Capítulo 3. Procesos

Dado un proceso P1. Es un espacio en memoria que tiene un descriptor asignado y este indica que recursos tiene asignado ese proceso a traves del S.O

Proceso

¿Que hay en la memoria de un proceso?

CODIGO , STACK , HEAD, CONSTANTES Y VARIABLES GLOBALES

Una instancia en ejecución de un programa se llama *proceso*. Si tiene dos ventanas de terminal en la pantalla, probablemente esté ejecutando el mismo programa de terminal dos veces: tiene dos procesos de terminal. Es probable que cada ventana de terminal esté ejecutando un shell; cada shell en ejecución es otro proceso. Cuando invoca un comando desde un shell, el programa correspondiente se ejecuta en un nuevo proceso; el proceso de shell se reanuda cuando ese proceso se completa.

Los programadores avanzados a menudo usan múltiples procesos de cooperación en una sola aplicación para permitir que la aplicación haga más de una cosa a la vez, para aumentar la solidez de la aplicación y para hacer uso de programas ya existentes.

La mayoría de las funciones de manipulación de procesos descritas en este capítulo son similares a las de otros sistemas UNIX. La mayoría se declara en el archivo de encabezado < unistd.h> ; compruebe la página de manual de cada función para estar seguro.

3.1 Observación de procesos

Incluso mientras se sienta frente a su computadora, hay procesos en ejecución. Cada programa en ejecución utiliza uno o más procesos. Comencemos por echar un vistazo a los procesos que ya están en su computadora.

3.1.1 ID de proceso

Cada proceso en un sistema Linux se identifica por su única *identificacion de proceso,* a veces referido como *pid.* Los ID de proceso son números de 16 bits que Linux asigna secuencialmente a medida que se crean nuevos procesos.

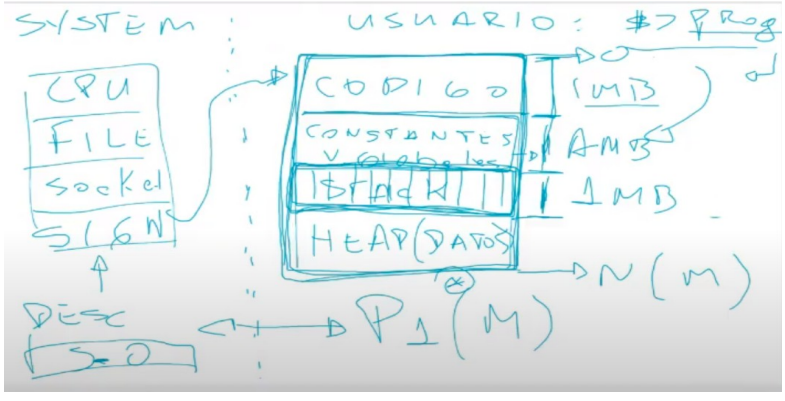
Cada proceso también tiene un proceso padre (excepto el especial init proceso, descrito en Sección 3.4.3 , "Procesos Zombie"). Por lo tanto, puede pensar en los procesos de un sistema Linux como organizados en un árbol, con la en eso proceso en su raíz. los *ID del proceso principal,* o *ppid,* es simplemente el ID de proceso del padre del proceso.

Al hacer referencia a los ID de proceso en un programa C o C ++, utilice siempre el pid_t typedef, que se define en < sys / types.h> . Un programa puede obtener el ID de proceso del proceso en el que se está ejecutando con el getpid () llamada al sistema, y puede obtener el ID de proceso de su proceso padre con la getppid () llamada al sistema. Por ejemplo, el programa en Listado 3.1 imprime su ID de proceso y su ID de proceso principal.

Listado 3.1 (print-pid.c) Impresión del ID del proceso

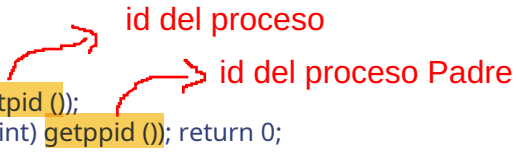
```
# incluye <stdio.h>
# incluye <unistd.h>

int main ()
```



```
{
    printf ("El ID del proceso es% d \ n", (int) getpid ());
    printf ("El ID del proceso padre es% d \ n", (int) getppid ()); return 0;

}
```



Observe que si invoca este programa varias veces, se informa un ID de proceso diferente porque cada invocación está en un proceso nuevo. Sin embargo, si lo invoca cada vez desde el mismo shell, el ID del proceso principal (es decir, el ID del proceso del shell) es el mismo.

3.1.2 Visualización de procesos activos

los ps comando muestra los procesos que se están ejecutando en su sistema. La versión GNU / Linux de ps tiene muchas opciones porque intenta ser compatible con versiones de ps en varias otras variantes de UNIX. Estas opciones controlan qué procesos se enumeran y qué información se muestra sobre cada uno.

