



Apunte de Procesos

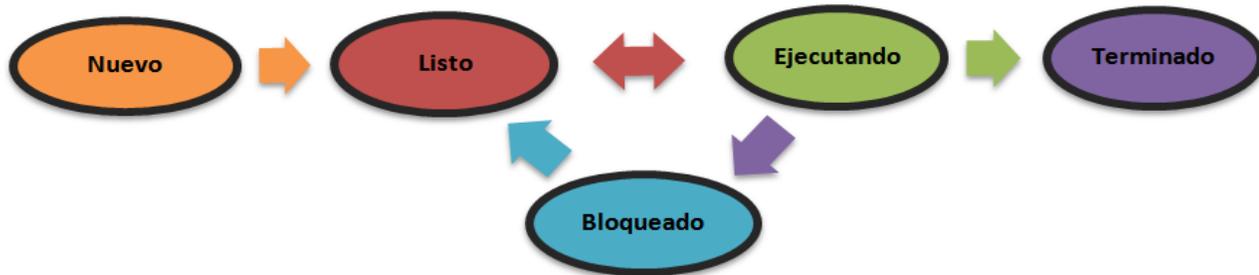
Programación Concurrente

Ing. Dario Hirschfeldt

PROCESOS

Ciclo de vida

Un proceso puede definirse como **un programa en ejecución mas su contexto**. Durante su paso por el sistema operativo (S.O.) un proceso pasará por diferentes estados. Por ejemplo en un modelo de cinco estados:



Nuevo	El proceso fue creado pero aún se encuentra a la espera de que se le asignen los recursos necesarios para su ejecución, aún no compite por el procesador.
Listo	El proceso se encuentra en la cola de listos, a la espera de que el dispatcher lo habilite para utilizar el procesador.
Ejecutando	El proceso está utilizando el procesador.
Bloqueado	El proceso se encuentra a la espera de algún evento, por ejemplo la finalización de una operación de Entrada/Salida, para volver a la cola de listos.
Terminado	El proceso fue excluido del conjunto de procesos ejecutables y se encuentra a la espera de que el S.O. libere sus recursos.

Si bien no se muestra en el gráfico, puede darse la transición a Terminado desde cualquier otro estado, por ejemplo en el caso de que un proceso finalice a pedido de su proceso padre o si éste finaliza.

Cambio de contexto

Con la necesidad de ejecutar procesos de manera concurrente, surge la necesidad de realizar un cambio de contexto, esto es básicamente salvar el estado de ejecución del proceso en ejecución, quien será desalojado, y cargar el estado de ejecución previo del proceso que ejecutará a continuación.

Cambio de proceso

Es el caso en el que un proceso sea desalojado para dar paso a la ejecución de otro proceso, habrá que realizar un cambio de contexto, esto involucra almacenar toda la información contenida en el **Process Control Block (PCB)** del proceso en ejecución y cargar toda la información del PCB del proceso a ejecutar.

En el caso en el que un proceso sea interrumpido, pero no desalojado, para que el S.O. ejecute una rutina de atención de interrupción, es decir luego de la ejecución de la rutina continuará con su ejecución, en este caso no hay un cambio de proceso, pero si hay un cambio de contexto (liviano), que es menos costoso en overhead (trabajo del procesador que no es útil en cuanto a la ejecución del proceso usuario), dado que solo hay que salvar el estado de los registros de la CPU y el contador de programa, valores que seguramente modificará la ejecución de la rutina, pero no es necesario salvar todo el PCB, ya que el mismo proceso volverá a ejecución luego de la ejecución de la rutina.

Process Control Block (PCB)

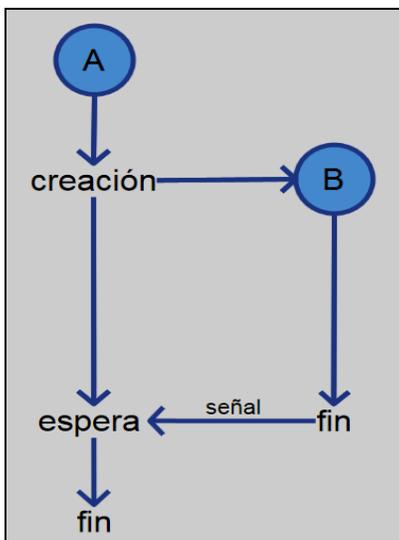
El PCB es vital para poder ejecutar procesos de manera concurrente, almacena el contexto de ejecución de un proceso, por lo tanto el S.O. deberá crear, mantener y destruir uno por cada proceso que se encuentre en el sistema. Para esto el S.O. puede mantener una array o lista enlazada de PCBs, donde el PCB puede ser implementado como una estructura.

Si bien la información contenida en el PCB varía de S.O. en S.O., se puede identificar en líneas generales lo siguiente:

Identificador del proceso (PID).
PIDs de procesos emparentados, padre, hijos.
Estado (Nuevo, Listo, etc).
Contador de programa.
Registros del procesador.
Información para la planificación (prioridad por ejemplo).
Información para la administración de memoria (Tabla de páginas, segmentos, etc).
Información de Entrada/Salida (archivos en uso por ejemplo).
Información para las estadísticas (Utilización de procesador por ejemplo).

