Administrador de Procesos

Práctica

Comandos

System call

Inspección / Monitoreo

Course

Operating System (with focus on Security)

Instructor

Acosta Bermejo Raúl

Lecture notes

Table of contents (outline)

Tabla de contenido

1. Introducción
2. System call
3. Comandos

Administrador de Procesos

Comandos

Comandos

Inspección

Comando ps

Report a snapshot of the current processes

- The ps utility displays a header line, followed by lines containing information about all of your processes that have controlling terminals.
- A different set of processes can be selected for display by using any combination of others (-a, -G, -g, -p, -T, -t, -U, -u) options.

$ ps -fea

-f full (en los campos),

-e extended (todos los procesos)

-a (similar a -e y -a)

```
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.1  0.2   4452  2568 ?        Ss   13:41   0:01 /sbin/init
root          2  0.0  0.0      0     0 ?        S    13:41   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        S    13:41   0:00 [ksoftirqd/0]
root          5  0.0  0.0      0     0 ?        S<   13:41   0:00 [kworker/0:0H]
root          7  0.0  0.0      0     0 ?        S    13:41   0:00 [rcu_sched]
root          8  0.0  0.0      0     0 ?        S    13:41   0:00 [rcu_bh]
root          9  0.0  0.0      0     0 ?        S    13:41   0:00 [migration/0]
root         10  0.0  0.0      0     0 ?        S    13:41   0:00 [watchdog/0]
root         11  0.0  0.0      0     0 ?        S<   13:41   0:00 [khelper]
root         12  0.0  0.0      0     0 ?        S    13:41   0:00 [kdevtmpfs]
root         13  0.0  0.0      0     0 ?        S<   13:41   0:00 [netns]
root         14  0.0  0.0      0     0 ?        S<   13:41   0:00 [writeback]
root         15  0.0  0.0      0     0 ?        S<   13:41   0:00 [kintegrityd]
root         16  0.0  0.0      0     0 ?        S<   13:41   0:00 [bioset]
root         17  0.0  0.0      0     0 ?        S<   13:41   0:00 [kworker/u3:0]
root         18  0.0  0.0      0     0 ?        S<   13:41   0:00 [kblockd]
root         19  0.0  0.0      0     0 ?        S<   13:41   0:00 [ata_sff]
root         20  0.0  0.0      0     0 ?        S    13:41   0:00 [khubd]
root         21  0.0  0.0      0     0 ?        S<   13:41   0:00 [md]
```

Comandos

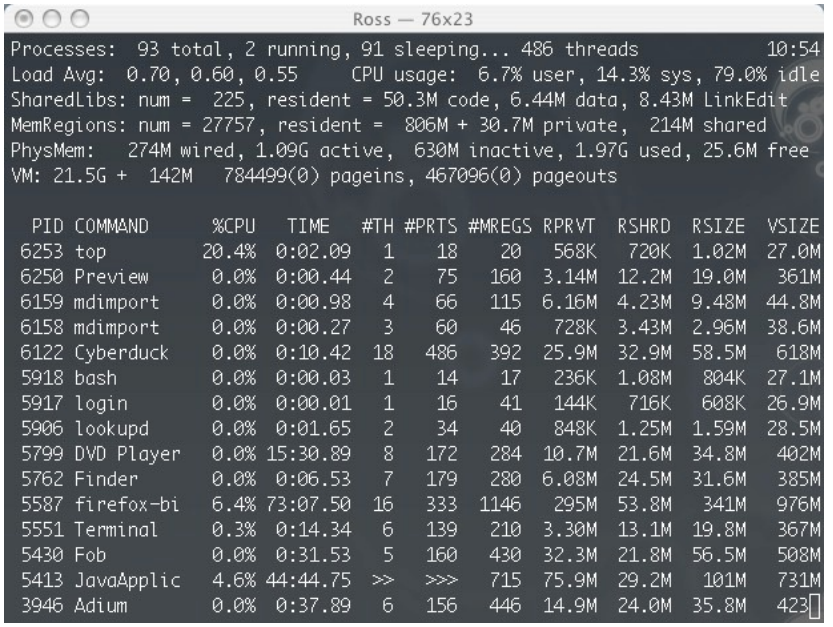
Inspección

Comando top

Display and update (dynamic real-time view) sorted information about processes.

- The default sorting key is pid.
- With the option ‘m’ change different views of memory.