Por defecto, invocando ps muestra los procesos controlados por la terminal o ventana de terminal en la que ps se invoca. Por ejemplo:

```
% ps
PID TTY HORA CMD
21693 pts / 8 00:00:00 bash
21694 pts / 8 00:00:00 ps
```

Esta invocación de ps muestra dos procesos. El primero, intento , es el shell que se ejecuta en este terminal. El segundo es la instancia en ejecución del PD programa en sí. La primera columna, etiquetada PID , muestra el ID de proceso de cada uno.

Para obtener una visión más detallada de lo que se está ejecutando en su sistema GNU / Linux, invoque esto:

```
% ps -e -o pid, ppid, comando
```

los - mi la opción instruye PD para mostrar todos los procesos que se ejecutan en el sistema. los - o pid, ppid, comando la opción dice PD qué información mostrar sobre cada proceso; en este caso, el ID del proceso, el ID del proceso principal y el comando que se ejecuta en este proceso.

Formatos de salida ps

Con el - o opción a la PD comando, especifica la información sobre los procesos que desea en la salida como una lista separada por comas. Por ejemplo, ps -o pid, usuario, hora_inicio, comando muestra el ID del proceso, el nombre del usuario que posee el proceso, la hora del reloj de pared a la que se inició el proceso y el comando que se ejecuta en el proceso. Consulte la página de manual para PD para obtener la lista completa de códigos de campo. Puedes usar el - F (listado completo), - l (lista larga), o - j (lista de trabajos) opciones en su lugar para obtener tres formatos de lista preestablecidos diferentes.

Aquí están las primeras líneas y las últimas líneas de salida de este comando en mi sistema. Es posible que vea una salida diferente, dependiendo de lo que se esté ejecutando en su sistema.

```
% ps -e -o pid, ppid, comando
   COMANDO PID PPID
      1 0 inicialización [5]
      2 1 [kflushd]
```

```
      3          1 [kupdate]
. . .
21725 21693 xterm
21727 21725 bash
21728 21727 ps -e -o pid, ppid, comando
```

Tenga en cuenta que el ID de proceso principal del PD comando, 21727, es el ID de proceso de intento , el caparazón desde el que invoqué PD . El ID de proceso principal de intento es a su vez 21725, el ID de proceso del xterm programa en el que se ejecuta el shell.

3.1.3 Eliminación de un proceso

Puede matar un proceso en ejecución con el kill mando. Simplemente especifique en la línea de comando el ID de proceso del proceso que se eliminará.

Cada vez que el proceso recibe una señal se guarda en el descriptor

los kill El comando funciona enviando el proceso a SIGTERM , o terminación, señal. [1] Esto hace que el proceso termine, a menos que el programa en ejecución maneje o enmascare explícitamente la SIGTERM señal. Las señales se describen en Sección 3.3 , "Señales".

[1] También puede utilizar el matar comando para enviar otras señales a un proceso. Esto se describe en Sección 3.4 , "Terminación del proceso".

3.2 Creación de procesos

Se utilizan dos técnicas comunes para crear un nuevo proceso. El primero es relativamente simple pero debe usarse con moderación porque es ineficiente y tiene considerables riesgos de seguridad. La segunda técnica es más compleja pero proporciona mayor flexibilidad, velocidad y seguridad.

3.2.4 Uso *system*

los system La función en la biblioteca C estándar proporciona una manera fácil de ejecutar un comando desde dentro de un programa, como si el comando se hubiera escrito en un shell. De hecho, system crea un subproceso que ejecuta el shell Bourne estándar (/ bin / sh) y entrega el comando a ese shell para su ejecución. Por ejemplo, este programa en Listado 3.2 invoca el ls comando para mostrar el contenido del directorio raíz, como si escribiera ls -l / en un caparazón.

Listado 3.2 (system.c) Uso de la llamada al sistema

```
# incluye <stdlib.h>

int main ()
{
  int return_value;
  return_value = system ("ls -l /"); return
  return_value;
}
```

los sistema La función devuelve el estado de salida del comando de shell. Si el shell en sí no se puede ejecutar, sistema devuelve 127; si ocurre otro error, sistema devuelve -1.

Porque el sistema La función usa un shell para invocar su comando, está sujeta a las características, limitaciones y fallas de seguridad del shell del sistema. No puede confiar en la disponibilidad de ninguna versión particular del shell Bourne. En muchos sistemas UNIX, / bin / sh es un enlace simbólico a otro caparazón. Por ejemplo, en la mayoría de los sistemas GNU / Linux, / bin / sh puntos a intento (Bourne- Again SHell), y diferentes distribuciones GNU / Linux usan diferentes versiones de intento . Invocar un programa con privilegios de root con el sistema función, por ejemplo, puede tener diferentes resultados en

diferentes sistemas GNU / Linux. Por tanto, es preferible utilizar el tenedor y ejecutivo método de creación de procesos.