Creación de procesos

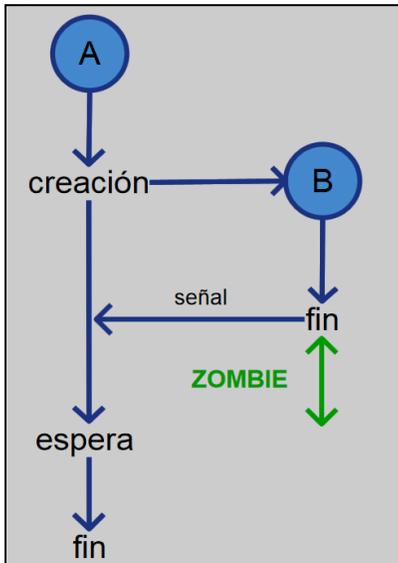
Podemos crear uno o varios procesos hijos desde un proceso padre para ejecutar ciertas tareas de manera concurrente. El procedimiento en general sería algo así como lo que se muestra en el siguiente gráfico:



En algún momento de su ejecución un proceso A creará a un proceso B, si B se crea exitosamente ambos ejecutarán de manera concurrente.

El proceso A deberá esperar la finalización del proceso B, para poder desasignar sus recursos, y luego finalizará.

El proceso B, de manera automática al finalizar su ejecución, emitirá una señal que desbloqueará la espera del proceso A.



Como es de esperarse la emisión de la señal y la espera seguramente no coincidirán en el tiempo, entonces cabe destacar que si B finaliza antes de que A llegue a la espera, todo ese lapso de tiempo el proceso B estará en un estado de Zombie, ya que finalizó (murió) pero todavía tiene los recursos asignados.

Creación de procesos en el Lenguaje C:

La función que nos permite crear procesos por programa es **fork()**, esta función duplica al proceso que la invoca, a quien llamaremos padre, generando un nuevo proceso (hijo). Dicha función tiene la particularidad de retornar 2 valores distintos en caso de éxito, uno para el proceso padre y otro para el hijo. Cabe destacar que ambos procesos no comparten memoria, es decir; las variables de un proceso no tienen que ver con las variables del otro, aunque en el código veamos que tienen el mismo nombre, ya que el código es el mismo, el área de datos es distinta.

```
pid_t fork(void);
#include <unistd.h>
```

Valores de retorno de fork():

ERROR: En el caso de que no se pueda crear el nuevo proceso, retorna un **entero negativo** al proceso llamador.

ÉXITO: En este caso retorna 2 valores enteros, uno al proceso llamador y otro al nuevo procesos:

Proceso llamador (Padre): Recibe un **entero positivo**, siendo éste el **PID** del nuevo proceso.

Nuevo proceso (Hijo): Recibe el valor **cero (0)**.

Funciones de la familia wait:

Las funciones de la familia wait permiten esperar por un cambio de estado en alguno de los hijos del proceso llamador, los cambios de estado pueden ser pasar a finalizado, ser detenido por una señal o ser reanudado por una señal. En el caso de pasar a finalizado, cualquier de las funciones de la familia wait permite al sistema liberar los recursos del proceso hijo finalizado.

Cuando creamos un proceso hijo mediante un proceso padre con **fork()**, es esperable que el padre quede a la espera de la finalización del hijo para que se desasignen sus recursos, salvo que querramos a drede crear un proceso huérfano. La espera por parte del padre se realiza efectuando una llamada a alguna de las funciones de la familia wait, las cuales se detallan a continuación:

```
pid_t wait(int *status):
#include <sys/wait.h>
```

Bloquea la ejecución del proceso llamador hasta que reciba una **señal SIGCHLD** proveniente de **cualquier proceso hijo**.

Ing. Dario Hirschfeldt

También permite obtener información sobre el estado de finalización del proceso hijo si se pasa como parametro un puntero a un entero. Dicho estado puede consultarse a través de unas macros que provee la biblioteca y trabajan sobre dicho entero. Algunos ejemplos de dichas macros son:

WIFEXITED	Retorna True si el hijo finalizó normalmente.
WIFSIGNALED	Retorna True si el hijo finalizó a causa de una señal.
WTERMSIG	Retorna el número de señal que causó la finalización del proceso hijo.
WCOREDUMP	Retorna True si el hijo produjo un core dump.

Valores de retorno:

PID del proceso hijo que ha terminado.

-1 en caso de error.

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

Es similar a la anterior solo que espera la finalización de **un proceso hijo en particular** especificado por el valor de pid, dicho primer parámetro puede tomar los siguientes valores:

> 0	Espera por el proceso hijo cuyo PID es igual al valor especificado
-1	Espera por cualquier proceso hijo.
0	Espera por cualquier proceso hijo cuyo group ID (GID) al del proceso padre.
< -1	Espera por cualquier proceso hijo cuyo GID es igual al valor absoluto del valor especificado.

También se pueden especificar opciones (modificadores de comportamiento), por ejemplo si se especifica la opción WNOHANG, la función deja de ser bloqueante.

Valores de retorno:

PID del proceso hijo que ha terminado. En el caso de que se haya especificado WNOHANG y no haya ningun hijo finalizado, entonces retorna cero (0) en vez de retornar el PID.

-1 en caso de error.

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options):

(disponible desde Linux 2.6.9)

Provee un control mas preciso sobre que hijos han finalizado, como primer parametro se especifica el tipo de ID que estamos buscando, que puede ser por ejemplo;

P_PID, si se busca el hijo que haya finalizado con el PID especificado en el segundo parametro.

P_PGID, si se busca cualquier hijo que haya finalizado con el GID especificado en el segundo parametro.

P_ALL, si se busca cualquier hijo, en cuyo caso el segundo parametro es ignorado.

Como tercer parametro se provee un puntero a una estructura, donde en caso de haber ejecutado sin errores, waitid llenara con cierta información sobre el hijo finalizado, por ejemplo el PID, si finalizó por una señal, core dump, etc. En cierta forma es similar a waitpid.