$ top



Comandos

Inspección

Comando nice

It runs utility at an altered scheduling priority.

- If an increment is given, it is used; otherwise an increment of 10 is assumed.
- The super-user can run utilities with priorities higher than normal by using a negative increment.
- The priority can be adjusted over a range of -20 (the highest) to 20 (the lowest).

$ nice –n increment pid

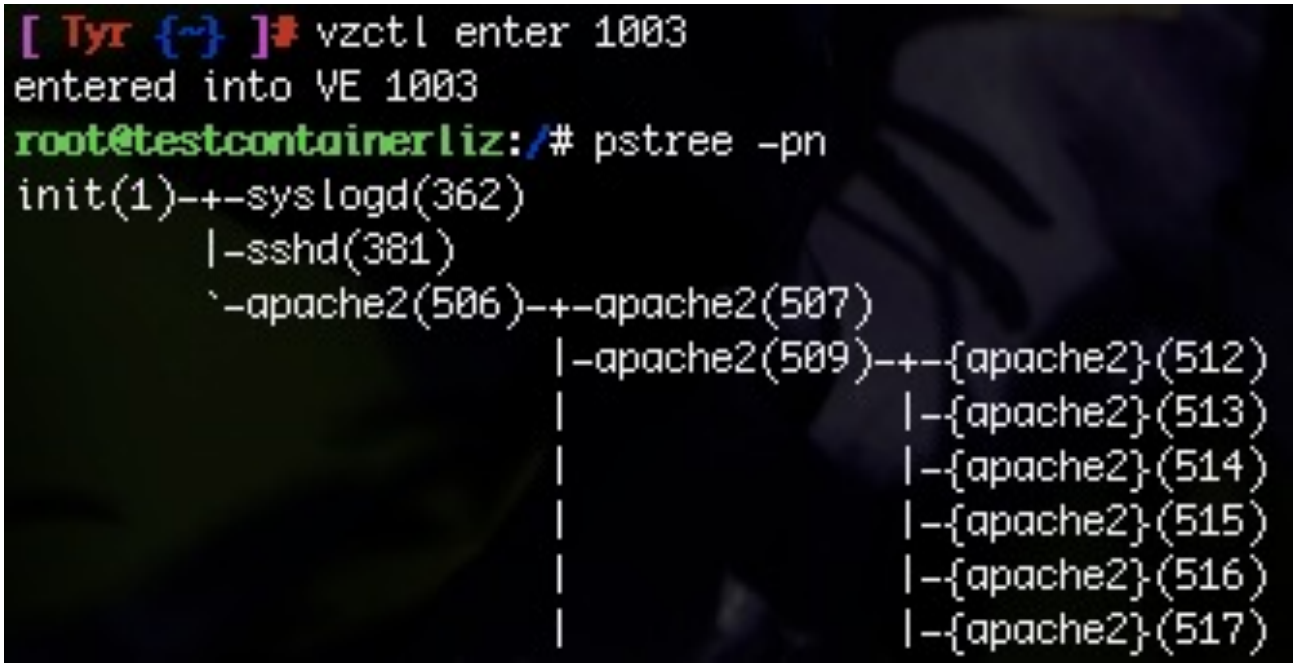
Comandos

Inspección

Comando pstree

Display a tree of process.

$ pstree



Comandos

Inspección

Comando pstree

Implementation on Linux

1. Read the /proc directory.
2. Use the syscall for File System.
   - Opendir
   - readdir
3. Build the tree (snapshot).

Links

- En lenguaje C procesando la salida del comando ps
  - http://www.thp.uni-duisburg.de/pstree/pstree.c
- En lenguaje C usando syscalls.
  - https://github.com/posborne/linux-programming-interface-exercises/blob/master/12-system-and-process-information/pstree.c

Comandos

Inspección

Comando kill

It terminates or signals a process.

- Only the super-user may send signals to other users' processes.

$ kill -9 pid # Envia la señal 9 (matar un proceso)

$ kill –l # Muestra todas las señales

Administrador de Procesos

Llamadas al sistema

Llamadas al sistema

fork

Syscall

```
pid_t fork();                    #include <sys/types.h>
```

Hace que el proceso actual se **duplique (clonación)**. Los procesos ocupan zonas de memoria diferentes pero con la misma información (misma variables, invocaciones de funciones , archivos abiertos, etc). La única diferencia entre padre e hijo es el valor regresado :

```
pid_t pid;

   pid = fork();
   switch(pid){
     case -1: Error de creación
     case  0: Código del hijo
     default: Código del padre
   }
```

Ambos procesos ejecutan la instrucción que se encuentra después del fork.

