

TEMA # 01

“PROCESOS”

1. OBJETIVO

Entender todos los conceptos relacionados con los procesos. Además de cómo identificarlos en un entorno Linux y de las funciones **system()**, **fork()** y **exec()** para su creación .

2. INTRODUCCIÓN

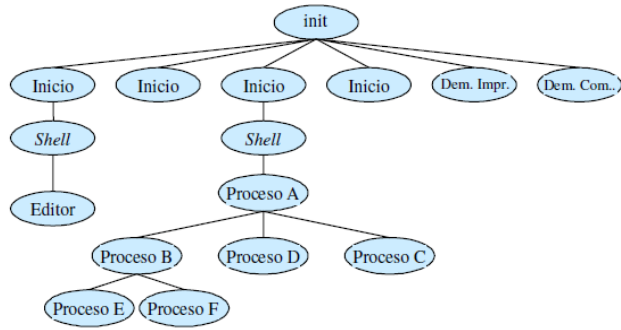
Para que la computadora realice alguna tarea, necesita ciertas instrucciones que le digan qué hacer. Un proceso es una instancia de un programa que se está ejecutando. Un explorador y un editor de texto —ejecutándose en la computadora— pueden estar corriendo uno o más procesos de tal manera que se estén realizando varias tareas.

Un proceso se compone de tres bloques:

- ✓ **Segmento de texto:** Contiene las instrucciones que entiende la CPU.
- ✓ **Segmento de datos:** Contiene los datos que deben ser cargados al iniciar el proceso.
- ✓ **Segmento de pila:** Es una serie de bloques lógicos —marcos de pila— , son introducidos cuando se llama a una función y son sacados cuando se vuelve de la función. Los marcos de pila se componen de los parámetros de la función, las variables locales y la información necesaria para restaurar el marco de la pila anterior a la llamada de la función.

Linux es un sistema de tiempo compartido que permite la ejecución de varios procesos a la vez (multiproceso). El planificador es la parte del núcleo encargada de gestionar la CPU y determinar qué proceso pasa a ocupar la CPU en un determinado instante.

Un proceso en Linux es una entidad creada tras la llamada *fork*, Todos los procesos, excepto el primero (proceso *init*), son creados mediante una llamada a *fork*. El proceso que llama a *fork* se conoce como proceso padre, y el proceso creado es el proceso hijo. Todos los procesos pueden tener varios procesos hijos, pero el proceso padre es único.



En Linux, cada proceso es identificado por un *ID de proceso* o *pid*. El ID de proceso es un número entero positivo que es asignado cada que un nuevo proceso es creado.

3. DESARROLLO

>>Ejemplo 1 – Número de procesos

En C, se puede obtener el número de proceso; así como el número de su proceso padre. Para obtener estos datos se usa la llamada al sistema **getpid()** (número de proceso) y **getppid()** (número de proceso padre). El siguiente código muestra el uso de las funciones.

```
#include <stdio.h>
#include <unistd.h> //Declara funciones que son parte del estándar POSIX

int main ()
{
    printf ("The process ID is %d\n", getpid ());
    printf ("The parent process ID is %d\n", (int) getppid ());
    return 0;
}
```

Código 1 Numeración de procesos (print-pid.c)

Para referencia de las cabeceras de C para UNIX, pueden referirse a la siguiente página:

<http://pubs.opengroup.org/onlinepubs/7908799/headix.html>

Una vez que se compila y se ejecuta el código anterior varias veces, se puede observar que el proceso padre no cambia, ya que se sigue ejecutando en el mismo Shell, pero lo que sí cambia es el número de proceso hijo.