3.2.5 Uso *de fork* y *exec*

La API de DOS y Windows contiene la Aparecer familia de funciones. Estas funciones toman como argumento el nombre de un programa a ejecutar y crean una nueva instancia de proceso de ese programa. Linux no contiene una sola función que haga todo esto en un solo paso. En cambio, Linux proporciona una función, tenedor , que crea un proceso hijo que es una copia exacta de su proceso padre. Linux proporciona otro conjunto de funciones, el ejecutivo familia, que hace que un proceso en particular deje de ser una instancia de un programa y en su lugar se convierta en una instancia de otro programa. Para generar un nuevo proceso, primero usa tenedor para hacer una copia del proceso actual. Entonces usas ejecutivo para transformar uno de estos procesos en una instancia del programa que desea generar.

Calling fork

Cuando un programa llama fork , un proceso duplicado, llamado *proceso hijo,* es creado. El proceso padre continúa ejecutando el programa desde el punto en que fork fue llamado. El proceso hijo también ejecuta el mismo programa desde el mismo lugar.

Entonces, ¿en qué se diferencian los dos procesos? Primero, el proceso hijo es un proceso nuevo y, por lo tanto, tiene una nueva ID de proceso, distinta de la ID de proceso de su padre. Una forma de que un programa distinga si está en el proceso principal o en el proceso secundario es llamar getpid . sin embargo, el fork La función proporciona diferentes valores de retorno a los procesos padre e hijo: un proceso "entra" en el fork call, y dos procesos "salen", con diferentes valores de retorno. El valor de retorno en el proceso padre es el ID de proceso del hijo. El valor de retorno en el proceso hijo es cero. Debido a que ningún proceso tiene un ID de proceso de cero, esto facilita al programa si ahora se está ejecutando como proceso padre o hijo.

Listado 3.3 es un ejemplo de uso fork para duplicar el proceso de un programa. Tenga en cuenta que el primer bloque del if La sentencia se ejecuta solo en el proceso padre, mientras que else La cláusula se ejecuta en el proceso hijo.

Listado 3.3 (fork.c) Uso de fork para duplicar el proceso de un programa

```
# incluye <stdio.h>
# incluye <sys / types.h>
# incluye <unistd.h>

int main ()
{
  pid_t child_pid;

  printf ("el ID del proceso del programa principal es% d \ n", (int) getpid ());

  child_pid = fork ();
if (child_pid! = 0) {
      printf ("este es el proceso padre, con id% d \ n", (int) getpid ()); printf ("el ID del proceso del
      niño es% d \ n", (int) child_pid);
}
  demás
    printf ("este es el proceso hijo, con id% d \ n", (int) getpid ());

  return 0;
}
```

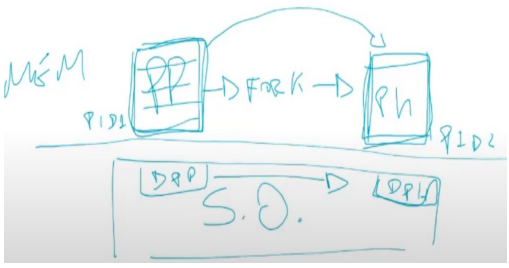
Referencia al id del proceso

indica que este proceso en particular se debe duplicar

Al proceso padre le retorna el id del proceso hijo y al proceso hijo le retorna el id 0

La función `fork` se utiliza para crear un nuevo proceso que represente la duplicación del proceso de la persona que llama. Tenga en cuenta que el proceso de llamada se denomina convencionalmente proceso padre y uno recién creado, proceso hijo. Aunque mencionamos anteriormente que el proceso hijo es un duplicado del padre, existen algunas diferencias, como que el proceso hijo tiene su propio PID único (los detalles completos sobre

Using the exec Family



Reemplaza al proceso actual con un nuevo programa, pero el siempre sabe quien es su papa por eldescriptor

El exec las funciones reemplazan el programa que se ejecuta en un proceso con otro programa. Cuando un programa llama a un exec función, ese proceso deja inmediatamente de ejecutar ese programa y comienza a ejecutar un nuevo programa desde el principio, asumiendo que el exec la llamada no encuentra un error.

Dentro de exec familia, hay funciones que varían ligeramente en sus capacidades y cómo se llaman.

- Funciones que contienen la letra *pag* en sus nombres execvp y execlp) aceptar un nombre de programa y buscar un programa por ese nombre en la ruta de ejecución actual; funciones que no contienen el *pag* Se debe proporcionar la ruta completa del programa que se ejecutará.

- Funciones que contienen la letra v en sus nombres execv , execvp , y ejecutivo) acepte la lista de argumentos para el nuevo programa como una matriz de punteros a cadenas terminada en NULL. Funciones que contienen la letra *l* (execl , execlp , y execle) Acepte la lista de argumentos usando el mecanismo de varargs del lenguaje C.

