

# PROGRAMACIÓN DE SISTEMAS: PROCESOS

Sexta parte

Abril 2023

## Resultados de aprendizaje:

- ✓ Definir proceso para diferenciarlo de otros elementos de Linux.
- ✓ Identificar y caracterizar pasos para creación de procesos.
- ✓ Reconocer y caracterizar la supresión de procesos para diferenciar las posibilidades (killing).
- ✓ Tipificar las diferentes formas de manipulación de procesos para adquirir versatilidad operativa y valorar la potencialidad de Linux.

# CARACTERIZACIÓN GENERAL

**Aproximación al concepto de proceso:** es un programa cargado en memoria destinado a ejecutarse.

**Definición formal de proceso:** Es una instancia de la ejecución de un programa con un determinado espacio de direcciones.

**Nota:** Es la unidad básica de programación del sistema operativo. Un proceso se puede asimilar a un programa en ejecución. Además un programa puede iniciar varios procesos al mismo tiempo.

## ELEMENTOS DE UN PROCESO

Un proceso consta de los siguientes elementos:

- El **contexto** del programa en curso, que es el estatus corriente de ejecución del programa.
- El **directorio** corriente de trabajo del programa.
- Los **archivos y directorios** a los cuales tiene acceso el programa.
- Las **credenciales** o derechos de acceso del programa tales como su modo de archivo y propiedad.
- La **cantidad de memoria** y otros recursos del sistema asignados al proceso.

# Aclaraciones:

- Un proceso consta de: un programa ejecutable, los datos, su pila, sus punteros de pila y programa, registros y toda información necesaria para que pueda ser ejecutado, teniendo siempre una vida limitada en la memoria de la arquitectura.
- Los procesos son la unidad básica de programación de Linux. El kernel los utiliza para controlar el acceso a la CPU y a otros recursos del sistema.
- Los procesos de Linux determinarán que programas correrán en la CPU, por cuánto tiempo, y con qué características.

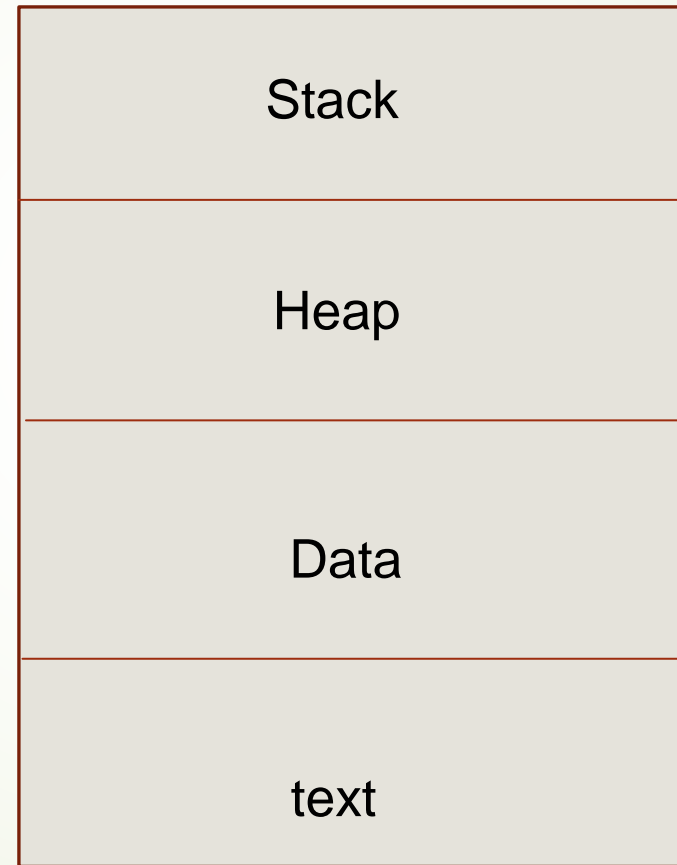


# Aclaraciones:

Continuación

- El fijador de tiempos de kernel distribuye los tiempos de ejecución a cargo de la CPU, denominados cuotas, entre todos los procesos, apropiándose de cada uno de ellos sucesivamente cuando su cuota de tiempo expira.
- Las cuotas de tiempo son lo suficientemente pequeñas como para que, en un sistema mono procesador, dé la impresión de que varios procesos se están ejecutando simultáneamente.
- Cada proceso tiene la suficiente información sobre sí mismo como para que el kernel pueda activarlo y desactivarlo según sea necesario.