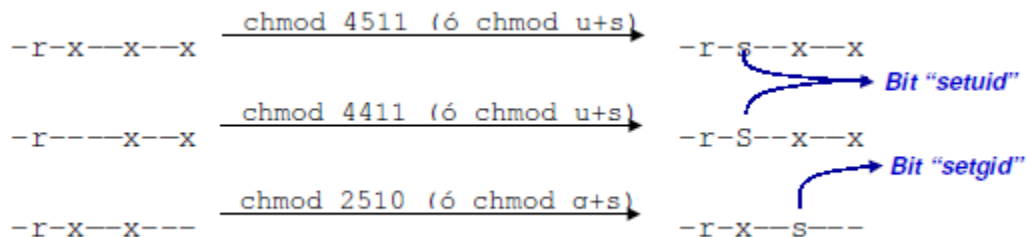
```
[root@vesta CLinux]# ./exe
The process ID is 26162
The parent process ID is 26076
[root@vesta CLinux]# ./exe
The process ID is 26165
The parent process ID is 26076
[root@vesta CLinux]# ./exe
The process ID is 26166
The parent process ID is 26076
[root@vesta CLinux]# ./exe
The process ID is 26167
The parent process ID is 26076
[root@vesta CLinux]# ./exe
The process ID is 26168
The parent process ID is 26076
[root@vesta CLinux]# ./exe
The process ID is 26169
The parent process ID is 26076
[root@vesta CLinux]# ./exe
The process ID is 26170
The parent process ID is 26076
[root@vesta CLinux]#
```

Además de los identificadores PID y PPID, cada proceso tiene otros que se muestran en la tabla siguiente.

Atributo	Tipo	Función
ID de proceso	pid_t	getpid(void);
ID de proceso padre	pid_t	getppid(void);
ID real del usuario	uid_t	getuid(void);
ID efectivo del usuario	uid_t	geteuid(void);
ID real de grupo	gid_t	getgid();
ID efectivo de grupo	gid_t	getegid();

Cada proceso tiene dos IDs de usuario y dos IDs de grupo. Son usados principalmente por cuestiones de seguridad, como la asignación de permisos de acceso a archivos. Los ID real de usuario y de grupo indican al usuario real, tal y como se encuentra en el archivo */etc/passwd*. Los ID efectivo de usuario y de grupo se usan para acceder a archivos de otros usuarios, enviar señales a procesos o ejecutar programas “*setuid*”.

Cuando un proceso ejecuta un programa “*setuid*”, el núcleo asigna al EUID del proceso el indicativo de propietario de dicho programa y al EGID del proceso, el indicativo del grupo del propietario de dicho programa.



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

int main(void)
{
    printf("Real user ID: %d\n", getuid());
    printf("Effective user ID: %d\n", geteuid());
    printf("Real group ID: %d\n", getgid());
}
```

```
printf("Effective group ID: %d\n", getegid());
return 0;
}
```

Código 2 Identificadores de procesos (ids.c)

```
[gkno@vesta CLinux]$ ./exe
Real user ID: 501
Effective user ID: 501
Real group ID: 501
Effective group ID: 501
```

>>Ejemplo 2 – Comando ps de Linux

En Linux existe un comando llamado ps que sirve para desplegar los procesos que se encuentran en ejecución en el sistema.

```
[root@vesta CLinux]# ps
  PID TTY          TIME CMD
 26076 pts/1        00:00:00 bash
 26219 pts/1        00:00:00 ps
```

La primera columna muestra el identificador del proceso. El primer proceso es el Shell que está corriendo y el segundo proceso es el programa *ps*. La siguiente ejecución de *ps* muestra parte de todos los sistemas que están corriendo en el sistema (opción -e); además de especificar la salida (opción -o pid, ppid, command).

```
[root@vesta CLinux]# ps -e -o pid,ppid,command
  PID  PPID  COMMAND
    1     0   init [3]
    2     1 [migration/0]
    3     1 [ksoftirqd/0]
    4     1 [watchdog/0]
    5     1 [migration/1]
    6     1 [ksoftirqd/1]
    7     1 [watchdog/1]
    8     1 [events/0]
    9     1 [events/1]
   10     1 [khelper]
```

>>Ejemplo 3 – Creación de procesos

En Linux hay tres maneras de crear procesos. La primera forma es a través del método **system()** que es parte de las librerías de C. También están **fork()** y **exec()**, que es la manera en que Linux lo hace.

+Uso de system()

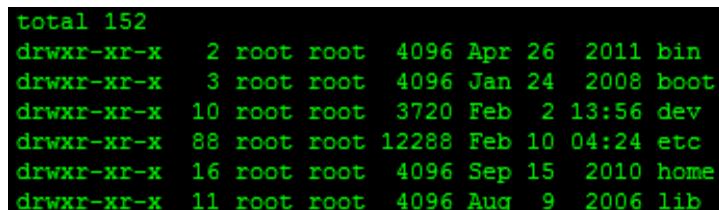
La función **system()** provee una manera sencilla de ejecutar un comando desde un programa, como si fuera invocado desde un Shell. Invocando un programa con los privilegios de root usando la función **system()**, puede tener diferentes resultado de un sistema Linux a otro, debido a que es dependiente de la versión de Shell que se ocupe.

La sintaxis de la función **system()** es la siguiente:

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