- Funciones que contienen la letra *mi* en sus nombres ejecutivo y execle) aceptar un argumento adicional, una matriz de variables de entorno. El argumento debe ser una matriz terminada en NULL de punteros a cadenas de caracteres. Cada cadena de caracteres debe tener la forma " VARIABLE = valor ".

Porque ejecutivo reemplaza el programa que llama por otro, nunca regresa a menos que ocurra un error.

La lista de argumentos que se pasa al programa es análoga a los argumentos de la línea de comandos que especifica a un programa cuando lo ejecuta desde el shell. Están disponibles a través del argc y argv parámetros a principal . Recuerde, cuando se invoca un programa desde el shell, el shell establece el primer elemento de la lista de argumentos. argv [0]) al nombre del programa, el segundo elemento de la lista de argumentos (argv [1]) al primer argumento de la línea de comandos, y así sucesivamente. Cuando usa un ejecutivo función en sus programas, usted también debe pasar el nombre de la función como el primer elemento de la lista de argumentos.

Usando fork y exec juntos

Un patrón común para ejecutar un subprograma dentro de un programa es primero bifurcar el proceso y luego ejecutar el subprograma. Esto permite que el programa de llamada continúe la ejecución en el proceso padre mientras que el programa de llamada es reemplazado por el subprograma en el proceso hijo.

El programa en Listado 3.4 , igual que Listado 3.2 , enumera el contenido del directorio raíz utilizando la ls mando. Sin embargo, a diferencia del ejemplo anterior, invoca la ls comando directamente, pasándole los argumentos de la línea de comandos - l y / en lugar de invocarlo a través de un caparazón.

Listing 3.4 (fork-exec.c) Using fork and exec Together

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

/* Spawn a child process running a new program. PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list. Returns  the process ID of
   the spawned process.  */

int spawn (char* program, char** arg_list)
```

```
{
  pid_t child_pid;

  /* Duplicate this process. */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process. */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program,  arg_list);
    /* The execvp  function returns only if an error occurs.  */
    fprintf (stderr,  "an error occurred in execvp\n");
    abort ();
  }
}

int main ()
{
  /*  The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };

  /* Spawn a child process running the "ls" command. Ignore the
     returned child process ID.  */
  spawn ("ls", arg_list);

  printf ("done with main program\n");

  return 0;
}
```

Duplica el proceso Padre

Estoy en el proceso padre

Reemplazo al programa padre por un nuevo programa

Para generar un nuevo hijo el padre necesita saber cual es el programa, los argumentos e invocarle a ese programa sin perder el padre

→ Genera un nuevo proceso hijo

3.2.6 Process Scheduling

Linux programa los procesos padre e hijo de forma independiente; no hay garantía de cuál se ejecutará primero, o cuánto tiempo se ejecutará antes de que Linux lo interrumpa y deje que el otro proceso (o algún otro proceso en el sistema) se ejecute. En particular, ninguno, parte o todos los ls El comando puede ejecutarse en el proceso hijo antes de que el padre se complete. [2] Linux promete que cada proceso se ejecutará eventualmente; ningún proceso se verá privado por completo de recursos de ejecución.

[2] Un método para serializar los dos procesos se presenta en Sección 3.4.1 , "Esperando la terminación del proceso".

Puede especificar que un proceso es menos importante, y debe recibir una prioridad más baja, asignándole una mayor *amabilidad* valor. Por defecto, cada proceso tiene una bondad de cero. Un valor de bondad más alto significa que al proceso se le da una prioridad de ejecución menor; a la inversa, un proceso con una amabilidad menor (es decir, negativa) obtiene más tiempo de ejecución.

Para ejecutar un programa con una bondad distinta de cero, utilice la bonito comando, especificando el valor de bondad con el - norte opción. Por ejemplo, así es como puede invocar el comando " ordenar input.txt> output.txt ", una operación de clasificación larga, con una prioridad reducida para que no ralentice demasiado el sistema:

```
% nice -n 10 sort input.txt > output.txt
```

Cambia la prioridad a un proceso para que no sobrepase al padre

You can use the renice command to change the niceness of a running process from the command line.

Para cambiar nuevamente prioridad

Para cambiar la bondad de un proceso en ejecución mediante programación, utilice el bonito función. Su argumento es un valor de incremento, que se suma al valor de bondad del proceso que lo llama. Recuerde que un valor positivo aumenta el valor de bondad y, por lo tanto, reduce la prioridad de ejecución del proceso.

Tenga en cuenta que solo un proceso con privilegios de root puede ejecutar un proceso con un valor de bondad negativo o reducir el valor de bondad de un proceso en ejecución. Esto significa que puede especificar valores negativos a la bonito y re bueno comandos solo cuando se inicia sesión como root, y solo un proceso que se ejecuta como root puede pasar un valor negativo al bonito función. Esto evita que los usuarios normales alejen la prioridad de ejecución de otros usuarios del sistema.

3.3 Singnals

Signals son mecanismos para comunicarse y manipular procesos en Linux. El tema de las señales es amplio; aquí discutimos algunas de las señales y técnicas más importantes que se utilizan para controlar procesos.

Una señal es un mensaje especial enviado a un proceso. Las señales son asincrónicas; cuando un proceso recibe una señal, procesa la señal inmediatamente, sin terminar la función actual o incluso la línea de código actual. Hay varias docenas de señales diferentes, cada una con un significado diferente. Cada tipo de señal se especifica por su número de señal, pero en los programas, normalmente se hace referencia a una señal por su nombre. En Linux, estos se definen en / usr / include / bits / signum.h . (No debe incluir este archivo de encabezado directamente en sus programas; en su lugar, usa < señal.h> .)

Cuando un proceso recibe una señal, puede hacer una de varias cosas, dependiendo de la señal *disposición.* Para cada señal, hay un *disposición predeterminada,* que determina qué sucede con el proceso si el programa no especifica algún otro comportamiento. Para la mayoría de los tipos de señales, un programa puede especificar algún otro comportamiento, ya sea para ignorar la señal o para llamar a un especial *manejador de señales* función para responder a la señal. Si se usa un manejador de señales, el programa que se está ejecutando actualmente se pausa, se ejecuta el manejador de señales y, cuando el manejador de señales regresa, el programa se reanuda.

El sistema Linux envía señales a los procesos en respuesta a condiciones específicas. Por ejemplo, SIGBUS (error de bus), SIGSEGV (violación de segmentación), y SIGFPE (excepción de punto flotante) pueden enviarse a un proceso que intenta realizar una operación ilegal. La disposición predeterminada para estos le indica que debe terminar el proceso y producir un archivo central.

Un proceso también puede enviar una señal a otro proceso. Un uso común de este mecanismo es finalizar otro proceso enviándole un SIGTERM o SIGKILL señal. [3] Otro uso común es enviar un comando a un programa en ejecución. Se reservan dos señales "definidas por el usuario" para este propósito: SIGUSR1 y SIGUSR2 . los SIGHUP La señal a veces también se usa para este propósito, comúnmente para activar un programa inactivo o hacer que un programa vuelva a leer sus archivos de configuración.

[3] a' The diferencia? los SIGTERM la señal le pide a un proceso que termine; el proceso puede ignorar la solicitud enmascarando o ignorando la señal. los SI w GRAMO h K La señal siempre mata el proceso inmediatamente porque el proceso no puede enmascarar o ignorar SIGKILL .

los sigaction La función se puede utilizar para establecer una disposición de señal. El primer parámetro es el número de señal. Los siguientes dos parámetros son indicadores de sigaction estructuras; el primero de ellos contiene la disposición deseada para ese número de señal, mientras que el segundo recibe la disposición anterior. El campo más importante en el primero o segundo sigaction la estructura es sa_handler . Puede tomar uno de tres valores:

- SIG_DFL , que especifica la disposición predeterminada para la señal.
- SIG_IGN , que especifica que la señal debe ignorarse.
- Un puntero a una función de manejador de señales. La función debe tomar un parámetro, el número de señal y devolver vacío .

Debido a que las señales son asíncronas, el programa principal puede estar en un estado muy frágil cuando se procesa una señal y, por lo tanto, mientras se ejecuta una función de manejo de señales. Por lo tanto, debe evitar realizar operaciones de E / S o llamar a la mayoría de las funciones de la biblioteca y del sistema desde los manejadores de señales.

Un manejador de señales debe realizar el trabajo mínimo necesario para responder a la señal y luego devolver el control al programa principal (o terminar el programa). En la mayoría de los casos, esto consiste simplemente en registrar el hecho de que ocurrió una señal. A continuación, el programa principal comprueba periódicamente si se ha producido una señal y reacciona en consecuencia.

Es posible que un gestor de señales sea interrumpido por la entrega de otra señal. Si bien esto puede parecer una ocurrencia poco común, si ocurre, será muy difícil diagnosticar y depurar el problema. (Este es un ejemplo de una condición de carrera, discutida en Capítulo 4 , "Hilos," Sección 4.4 , "Sincronización y secciones críticas"). Por lo tanto, debe tener mucho cuidado con lo que hace su programa en un manejador de señales.

Incluso asignar un valor a una variable global puede ser peligroso porque la asignación puede en realidad llevarse a cabo en dos o más instrucciones de máquina, y puede ocurrir una segunda señal entre ellas, dejando la variable en un estado corrupto. Si usa una variable global para marcar una señal de una función manejadora de señales, debe ser del tipo especial sig_atomic_t . Linux garantiza que las asignaciones a variables de este tipo se realizan en una sola instrucción y, por lo tanto, no se pueden interrumpir a la mitad. En Linux, sig_atomic_t es un ordinario En t ; de hecho, las asignaciones a tipos enteros del tamaño de En t o más pequeños, o apuntadores, son atómicos. Sin embargo, si desea escribir un programa que sea portátil a cualquier sistema UNIX estándar, use sig_atomic_t para estas variables globales.

Este programa esqueleto en Listado 3.5 , por ejemplo, usa una función de manejador de señales para contar el número de veces que el programa recibe SIGUSR1 , una de las señales reservadas para uso de la aplicación.

Listado 3.5 (sigusr1.c) Usando un manejador de señales