# Segmento de un proceso en Linux



# ATRIBUTOS DE UN PROCESO

Los procesos tienen atributos o características que los identifican y definen su comportamiento



# Atributos de un proceso

Continuación

- El kernel también guarda internamente una gran cantidad de información acerca de cada proceso y contiene una interfaz o grupo de llamadas a funciones que le permiten obtener dicha información.
- Esta información consta de:
  - a) Identificadores de proceso
  - b) Estados y tipos de procesos

# Identificadores de proceso

- ❖ Los atributos básicos de un proceso son su identificador o ID (PID) y el identificador o ID de su proceso padre (PPID). **PID y PPID son números enteros positivos y distintos de cero.**
- ❖ **Un PID identifica a un proceso de forma unívoca e inequívoca** (garantiza unicidad dentro de una máquina). Es único para cada proceso que esté en ejecución. Rango de valores de PID:  
$$0 < \text{PID} < 65536$$
- ❖ Entero 0 cuando se arranca el sistema y volviendo a comenzar en 0 cuando alcanza el máximo, asignando un PID que no esté asociado a un proceso activo al momento de iniciar la ejecución de un nuevo proceso.
- ❖ Cuando un proceso (proceso padre) crea un nuevo proceso, se dice que tiene un proceso hijo.

# Identificadores de proceso

Continuación

- ❖ Se puede trazar la ascendencia de un proceso hasta llegar al proceso que tiene el PID 1, llamado proceso init (padre de todos los procesos).
- ❖ El proceso init es el primer proceso que aparece después que de que arranca el kernel.
- ❖ El proceso de init pone en funcionamiento el sistema, comienza los daemons y ejecuta los programas que se deban correr.

# Estados y Tipos de Procesos

En los S.O multitarea los procesos pueden tener alguno de los siguientes estados:

- ❖ Ejecutándose
- ❖ Listo para ejecutar
- ❖ Esperando

# Estados y Tipos de Procesos

Continuación

Definición de los estados:

- ❖ **Ejecutándose:** Únicamente un proceso utiliza la CPU. Sus modos son:
  - Modo user: ejecuta instrucciones del programa de usuario.
  - Modo kernel: ejecuta instrucciones del kernel (recordar: que estas son llamadas al S.O).
- ❖ **Listo para ejecutar:** varios procesos pueden tener este estado (en memoria ram o disco), este estado implica que hay varios procesos disputándose el uso CPU y sólo pasan a ejecutarse cuando reciben permiso.