> 0	Espera por el proceso hijo cuyo PID es igual al valor especificado
-1	Espera por cualquier proceso hijo.
0	Espera por cualquier proceso hijo cuyo group ID (GID) al del proceso padre.
< -1	Espera por cualquier proceso hijo cuyo GID es igual al valor absoluto del valor especificado.

Valores de retorno:

0 en caso de éxito o en el caso de que se haya especificado WNOHANG y no haya ningun hijo finalizado.

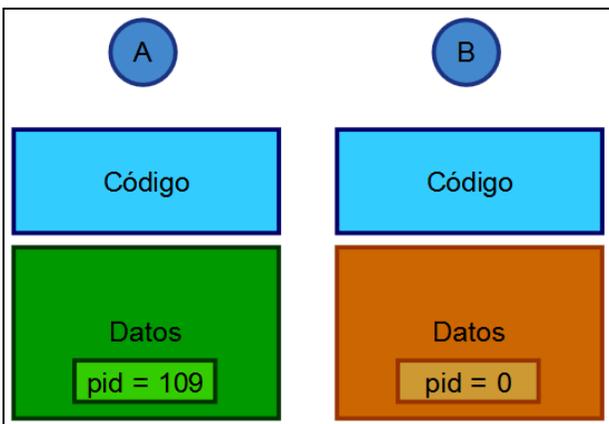
-1 en caso de error.

Ejemplo de código 1:

```
Settings x Shell x main.c x +
main.c > ...
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int main(int argc, char *argv[])
7 {
8     pid_t pid;
9
10    pid = fork();
11
12    if( pid < 0 )
13    {
14        printf("Error al crear el nuevo proceso\n");
15        return EXIT_FAILURE;
16    }
17
18    if( pid )
19    {
20        printf("Soy el proceso padre. PID: %d\n",getpid());
21        wait(NULL);
22    }
23    else
24    {
25        printf("Soy el proceso hijo. PID: %d\n",getpid());
26        printf("Mi padre es el proceso con PID: %d\n",getppid());
27        return EXIT_SUCCESS;
28    }
29
30    return EXIT_SUCCESS;
31 }
```

```
>_ Console x +
> make -s
> ./main
Soy el proceso padre. PID: 108
Soy el proceso hijo. PID: 109
Mi padre es el proceso con PID: 108
> □
```

En el ejemplo vemos que inmediatamente despues de fork(), debemos consultar el valor retornado por fork (la variable pid), si el nuevo proceso pudo ser creado, a traves del valor de (pid) podremos saber si somos el proceso padre o el proceso hijo para saber que código se debe ejecutar, recordemos que el código es el mismo ya sea compartido o una copia, es el mismo, entonces la única manera de saber que porción de código debe ejecutar cada proceso es el valor que fork retornó a cada uno de ellos, visualicemos el ejemplo anterior:



En este caso A sería el proceso padre y B el proceso hijo.

Si bien en el código vemos que la variable pid es una sola, en el área de datos de cada proceso existirá una variable pid, una independiente de la otra.

Veamos que parte del código ejecuta cada proceso:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int main(int argc, char *argv[])
7 {
8     pid_t pid;
9
10    pid = fork();
11
12    if( pid < 0 )
13    {
14        printf("Error al crear el nuevo proceso\n");
15        return EXIT_FAILURE;
16    }
17
18    if( pid )
19    {
20        printf("Soy el proceso padre. PID: %d\n",getpid());
21        wait(NULL);
22    }
23    else
24    {
25        printf("Soy el proceso hijo. PID: %d\n",getpid());
26        printf("Mi padre es el proceso con PID: %d\n",getppid());
27        return EXIT_SUCCESS;
28    }
29
30    return EXIT_SUCCESS;
31 }
```

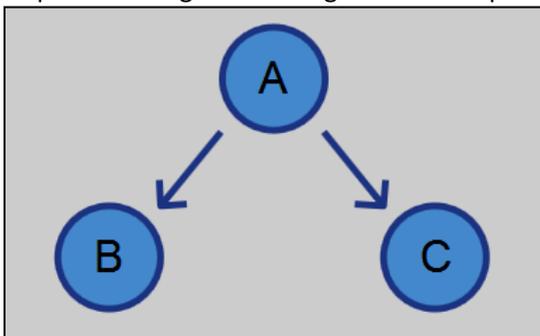
PROCESO A

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int main(int argc, char *argv[])
7 {
8     pid_t pid;
9
10    pid = fork();
11
12    if( pid < 0 )
13    {
14        printf("Error al crear el nuevo proceso\n");
15        return EXIT_FAILURE;
16    }
17
18    if( pid )
19    {
20        printf("Soy el proceso padre. PID: %d\n",getpid());
21        wait(NULL);
22    }
23    else
24    {
25        printf("Soy el proceso hijo. PID: %d\n",getpid());
26        printf("Mi padre es el proceso con PID: %d\n",getppid());
27        return EXIT_SUCCESS;
28    }
29
30    return EXIT_SUCCESS;
31 }
```