```
# incluye <señal.h>
# incluye <stdio.h>
# incluye <string.h>
# incluye <sys / types.h>
# incluye <unistd.h>

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
  ++sigusr1_count;
}

int  main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof  (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* Do some lengthy  stuff here. */
  /*  ... */
```

Para manejar una Señal

Dirección de memoria de la función

```
    printf ("SIGUSR1 se elevó% d veces \ n", sigusr1_count); return 0;
}
```

3.4 Terminación del proceso

Normalmente, un proceso termina de una de dos formas. O el programa en ejecución llama al Salida función, o el programa principal devuelve la función. Cada proceso tiene un código de salida: un número que el proceso devuelve a su padre. El código de salida es el argumento que se pasa al Salida función, o el valor devuelto por principal .

Un proceso también puede terminar de manera anormal, en respuesta a una señal. Por ejemplo, el SIGBUS , SIGSEGV , y SIGFPE las señales mencionadas anteriormente hacen que el proceso termine. Otras señales se utilizan para terminar un proceso de forma explícita. los SIGINT La señal se envía a un proceso cuando el usuario intenta finalizarlo escribiendo Ctrl + C en su terminal. los SIGTERM la señal es enviada por el matar mando. La disposición predeterminada para ambos es finalizar el proceso. Llamando al abortar función, un proceso se envía a sí mismo el SIGABRT signal, que finaliza el proceso y produce un archivo principal. La señal de terminación más potente es SIGKILL , que finaliza un proceso inmediatamente y no puede ser bloqueado ni manejado por un programa.

Cualquiera de estas señales se puede enviar utilizando el matar comando especificando una bandera de línea de comando adicional; por ejemplo, para finalizar un proceso problemático enviándole un SIGKILL , invocar lo siguiente, donde pid es su ID de proceso:

```
% kill -KILL pid
```

Para enviar una señal desde un programa, utilice el matar función. El primer parámetro es el ID del proceso de destino. El segundo parámetro es el número de señal; usar SIGTERM para simular el comportamiento predeterminado del matar mando. Por ejemplo, donde pid niño contiene el ID de proceso del proceso hijo, puede utilizar el matar función para terminar un proceso hijo del padre llamándolo así:

```
matar (child_pid, SIGTERM);
```

Incluir la < sys / types.h> y < señal.h> encabezados si usa el matar función.

Por convención, el código de salida se usa para indicar si el programa se ejecutó correctamente. Un código de salida de cero indica una ejecución correcta, mientras que un código de salida distinto de cero indica que ocurrió un error. En el último caso, el valor particular devuelto puede dar alguna indicación de la naturaleza del error. Es una buena idea ceñirse a esta convención en sus programas porque otros componentes del sistema GNU / Linux asumen este comportamiento. Por ejemplo, los shells asumen esta convención cuando conecta varios programas con el && (lógico y) y || operadores (lógicos o). Por lo tanto, debe devolver explícitamente cero de su principal función, a menos que se produzca un error.

Con la mayoría de los shells, es posible obtener el código de salida del programa ejecutado más recientemente usando la función especial PS variable. He aquí un ejemplo en el que ls El comando se invoca dos veces y su código de salida se muestra después de cada invocación. En el primer caso, ls se ejecuta correctamente y devuelve el código de salida cero. En el segundo caso, ls encuentra un error (porque el nombre de archivo especificado en la línea de comando no existe) y, por lo tanto, devuelve un código de salida distinto de cero.

```
% ls /
compartimiento  coda, etc.          lib                    misc nfs proc sbin usr
boot dev            casa perdida + encontrada mnt          opt root tmp var
% echo $?
```

