Práctica 6: comandos para identificar procesos

Dr. Hermes Francisco Montes Casiano

1. Introducción

En un sistema multiprogramado o de tiempo compartido, un *proceso* es la imagen de memoria de un programa, junto con la información relacionada con el estado de su ejecución. Un programa (el archivo en disco) es una *entidad pasiva*, una lista de instrucciones; un proceso es una *entidad activa*.

En contraposición con proceso, en un sistema por lotes se habla de *tareas*. Una tarea requiere mucho menos estructura, típicamente es suficiente guardar la información relacionada con la utilización de recursos empleados. Una tarea no es interrumpida en el transcurso de su ejecución.

Si bien el sistema operativo da la ilusión de que muchos procesos se están ejecutando al mismo tiempo, la mayor parte de ellos están esperando que le sea otorgado tiempo de cómputo para continuar su ejecución, en un momento determinado sólo se está ejecutando un número de procesos igual o menos al número de procesadores que tenga el equipo de cómputo.

2. Monitoreando procesos

Siempre hay varios procesos ejecutándose en el sistema, a pesar de que el usuario no inicie ningún proceso. Existen diversas herramientas para monitorear los procesos que están en ejecución, algunas gráficas y otras en línea de comandos.

Cuando un programa se ejecuta en el sistema, se le denomina proceso. Para examinar los procesos, deberá familiarizarse con el comando *ps*, la *navaja suiza* de las utilidades. Puede producir mucha información sobre todos los programas que se ejecutan en su sistema.

Desafortunadamente, con esta solidez viene la complejidad, en forma de numerosos parámetros, lo que hace que el comando *ps* sea probablemente uno de los comandos más difíciles de dominar. La mayoría de los administradores de sistemas encuentran un subconjunto de estos parámetros que proporcionan la información que desean y luego se limitan a utilizarlos únicamente.

El comando *ps* básico no proporciona tanta información:

```
1 $ ps
2  PID TTY      TIME          CMD
3  785 pts/0    0:00.26      bash
4  824 pts/0    0:00.25      ps
5 $
```

Ejemplo 1: Salida básica del comando *ps*

De forma predeterminada, el comando *ps* muestra solo los procesos que pertenecen al usuario actual y que se están ejecutando en el terminal actual. La salida básica muestra el ID de proceso (PID) de los programas, el terminal (TTY) desde el que se ejecutan y el tiempo de CPU que ha utilizado el proceso.

Actividad 1: El comando *ps*

1. Ejecute el comando *ps*, observe la salida, compare con la salida mostrada en el ejemplo 1 y escriba sus comentarios.
2. Utilice el comando *man* para conocer los parámetros que se pueden proporcionar al comando *ps*, describa las diez opciones que considere más útiles, justifique su elección.

El comando *ps* es excelente para obtener información sobre los procesos que se ejecutan en el sistema, pero tiene un inconveniente, el comando *ps* muestra la información de los procesos que se ejecutan al momento que se ejecuta el comando *ps*.

Si está tratando de observar el comportamiento de procesos que se intercambian con frecuencia dentro y fuera de la memoria, es difícil hacerlo con el comando *ps*. En cambio, el comando *top* puede resolver este problema. El comando *top* muestra la información del proceso de manera similar al comando *ps*, pero lo hace en modo de tiempo real, ver figura 1.

La primera sección de la salida muestra información general del sistema. La primera línea muestra la hora actual, cuánto tiempo ha estado activo el sistema, la cantidad de usuarios conectados y el promedio de carga en el sistema. Cuanto más altos sean los valores, más carga está experimentando el sistema. No es raro que el primer valor de carga sea alto para periodos breves de actividad. Si el tercer valor de carga es alto, su sistema puede tener problemas.

La segunda línea muestra información general del proceso: cuántos procesos están en ejecución, inactivos, detenidos y zombis (procesos que han finalizado pero su proceso principal no ha respondido). La siguiente línea muestra información general de la CPU. La pantalla superior desglosa la utilización de la CPU en varias categorías según el propietario del proceso (procesos de usuario frente a procesos del sistema) y el estado de los procesos (en ejecución, inactivos o en espera).

```

alumno@debian: ~
File Edit View Search Terminal Help
top - 23:35:15 up 1 day, 2:37, 3 users, load average: 0.57, 0.22, 0.08
Tasks: 173 total, 2 running, 171 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.0 us, 0.0 sy, 0.0 ni, 99.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2052524 total, 155440 free, 981372 used, 915712 buff/cache
KiB Swap: 2096124 total, 2096124 free, 0 used, 847836 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 1289 alumno    20   0 1520600 353008 74892 R   2.0  17.2  18:37.54 cinnamon
 1159 alumno    20   0  183768   2912  2468 S   0.7   0.1   5:45.34 VBoxClient
   733 root        20   0  435640 102552 38480 S   0.3   5.0   6:33.41 Xorg
 1386 alumno    20   0  710932  58476 36156 S   0.3   2.8   3:40.79 cinnamon-screen
 4685 alumno    20   0 50.629g 222968 106068 S   0.3  10.9   0:54.50 code
 5046 alumno    20   0  628380  15768  9076 S   0.3   0.8   0:09.66 cpptools-srv
 9558 root         20   0     0     0     0 S   0.3   0.0   0:00.06 kworker/0:1
    1 root         20   0  139156   6936  5216 S   0.0   0.3   0:01.78 systemd
    2 root         20   0     0     0     0 S   0.0   0.0   0:00.00 kthreadd
    3 root         20   0     0     0     0 S   0.0   0.0   0:01.91 ksoftirqd/0
    5 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 kworker/0:0H
    7 root         20   0     0     0     0 S   0.0   0.0   0:15.37 rcu_sched
    8 root         20   0     0     0     0 S   0.0   0.0   0:00.00 rcu_bh
    9 root         rt   0     0     0     0 S   0.0   0.0   0:00.00 migration/0
   10 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 lru-add-drain
   11 root         rt   0     0     0     0 S   0.0   0.0   0:01.35 watchdog/0
   12 root         20   0     0     0     0 S   0.0   0.0   0:00.00 cpuhp/0
   13 root         20   0     0     0     0 S   0.0   0.0   0:00.00 kdevtmpfs
   14 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 netns
   15 root         20   0     0     0     0 S   0.0   0.0   0:00.18 khungtaskd
   16 root         20   0     0     0     0 S   0.0   0.0   0:00.00 oom_reaper
   17 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 writeback
   18 root         20   0     0     0     0 S   0.0   0.0   0:00.00 kcompactd0
   19 root         25   5     0     0     0 S   0.0   0.0   0:00.00 ksm
   21 root         39  19     0     0     0 S   0.0   0.0   0:00.00 khugepaged
   22 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 crypto
   23 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 kintegrityd
   24 root         0 -20     0     0     0 S   0.0   0.0   0:00.00 bioset

