PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 2/2017
Soluciones Interrogación 1

Lunes 29-Agosto-2017

Duración: 2 horas
SIN CALCULADORA

1. **[21p]** Responda **brevemente** las siguientes preguntas

   1.1) **[5p]** De las siguientes operaciones, indique cuáles deben ser ejecutados en modo *kernel*, y cuáles pueden ser ejecutadas en modo *user*. Justifique **brevemente**.

   a) Invocar una *syscall* desde un programa en C
   R. *User*. Si el usuario no puede invocar una *syscall* desde un programa, no puede acceder a los recursos del sistema operativo.

   b) Copiar el espacio de direcciones de un proceso desde la RAM al disco
   R. *Kernel*. Si el usuario puede copiar cualquier espacio de direcciones desde la RAM al disco, entonces podría guardar el estado de cualquier proceso y leer sus datos.

   c) Leer un mensaje recibido por la interfaz de red
   R. *Kernel*. Si el usuario puede leer cualquier mensaje recibido por la interfaz de red, podría leer mensajes destinados a cualquier proceso. Debe ser el sistema operativo el que dice quién puede leer qué.

   d) Establecer el valor del *timer* para que genere una interrupción
   R. *Kernel*. Si el usuario puede establecer el valor del *timer*, podría evitar postergar una interrupción y utilizar la CPU por el tiempo que quiera.

   e) Modificar el registro PC (*Program Counter*)
   R. *User*. El PC se modifica en modo usuario después de cada ejecución. Más aún, el proceso de usuario podría desplazar el PC a cualquier valor. Sin embargo, si lo establece en una dirección de memoria no permitida, se generará una interrupción.

   1.2) **[2p]** En los años 1960s se introdujo la multiprogramación a los sistemas computacionales, lo que trajo una ventaja importante en términos de eficiencia. ¿Por qué la introducción de la multiprogramación fue una ventaja?

   R. Antes de la multiprogramación, un proceso debía ser ejecutado completamente antes de empezar a ejecutar el siguiente. Esto implicaba que si un proceso ejecutaba operaciones de E/S, la CPU no podía hacer nada hasta que esa operación terminara. Con la multiprogramación, ahora la CPU podía ejecutar instrucciones de otro procesos mientras el proceso que estaba esperando E/S quedaba bloqueado, lo que aumentó la utilización de la CPU.

   1p. Mencionar que antes no se cargaba más de un proceso en memoria.

   1p. Mencionar que se mejoró el uso de CPU, o que ahora el proceso que ejecuta E/S pasa a estado *waiting*, o que ahora puede haber siempre un proceso usando la CPU.

   1.3) **[2p]** En los sistemas modernos, parte del procesamiento de un cambio de contexto se escribe en *assembler*, en lugar de un lenguaje de alto nivel (como C). ¿Por qué?

   R. Ejecutar un cambio de contexto implicar modificar registros del *hardware*, lo que es dependiente de la arquitectura. Aún en C, esto se programa insertando explícitamente instrucciones de *assembler*, debido a que son instrucciones específicas para cada arquitectura.

   1p. Argumentar que hay dependencia de la arquitectura.

   1p. Argumentar que se deben modificar registros (al menos el PC).

NO es correcto decir que “es más eficiente”

NO es correcto decir que “son instrucciones protegidas”

1.4) **[2p]** ¿Es posible implementar un sistema operativo sin utilizar *timer*s? Justifique su respuesta.

R. Sí es posible. En ese caso el sistema debe incluir alguna *syscall* que permita que los procesos entreguen la CPU voluntariamente, y confiar en que los procesos la usarán. Tiene el riesgo que no hay cómo forzar a un proceso a terminar en caso que éste entre en un ciclo infinito. Este tipo de sistema se conoce como sistema con *scheduling cooperativo*. Windows 3.1 y las primeras versiones de MacOS (antes de X) funcionaban de esta manera.

1p. Mencionar que sí es posible.

1p. Argumentar debilidades que tendría este modelo, como la posibilidad que un proceso bloquee todo el sistema.

1.5) **[2p]** ¿Qué diferencia hay entre los sistemas *batch* y los sistemas interactivos?