```
0
% ls archivo falso
ls: bogusfile: ¿No existe ese archivo o directorio%
echo $?
1
```

Tenga en cuenta que aunque el tipo de parámetro del Salida la función es En t y el principal la función devuelve un En t , Linux no conserva los 32 bits completos del código de retorno. De hecho, debe usar códigos de salida solo entre cero y 127. Los códigos de salida superiores a 128 tienen un significado especial: cuando un proceso termina con una señal, su código de salida es 128 más el número de la señal.

3.4.1 Esperando la terminación del proceso

Si escribió y ejecutó el tenedor y ejecutivo ejemplo en Listado 3.4 , es posible que haya notado que la salida del ls El programa aparece a menudo después de que el "programa principal" ya se ha completado. Eso es porque el proceso hijo, en el que ls se ejecuta, se programa independientemente del proceso principal. Debido a que Linux es un sistema operativo multitarea, ambos procesos parecen ejecutarse simultáneamente y no se puede predecir si el ls El programa tendrá la oportunidad de ejecutarse antes o después de que se ejecute el proceso principal.

En algunas situaciones, sin embargo, es deseable que el proceso principal espere hasta que se hayan completado uno o más procesos secundarios. Esto se puede hacer con el Espere familia de llamadas al sistema. Estas funciones le permiten esperar a que un proceso termine de ejecutarse y permiten que el proceso padre recupere información sobre la terminación de su hijo. Hay cuatro llamadas al sistema diferentes en el Espere familia; puede optar por obtener poca o mucha información sobre el proceso que terminó, y puede elegir si le importa qué proceso secundario terminó.

3.4.2 El *Espere* Llamadas al sistema

La función más simple de este tipo se llama simplemente Espere . Bloquea el proceso de llamada hasta que uno de sus procesos secundarios sale (o se produce un error). Devuelve un código de estado a través de un argumento de puntero entero, del cual puede extraer información sobre cómo salió el proceso hijo. Por ejemplo, el WEXITSTATUS macro extrae el código de salida del proceso hijo.

Puedes usar el WIFEXITED macro para determinar a partir del estado de salida de un proceso hijo si ese proceso salió normalmente (a través del Salida función o regresando de principal) o murió por una señal no manejada. En el último caso, utilice el WTERMSIG macro para extraer de su estado de salida el número de señal por el que murió.

Aquí está el principal función de la tenedor y ejecutivo ejemplo de nuevo. Esta vez, el proceso padre llama Espere esperar hasta el proceso hijo, en el que el ls el comando se ejecuta, está terminado.

```
int main ()
{
  int child_status;

  / * La lista de argumentos para pasar al comando "ls". * / char *
  lista_arg [] == {
    "ls", / * argv [0], el nombre del programa. * /
    "-l",
    "/",
    NULL / * La lista de argumentos debe terminar con NULL. * /
  };

  / * Genera un proceso hijo ejecutando el comando "ls". Ignora el
       ID de proceso hijo devuelto. * /
```

```
    spawn ("ls", lista_arg);

    / * Espere a que se complete el proceso hijo. * / esperar (&
    child_status);
    si (WIFEXITED (child_status))
        printf ("el proceso hijo salió normalmente, con el código de salida% d \ n",
                WEXITSTATUS (child_status));
    demás
        printf ("el proceso hijo salió anormalmente \ n");

    return 0;
}
```

Varias llamadas de sistema similares están disponibles en Linux, que son más flexibles o brindan más información sobre el proceso hijo que sale. los esperar La función se puede utilizar para esperar a que salga un proceso hijo específico en lugar de cualquier proceso hijo. los espera3 La función devuelve estadísticas de uso de CPU sobre el proceso hijo saliente y el esperar4 La función le permite especificar opciones adicionales sobre qué procesos esperar.

3.4.3 Procesos Zombie

Si un proceso hijo termina mientras su padre llama a un Espere función, el proceso hijo desaparece y su estado de terminación se pasa a su padre a través de la Espere llama. Pero, ¿qué sucede cuando un proceso hijo termina y el padre no llama ¿Espere? ¿Simplemente desaparece? No, porque entonces se perdería información sobre su terminación, como si salió normalmente y, de ser así, cuál es su estado de salida. En cambio, cuando un proceso hijo termina, se convierte en un proceso zombi.

A *proceso zombie* es un proceso que ha terminado pero que aún no se ha limpiado. Es responsabilidad del proceso padre limpiar a sus hijos zombis. los Espere Las funciones también hacen esto, por lo que no es necesario rastrear si su proceso hijo todavía se está ejecutando antes de esperarlo. Supongamos, por ejemplo, que un programa bifurca un proceso hijo, realiza algunos otros cálculos y luego llama Espere . Si el proceso hijo no ha terminado en ese momento, el proceso padre se bloqueará en el Espere llamar hasta que finalice el proceso hijo. Si el proceso hijo finaliza antes de que el proceso padre llame Espere , el proceso hijo se convierte en un zombi. Cuando el proceso padre llama Espere , Se extrae el estado de terminación del niño zombi, se elimina el proceso secundario y se Espere la llamada regresa inmediatamente.

¿Qué sucede si el padre no limpia a sus hijos? Permanecen en el sistema, como procesos zombies. El programa en Listado 3.6 bifurca un proceso hijo, que termina inmediatamente y luego se duerme durante un minuto, sin siquiera limpiar el proceso hijo.

Listado 3.6 (zombie.c) Creación de un proceso zombie