```

Figura 1: Salida por defecto del comando *top*.

Actividad 2: El comando *top*

1. Ejecute cada el comando *top*, observe la salida, compare con la salida mostrada en la figura 1 y escriba sus comentarios.
2. Utilice el comando *man* para conocer los parámetros que se pueden proporcionar al comando *top*, describa las cinco opciones que consideres más útiles, justifique su elección.

3. La llamada al sistema *fork*

La llamada al sistema *fork* crea un nuevo proceso. El nuevo proceso (proceso hijo) es una copia exacta del proceso que realiza la llamada (proceso padre) excepto en lo siguiente:

- El identificador del proceso.
- El identificador de proceso padre.
- El proceso hijo tiene su propia copia de los descriptores del padre.
- Las estadísticas de utilización de recursos.

La llamada *fork* devuelve 0 al proceso hijo, mientras que al proceso padre le devuelve el *PID* del proceso hijo. En caso de que ocurra un error en la creación del proceso le devolvera

-1 al proceso que realiza la llamada y el proceso hijo no se crea, ver ejemplo 2.

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<unistd.h>
4 #include<sys/types.h>
5 #include<sys/wait.h>
6
7 int main() {
8     pid_t pid;
9     int i=0, estado;
10    pid = fork();
11    if(pid==-1) {
12        perror("Error en la creación del proceso\n");
13        return -1;
14    } else if(pid == 0) {
15        printf("Soy el proceso hijo - PID: %d, PPID: %d, i: %d\n",
16            getpid(), getppid(), i++);
17        return 0;
18    } else {
19        printf("Soy el proceso padre - PID: %d, PPID: %d, i: %d\n",
20            getpid(), getppid(), i--);
21        wait(&estado);
22        return 0;
23    }
24 }
```

Ejemplo 2: Creación de procesos utilizando la llamada *fork*

Todo proceso que termina debe llamar a la función *exit*. El parámetro de *exit* se le envía a su proceso padre y éste puede recibirlo en el parámetro de la llamada *wait* para saber como terminó su proceso hijo. Si el proceso padre muere antes de que termine el proceso hijo, el hijo quedará huérfano y será adoptado por el proceso INIT.

Con la llamada al sistema *fork* también es posible crear dos o mas procesos hijos dentro de cada proceso, pudiendo formar un árbol de procesos con distintas ramificaciones, ver ejemplo 3.

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<unistd.h>
4
5 int main() {
6     pid_t pid;
7     int j, estado;
8     for(j=0; j<5; j++) {
```

```

9         pid = fork();
10        if(pid != 0) {
11            break;
12        }
13    }
14    wait(&estado);
15    printf("El padre del proceso %d es %d, j=%d\n", getpid(),
16          getpid(), j);
17    sleep(3);
18 }

```

Ejemplo 3: Creación de múltiples procesos utilizando la llamada *fork*

Actividad 3: La llamada a *fork*

1. Transcriba el código del ejemplo 2, compilelo, ejecútelo y describa la salida del programa.
2. Modifique el programa del ejemplo 2 de modo que el proceso hijo quede huérfano e imprima el *PID* de su padre adoptivo.
3. Transcriba el código del ejemplo 3, compilelo, ejecútelo y describa la salida del programa (utilice el comando *pstree* para observar el árbol de procesos).
4. Modifique el código del ejemplo 3 de forma tal que todos los procesos sean creados por el mismo proceso padre y que cada proceso imprima su *PID* y su *PPID*.

Pregunta 1

¿Si el ciclo *for* del código del ejemplo 3 va de 0 a 4 por que hay 6 impresiones?

Pregunta 2

Analice la contribución de la función *break* en el código del ejemplo 3, ¿Cuántos procesos se crearían si se eliminara?