```
#include <stdlib.h>
int main ()
{
    int return_value;
    return_value = system ("ls -l /");
    return return_value;
}
```

Código 3 Uso de la función system() (system.c)



```
total 152
drwxr-xr-x  2 root root  4096 Apr 26  2011 bin
drwxr-xr-x  3 root root  4096 Jan 24  2008 boot
drwxr-xr-x 10 root root  3720 Feb  2 13:56 dev
drwxr-xr-x 88 root root 12288 Feb 10 04:24 etc
drwxr-xr-x 16 root root  4096 Sep 15  2010 home
drwxr-xr-x 11 root root  4096 Aug  9  2006 lib
```

+ Uso de fork()

Otra manera de crear procesos es usando la función **fork()**. La llamada a esta función crea un nuevo proceso. El nuevo proceso o proceso hijo será una copia del proceso padre excepto por el PID y el PPID.

Cuando realizamos una llamada a **fork()**, el núcleo del sistema realiza las siguientes operaciones:

- ✓ Buscar una entrada libre en la tabla de procesos y la reserva para el proceso hijo.
- ✓ Asigna un PID al proceso hijo, el cual es invariable y único durante toda la vida del proceso; además constituirá la clave para poder controlarlo desde otros procesos. Realiza una copia del contexto del nivel de usuario del proceso padre para el proceso hijo.

- ✓ También se copiarán las tablas de control de archivos locales del proceso padre al proceso hijo. Vuelve al proceso padre el PID del proceso hijo y el proceso hijo le devuelve el valor 0.

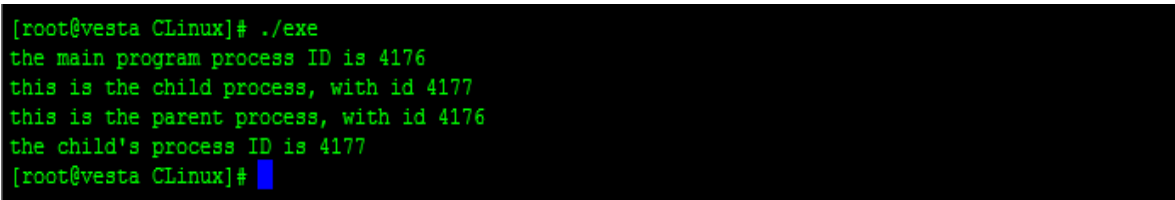
Mientras el proceso hijo es creado, el proceso padre se sigue ejecutando desde el punto donde la función **fork()** fue invocada. El nuevo proceso hijo —al igual que el padre— ejecuta el programa desde el mismo punto.

La sintaxis para el uso de la función **fork()** es:

pid_t fork(void)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
    pid_t child_pid;
    printf ("the main program process ID is %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid != 0) {
        printf ("this is the parent process, with id %d\n", (int) getpid ());
        printf ("the child's process ID is %d\n", (int) child_pid);
    }
    else
        printf ("this is the child process, with id %d\n", (int) getpid ());
    return 0;
}
```

Código 4 Creación de procesos (fork.c)



```
[root@vesta CLinux]# ./exe
the main program process ID is 4176
this is the child process, with id 4177
this is the parent process, with id 4176
the child's process ID is 4177
[root@vesta CLinux]#
```

A continuación una variante del código anterior:

```
Applications Places System Wed Aug 22, 11:16 PM Carlos Roman Zamitiz
Browse and run installed applications carlos@localhost:~
Window Menu Terminal Help
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    pid_t id_hijo;
    printf( "El id del proceso main es: %d\n", getpid() );
    id_hijo = fork();
    printf( "El id que devuelve fork es:%d\n", id_hijo );
    if( id_hijo != 0 )
    {
        printf( "El id del padre %d\n", getpid() );
        printf( "El id_hijo != 0 %d\n", id_hijo );
    }
    else
    {
        printf( "El id del padre %d\n", getpid() );
        printf( "El id_hijo == 0 %d\n", id_hijo );
    }
}

"fork.c" 20L, 440C written
```