PROCESO B

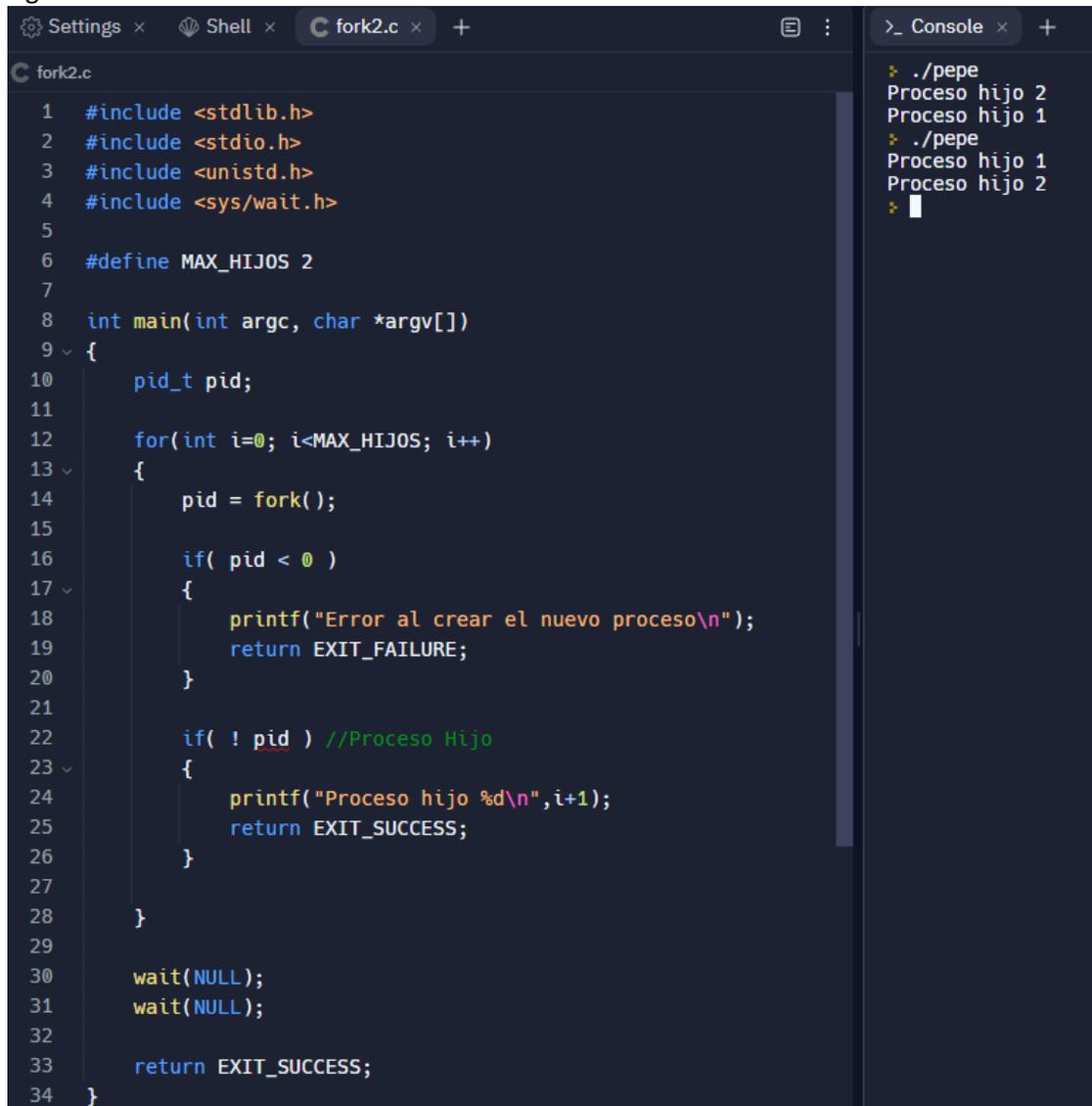
Notese que el proceso B (Hijo) no ejecuta el fork(), porque al momento de ser creado hereda el contexto de ejecución del proceso padre, y su contador de programa ya está apuntado a la siguiente instrucción después del fork().

Si quisieramos generar el siguiente árbol por ejemplo, un padre y dos hijos:



El código sería el siguiente:

Ejemplo de código 2:



```
Settings x Shell x fork2.c x +
fork2.c
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 #define MAX_HIJOS 2
7
8 int main(int argc, char *argv[])
9 {
10     pid_t pid;
11
12     for(int i=0; i<MAX_HIJOS; i++)
13     {
14         pid = fork();
15
16         if( pid < 0 )
17         {
18             printf("Error al crear el nuevo proceso\n");
19             return EXIT_FAILURE;
20         }
21
22         if( ! pid ) //Proceso Hijo
23         {
24             printf("Proceso hijo %d\n",i+1);
25             return EXIT_SUCCESS;
26         }
27     }
28
29     wait(NULL);
30     wait(NULL);
31
32     return EXIT_SUCCESS;
33 }
34 }
```

```
>_ Console x +
./pepe
Proceso hijo 2
Proceso hijo 1
./pepe
Proceso hijo 1
Proceso hijo 2
>
```

Notesé que la espera de los hijos por parte del padre se realiza con dos ejecuciones de wait, también se podría hacer un ciclo desde cero hasta MAX_HIJOS y que en cada pasada del mismo se ejecute un wait, lo mas políticamente correcto es utilizar un manejador de señales:

```
int sigaction(int signum, const struct sigaction *restrict act, struct sigaction
*restrict oldact);
#include <signal.h>
```

A traves de **sigaction** podemos cambiar el comportamiento predefinido en el proceso llamador de cualquier señal excepto las señales SIGKILL y SIGSTOP.