R. Un sistema *batch* recibe una lista de procesos y los ejecuta de acuerda a algún orden. Los procesos se ejecutan completamente, si bien pueden ejecutar operaciones de E/S con dispositivos como disco o red, pero no pueden efectuar interacción con el usuario. Un sistema interactivo, en cambio ejecuta acciones dependiendo de su interacción con el usuario (pantalla, teclado, mouse).

1p. Características de sistema *batch*, donde alguna contraste con el sistema interactivo.

1p. Características de sistema *interactivo*, donde alguna contraste con el sistema *batch*.

Esta pregunta no tiene que ver necesariamente con algoritmos de *scheduling*. Se puede tener un sistema interactivo, que utiliza un *scheduler* FIFO, lo que funcionaría a pesar de tener un pésimo tiempo de respuesta.

1.6) **[2p]** ¿Qué diferencia hay entre las señales `SIGTERM` y `SIGKILL`, que pueden ser enviadas con la *syscall* `kill`?

R. SIGTERM indica al proceso que debe terminar, y es capturable por el proceso. SIGKILL es una señal recibida por el sistema operativo y no por el proceso, por lo tanto el proceso no puede capturarla.

1p. Mencionar que SIGTERM es capturable, e indica al proceso que debe termianr.

1p. Mencionar que SIGKILL no es capturable, o que va directamente al sistema operativo.

1.7) **[6p]** Se desea introducir un nuevo tipo de interfaz: LightUSB, para lo cual el sistema operativo debe ser capaz de reconocer estos dispositivos y utilizar sus *drivers*. Suponiendo que posee el código fuente de los *drivers* necesarios, ¿qué pasos debería seguir para incorporar los *drivers* a su sistema operativo? Mencione una ventaja y una desventaja para cada uno de estos casos

a) Sistema monolítico

R. 1p. Se debe incorporar el *driver* junto con el código fuente del sistema operativo, y recompilar el kernel.

1p. Ventaja. Ejecución más eficiente.

1p. Desventaja. Trabajo de incorporar otro código fuente y de recompilar.

b) Sistema monolítico con módulos

R. 1p. Se debe compilar el *driver* y utilizar una función (*syscall*) del *kernel* para cargar este módulo como parte del *kernel*.

1p. Ventaja. Ejecución más eficiente y menos trabajo para incorporar el módulo ya que no hay que recompilar.

1p. Desventaja. Es (un poco) menos eficiente que tener el *driver* incorporado al *kernel*

c) Sistema basado en microkernel

R. 1p. Se debe compilar el *driver* y utilizar una función (*syscall*) del *kernel* para cargarlo y marcarlo como módulo, el cual sin embargo queda ejecutando como un proceso de usuario.

1p. Ventaja. Más flexible y fácil de cargar o descargar como extensión del *kernel*.

1p. Desventaja. Ejecuta en modo *user* y está limitado a interactuar con el resto del *kernel* vía *syscalls*, lo que lo hace menos eficiente.

2. **[9p]** Se proponen dos alternativas para implementar una instrucción que bloquea un proceso por `T` segundos, donde `GetTime()` es una función que retorna un entero con el tiempo actual (un *timestamp*) en segundos.

```c
// Alternativa A
void blockFor(unsigned int T) {
    unsigned int t = GetTime();
    while (GetTime() - t < T)
        ; // do nothing in loop
}
```

```c
// Alternativa B
#include <unistd.h>
void blockFor(unsigned int T) {
    sleep(T);
}
```

Describa la diferencia entre ambas alternativas en términos de utilización de CPU, estado de los procesos, y comportamiento en las colas del sistema operativo.

R. 3p. Respecto a utilización de la CPU la alternativa A utiliza la CPU durante todo el tiempo que dura la espera. Realiza una "espera ocupada" (*busy waiting*). La alternativa B permite que otro proceso use la CPU, pues la instrucción `sleep` llama una *syscall* que deja al proceso en la cola *waiting*. Si bien ambas podrían mantener la CPU al 100 % (en el caso de B dejando que otro proceso la ocupe), la alternativa A no hace trabajo útil para el usuario, ni deja que otro proceso lo haga.

3p. La alternativa A mantiene al proceso en estado *running*, salvo por una eventual interrupción por *timer*. La alternativa B deja al proceso en estado *waiting*.

3p. La alternativa A mantiene al proceso en la cola *ready* o en uso de la CPU. La alternativa B deja al proceso en la cola *waiting* hasta que se cumple el tiempo solicitado.