```
# incluye <stdlib.h>
# incluye <sys / types.h>
# incluye <unistd.h>

int main ()
{
    pid_t child_pid;

    / * Crea un proceso hijo. * / child_pid =
    fork ();
    if (child_pid> 0) {
        / * Este es el proceso padre. Duerme un minuto. * / dormir (60);

    }
    demás {
```

```
        / * Este es el proceso hijo. Sal de inmediato. * / salir (0);

    }
    return 0;
}
```

Intente compilar este archivo en un ejecutable llamado hacer-zombi . Ejecútelo y, mientras aún se está ejecutando, enumere los procesos en el sistema invocando el siguiente comando en otra ventana:

```
% ps -e -o pid, ppid, stat, cmd
```

Esto enumera el ID del proceso, el ID del proceso principal, el estado del proceso y la línea de comando del proceso. Observe que, además del padre hacer-zombi proceso, hay otro hacer-zombi proceso enumerado. Es el proceso del niño; tenga en cuenta que su ID de proceso padre es el ID de proceso de la hacer-zombi proceso. El proceso hijo está marcado como < difunto> , y su código de estado es Z, para zombie.

¿Qué sucede cuando el principal hacer-zombi El programa finaliza cuando el proceso padre sale, sin siquiera llamar Espere ? ¿Se queda el proceso zombi? No, intenta correr PD de nuevo, y tenga en cuenta que ambos hacer-zombi los procesos se han ido. Cuando un programa sale, sus hijos son heredados por un proceso especial, el en eso programa, que siempre se ejecuta con el ID de proceso 1 (es el primer proceso que se inicia cuando Linux arranca). los en eso proceso limpia automáticamente cualquier proceso hijo zombi que herede.

3.4.4 Limpieza asincrónica de niños

Si está utilizando un proceso hijo simplemente para ejecutivo otro programa, está bien llamar Espere inmediatamente en el proceso padre, que se bloqueará hasta que se complete el proceso hijo. Pero a menudo, querrá que el proceso padre continúe ejecutándose, ya que uno o más hijos se ejecutan sincrónicamente. ¿Cómo puede estar seguro de limpiar los procesos secundarios que se han completado para no dejar los procesos zombis, que consumen recursos del sistema, por ahí?

Un enfoque sería que el proceso padre llamara espera3 o esperar4 periódicamente, para limpiar a los niños zombies. Vocación Espere para este propósito no funciona bien porque, si ningún hijo ha terminado, la llamada se bloqueará hasta que uno lo haga. Sin embargo, espera3 y esperar4 tomar un parámetro de bandera adicional, al que puede pasar el valor de la bandera WNOHANG . Con esta bandera, la función se ejecuta en *modo sin bloqueo* —Limpiará un proceso secundario terminado si lo hay, o simplemente regresará si no lo hay. El valor de retorno de la llamada es el ID de proceso del hijo terminado en el primer caso, o cero en el último caso.

Una solución más elegante es notificar al proceso padre cuando un hijo termina. Hay varias formas de hacer esto usando los métodos discutidos en Capítulo 5 , "Comunicación entre procesos", pero afortunadamente Linux lo hace por usted, utilizando señales. Cuando un proceso hijo termina, Linux envía al proceso padre el SIGCHLD señal. La disposición predeterminada de esta señal es no hacer nada, por lo que es posible que no lo haya notado antes.

Por lo tanto, una forma sencilla de limpiar los procesos secundarios es manipular SIGCHLD . Por supuesto, al limpiar el proceso hijo, es importante almacenar su estado de terminación si se necesita esta información, porque una vez que el proceso se limpia usando Espere , esa información ya no está disponible. Listado 3.7 es lo que parece que un programa utilice un SIGCHLD handler para limpiar sus procesos secundarios.

Listado 3.7 (sigchld.c) Limpieza de niños manipulando SIGCHLD

```
# incluye <señal.h>
# incluye <string.h>
# incluye <sys / types.h>
```

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

sig_atomic_t child_exit_status;

void clean_up_child_process (int número_señal) {

    / * Limpiar el proceso hijo. * / estado int;

    esperar (& estado);
    / * Almacena su estado de salida en una variable global. * /
    child_exit_status = estado;
}

int main ()
{
    / * Maneja SIGCHLD llamando a clean_up_child_process. * / struct sigaction
    sigchld_action;
    memset (& sigchld_action, 0, sizeof (sigchld_action));
    sigchld_action.sa_handler = & clean_up_child_process; sigaction
    (SIGCHLD, & sigchld_action, NULL);

    / * Ahora haga cosas, incluso bifurcar un proceso hijo. * / / * ... * /


  return 0;
}
```

Observe cómo el manejador de señales almacena el estado de salida del proceso hijo en una variable global, desde la cual el programa principal puede acceder a ella. Debido a que la variable se asigna en un manejador de señales, su tipo es sig_atomic_t .