Modificando el ejemplo anterior:

```
fork2.c x +
fork2.c
9  int finalizados = 0;
10
11 void manejador( int sig ){
12     switch( sig )
13     {
14         case SIGCHLD:
15             wait( NULL );
16             finalizados++;
17             break;
18     }
19 };
20
21 int main(int argc, char *argv[])
22 {
23     struct sigaction psa;
24     psa.sa_handler = manejador;
25     sigaction(SIGCHLD, &psa, NULL); //Asociación con el manejador propio
26
27     pid_t pid;
28
29     for(int i=0; i<MAX_HIJOS; i++)
30     {
31         pid = fork();
32
33         if( pid < 0 )
34         {
35             printf("Error al crear el nuevo proceso\n");
36             return EXIT_FAILURE;
37         }
38
39         if( ! pid ) //Proceso Hijo
40         {
41             printf("Proceso hijo %d\n",i+1);
42             return EXIT_SUCCESS;
43         }
44     }
45
46     while( finalizados < MAX_HIJOS )
47         usleep(100000); //Duerme 100 miliseg para mitigar la espera activa
48
49     return EXIT_SUCCESS;
50 }
51
```

```
>_ Console x Shell x +
> gcc -o pepe fork2.c
> ./pepe
Proceso hijo 1
Proceso hijo 2
>
```

Funciones de la familia exec():

Las funciones de la familia exec() reemplazan la imagen del proceso llamador con la imagen de otro proceso.

```
int execve(const char *file, char *const args[], char *const envp []);
#include <unistd.h>
```

Ejecuta el binario ejecutable o script especificado en **file**. Los parametros al binario o script se pasan a traves de **args**. El parámetro **envp** se utiliza para pasar variables de entorno.

```
int execl(const char *pathname, char *const argv[] );
int execl(const char *path, const char *arg0, const char *arg1, ...);
```

Ing. Dario Hirschfeldt

```
int execlp(const char *file, const char *arg0, const char *arg1, ...);
int execl(const char *path, const char *arg0, ..., char * const envp[]);
int execvp(const char *file, char *const argv[]);
```

En el siguiente ejemplo utilizaremos la función `execv`, para facilitar el ejemplo:

Ejemplo de código 3:

```

B.bin x A.c x B.c x +
A.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <errno.h>
5
6 int main( int argc, char *argv[] )
7 {
8     printf("Proceso A. PID: %d\n", getpid() );
9
10    char *arg[] = { "B.bin", "Hola mundo", NULL };
11
12    execv("B.bin", arg );
13
14    //Si se ejecuta de aqui en adelante es porque
15    //hubo error en el execv
16
17    perror("execv");
18
19    exit(EXIT_FAILURE);
20 }

_ Console x Shell x +
> gcc -o A.bin A.c
> gcc -o B.bin B.c
> ./A.bin
Proceso A. PID: 633
Programa B ejecutado con execv. PID: 633
Nombre: B.bin
Arg 1: Hola mundo
>

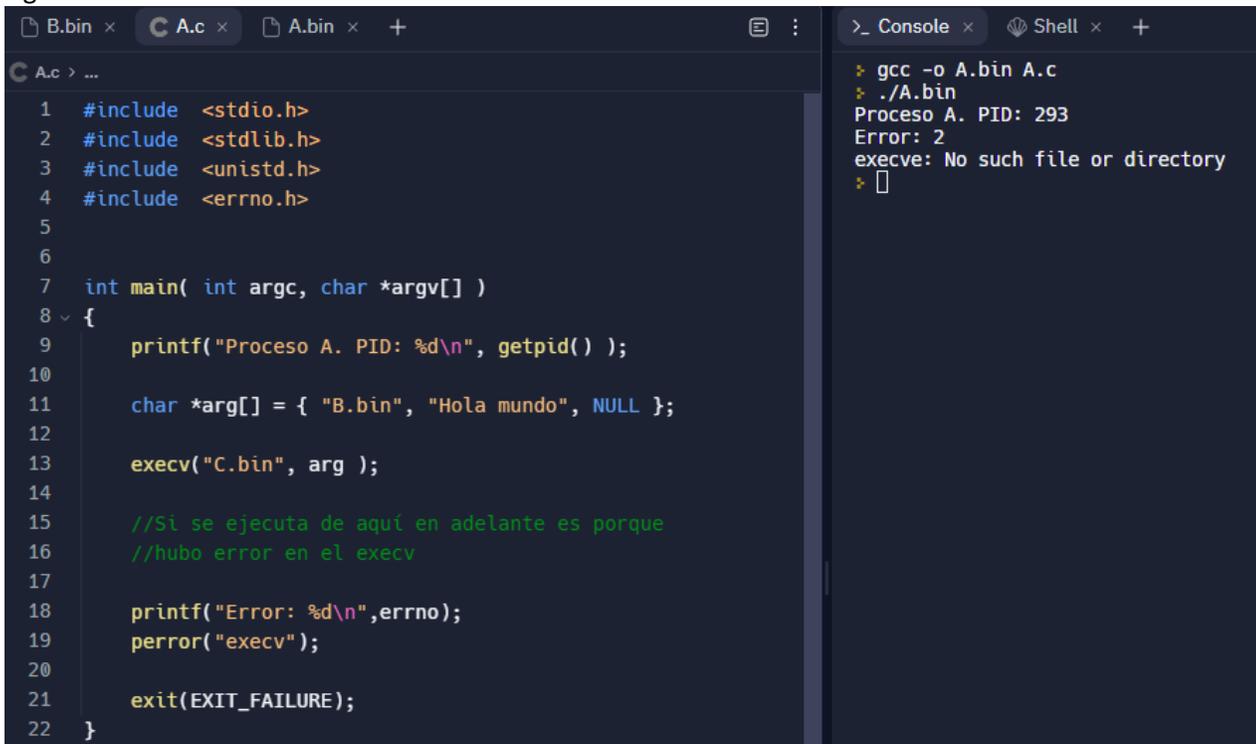
B.bin x A.c x B.c x +
B.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int main( int argc, char *argv[] )
6 {
7     printf("Programa B ejecutado con execv. PID: %d\n", getpid() );
8
9     printf("Nombre: %s\n", argv[0]);
10    printf("Arg 1: %s\n", argv[1]);
11
12    return EXIT_SUCCESS;
13 }
```

En caso de ejecutarse exitosamente, el `execve` no retorna, a partir de ese momento ya se reemplazó la imagen del proceso llamador de `execv` con la imagen del nuevo proceso.