3. **[15p]** De acuerdo al estado del sistema presentado en la hoja adjunta, responda las siguientes preguntas:

   3.1) **[3p]** En la situación A, los usuarios `cruz` y `jlopez` están ejecutando el proceso `tom`, el cual hace uso intensivo de CPU. ¿Cómo puede explicar el hecho que haya varios procesos en ejecución y todos posean 100 % de uso de CPU?

   R. Esto ocurre porque el sistema debe tener más de una CPU (y al menos cuatro), ya que cada uno de los cuatro `tom` hace uso completa de la CPU.

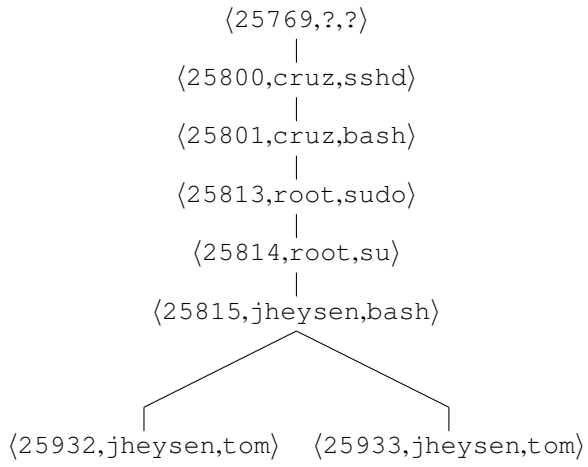
   3.2) **[4p]** En la situación B, el usuario `jheysen` ha iniciado también el proceso `tom`. ¿Por qué esto hace que el uso de CPU de todos los procesos baje? El proceso `tom` se ejecutó de la misma manera, y con los mismos argumentos que utilizaron los usuarios `cruz` y `jlopez`.

   R. Esto ocurre porque el sistema debe tener menos de 5 CPUs, ya que si se trata del mismo proceso que los anteriores, también debe intentar ocupar el 100 % de la CPU, pero como no hay CPUs suficientes, el *scheduler* debe distribuir el tiempo de las 4 CPUs entre los procesos que hay disponibles.

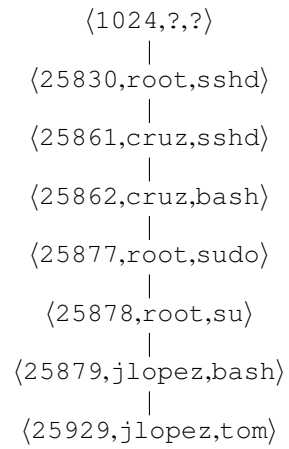
   3.3) **[4p]** Dibuje, con la información disponible, un árbol de procesos para la situación B. Para cada proceso incluya el par $\langle$ *PID*, *command* $\rangle$

   R. En los árboles que se presentan a continuación, la suma es 5p. Se evalúa cada árbol, pero la suma máxima de esta pregunta no puede ser más de 4p.

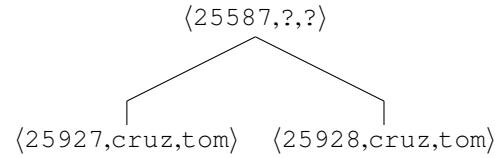
   1p por el árbol que ejecuta `sudo su jheysen; tom`.



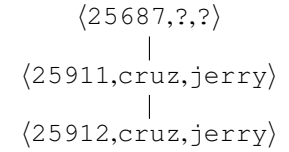
1p por el árbol que ejecuta `sudo su jlopez; tom`



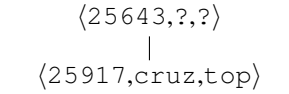
1p. por el árbol de `cruz` que ejecuta dos procesos `tom`.



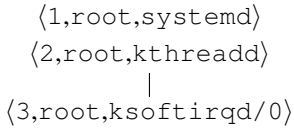
1p. por el árbol de `cruz` que ejecuta los procesos `jerry`.



0.5p por el árbol de `top`.



0.5p por el árbol de `systemd` ó por el árbol de `kthreadd`. Si están ambos, solo se suma 0.5p.



3.4) **[4p]** En la situación C, solo el usuario `cruz` se encuentra ejecutando sus procesos. Además, él se encuentra ejecutando dos procesos `jerry`. ¿Por qué estos procesos no consumen CPU?