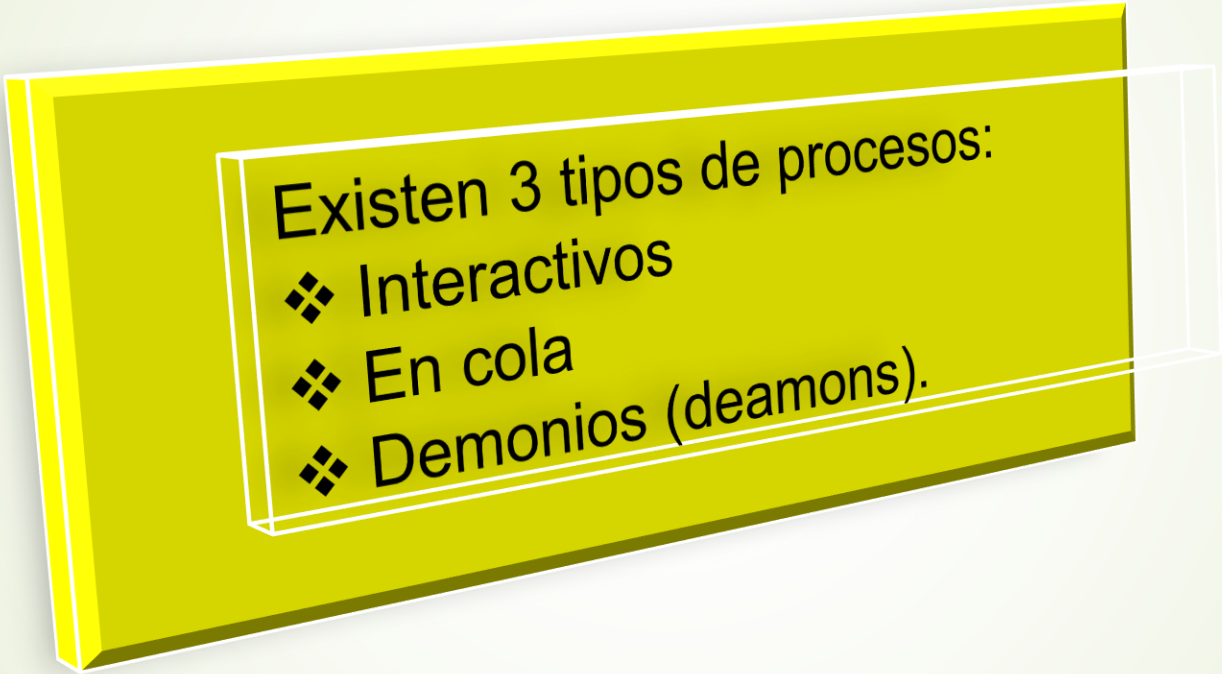
# Estados y Tipos de Procesos

Continuación

- ❖ **Esperando** (o Durmiendo). Varios programas pueden estar en este estado. “Esperando” se entiende como a la espera de obtener algún recurso o de que ocurra un suceso. En este estado puede alojarse en ram o disco. Una vez que el proceso sale del estado de espera, sea porque obtuvo el recurso sea que el suceso se produjo, el proceso entra en una cola de procesos llamada “preparados para ejecutarse” (recordar que un proceso en este estado no podrá ejecutarse aunque se le asigne CPU).



# Tipos de procesos



Existen 3 tipos de procesos:

- ❖ Interactivos
- ❖ En cola
- ❖ Demonios (deamons).

# Tipos de procesos

Continuación

Caracterización de los tipos de procesos:

- **Interactivos**: Pueden ejecutarse en foreground (primer plano) o background (segundo plano). Son iniciados y controlados por shell.
- **En cola**: están en cola y allí esperan ser ejecutados secuencialmente. (no están asociados a terminal).
- **Deamons**: son procesos lanzados al iniciar el sistema ejecutándose en background.

# Tipos de procesos

Continuación

**Linux construye una tabla**, llamada “tabla de procesos” en la que registra información de los procesos que se están ejecutándose. Para ello el usuario puede usar el comando **ps** (visualiza el estado de procesos) sin argumento para acceder a datos de los procesos en ejecución en su terminal (un usuario).

# Tipos de procesos

Continuación

La tabla se presenta de la siguiente forma:

PID: id del proceso    TIME: es el tiempo de CPU empleado por el proceso desde que se ejecutó.

TTY: terminal del que lee y al que escribe cada proceso. El símbolo “?” significa que el proceso no está asociado a un terminal .

COMMAND: Muestra la orden que lanzó ese proceso.