En caso de error retorna -1 y actualiza la variable `errno`.

En el siguiente ejemplo se muestra un ejemplo de ejecución errónea al pasar un path inexistente (C.bin):

Ejemplo de código 4:



```
A.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <errno.h>
5
6
7 int main( int argc, char *argv[] )
8 {
9     printf("Proceso A. PID: %d\n", getpid() );
10
11     char *arg[] = { "B.bin", "Hola mundo", NULL };
12
13     execv("C.bin", arg );
14
15     //Si se ejecuta de aquí en adelante es porque
16     //hubo error en el execv
17
18     printf("Error: %d\n",errno);
19     perror("execv");
20
21     exit(EXIT_FAILURE);
22 }
```

```
>_ Console x Shell x +
> gcc -o A.bin A.c
> ./A.bin
Proceso A. PID: 293
Error: 2
execve: No such file or directory
> []
```

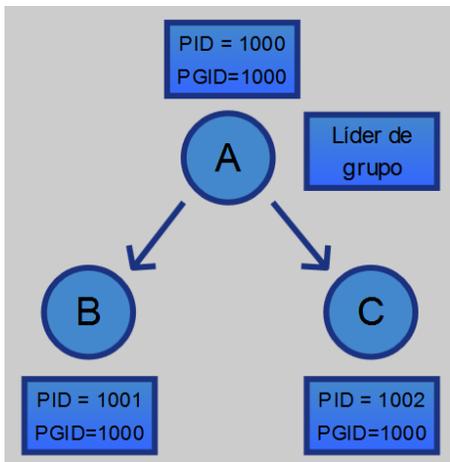
También se puede combinar fork y exec, ejecutando exec en el hijo reemplazamos la imagen del proceso hijo con la imagen de otro proceso, a continuación se detalla un ejemplo:

Ejemplo de código 5:

```
C A.c x +
C A.c > f main
/
int main(int argc, char *argv[])
8 {
9     char *arg[] = { "B.bin", "Hola mundo", NULL };
10    pid_t pid;
11    int estado;
12
13    pid = fork();
14
15    if( pid < 0 )
16    {
17        printf("Error al crear el nuevo proceso\n");
18        return EXIT_FAILURE;
19    }
20
21    if( pid )
22    {
23        printf("Soy el proceso padre. PID: %d\n",getpid());
24
25        wait( &estado );
26
27        if( estado == 0 )
28        {
29            printf("Hijo finalizado con éxito\n");
30        }
31        else
32        {
33            printf("Hijo finalizado con error\n");
34        }
35    }
36    else
37    {
38        execv("B.bin", arg );
39        printf("Error: %d\n",errno);
40        perror("execv");
41        exit(EXIT_FAILURE);
42    }
}

>_ Console x Shell x +
> gcc -o A.bin A.c
> ./A.bin
Soy el proceso padre. PID: 2083
Programa B ejecutado con execv. PID: 2084
Nombre: B.bin
Arg 1: Hola mundo
Hijo finalizado con éxito
> []
```

Process Group ID (PGID):



El PGID es un identificador que permite rápidamente identificar a un grupo de procesos. De nuevo como ejemplo si tenemos un padre (main) y dos hijos, al crear a los hijos estos heredan el PGID del padre. El PGID del padre se establece con el mismo número de PID al momento de ser creado.

En este caso el líder de grupo es el proceso padre creado cuando se ejecutó el programa principal.

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

```
pid_t getpgrp(void);
```

retorna el PGID del proceso llamador.

```
int setpgid(pid_t pid, pid_t pgid);
```

permite cambiar el PGID de un proceso en particular

```
int killpg(int pgrp, int sig);
```