R. Estos procesos están en estado *sleeping*, lo que significa que deben estar esperando algún evento (término de un hijo, o alguna E/S) para continuar. Como los procesos están en la cola *waiting*, no están consumiendo CPU, por lo tanto el uso que se ve es normal.

4. **[15p]** Se solicita escribir un programa *duplicador de comandos* que cumpla los siguientes requisitos. R1: El programa debe leer dos comandos y sus respectivos argumentos. R2: el programa debe ejecutar ambos comandos **en paralelo** (esto es, que puedan estar en la cola *ready* al mismo tiempo), y luego esperar más comandos. R3: si algún comando termina de manera anormal (*exit code* distinto a 0), debe reejecutarlo solo una vez más, y luego imprimir los PID de aquellos que fallaron ambas veces. R4: el programa no debe terminar si uno de los procesos falla.

```
1  // suponga que todos los headers necesarios han sido incluidos
2  char *command1; char *command2;
3  char *arguments1; char *arguments2
4  int *exitCode; pid_t p1, p2;
5  // puede agregar mas variables aqui
6  while('r') {
7    // lee correctamente ambos comandos y argumentos
8    read_command(&command1, &arguments1, &command2, &arguments2);
9    exec(command1, arguments1);
10   if ( (p1 = fork()) > 0 ) { waitpid(p1, &exitCode); }
11   exec(command2, arguments2);
12   if ( (p2 = fork()) > 0 ) { waitpid(p2, &exitCode); }
13 }
```

4.1) **[4p]** El fragmento de código presentado no cumple con todos los requisitos. Explique cuáles **NO** cumple y por qué.

R. No cumple R2. No ejecuta ambos comandos en paralelo. Después que ejecuta el primer `exec` (línea 9), el programa no continúa, pues se "convierte" al ejecutable de `command1`.

R3, R4: No cumple R3 ni R4 porque no llega hasta la línea 10.

4.2) **[7p]** Modifique el código presentado para que cumpla todos los requisitos.

Esta es una solución posible:

```
1  // suponga que todos los headers necesarios han sido incluidos
2  char *command1; char *command2;
3  char *arguments1; char *arguments2
4  int *exitCode1; int *exitCode2;
5  pid_t p1, p2;
6  while('r') {
7    // lee correctamente ambos comandos y argumentos
8    read_command(&command1, &arguments1, &command2, &arguments2);
9    // esto cumple R2
10   p1 = fork(); // crea un hijo para ejecutar command1
11   if (p1 > 0) { // padre
12           p2 = fork(); // crea un hijo para ejecutar command2
13         if (p2 > 0) { // padre
14           // espera por ambos (en cualquier orden)
15       waitpid(p1, &exitCode1);
16       waitpid(p2, &exitCode2);
17     }
```

```
    else {
      // hijo para command2
      exec(command2, arguments2);
    }
  }
  else {
    // hijo para command1
    exec(command1, arguments1);
  }
  // esto cumple R3 y R4
  if(exitCode1 != 0){
    // crea otro hijo para ejecutar command1
    if( (p1 = fork()) > 0) { waitpid(p1, &exitCode1); }
    if( exitCode1 != 0) { printf("%d\n", p1); }
  }
  if(exitCode2 != 0){
    // crea otro hijo para ejecutar command2
    if( (p2 = fork()) > 0) { waitpid(p2, &exitCode2); }
    if( exitCode2 != 0) { printf("%d\n", p2); }
  }
}
```

NO es correcto ejecutar algo como `p1=fork(); p2=fork(); if(p1>0) ...`. Esto hace que ejecute dos veces `p2=fork();`.

NO es correcto hacer `p1=fork(); if(p1>0) waitpid(p1,&exitCode1); p2=fork();...`, porque esta secuencia espera que `p1` termine antes de crear a `p2` y por lo tanto no se cumple R2.

1p. Crear 2 hijos (y no más) en algún momento.

1p. Proceso principal no debe hacer `exec()`, solo los hijos. De lo contrario no se puede esperar al que hizo `exec()`, ni tampoco recuperar su `exitCode`.

2p. Ambos procesos deben poder estar creados simultáneamente.

1p. Obtener el `exitCode` de cada proceso usando `wait()`.

2p. Reejecutar una vez cada proceso que retorna `exitCode != 0`, e imprimir PID de los que fallaron dos veces.

