Procesos y señales en C-Linux

Armando Rivera

20 de octubre de 2018

1. Depurador gdb

Para trabajar con este depurador se tiene que compilar el programa a depurar con la bandera -g

```
$ gcc -g -o test test.c
```

Para usar este depurador ejecutamos el comando gdb seguido del nombre de nuestro programa: *gdb test*. Comandos:

- **l** *(list)*: Este comando muestra el código fuente del programa que estoy depurando.
- **r** *(run)*: Corre el programa
- **b n** *(break)*: Pone un breakpoint donde n es la linea donde queremos dicho breakpoint
- **c***(continue)*: Continua con el programa despues de un breakpoint.
- **p var** *(print variable)*: Imprime el valor de una variable.
- **delate n**: Elimina el break point que se encuentra en la linea n.
- **n** *(next)*: Continúa con la siguiente linea y avanza con el código linea por línea.

2. Comando top

Este comando nos proporciona información de los procesos que están corriendo en tiempo real.

1. `Primera línea`: muestra una serie de datos referidos al sistema.
   - **Hora actual**
   - **Tiempo que el sistema ha estado corriendo**
   - **Cantidad de usuarios conectados**

- **load average**: los números indican el estado de us del CPU. Si los números son menores a 1 esto quiere decir que el CPU no tiene que esperar para poder ejecutar una instrucción

2. `Segunda línea`: Muestra el total de procesos que están corriendo y los divide por estados, Running, slepping, Stopped, Zombie.
3. `Tercera línea`: Muestra el uso del CPU.
   - %us: muestra el uso de CPU del usuario.
   - %sy: Uso del CPU del sistema
   - %id: Muestra el uso del CPU disponoble para utilizar.
   - %wa: Muestra en porcentaje el tiempo en espera del CPU.
4. `Cuarta línea`: Muestra los valores referentes a la memoria fisica del equipo.
   - **Total**: Valor total de la memoria fisica.
   - **used**: memoria utilizada.
   - **Free**: Memoria libre.
   - **Buffered**: Memoria que está en el buffer
5. `Quinta línea` Muestra valores referentes al uso de memoria swap.
6. **PID**: Process ID del proceso
7. **USER**: Usuario que esta corriendo dicha aplicación.
8. **PR**: Prioridad del proceso.
9. **NI**: valor por el cual se establece una prioridad para el proceso.
10. **VIRT**: Total de la memoria virtual utilizada.
11. **RES**: Resident task size.
12. **SHR**: Estado del proceso
13. **%CPU %MEM**: Porcentajes de memoria y cpu utilizadas.
14. **Time**: Tiempo de uso del procesador para ese proceso.
15. **Comand**: Comando que esta siendo ejecutado por el Deamon

3. Procesos

3.1. Definición de procesos

Un programa es una colección de instrucciones y de datos que se encuentran almacenados en un fichero ordinario.
Un programa generalmente consta de las siguientes partes:

- Un conjunto de cabeceras que describen atributos del fichero
- Un bloque donde se encuentran las instrucciones en lenguaje máquina del programa

Cuando un programa es leído del disco por el núcleo y es cargado en memoria para ejecutarse, se convierte en un proceso. A cada proceso se le asigna un un número que lo identifica, este numero es llamado *PID*. Un proceso se compone de tres bloques fundamentales conocidos como segmentos:

- *Segmento de texto*: Contiene las instrucciones que entiende la CPU de nuestra máquina.
- *Segmento de datos*: Contiene los datos que deben ser inicializados al arrancar el proceso.
- *Segmento de pila*: Se crea al arrancar el proceso y su tamaño es gestionado dinámicamente.

3.2. Estados de un proceso

El tiempo de vida de un proceso se puede dividir en un conjunto de estados, cada uno con unas características determinadas. Los estados de un proceso son:

- *new (Nuevo)*: Cuando el proceso es creado
- *Ejecutando (running)*: El proceso tiene asignado un procesador y está ejecutando sus instrucciones
- *Bloqueo (waiting)*: El proceso está esperando por un evento.
- *Listo (ready)*: El proceso está listo para ejecutar.
- *Finalizado*: El proceso finalizó su ejecución

Al crearse un proceso inmediatamente pasa al estado listo.
Ante una interrupción que se genere, el proceso puede perder el recurso del procesador y pasar al estado de listo En el estado listo, El planificador será el encargado de seleccionar el próximo proceso a ejecutar.
De ejecutando a bloqueado: A medida que el proceso ejecuta instrucciones realiza pedidos en distintos componentes. Teniendo en cuenta que el pedido puede demorar y, además, si está en un sistema multiprogramado, el proceso es puesto

en una cola de espera hasta qe se complete su pedido. De esta forma se logra utilizar en forma mas eficiente el procesador.
De bloqueado a listo: Una vez que ocurre el evento que el proceso estaba esperando, es puesto nuevamente en la cola de procesos listos.