Llamadas al sistema

fork

Syscall

```
pid_t getpid(void);
```

```
#include <sys/types.h>
```

Regresa el identificador del proceso que realiza la llamada.

```
pid_t pid=getpid(void);
printf(”PID=%d \n", pid);
```

Syscall

```
pid_t getppid();
```

Regresa el identificador del padre del proceso que realiza la llamada, 1 si ya no existe.

Un programa que se puede utilizar para detectar cuando termina el padre es el siguiente:

```
while( getppid() ¡= 1 )
   sleep(1);
```

Cual es el problema de este código?

Se dice que es un programa que hace *polling*.

Llamadas al sistema

fork

Syscall

```
unsigned int sleep(unsigned int);
```

Duerme por un número de segundos dado como parámetro.
Tambien se puede despertar si llega una **señal** que no sea ignorada.

Otras opciones

- int usleep(useconds_t usec) Older POSIX and deprecated in many systems.
- int nanosleep(const struct timespec *req, struct timespec *rem);

```
struct timespec {
  time_t tv_sec; /* seconds */
  long tv_nsec; /* nanoseconds */
};
```

Llamadas al sistema

fork

```
for(i=1; i<=3; i++){
     pid = fork();
     switch( pid ){
        case -1 : //ERROR
                  printf("Error\n");
                  return(1);
        case 0 :  //HIJO
                  for(k=MAX; k>=1; k--){
                    printf("H=%d \n", getpid());
                  }
                  // Y si hay más tipos de hijos?
                  // se requiere otra lógica!
        default : //PADRE
                  …
     }
  }
```

El código de los hijos debe estar encapsulado y este no lo está. Hay que usar funciones y un exit.

Llamadas al sistema

fork

```
for(i=1; i<=3; i++){
     pid = fork();
     switch( pid ){
        case -1 : //ERROR
                  printf("Error\n");
                  return(1);
        case 0 :
                  hijo();
                  exit 1;
        default :
                  padre();
     }
}

hijo()
{
}
```

No se ejecuta más código

Llamadas al sistema

fork

Creación de procesos en el shell

```
$ comando
$ echo $?
```

El shell crea un proceso y captura el valor regresado .

Leer la variable del shell que contiene el valor de retorno.

```
int main (int argc, char **argv)
{
   printf("argc= %d \n", argc);
   if( argc==1 )
      return 0;
   else
      return argc;
}
```

Tipos de Shell

Bourn Shell, Korn Shell, C Shell, etc

Comandos bg, fg, jobs

```
$ comando <CTL-Z>     El proceso se suspende y regresa un Id.
$ bg <ENTER>    Pasa el proceso suspendido en background (sin terminal)
$ fg #                  Pasa un proceso a foreground
$ comando &             Ejecuta un proceso en background
```

Llamadas al sistema

fork

Cambio de contexto

```
main(){
for(i=1; i<=N; i++)
     pid = fork(),
     switch(pid){
               proceso(i)
     }
}

proceso(unsigned pid)
{
 unsigned i=1;

   while(1){
      for(t=0; t<=pid; t++)
         printf("\t");
      printf("<%d,%d>\n", getpid(), i++);
   }
}
```

Crear **N** procesos

<43,1>
<43,2>
<43,3>
<43,4>

Cambio de contexto

<98,1>
<98,2>
<98,3>
<98,4>
<98,5>
<98,6>
<98,7>

<43,6>
<43,7>
…

<103,1>
<103,2>

Tarea: Investigar el tiempo de un Quantum en algun SO o como medirlo.

Llamadas al sistema

fork

Problema al crear procesos

```
for(i=1; i<=3; i++){
    if( fork() == -1 ){
       printf("Error");
    }
  }
```

Cuantos procesos se crean?

Encontrar la respuesta haciendo una corrida a mano.

Sugerencia: al mísmo tiempo hacer un árbol.