4.3) **[4p]** Si eliminamos el requisito R3, ¿cómo podría modificar el código del ítem anterior para que los procesos creados quedaran en estado *zombie*? No es necesario que escriba el código, sino indicar las modificaciones (si sale más sencillo reescribirlo, reescríbalo). ¿Qué implicancias tendría esto para el sistema operativo?

R.

2p. Si ya no es importante recuperar el `exitCode` de los hijos, el proceso padre podría **no** ejecutar `wait()`, y continuar inmediatamente con la siguiente iteración hasta el próximo `readCommand()`. De esta manera, cuando los procesos hijos terminan, ellos ejecutarán `exit()` pero no su `exitCode` no será leído por el padre.

Podría quedar de esta manera:

```
// suponga que todos los headers necesarios han sido incluidos
char *command1; char *command2;
char *arguments1; char *arguments2
pid_t p1, p2;
while('r') {
  // lee correctamente ambos comandos y argumentos
  read_command(&command1, &arguments1, &command2, &arguments2);
  // esto cumple R2
```

```
  p1 = fork(); // crea un hijo para ejecutar command1
  if (p1 > 0) { // padre
          p2 = fork(); // crea un hijo para ejecutar command2
        if (p2 > 0) { // padre
          // ya no necesita esperar por nadie
    }
    else {
      // hijo para command2
      exec(command2, arguments2);
    }
  }
  else {
    // hijo para command1
    exec(command1, arguments1);
  }
  // ya no es necesario verificar los exitCode
}
```

2p. La implicancia que tiene para el sistema operativo es que en cada iteración se acumularán procesos *zombie*, lo que significa mantener todos los PCBs en la tabla de procesos, y ocupando algún espacio de memoria del sistema operativo. En cualquier caso, la memoria asignada al proceso sí se borra, así que eso no es un problema.

API de procesos

- `pid_t fork()` retorna 0, en el contexto del hijo; retorn *pid* del hijo, en el contexto del padre.
- `int exec(char *command, char *argumentos)` recibe como parámetro un *string* con la ruta del archivo a ejecutar y sus argumentos. Si hay error retorna -1. De lo contario, no retorna.
- `pid_t wait(pid_t p, int *exitStatus)` espera por el proceso `p`, y guarda el estado de salida de `p` en `exitStatus`. Si `p` es -1, espera por cualquiera. Retorna el `pid` del proceso que hizo `exit`.

Situación A

```
top - 09:18:43 up 27 days, 17:12,  5 users,  load average: 0.77, 0.60, 0.64
Tasks: 164 total,   5 running, 159 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st

  PID USER      PR    VIRT    RES    SHR S  %CPU %MEM     TIME+  PPID COMMAND
25927 cruz      20    6520    724    644 R 100.0  0.0   0:10.69 25587 tom
25928 cruz      20    6520    756    676 R 100.0  0.0   0:10.69 25587 tom
25930 jlopez    20    6520    680    600 R 100.0  0.0   0:07.47 25879 tom
25929 jlopez    20    6520    720    636 R  99.7  0.0   0:07.45 25879 tom
    1 root      20   37644   5736   4008 S   0.0  0.1   0:56.78     0 systemd
    2 root      20       0      0      0 S   0.0  0.0   0:00.50     0 kthreadd
25586 cruz      20   95480   3352   2316 S   0.0  0.0   0:00.10 25504 sshd
25587 cruz      20   22580   5444   3560 S   0.0  0.1   0:00.20 25586 bash
25862 cruz      20   22572   5192   3324 S   0.0  0.1   0:00.11 25861 bash
25877 root      20   52704   3928   3464 S   0.0  0.0   0:00.00 25862 sudo
25878 root      20   52284   3552   3136 S   0.0  0.0   0:00.00 25877 su
25879 jlopez    20   22616   5424   3504 S   0.0  0.1   0:00.12 25878 bash
25911 cruz      20    4356    688    612 S   0.0  0.0   0:00.00 25687 jerry
25912 cruz      20    4356     84      0 S   0.0  0.0   0:00.00 25911 jerry
```