PID	TIME	TTY	COMMAND
1	0.2	?	init
5	0.2	?	update

# Tipos de procesos

Algunos argumentos del comando **ps** (seguido de distintos argumentos permiten obtener otro datos además de los que trae la tabla):

-f ofrece información adicional UID (id del propietario del proceso), PPID (id del proceso padre), C cantidad de recursos de procesador que el proceso utilizó recientemente (este número permite determinar el proceso de menor C para asignar CPU, antes que otro de mayor C, estableciendo un criterio de asignación de CPU), STIME hora del día que se inició el proceso (si no se corresponde con el día en curso, muestra día y mes de inicio)



# Tipos de procesos

Continuación

Algunos argumentos del comando **ps** (seguido de distintos argumentos permiten obtener otro datos además de los que trae la tabla):

- e muestra todos los procesos que estén ejecutándose (activos) en el sistema (se utiliza como argumento para diagnosticar el sistema: conocer el tiempo que lleva ejecutándose, ver si hay procesos “colgados”, conocer PID, los recursos empleados, prioridad relativa).
- r muestra sólo procesos con estado “ejecutándose”
- u muestra los procesos activos de un usuario NN
- l muestra los Flags asociados con el proceso, estado del proceso (S durmiendo, R corriendo), prioridad (dinámica) del proceso (PRI), valor empleado en cálculo de prioridad, Dirección de RAM, número de bloques de la imagen del proceso (SZ), dirección del proceso por el cual espera (WCHAN) cuando el proceso está en espera.



# Funciones getpid y getppid

Las funciones que permiten que un proceso obtenga su PID y su PPID son getpid y getppid

Los códigos de estas funciones están en <unistd.h>, sus prototipos son:

```
pid_t getpid ( void),  
pid_t getppid (void);
```

- getpid devuelve el número PID del proceso que efectuó el llamado
- getppid retorna el número PPID de quien la llamó, que sería el PID del padre del proceso que llamó a getpid.

# Funciones getpid y getppid

Continuación

Algunos motivos para que un proceso conozca su pid o pid de su padre:

- ❑ Un uso común de un PID es crear archivos o directorios que sean únicos. Luego de una llamada a getpid, el proceso podría utilizar su PID para crear un archivo temporario.
- ❑ Otra funcionalidad es escribir el PID en un archivo de registro de actividades de un programa como parte de un mensaje del mismo, para dejar en claro qué proceso fue el que grabó el mensaje del registro.
- ❑ Un proceso puede utilizar su PID para enviar una señal u otro mensaje a su proceso padre.

# Funciones getpid y getppid

Continuación

EJEMPLO: El programa muestra por pantalla su PID y el PPID del mismo

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main ( void) { printf ( "\n PID = %d", getpid ( ) );
printf ( "\n PPID = %d", getppid ( ) ); exit (EXIT_SUCCESS);}
```

La salida variará según el sistema en el que se encuentre.

# Funciones getpid y getppid

Continuación

Aclaraciones del código:

`#include <unistd.h>` declara funciones que son parte del estándar POSIX (Portable Operating System Interface eXtensions).

POSIX es una familia de normas que definen los servicios y capacidades que debe proveer un sistema operativo para ser considerado “según POSIX”

El acatamiento a POSIX es importante porque asegura (en teoría) que las aplicaciones escritas para funcionar en un tipo de sistema sean sencillas de transportar a otro sistema.

# Identificaciones reales y efectivas

Además de su PIDs y su PPIDs cada proceso tiene otros atributos de identificación que se muestran en la Tabla de Atributos de Proceso junto a su tipo en C y las funciones en que los retornan.

Para utilizarlas se debe incluir en el código las librerías `<sys/types.h>` como `<unistd.h>`.

# Identificaciones reales y efectivas

Continuación

Tabla de atributos de proceso:

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



# Identificaciones reales y efectivas

Continuación

Cada proceso tiene 3 IDs de usuarios (UIDs) y 3 IDs de grupo (GIDs).

Se emplean principalmente por razones de seguridad: asignar permisos de acceso a archivos, y limitar quien puede ejecutar ciertos programas.

El ID de usuario real y el ID de grupo real indican quién es el usuario concreto.

Son leídos de `/etc/passwd` cuando uno ingresa al sistema. Constituyen las representaciones numéricas del nombre de acceso y la principal pertenencia a un grupo del usuario que está ingresando.

# Identificaciones reales y efectivas

Continuación

¿Por qué la necesidad una identificación de usuario real y efectiva?

Porque puede ser posible que en algunas circunstancias sea necesario tomar temporalmente la identidad de otro usuario (generalmente sería root , pero podría ser cualquier usuario). Si solo existiera una identificación de usuario, entonces no habría forma de volver a cambiar a su identificación de usuario original después.

**Nota:** la identificación de usuario real es quién es realmente (el propietario del proceso), y la identificación de usuario efectiva es lo que el sistema operativo observa para tomar una decisión sobre si se le permite o no hacer algo (la mayoría de las veces, hay algunas excepciones).

# Identificaciones reales y efectivas

Continuación

Ejemplo: muestra los UIDs y los GIDs reales y efectivos del proceso.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main ( void) { printf ( "\n ID de usuario real = %d", getuid ( ) );
printf ( "\n ID de usuario efectivo = %d", geteuid ( ) );
printf ( "\n ID de grupo real = %d", getgid ( ) );
printf ( "\n ID de grupo efectivo = %d", getegid ( ) );
exit (EXIT_SUCCESS);}
```