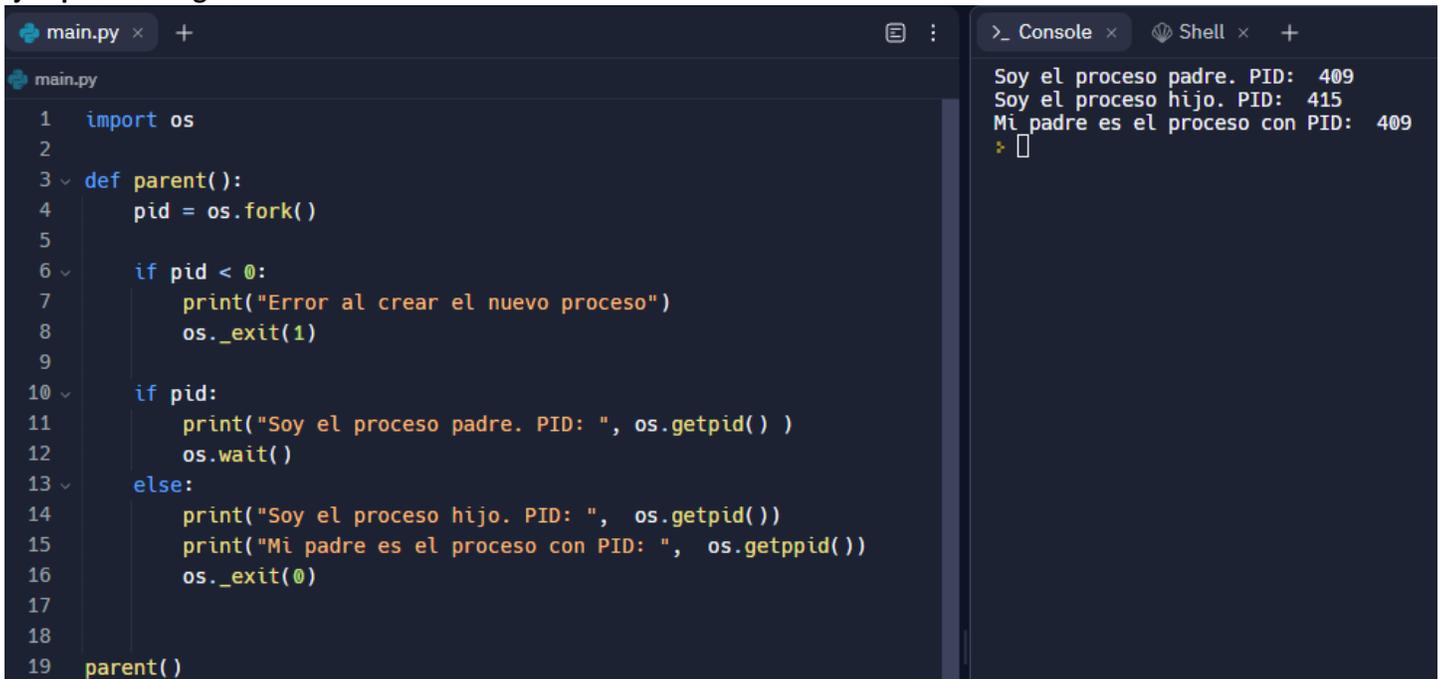
Ing. Dario Hirschfeldt

permite enviar una señal a todos los procesos que tengan el PGID especificado.

Creación de procesos en Python:

Para poder ejecutar las operaciones de `fork()`, `wait()`, etc, debemos **importar el modulo OS**, que nos permitirá interactuar con el sistema operativo, independientemente de si es MS Windows, Macintosh o Linux.

Ejemplo de código 6:



```

main.py x +
main.py
1 import os
2
3 def parent():
4     pid = os.fork()
5
6     if pid < 0:
7         print("Error al crear el nuevo proceso")
8         os._exit(1)
9
10    if pid:
11        print("Soy el proceso padre. PID: ", os.getpid() )
12        os.wait()
13    else:
14        print("Soy el proceso hijo. PID: ", os.getpid())
15        print("Mi padre es el proceso con PID: ", os.getppid())
16        os._exit(0)
17
18
19 parent()
    
```

```

>_ Console x Shell x +
Soy el proceso padre. PID: 409
Soy el proceso hijo. PID: 415
Mi padre es el proceso con PID: 409
>
    
```

Referencia al módulo os

<https://docs.python.org/es/3.10/library/os.html>

Métodos relacionados

- `os.fork()`
- `os.wait()` Retorna una tupla con [PID del hijo finalizado, estado de finalizacion]
- `os.waitpid(PID, opciones)` Mismo retorno que `os.wait()`
- `os._exit(estado)` Finaliza la ejecución del proceso llamador
- `os.execv(ruta, argumentos)` Los errores se informarán como excepciones (`OSError`)
- `os.kill(PID, nro de señal)` Envía una señal al proceso correspondiente al PID especificado
- `os.killpg(PGID, nro de señal)` Idem anterior pero a todos los procesos con grupo igual al valor de PGID

Ejemplo de código 7:

```
A.py
1 import os
2
3 def parent():
4     print("Proceso A. PID: ", os.getpid() )
5
6     try:
7         os.execv("/bin/ls", ["/bin/ls", "-l"] )
8
9     except Exception as error:
10        print("Error: ", os.strerror(error.errno) )
11        os._exit(1)
12
13 parent()
```

```
~/UsefulSteepWorkspace$ python A.py
Proceso A. PID: 10263
total 36
-rwxrwxrwx 1 runner runner 217 Feb 24 15:53 A.py
-rwxrwxrwx 1 runner runner 345 Feb 24 15:07 main.py
-rw-r--r-- 1 runner runner 17848 Jan 20 17:55 poetry.lock
-rw-r--r-- 1 runner runner 473 Jan 20 17:56 pyproject.toml
-rw-r--r-- 1 runner runner 655 Feb 16 01:29 replit.nix
drwxr-xr-x 1 runner runner 56 Jan 20 17:52 venv
~/UsefulSteepWorkspace$
```

En el código se muestra la utilización de **execv()** para ejecutar el comando ls, además se ejemplifica minimamente el manejo de errores. Para mayor información sobre el tratamiento de excepciones se puede consultar el siguiente link: <https://docs.python.org/3/library/exceptions.html>

```
B.py
1 import os
2
3 def parent():
4     print("Proceso A. PID: ", os.getpid() )
5
6     try:
7         os.execv("", ["/bin/ls", "-l"] )
8
9     except Exception as error:
10        print("Error: ", os.strerror(error.errno) )
11        os._exit(1)
12
13 parent()
```

```
~/UsefulSteepWorkspace$ python B.py
Proceso A. PID: 2826
Error: No such file or directory
~/UsefulSteepWorkspace$
```

Aquí ejecutamos generando un error.