Padre 1 i == 1
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){ ≠0
        Tratamiento del error;
    }
}
```

Hijo 2 i == 1
```
for(i=1; i<=3; i++){
 if( fork() == -1 ){ =0
        Tratamiento del error;
   }
}
```

Padre i == 2
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){
     Tratamiento del error;
  }
}
```

Hijo 2 i == 2
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){
     Tratamiento del error;
  }
}
```

Padre i == 2
```
for(i=1; i<=3; i++){
 if( fork() == -1 ){≠0
     Tratamiento del error;
   }
}
```

Hijo 2 i == 2
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){≠0
     Tratamiento del error;
   }
}
```

Hijo 3 i == 2
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){=0
     Tratamiento del error;
   }
}
```

Hijo 4 i == 2
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){=0
     Tratamiento del error;
   }
}
```

Padre i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){
     Tratamiento del error;
  }
}
```

Hijo 2 i == 3
```
for(i=1; i<=3; i++){
   if( fork() == -1 ){≠0
     Tratamiento del error;
   }
}
```

Hijo 3 i == 3
```
for(i=1; i<=3; i++){
   if( fork() == -1 ){
     Tratamiento del error;
   }
}
```

Hijo 4 i == 3
```
for(i=1; i<=3; i++){
   if( fork() == -1 ){
     Tratamiento del error;
   }
}
```

Padre i == 3
```
for(i=1; i<=3; i++){
 if( fork() == -1 ){≠0
     Tratamiento del error;
   }
}
```

Hijo 2 i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){≠0
     Tratamiento del error;
   }
}
```

Hijo 3 i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){≠0
     Tratamiento del error;
   }
}
```

Hijo 4 i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){≠0
     Tratamiento del error;
   }
}
```

Hijo 5 i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){ =0
     Tratamiento del error;
   }
}
```

Hijo 6 i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){ =0
     Tratamiento del error;
   }
}
```

Hijo 7 i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){ =0
     Tratamiento del error;
   }
}
```

Hijo 8 i == 3
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){ =0
     Tratamiento del error;
   }
}
```

Padre i == 4
```
for(i=1; i<=3; i++){
  if( fork() == -1 ){
     Tratamiento del error;
   }
}
```

