

Procesos en UNIX

LABORATORIO DE SISTEMAS DE OPERACIÓN I
(ci-3825)

Prof. Yudith Cardinale

Procesos en UNIX

- Un proceso en Unix es un programa en ejecución que tiene los siguientes atributos:
 - Área de código, Área de datos, Área de stack, Un identificador de proceso único (PID), Área de usuario: está en el kernel y consiste de la tabla de descriptores de archivos, información sobre el consumo de CPU, manejadores de señales, entre otros, Tabla de páginas.
- Cuando Unix comienza, el ``bootstrap program" inicia los primeros procesos, ``Sched" o ``Swapper" (PID = 0), ``init" (PID=1) y ``pagedaemon" (PID=2). A partir de ``init" se van creando el resto de los procesos.
- En Unix todo proceso, excepto ``init" tiene un padre. Para crear un nuevo proceso hijo, el proceso padre se duplica, en este caso tanto el padre como el hijo son iguales excepto en sus PIDs.
- Un proceso padre puede reemplazar el código del hijo por un código ejecutable.
- Cuando un proceso hijo termina, se lo anuncia al padre (signal). Es común que un proceso padre espere hasta que el hijo termine.

Procesos en UNIX

PROCESOS EN BACKGROUND Y EN FOREGROUND

- Un proceso en background corre concurrentemente con su ``shell" padre y no toma control del teclado. Para poner a ejecutar un proceso en background se usa el metacaracter &. Si una orden, pipeline, secuencia, secuencia de pipelines o grupo de órdenes es seguida por &, se crea un subshell para ejecutar tal orden como un proceso en background.
- Los procesos en background son muy útiles para realizar varias tareas simultáneamente, siempre que tales tareas no requieran entrada por teclado. En un ambiente de ventanas, es común correr en cada ventana un comando u otros comandos en background.
- Cuando un proceso en background se crea, el ``shell" muestra cierta información que puede ser necesaria para posterior manipulación.
\$> command &
[2] 21168
- Se puede redireccionar la salida de un proceso en background:
\$> command > file.txt &
- Un procesos en Foreground es el proceso con el que se puede interactuar y tiene control sobre teclado.

Procesos en UNIX

Órdenes del Shell para manipular procesos:

- ps: permite monitorear el estado de los procesos

```
$> ps -aelfc
```

Genera una lista de información sobre el estado de los procesos. Por defecto la salida está limitada a procesos creados por el ``shell" actual.

La opción -a permite incluir procesos que son propiedad de otros usuarios.

La opción -e restringe la salida sólo para procesos que están ejecutándose.

La opción -l produce una salida larga que incluye el propietario de cada proceso.

La opción -c imprime información referente al scheduler, como la prioridad.

La opción -f produce una salida con más información que la opción por defecto.

Información de salida por defecto y extendida de ps:

```
PID TT STAT TIME COMMAND
```

```
USER PID %CPU %MEM SZ RSS TT STAT START TIME COMMAND
```

STAT puede ser: R: running, T: suspendido, P: esperando por entrada de una página, D: en espera no interrumpible, tal como E/S de disco, S: durmiendo por corto período, menos de 20 segundos, L: durmiendo por largo período, más de 20 segundos, Z; proceso zombie.

Procesos en UNIX

Órdenes del Shell para manipular procesos:

- `kill [-signalid] pid`

Se usa para terminar procesos antes que terminen normalmente

`kill -9 21254`

`kill 0` : mata todos los procesos del shell.

`kill -l` : lista todos las posibles señales (1 a 36)

- `top`: lista los procesos que están en el sistema con más información que `ps`.

Procesos en UNIX

Llamadas al sistema para manipular procesos:

- `pid_t fork(void)`: causa que un proceso se duplique. El proceso hijo es casi exacto, excepto por el PID.
Si `fork()` es exitoso, retorna el PID del hijo al proceso padre y 0 al proceso hijo. Si falla retorna -1 al padre y el proceso hijo no se crea.
- `pid_t getpid(void)`, `pid_t getppid(void)`: retornan el PID del proceso y del proceso padre que realiza la llamada. Estas llamadas siempre tienen éxito.
- `void exit(int status)`: cierra todos los descriptores de archivo de un proceso, desasigna código, datos y stack y termina el proceso. Cuando un proceso hijo termina, envía a su padre un `SIGCHILD` y espera por que su código de status de terminación sea aceptado. `exit()` nunca retorna.

Procesos en UNIX

Llamadas al sistema para manipular procesos:

- `int wait(int *status)`: causa que un proceso se suspenda hasta que sus hijos terminen. Una llamada exitosa a `wait()` retorna el PID del hijo que terminó y coloca un código de status codificado como sigue:

Si el byte más a la derecha de status es 0, el byte más a la izquierda contiene los 8 bits más bajos del valor retornado por la salida (`exit()` o `return()`) del hijo.

Si el byte más a la derecha de status no es 0, los siete bits más a la derecha son iguales al número de la señal que causó que el hijo terminara, y el restante bit del byte más a la derecha es colocado en 1 si el hijo produjo un core dump.

Si un proceso ejecuta un `wait()` y no tiene hijos, se retorna inmediatamente -1. Si un proceso ejecuta un `wait()` y uno o más de sus hijos ya son zombies, se retorna el status de uno de los zombies.

Procesos en UNIX

Llamadas al sistema para manipular procesos:

- Familia exec():

```
int execl(char* path, char* arg0, char* arg1,...,char* argn,NULL)
```

```
int execv(char* path,char* argv[])
```

```
int execlp(char* path, char* arg0, char* arg1,...,char* argn,NULL)
```

```
int execvp(char* path,char* argv[])
```

La familia exec() reemplaza el código, datos y stack del proceso llamante por uno ejecutable que se encuentra en path.

execl() y execlp son idénticas, así como execv() y execvp() también son idénticas. Excepto que execl() y execv() requieren el pathname absoluto o relativo del ejecutable, mientras que los otros dos usan la variable de ambiente \$PATH para encontrar el path.

Si el archivo ejecutable no es encontrado se retorna -1. Un exec() exitoso no retorna.

Procesos en UNIX

Llamadas al sistema para manipular procesos:

- `int chdir(char* pathname)`, se usa para cambiar de directorio. Retorna 0 si tiene éxito o -1 en caso de error.
- `int nice(int delta)`, se usa para cambiar la prioridad a los procesos. Los procesos hijos heredan la prioridad del padre. Retorna 0 si tiene éxito o -1 en caso de error (un usuario común quiere obtener una prioridad negativa).
- `uid_t getuid()`: retorna el id del usuario real.
- `uid_t geteuid()`: retorna el id del usuario efectivo.
- `gid_t getgid()`: retorna el id del grupo real.
- `gid_t getegid()`: retorna el id del grupo efectivo.
Estas llamadas siempre son exitosas.
- `int setuid(uid_t id)`: coloca el id del usuario real.
- `int seteuid(uid_t id)`: coloca el id del usuario efectivo.
- `int setgid(gid_t id)`: coloca el id del grupo real.
- `int setegid(gid_t id)`: coloca el id del grupo efectivo.
Sólo pueden ser usadas por el superusuario. Retornan 0 si son exitosas o -1 en caso de error.