Creación de procesos en JAVA:

La clase que nos permite crear nuevos procesos es **ProcessBuilder**.

A continuación un pequeño programa que es el que se ejecutará con un nuevo proceso:

Ejemplo de código 8:

```
HolaMundo x Main.java x HolaMundo.java x +
HolaMundo.java
1 class HolaMundo
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hola Mundo!");
6     }
7 }
```

Ejemplo de código 9:

```
HolaMundo x Main.java x HolaMundo.java x +
Main.java
1 import java.io.IOException;
2
3 public class Main
4 {
5     public static void main(String[] args)
6     {
7         try
8         {
9             ProcessBuilder pb = new ProcessBuilder("java", "HolaMundo.java");
10
11             pb.redirectErrorStream(true);
12             pb.inheritIO();
13
14             Process proceso = pb.start();
15
16             int error = proceso.waitFor();
17
18             if( error != 0 )
19             {
20                 System.out.println("Error al ejecutar el programa");
21             }
22         }
23         catch (IOException io_ex) {}
24         catch (InterruptedException int_ex) {}
25     }
26 }
```

```
~/ProcessJava$ java Main.java
Hola Mundo!
~/ProcessJava$
```

Primero instanciamos un objeto de la clase **ProcessBuilder** con el programa o comando a ejecutar y sus parámetros. Con el método **start()** se inicia el nuevo proceso. Con el método **waitFor()** esperamos la finalización del nuevo proceso desde el proceso llamador (Padre).

Cabe destacar que como los métodos **start()** y **waitFor()** pueden producir excepciones, hay que agregar el manejo de las mismas como en el ejemplo anterior. También se pueden introducir de la siguiente manera:

```
public class Main
{
    public static void main(String[] args) throws IOException, InterruptedException
    {
        try
        {
```

Y no harían falta las líneas de código de catch, de todas maneras la idea es siempre utilizar los manejadores de excepciones.

Creación de proceso

Anexo:

Funciones wait3 y wait4:

Ing. Dario Hirschfeldt

Si quisieramos obtener información sobre performance y la utilización de los recursos por parte de los procesos hijos, se pueden utilizar algunas de las funciones `wait3` y `wait4`, las cuales almacenarán dicha información en una estructura del tipo `rusage`.

```
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

A continuación se detalla un ejemplo:

```

struct_rusage.h x main.c x +
main.c > f main
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/resource.h>
5 #include <sys/wait.h>
6 #include <time.h>
7
8 #define TAM_ARRAY 1000
9
10 int main (int argc, char *argv[])
11 {
12     struct rusage uso_hijo;
13     int array[ TAM_ARRAY ];
14
15     srand( time(NULL) );
16     for(int i=0; i<TAM_ARRAY; i++)
17     {
18         array[ i ] = rand()%100;
19     }
20
21     if( ! fork() ) //Hijo
22     {
23         int suma=0;
24         for(int i=0; i<TAM_ARRAY; i++)
25         {
26             suma+=array[i];
27         }
28         printf("Suma del array: %d\n\n",suma);
29         return EXIT_SUCCESS;
30     }
31
32     if( wait3(NULL,0,&uso_hijo) < 0 )
33     {
34         perror("wait3");
35         return EXIT_FAILURE;
36     }
37
38     printf("Tiempo CPU sistema: %ld.%06ld\n", uso_hijo.ru_stime.tv_sec, uso_hijo.ru_stime.tv_usec);
39     printf("Tiempo CPU usuario: %ld.%06ld\n", uso_hijo.ru_utime.tv_sec, uso_hijo.ru_utime.tv_usec);
40     printf("Soft Page Faults: %ld\n",uso_hijo.ru_minflt);
41     printf("Hard page faults: %ld\n",uso_hijo.ru_majflt);
42     printf("Señales recibidas: %ld\n",uso_hijo.ru_nsignals);
43     printf("CS voluntario: %ld\n",uso_hijo.ru_nvcsw);
44     printf("CS involuntario: %ld\n",uso_hijo.ru_nivcsw);
45
46     return EXIT_SUCCESS;
47 }

```

```

>_ Console x Shell x +
> make -s
> ./main
Suma del array: 50245

Tiempo CPU sistema: 0.000000
Tiempo CPU usuario: 0.000359
Soft Page Faults: 39
Hard page faults: 0
Señales recibidas: 0
CS voluntario: 1
CS involuntario: 4

```

En el ejemplo se genera un proceso hijo que realiza la suma de todos los valores de un array e informa el resultado, posteriormente el padre recolecta la información de utilización de los recursos a través de la función `wait3`, dicha información queda almacenada en una estructura del tipo `rusage`.

A continuación se detalla la estructura `rusage`:

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long ru_maxrss; /* maximum resident set size */
    long ru_ixrss; /* integral shared memory size */
    long ru_idrss; /* integral unshared data size */
    long ru_isrss; /* integral unshared stack size */
    long ru_minflt; /* page reclaims (soft page faults) */
    long ru_majflt; /* page faults (hard page faults) */
    long ru_nswap; /* swaps */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
    long ru_msgsnd; /* IPC messages sent */
    long ru_mmsgcv; /* IPC messages received */
    long ru_nsignals; /* signals received */
    long ru_nvcsw; /* voluntary context switches */
    long ru_nivcsw; /* involuntary context switches */
};
```

Los tiempos son almacenados en una estructura del tipo timeval:

```
struct timeval {
    time_t tv_sec; // Number of whole seconds of elapsed time
    long int tv_usec; // Number of microseconds of rest of elapsed time minus tv_sec. Always
less than one million
};
```