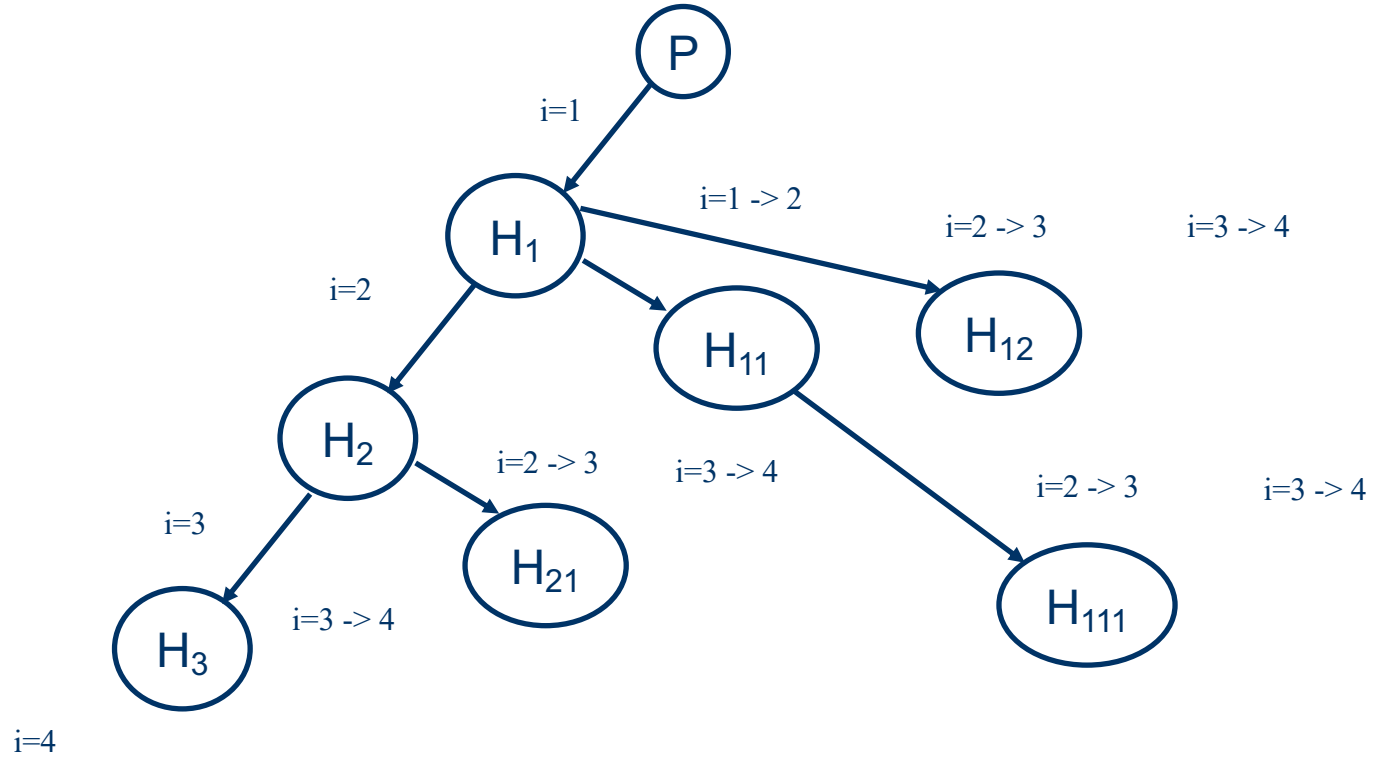
... Terminan todos

Suponiendo que no hay errores y los procesos se ejecutan en el orden de creación

Llamadas al sistema

fork

El árbol de procesos creado es:



Llamadas al sistema

fork

Syscall

Otra syscall similares a fork son:

- vfork()
  - It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created.
  - vfork() differs from fork in that the calling thread is suspended until the child terminates.
  - It is a special case of clone.
- clone()
  - It allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.
  - It exists only in Linux.

Llamadas al sistema

Sincronización de procesos

Syscall

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

```
#include <sys/types.h>
#include <sys/wait.h>
```

Espera que termine el primero de los hijos creados. Implementación con señales.

Función	pid	opciones	rusage	POSIX.1	SVR4	4.3 BSD
wait				•	•	•
waitpid	•	•		•	•	•
wait3		•	•		•	•
wait4	•	•	•			•

rusage: contiene estadísticas de los recursos que uso el proceso.

opciones: se dan mediante bits y macros, ejemplo WNOHANG el proceso que hace la invocación no se bloquea cuando no hay procesos hijos.

Llamadas al sistema

Sincronización con el padre

Syscall

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ...);
```

Crea un proceso y remplaza al que lo crea.

Función	pathname	filename	arglist	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•

Llamadas al sistema

Límites en el sistema

Comando ulimit

This programs allow to limit system-wide resource use.
$ ulimit –a

Este comando crea un número infinito de procesos y si no se tiene limite en el SO bloquea al sistema (bash forkbomb):

$:(){ :|:& };:

Pruebe el comando limitando los procesos y trate de crear más:

$ ulimit –u 30
$ ulimit -a

Llamadas al sistema

Límites en el sistema

Comando sysctl

- Another similar command to ulimit.

```
$ sysctl -a
```

Lee todas las variables del sistema, por ejemplo, en mi Mac:

- Listo 1141 lineas
- Las agrupo por tipos:
  - **user**.stream_max
  - **kern**.maxfiles
  - **vm**.loadavg
  - **vfs**.generic.maxtypenum
  - net, debug, hw, …

```
$ sysctl user # read a particular variable or family
```

- Por default se tienen ciertos valores de las variables y se pueden configurar en: /etc/sysctl.conf

Llamadas al sistema

Sincronización con el padre

Como siempre estos comandos son implementados mediante uno o varios syscall, en particular por:

Syscall

```
#include <unistd.h>
          <linux/sysctl.h>

int _sysctl(struct _sysctl_args *args);
```

- Read and write system parameters (kernel).
- This call does a search in a tree structure, possibly resembling a directory tree under /proc/sys, and if the requested item is found calls some appropriate routine to read or modify the value.
- On Linux, it is mandatory to use Procfs. A virtual file system for process.

Llamadas al sistema

Resumen

Syscall

1. fork: Crea un proceso. Hace que el proceso actual se duplique.
2. vfork, clone, etc. Crean procesos controlando los datos.
3. getpid y getppid. Obtiene el identificador de un proceso (el padre).
4. sleep, nanosleep. Duerme un proceso por un tiempo.
5. wait: Espera que termine el primero de los hijos creados.
6. Familia execl: Crea un proceso y remplaza al que lo crea.
7. sysctl: Lee y modifica variables del kernel.

Fin

Contacto

racostab@ipn.mx
racosta@cic.ipn.mx

57-29-60-00
Ext. 56652