Situación B

```
top - 09:21:51 up 27 days, 17:16,  5 users,  load average: 4.84, 2.45, 1.36
Tasks: 166 total,   7 running, 159 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st

  PID USER      PR    VIRT    RES    SHR S  %CPU %MEM     TIME+  PPID COMMAND
25933 jheysen   20    6520    756    672 R  72.1  0.0   0:25.98 25815 tom
25927 cruz      20    6520    724    644 R  65.4  0.0   3:05.27 25587 tom
25928 cruz      20    6520    756    676 R  65.4  0.0   3:06.59 25587 tom
25932 jheysen   20    6520    720    636 R  61.1  0.0   0:23.44 25815 tom
25929 jlopez    20    6520    720    636 R  58.5  0.0   3:00.88 25879 tom
    1 root      20   37644   5736   4008 S   0.0  0.1   0:56.78     0 systemd
    2 root      20       0      0      0 S   0.0  0.0   0:00.50     0 kthreadd
    3 root      20       0      0      0 S   0.0  0.0   0:00.70     2 ksoftirqd/0
25800 cruz      20   95404   3236   2300 S   0.0  0.0   0:00.03 25769 sshd
25801 cruz      20   22572   5256   3384 S   0.0  0.1   0:00.10 25800 bash
25813 root      20   52932   3956   3380 S   0.0  0.0   0:00.02 25801 sudo
25814 root      20   52284   3508   3088 S   0.0  0.0   0:00.00 25813 su
25815 jheysen   20   22672   5260   3316 S   0.0  0.1   0:00.12 25814 bash
25830 root      20   95480   7132   6096 S   0.0  0.1   0:00.04  1024 sshd
25861 cruz      20   95480   3436   2400 S   0.0  0.0   0:00.04 25830 sshd
25862 cruz      20   22572   5192   3324 S   0.0  0.1   0:00.11 25861 bash
25877 root      20   52704   3928   3464 S   0.0  0.0   0:00.00 25862 sudo
25878 root      20   52284   3552   3136 S   0.0  0.0   0:00.00 25877 su
25879 jlopez    20   22616   5424   3504 S   0.0  0.1   0:00.12 25878 bash
25911 cruz      20    4356    688    612 S   0.0  0.0   0:00.00 25687 jerry
25912 cruz      20    4356     84      0 S   0.0  0.0   0:00.00 25911 jerry
25917 cruz      20   41800   3736   3104 R   0.0  0.0   0:02.24 25643 top
```

Situación C

```
top - 09:28:59 up 27 days, 17:23,  5 users,  load average: 4.99, 4.92, 3.00
Tasks: 162 total,   3 running, 159 sleeping,   0 stopped,   0 zombie
%Cpu(s): 50.0 us,  0.0 sy,  0.0 ni, 50.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st

  PID USER      PR    VIRT    RES    SHR S  %CPU %MEM     TIME+  PPID COMMAND
25927 cruz      20    6520    724    644 R 100.0  0.0   7:56.72 25587 tom
25928 cruz      20    6520    756    676 R 100.0  0.0   7:56.85 25587 tom
    1 root      20   37644   5736   4008 S   0.0  0.1   0:56.78     0 systemd
    2 root      20       0      0      0 S   0.0  0.0   0:00.50     0 kthreadd
25504 root      20   95480   6896   5860 S   0.0  0.1   0:00.05  1024 sshd
25506 cruz      20   45248   4744   4044 S   0.0  0.1   0:00.03     1 systemd
25586 cruz      20   95480   3352   2316 S   0.0  0.0   0:00.16 25504 sshd
25587 cruz      20   22580   5444   3560 S   0.0  0.1   0:00.20 25586 bash
25611 root      20   95404   6868   5924 S   0.0  0.1   0:00.03  1024 sshd
25642 cruz      20   95404   3256   2316 S   0.0  0.0   0:00.77 25611 sshd
25643 cruz      20   22572   5188   3316 S   0.0  0.1   0:00.10 25642 bash
25655 root      20   95404   6808   5868 S   0.0  0.1   0:00.02  1024 sshd
25686 cruz      20   95404   3116   2180 S   0.0  0.0   0:00.00 25655 sshd
25687 cruz      20   22572   5124   3252 S   0.0  0.1   0:00.10 25686 bash
25911 cruz      20    4356    688    612 S   0.0  0.0   0:00.00 25687 jerry
25912 cruz      20    4356     84      0 S   0.0  0.0   0:00.00 25911 jerry
25917 cruz      20   41800   3736   3104 R   0.0  0.0   0:03.65 25643 top
```