba2.cpp

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4
5   int main () {
6       int pid;
7       pid = fork();
8
9       // Ambos procesos Padre e Hijo tienen el mismo codigo
10      if (pid == 0) {
11          // Pero esto solo lo ejecuta el Hijo
12          printf ("Soy el Hijo %d del Padre %d\n", getpid(), getppid());
13      } else {
14          // Y esto solo lo ejecuta el Padre
15          printf ("Soy el Padre %d del Hijo %d\n", getpid(), pid);
16      }
17      printf ("He terminado %d\n", getpid());
18      exit(0);
19  }
20
```

Lo más probable es que obtengas un resultado parecido a este. Si te fijas, el Padre ejecuta su código y termina; y después el Hijo ejecuta su código y termina. Uno detrás del otro. Pero resulta que en ocasiones es el Padre el que termina antes que el Hijo. Puede saberse porque se muestra el mensaje “Soy el Hijo 8036 del Padre 1”.



¿Por qué sucede esto? Porque los Procesos **no están sincronizados**. Recuerda, nosotros no podemos decir al Sistema Operativo qué Proceso debe ejecutar, sólo podemos influir en sus decisiones. En este caso, el Padre termina antes – algo que no debería ocurrir – y el Hijo se queda huérfano.

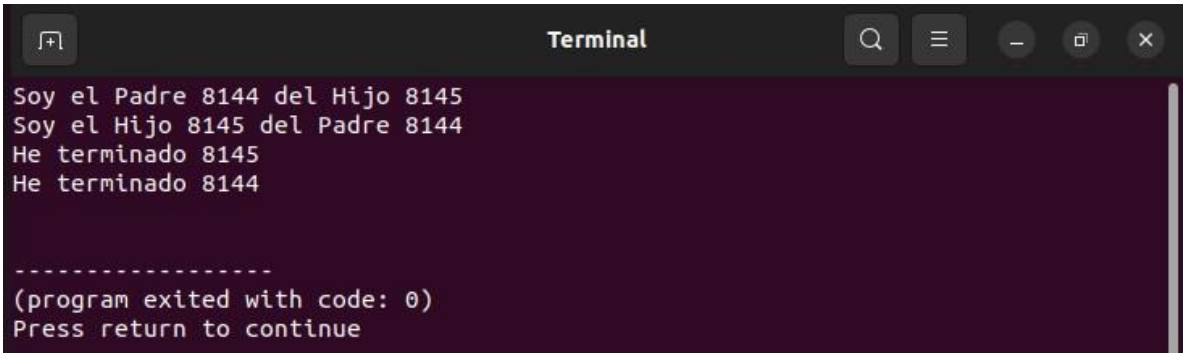
Ejemplo 3 - Creación de Procesos Síncronos

Vamos a obligar al Padre a terminar antes que el Hijo. Lo que haremos será utilizar la llamada **wait** que pone en espera a un Proceso, en este caso al Padre.

prueba3.cpp

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <errno.h>
5   #include <sys/types.h>
6   #include <sys/wait.h>
7
8   int main () {
9       int pid;
10      int status = 0;
11      pid = fork();
12
13      // Sincronicemos: el padre tiene que esperar a que el hijo acabe
14      if (pid == 0) {
15          // Pero esto solo lo ejecuta el Hijo
16          printf ("Soy el Hijo %d del Padre %d\n", getpid(), getppid());
17      } else {
18          // Y esto solo lo ejecuta el Padre
19          printf ("Soy el Padre %d del Hijo %d\n", getpid(), pid);
20          wait (&status); // Esperamos...
21      }
22      printf ("He terminado %d\n", getpid());
23      exit(0);
24  }
25
```

Así, la ejecución del programa siempre es similar a esto:



Ejemplo 4 – Cambiar Procesos

Un fork lo que crea en realidad son clones. Existe una forma para sustituir el código de un proceso por otro. De esta forma podemos hacer que un Proceso genere y ejecute otro Proceso totalmente distinto. La familia de llamadas al sistema **execl** hace un fork, **pero sustituye** su código por el de otro proceso. Si ocurre algún error, se devuelve el control al programa que la invoca devolviendo -1.

En el ejemplo, el Hijo muestra el contenido de la carpeta /bin.

prueba4.cpp

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

int main () {
    int pid;
    pid = fork();

    if (pid == 0) {
        // Sustituimos el codigo del Hijo con algo nuevo
        execl ("/bin/ls", "/bin/ls", "-1", NULL);
        // Esto solo se ve si FALLA el execl
        printf ("No se ha podido ejecutar execl \n");
        exit (-1);
    } else {
        // Padre
        printf ("Soy el Padre %d del Hijo %d\n", getpid(), pid);
        exit(0);
    }
```

Ejemplo 5 – Control de Señales

¿Cómo podemos **obligar** al Hijo a acabar antes que el Padre? Los Procesos pueden comunicarse entre ellos. Una forma de hacerlo es mediante las Señales. La instrucción **kill** no finaliza un Proceso, sino que envía una señal. Puede ocurrir, no obstante, que el comportamiento por defecto sea 'matar' al Proceso.

En el siguiente ejemplo, el Padre se pone a esperar **indefinidamente** si va a terminar antes que el Hijo. Entonces, el Hijo envía una Señal al Padre y termina, despertando al Padre, que también finaliza. Si hubiese acabado antes el Hijo, habría matado directamente al Padre.

prueba5.cpp

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

int main () {
    int pid;
    pid = fork();
    if (pid == 0) {
        // Hijo
        printf ("Soy el Hijo %d del Padre %d\n", getpid(), getppid());
        kill (getppid(), SIGKILL); // Enviamos la señal al Padre
    } else {
        // Padre
        printf ("Soy el Padre %d del Hijo %d\n", getpid(), pid);
        pause(); // Se pone a la espera
    }
    printf ("He terminado %d\n", getpid());
    exit(0);
}
```

Ejemplo 6 – Registro de Señales

Pero ‘matar’ un Proceso es tan sólo una de las posibilidades. Es posible cambiar el **comportamiento por defecto** de las Señales capturándolas y dándolas otro comportamiento. En el siguiente ejemplo, registramos SIGALRM y una función. La instrucción **alarm** lanza dicha señal cuando el tiempo se agote, por tanto, el Proceso no morirá sino que se ejecutará la función indicada.

prueba6.cpp

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <errno.h>
5   #include <sys/types.h>
6   #include <sys/wait.h>
7   #include <csignal>
8
9   void funcion (int signum){
10      printf ("He recibido la señal %d\n", signum);
11  }
12
13  int main () {
14      // Registra la señal SIGALRM
15      signal (SIGALRM, funcion);
16      alarm(3);
17      pause();
18      exit(0);
19  }
20
```

Ejercicio 2

Ejecuta el siguiente código. ¿Por qué nunca se muestra el texto de la línea 17?

prueba7.cpp

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <errno.h>
5   #include <sys/types.h>
6   #include <sys/wait.h>
7   #include <csignal>
8
9   int main () {
10      int seg = alarm (10);
11      if (seg > 0){
12          printf ("Ya tenias una alarma");
13      } else {
14          printf ("No tenias una alarma");
15      }
16      sleep (30);
17      printf ("Esto no sadrá nunca por pantalla...");
18      exit (-1);
19  }
20
```