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

int main(void)
{
    pid_t child;
    if((child = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if(child == 0) {
        puts("in child");
        printf("\tchild pid = %d\n", getpid());
        printf("\tchild ppid = %d\n", getppid());
        exit(EXIT_SUCCESS);
    } else {
        puts("in parent");
        printf("\tparent pid = %d\n", getpid());
        printf("\tparent ppid = %d\n", getppid());
    }
    exit(EXIT_SUCCESS);
}
```

[Código 5 Creación de procesos \(child.c\)](#)


```
[root@vesta CLinux]# ./exe
in child
    child pid = 4280
    child ppid = 4279
in parent
    parent pid = 4279
    parent ppid = 4141
[root@vesta CLinux]#
```

Algo que debe de notarse, es que no se puede predecir si un proceso padre se va a seguir ejecutando antes o después de la creación del proceso hijo y viceversa, ya que los procesos se ejecutan asincrónicamente debido al uso de la función **fork()**. Debido a este comportamiento no se debería de ejecutar código en el proceso hijo que dependa del proceso padre y viceversa; el hacer esto crea una condición de carrera (*race condition*), donde provocaría un comportamiento impredecible en el programa.

+Uso de **exec**

La función **exec()** —a diferencia de **fork()**— reemplaza el programa ejecutándose en un proceso por otro programa. Cuando un programa manda llamar la función **exec()**, su proceso termina inmediatamente y comienza a ejecutar el nuevo programa desde el inicio, asumiendo que **exec()** no encontró ningún error.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    char *args[] = {"/bin/ls", NULL};
    if(execve("/bin/ls", args, NULL) == -1) {
        perror("execve");
        exit(EXIT_FAILURE);
    }
    puts("shouldn't get here");
    exit(EXIT_SUCCESS);
}
```

Código 6 Uso de **exec** (execs.c)

```
[root@vesta CLinux]# ./exe
child.c exe execs.c fork.c ids.c print-pid.c system.c
[root@vesta CLinux]#
```

Se puede observar en la salida del programa no se hizo la impresión de: “**shouldn't get here**”. Cuando `exec` se ejecuta exitosamente, no regresa al proceso que lo llamó. En este caso, el proceso invocador fue reemplazado totalmente con el nuevo programa. Sin embargo, si `exec` falla, regresa un valor -1.

`Exec` es una familia de funciones que varía en sus capacidades dependiendo en cómo se mande llamar la función.

Las funciones `execl()`, `execv()`, `execle()` y `execve()` toman una ruta como su primer argumento. Las funciones `execlp()` y `execvp()` toman el nombre de un archivo, si el nombre no contiene una “/”, buscara dentro de `$PATH` un comando que coincida con el nombre para ser ejecutado.

Las tres funciones que contienen una letra *l* espera una lista de argumentos terminada con un apuntador `NULL`, que será pasada al programa que va a ser ejecutado. Las funciones que contienen un *v*, toman un arreglo de apuntadores terminando con una cadena `NULL`. Por ejemplo, si se quiere ejecutar los comandos `/bin/cat /etc/passwd /etc/group` usando alguna de las funciones con *l*, simplemente se pasa cada uno de los comandos de la lista y se termina con la cadena `NULL`

```
execl("/bin/cat", "/bin/cat", "/etc/passwd", "/etc/group", NULL);
```

Usando una de las funciones con *v*, se tiene que construir el arreglo y después pasarlo a la función `exec`.

```
char *argv[] = {"/bin/cat", "/etc/passwd", "/etc/group", NULL};
```

```
execv("/bin/cat", argv);
```

Finalmente, las dos funciones terminadas con *e* —`execve()` y `execle()`— permiten crear un ambiente especializado para ser ejecutado. El ambiente es guardado en la variable `envp`, que es un apuntador a una lista de cadenas terminada en `NULL`. Cada cadena en la lista toma la forma `nombre=valor`, donde `nombre` es el valor de la variable de ambiente y `valor`, su valor.

```
char *envp[] = "PATH=/bin:/usr/bin", "USER=joe", NULL};
```