La salida variará según el sistema en el que se encuentre.

# Información de usuario

Los usuarios trabajan y reconocen mejor los nombres que los números asociados a ellos. Existen maneras de convertir los UIDs en nombres legibles para los usuarios, una de estas maneras es a través de la función `getlogin ( )` la cual retorna el nombre de acceso del usuario que ejecuta un proceso.

`getlogin ( )` está declarada en `<unistd.h>` y su prototipo es `char * getlogin ( void);`

`getlogin` retorna un puntero a una cadena que contiene el nombre de acceso del usuario que se encuentra corriendo el proceso o `NULL` si la información no se encuentra disponible.

# Información de usuario

Continuación

Ejemplo: muestra la forma de usar getlogin

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
int main ( void) { char * nom;
/*nombre de acceso*/
if ( (nom = getlogin ( ) ) == NULL ) { perror ( "getlogin"); // muestra mensaje
                                     exit ( EXIT_FAILURE); }
printf ( "\n getlogin de retorno %s", nom );
exit (EXIT_SUCCESS);}
```

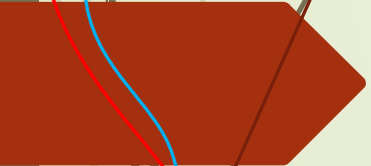
Si el if sale por NULL perror imprimirá un mensaje de error y el programa terminará.

EXIT\_FAILURE : El valor de esta macro suele ser -1. Se puede utilizar para indicar que el programa ha encontrado un error indeterminado.



# SESIONES Y GRUPOS DE PROCESOS

Existen situaciones en las cuales el modelo padre/hijo no alcanza para describir las relaciones entre procesos.



Un grupo de procesos es un conjunto de procesos relacionados generalmente una secuencia de comandos en una pipeline. Todos los procesos de un grupo de procesos tienen el mismo ID de grupo de proceso, o sea el PGID.



# Sesiones y Grupos de Procesos

Continuación

La finalidad de un grupo de procesos es la de facilitar el control de tareas, por ejemplo:

Supongamos que el usuario corre la pipeline de comandos:

```
ls -l /usr / include | sort | wc -l
```

si mientras la pipeline está en ejecución , se la quiere abortar (con Ctrl +C) la interfaz debe poder terminar todos los procesos (lo cual lo realiza abortando el grupo de procesos), en lugar de hacerlo proceso por proceso (cada proceso en forma individual).

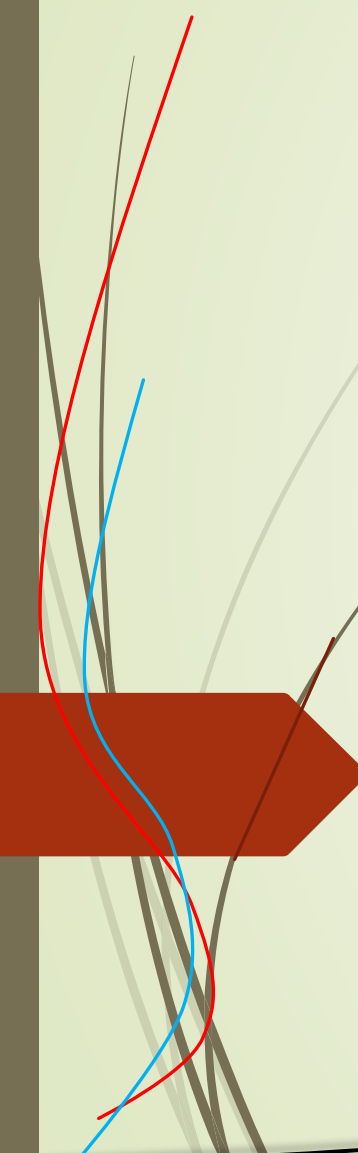
# Sesiones y Grupos de Procesos

Continuación

- ❑ Una SESIÓN consiste de uno o más grupos de procesos.
- ❑ Un líder de sesión es el proceso que crea la sesión, ésta tiene un único identificador, denominado ID de sesión, el cual es el PID del líder de sesión.
- ❑ Las sesiones cumplen el mismo propósito para los grupos de procesos que estos últimos para los procesos individuales

# MANIPULACIÓN DE PROCESOS

- Creación de procesos. Linux ofrece 3 maneras de hacerlo: llamadas a `system` (provista en por la biblioteca estándar de C) y llamadas a `fork` y `exec`.
- Eliminación de procesos.
  - ✓ Función `kill`, cuando se liquida un proceso.
  - ✓ Para liquidarse a si mismo un proceso usa: `exit` o `abort`



Las buenas prácticas de administración de recursos requieren que los procesos padres aguarden hasta que terminen sus hijos (funciones `wait`).

# Creación de Procesos

Continuación

Utilización de system: La función system ejecuta un comando de interfaz que se le transfiere. El prototipo está en <stdlib.h> es  
`int system (const char *string);`

system ejecuta el comando de interfaz string transfiriéndoselo a /bin/sh y retornando después de haber terminado de ejecutarse dicho comando.

Si por alguna razón system no lograra invocar a /bin/sh, retorna el código 127; si ocurriera algún otro tipo de error, system retornará -1. El valor normal de retorno de system es el código de salida del comando transferido en string.

Si string resulta ser NULL, system retornará 0 si /bin/sh se encontrase disponible o un valor distinto de 0 en caso contrario.

# Creación de Procesos

Continuación

## Mediante la utilización de fork

Cuando el usuario introduce un comando, una vez que el shell analiza la línea de comando, decide si se trata de un comando propio del shell o un comando externo que reside en disco.

Para la primera situación descripta anteriormente, el proceso se lleva a cabo mediante llamadas a sistema, que consiste en peticiones a servicios que proporciona el núcleo (única forma del proceso de acceder al hardware).

Para la segunda situación el shell emite una llamada a fork, haciendo que el kernel cree un clon del proceso padre que lanzó la llamada. **Esto quiere decir que el proceso de efectuar un fork involucra el copiado de toda la imagen de memoria del proceso padre al proceso hijo.**



# Creación de Procesos

Continuación

Mediante la utilización de fork

**La llamada a fork crea un nuevo proceso.** Este proceso hijo será una copia del proceso que hizo la llamada, o proceso padre. La función fork está declarada en `<unistd.h>` y su sintaxis es:

`pid_t fork ( void);`

- ✓ Si la llamada tiene éxito, fork retorna el PID del proceso hijo al proceso padre y retorna 0 al proceso hijo. Esto significa que aún si uno llamara a fork una sola vez, la misma retorna dos veces.
- ✓ El nuevo proceso que crea fork es una copia exacta del proceso padre, excepto por su PID y su PPID. fork realiza una copia completa del proceso padre, incluyendo los UIDs y GIDs reales y efectivos y los IDs de grupo de proceso y de sesión, el entorno, los límites de los recursos , archivos abiertos y segmentos de memoria compartida.

# Creación de Procesos

Continuación

Mediante la utilización de fork (o bifurcación crea una imagen de si mismo)

- ✓ Las diferencias entre el proceso padre y el hijo son ínfimas.
- ✓ El proceso hijo no hereda: las alarmas programadas en el proceso padre (realizadas mediante una llamada a alarm), los bloques de archivos creados por el proceso padre y las señales pendientes.
- ✓ El concepto clave a ser comprendido es que fork crea un nuevo proceso que es un duplicado exacto del proceso padre, a excepción de lo indicado.
- ✓ Es importante destacar que el shell es para cada usuario el padre de todos los procesos que él ejecute. A partir del shell se crea la estructura jerárquica de los procesos que se ejecutan desde cada terminal. Todo proceso proviene de un padre. Solamente existe una excepción y es el proceso 0 que es el primero que se crea al inicializarse el SO.

# Creación de Procesos

Continuación

Ejemplo sencillo de utilización de fork

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main ( void)
{ pid_t hijo;
  if ( (hijo = fork ( ) ) == -1 ) { perror ( "fork"); exit ( EXIT_FAILURE); }
  else if ( hijo == 0 ) { puts ( " En el proceso hijo");
    printf ( " \t pid = %d \n", getpid ( ) );
    printf ( " \t ppid = %d \n", getppid ( ) );
    exit (EXIT_SUCCESS);}
  else { puts ( " En el proceso padre");
    printf ( " \t pid = %d \n", getpid ( ) );
    printf ( " \t ppid = %d \n", getppid ( ) ); }
    exit (EXIT_SUCCESS);
  // otras acciones
}
```