3.3. Proceso Zombie

Cuando un proceso finaliza en sistemas Unix, toda su memoria y recursos asociados a él se desreferencian (típico exit) para que puedan ser usados por otros procesos. En ese espacio de tiempo, la entrada del proceso hijo en la tabla de procesos permanece un mínimo tiempo, hasta que el padre conoce que el estado de su proceso hijo es finalizado y entonces lo saca de la tabla de procesos.

Para que el proceso padre sepa el estado de su hijo, se le envía una señal SIGCHLD indicando que el proceso hijo ha finalizado. Esa señal es generada gracias a llamadas al sistema como wait() / waitpid() / waitid().

¿Qué pasa cuando no se usa esos manejadores para conocer el estado de los hijos (función wait() / waitpid() / waitid())? Pues que el padre no sabe que su hijo ha terminado y por lo tanto sigue en la lista de procesos. Los procesos zombie se generan por tanto, cuando el padre no recibe esa señal o bien la ignora, generalmente por bugs o aplicaciones mal programadas

Es posible, aunque algo poco común, que el padre esté muy ocupado y no pueda en ese momento matar al proceso. También podría ser que el padre decida tener un proceso zombie en la tabla para reservar ese PID, o que el padre esté interesado en eliminar los procesos hijos en un determinado orden,...

El tener procesos zombies en la tabla no suele ser un problema, al no ser que su número crezca exponencialmente y se ocupen todos los identificadores de procesos que el sistema operativo puede utilizar o bien se necesite el PID que el proceso fantasma ocupa.

4. Señales

4.1. Definición

Las señales son interrupciones que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o situación especial. Cuando un proceso recibe una señal puede reaccionar de de tres formas distintas:

- Ignora la señal con lo cual inmune la misma
- Invocar a la rutina de tratamiento por defecto: Esta rutina no la aporta el programador sino que es proporcionada por el núcleo
- Invocar a una rutina propia que se encarga de tratar la señal

4.2. Tipos de señales en Linux

Cada señal tiene un número asociado, por lo tanto al enviar una señal, en realidad se está enviando un número. Los tipos de señales más conocidos son:

- Señales relacionadas con la terminación de procesos.
- Señales relacionadas con las excepciones inducidas por los procesos. Por ejemplo: al intentar acceder a una dirección de memoria no permitida
- Señales originadas desde un proceso en ejecución. Un proceso envía una señal a otro via *kill*
- Señales relacionadas con la interacción de la terminal.
- Señales para ejecutar un proceso paso a paso

A continuación se muestran algunas señales importantes.

- `SIGHUP`: *Desconexión.* La acción por defecto de esta señal es terminar la ejecución del proceso que la recibe.
- `SIGINT`: *Interrupción.* Se envía a todo proceso asociado con una terminal de control cuando se pulsan las teclas de interrupción $Ctrl - C$
- `SIGALARM`: *Despertador.* Es enviada a un proceso cuando alguno de sus temporizadores descendentes llega a cero. Su acción por defecto es terminar el proceso
- `SIGTERM`: *Finalización controlada.* Es la señal utilizada para indicarle a un proceso que debe terminar su ejecución. La rutina de tratamiento de esta señal se encarga de tomar las acciones necesarias para dejar al proceso en un estado coherente y a continuación finalizar su ejecución con una llamada a exit
- `SIGUSR1`: *Señal numero 1 de usuario.* Señal numero 1 de usuario. Esta señal está reservada para uso del programador.
- `SIGTRAP`: *Trace trap.* Cuando un proceso se está ejecutando *paso a paso*, esta señal es enviada después de ejecutar cada instrucción. Es empleada por los programas de depuradores.

4.3. Envío de señales

El envio de señales se hace mediante el comando *kill* cuyo prototipo es:

```
#include < signal.h >
int kill (pid_t pid, int sig)
```

4.4. Tratamiento de señales

Para especificar qué tratamiento debe realizar un proceso al recibir una señal, se emplea la llamda `signal`.

```
#include < signal.h >
void*signal (int sig, void (*action ())) ();
```

4.5. Escucha de señales

En ocasiones puede interesar que un proceso suspenda su ejecución en espera de que ocurra algún evento exterior a él. Por ejemplo al ejecutar una entrada/salida. Para estas situaciones nos valemos de la llamada a pause

```
#include < unistd.h >
int pause (void);
```