Las otras funciones reciben implícitamente su ambiente a través de la variable `environ` que apunta a un arreglo de cadenas que contiene las variables de ambiente. Para manipular el ambiente se usa las siguientes funciones contenidas en `<stdlib.h>`.

```
int putenv(const char *string);
```

```
char *getenv(const char *name);
```

A continuación se muestra la sintaxis de la familia `exec`.

```
int execl(char *path, char *arg0, char *arg1,... char *argN, (char *)0);
int execv(char *path, char *argv[ ]);
int execl(char *path, char *arg0, char *arg1,... char *argN, (char *)0, char *envp[ ]);
int execve(char *path, char *argv[ ], char *envp[ ]);
int execlp(char *file, char *arg0, char *arg1,... char *argN, (char *)0);
int execvp(char *file, char *argv[ ]);
```

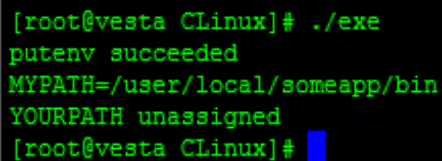
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char envval[] = {"MYPATH=/user/local/someapp/bin"};
    if(putenv(envval))
        puts("putenv failed");
    else
        puts("putenv succeeded");

    if(getenv("MYPATH"))
        printf("MYPATH=%s\n", getenv("MYPATH"));
    else
        puts("MYPATH unassigned");

    if(getenv("YOURPATH"))
        printf("YOURPATH=%s\n", getenv("YOURPATH"));
    else
        puts("YOURPATH unassigned");
    exit(EXIT_SUCCESS);
}
```

Código 7 Variables de ambiente (testenv.c)



```
[root@vesta CLinux]# ./exe
putenv succeeded
MYPATH=/user/local/someapp/bin
YOURPATH unassigned
[root@vesta CLinux]#
```

A continuación otro ejemplo utilizando `execve` para mostrar la ejecución de un programa secundario pasando su nombre como argumento desde otro programa primario:

```
/* myecho.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char *argv[])  
{  
    int j;  
  
    for (j = 0; j < argc; j++)  
    {  
        printf("argv[%d]: %s\n", j, argv[j]);  
    }  
  
    exit(EXIT_SUCCESS);  
}
```

~
~
~
~
~
~
~
~
~
~
~

"myecho.c" [New] 16L, 337C written

```
/* execve.c */

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

int main(int argc, char *argv[])
{
    char *newargv[] = { NULL, "hello", "world", NULL };
    char *newenviron[] = { NULL };

    if (argc != 2)
    {
        fprintf(stderr, "Sintaxis: %s <file-to-exec>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    newargv[0] = argv[1];

    execve(argv[1], newargv, newenviron);
    perror("execve"); /* execve() only returns on error */
    exit(EXIT_FAILURE);
}

~
~
~
~
"execve.c" [New] 23L, 671C written
```

Para ejecutarlos se compilan de la siguiente manera:

```
cc myecho.c -o myecho
cc execve.c -o execve
./execve ./myecho
```

Analizar el resultado.

4. CUESTIONARIO

El siguiente código muestra el uso de las funciones fork() y exec(). Describe qué es lo que hace y qué resultados arroja.

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

int spawn (char* program, char** arg_list)
{
```

```
pid_t child_pid;
child_pid = fork ();

if (child_pid != 0)
    return child_pid;
else {
    execvp (program, arg_list);
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
}
}

int main ()
{
    char* arg_list[] = {
        "ls",
        "-l",
        "/",
        NULL
    };

    spawn ("ls", arg_list);
    printf ("done with main program\n");
    return 0;
}
```

Código 8 Uso de las funciones fork() y exec() juntas (fork-exec.c)

5. BIBLIOGRAFÍA

- [1] Francisco Manuel Márquez García. “Unix: programación avanzada”. Editorial RA-MA
- [2] Richard Stevens. “Advanced Unix programming”. Editorial Addison Wesley
- [3] Kurt Wall. “Linux Programming by Example”. Editorial QUE