# Creación de Procesos

Continuación

Ejemplo sencillo de utilización de fork

La salida de este programa es similar a:

\$ ./hijo

En el proceso hijo

En el proceso padre

pid del hijo = 1150

ppid del hijo = 1149

pid del padre = 1149

ppid del padre = 706

# Creación de Procesos

Continuación

Ejemplo sencillo de utilización de fork

Análisis de la salida:

- ✓ Como puede apreciarse el PPID del proceso hijo (es el ID del proceso padre) es el mismo que el PID del padre, 1149.
- ✓ La salida del programa también muestra algo crítico en el uso de fork, no se puede predecir si un proceso padre se ejecutará antes o después del hijo, lo cual se ve en la salida del programa: la primera línea de salida proviene del proceso padre, las líneas 2da a 4ta provienen del proceso hijo, la 5ta y 6ta línea provienen nuevamente del proceso padre.
- ✓ El programa se ha ejecutado fuera de secuencia, es decir **ASÍNCRONAMENTE**.



# Creación de Procesos

Continuación

Ejemplo sencillo de utilización de fork  
Análisis de la salida:

Una llamada a fork puede fracasar porque hay muchos procesos corriendo en ese momento en el sistema o porque el proceso que está intentando generar procesos descendientes ha excedido el número de procesos que está permitido ejecutarse, si la llamada fracasa fork retorna -1 al proceso padre y no crea ningún proceso hijo.

# Creación de Procesos

Continuación

## Utilización de exec

Se trata en realidad de 6 funciones, con empleos ligeramente diferentes unas de otras. Son conocidas en bloque como la función exec. Está declarada en <unistd.h>. Los prototipos son los siguientes:

```
int execl ( const char * path, const char *arg, ...);  
int execlp ( const char * file, const char *arg, ...);  
int execl_e ( const char * path, const char *arg, char *const envp [ ] );  
int execv ( const char * path, char * const argv [ ] );  
int execve ( const char * path, char * const argv [ ], char * const envp [ ] );  
int execvp ( const char * file, char * const argv [ ] );
```

exec reemplaza completamente la imagen del proceso que efectuó la llamada con la del programa iniciado por exec.

# Creación de Procesos

Continuación

Ejemplo de uso:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main ( void)
{ char * args [ ] = { “ /bin/lS”, NULL};
if ( execve ( “/bin/lS”, args, NULL ) == -1) {
    perror ( “execve”);
    exit ( EXIT_FAILURE); } puts ( “\n No llegar aquí”);
exit (EXIT_SUCCESS); } //(La salida depende del entorno)
```

Si exec tiene éxito no regresa nunca al proceso que lo llamó → las 2 últimas líneas del programa nunca se ejecutarán. **Se deja al alumno explorar las diferencias entre las 6 funciones de la familia exec.**

# Esperas en proceso

Continuación

Luego de que se genera un proceso (mediante fork) o ejecuta un proceso nuevo (con exec), el proceso padre debe esperar a que el proceso hijo termine para recoger su condición de salida y evitar la generación de zombies.

Un zombie es un proceso hijo que termina sin que su proceso padre disponga recoger la condición de salida del mismo con wait o waitpid.

El proceso padre recoge la condición de salida de un proceso hijo utilizando una de las funciones wait para recuperar la condición de salida desde la tabla de procesos del kernel. Este tipo de proceso se llama zombie, porque está muerto pero sigue presente en la tabla de procesos.

# Esperas en proceso

Continuación

- El proceso hijo ha terminado, ha sido liberada la memoria y los recursos asignados al mismo, pero todavía ocupa un lugar en la tabla de procesos del kernel.
- El kernel almacena la condición de salida hijo hasta que el proceso padre se retire de allí.
- Tener algunos zombies no es un problema grave, pero si un programa ejecuta constantemente comandos fork y exec no recoge sus condiciones de salida, **la tabla de procesos se puede llenar**, lo que deteriora el desempeño y obliga a volver a arrancar el sistema.
- Otra condición no deseada son los procesos huérfanos.



# Esperas en proceso

Continuación

Para recoger la condición de salida de un proceso hijo se debe hacer una llamada a `wait` o `waitpid`.

Para ello hay que incluir `<sys/types.h>` y `<sys/wait.h>`.

Los prototipos son:

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

```
pid_t waitpid ( pid_t pid, int *status, int opción);
```

**status:** condición de salida del proceso hijo.

**opción:** indica cómo se deberá comportar la llamada a `wait`

**pid** es el PID del proceso por el cual se quiere aguardar, puede adoptar 4 valores

# Esperas en proceso

Continuación

Ejemplo de uso de wait.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
# include < sys/type.h>
int main ( void) { pid_t hijo; int condición;
  if ( (hijo = fork ( ) ) == -1 ) { perror ( "fork");
    exit ( EXIT_FAILURE); }
  else if ( hijo == 0 ) { puts ( " En proceso hijo");
    printf ( " \tpid de proceso hijo = %d\n ", getpid ( ) );
    printf ( stdout, " \tppid de proceso hijo = % d \n", getppid ( ) );
    exit (EXIT_SUCCESS); }
  else { waitpid ( hijo, &condición, 0);
    puts ( "El proceso padre: " );
    printf ( " \t pid del proceso padre = %d\n", getpid ( ) );
    printf ( " \t ppid del proceso padre = %d\n", getppid ( ) );
    printf ( " \t El proceso hijo retorno %d\n", condicion ); }
  exit ( EXIT_SUCCESS); }
```

# Eliminación (killing) de procesos

Un proceso puede terminar por una de 5 razones:

- Su función main llama a return
- Llamada a exit
- Llama a \_exit
- Llama a abort
- Es finalizado por una señal

# Eliminación (killing) de procesos

Continuación

Función kill : permite a un proceso poner fin a otro.

El prototipo de kill es:

```
int kill (pid_t pid, int sig);
```

Se debe incluir las librerías `<sys/types.h>` como `<signal.h>`

pid: indica el proceso que se desea eliminar

sig: es la señal que se desea enviar para realizarlo.

## BIBLIOGRAFÍA:

“Programación en Linux con ejemplos” Autor: Kurt Wall  
Editorial: 1° edición, Prentice Hall (2000)

“Aprendiendo C++ para Linux en 21 Días”. Autores: Jesse Liberty  
y David B. Horvath. Prentice Hall (2001)



# GRACIAS