Programa 2.4: Archivo `keeploglib.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

  /* Se ejecuta cmd y se guardan cmd y la hora de ejecución en la lista de
     historial. */
int runproc(char *cmd)
{
   data_t execute;

   time(&(execute.time)=;
   execute.string = cmd;
   if (system(cmd) == -1)
      return -1;
   return add_data(execute);
}
/*Se imprime la lista de historial contenida en el archivo f. */
void showhistory(FILE *f)
{
   data_t *infop;

   rewind_list();
   while ((infop = get data ()) != NULL)
      fprintf(f, "Comando: %s\nHora: %s\n", infop->string,
   return;
}
```

Programa 2.4

2.3 El proceso ID

UNIX identifica los procesos mediante un entero único denominado *ID del proceso*. El proceso que ejecuta la solicitud para la creación de un proceso recibe el nombre de *padre* del proceso, y el proceso creado se conoce como *hijo*. El ID del proceso padre identifica al padre del proceso. Para determinar estos procesos utilice las funciones `getpid` y `getppid`.

SINOPSIS

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

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

POSIX.1, Spec 1170

UNIX asocia cada proceso con un usuario particular conocido como *propietario* del proceso. El propietario tiene ciertos privilegios con respecto de los procesos. Cada usuario tiene un número de identificación único que se conoce como *ID del usuario*. Un proceso puede determinar el ID de usuario de su propietario con una llamada a `getuid`. El propietario es el usuario que ejecuta el programa. El proceso también tiene un *ID de usuario efectivo (effective user)*, que determina los privilegios que el proceso tiene para el acceso de recursos tales como archivos. El ID de usuario efectivo puede cambiar durante la ejecución de un proceso. El proceso puede determinar el ID de su usuario efectivo llamando a la función `geteuid`.

SINOPSIS

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

uid_t getuid(void);
uid_t geteuid(void);
```

POSIX.1, Spec 1170

Ejemplo 2.1

El siguiente programa imprime los ID del proceso, del padre del proceso y del propietario.

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

void main(void)
{
   printf("ID del proceso: %ld\n", (long)getpid());
   printf("ID del padre del proceso: %ld\n", (long)getppid());
   printf("ID del usuario propietario: %ld\n", (long)getuid());
}
```

Bajo POSIX, `getpid` devuelve un valor de tipo `pid_t` que puede ser un `int` o un `long`. Para evitar errores, haga que el valor devuelto por `getpid` sea de tipo `long`.

2.4 Estado de un proceso

El *estado* de un proceso indica la condición en que éste se encuentra en un momento determinado. Muchos sistemas operativos permiten alguna forma de los estados que aparecen en la tabla 2.2. El *diagrama de estados* es una representación gráfica de los estados permitidos de un proceso, así como de las transiciones permitidas entre éstos. La figura 2.3 muestra uno de estos diagramas. Los nodos de la gráfica representan los estados posibles, y las aristas las transiciones posibles. Un arco dirigido del estado **A** hacia el **B** significa que el proceso puede ir directamente del estado **A** al estado **B**. Las etiquetas sobre los arcos indican las condiciones que hacen que se presenten las transiciones entre los estados.

Estado	Significado
nuevo *(new)*	creación del proceso
ejecuta *(running)*	ejecución de las instrucciones del proceso
bloqueado *(blocked)*	proceso en espera de un evento, como una operación de E/S
listo *(ready)*	proceso en espera de ser asignado al procesador
hecho *(done)*	proceso terminado y recuperación de sus recursos

Tabla 2.2: Estados comunes de proceso.

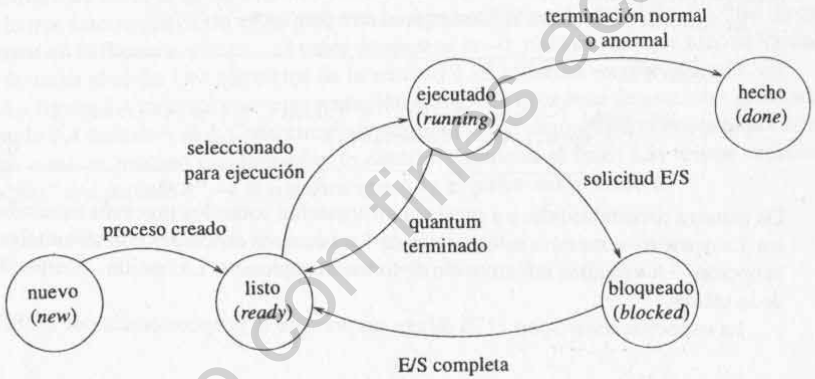

Figura 2.3: Diagrama de estado para un sistema operativo sencillo.

Se dice que un programa se encuentra en el estado *nuevo (new)* mientras experimenta la transformación que lo convertirá en un proceso activo. Cuando la transformación termina, el sistema operativo coloca el proceso en una cola de procesos que están listos para ser ejecutados. Entonces, el proceso se encuentra en el estado *listo (ready)*. En algún momento dado, el administrador de procesos lo seleccionará para ejecutarlo. El proceso se encontrará en el estado *ejecución (running)* cuando esté siendo ejecutado por el CPU.

Un proceso se encuentra en el estado bloqueado *(blocked)* si está en espera de un evento y no es considerado como candidato para ejecución por el administrador de procesos. Un proceso puede moverse voluntariamente al estado *bloqueado* haciendo una llamada a una función como `sleep`. Lo más común es que el proceso se mueva al estado *bloqueado* cuando lleva a cabo una solicitud de E/S. La entrada y la salida pueden ser miles de veces más lentas que las instrucciones ordinarias. La E/S es manejada por el sistema operativo, y un proceso lleva a cabo una operación E/S solicitando el servicio a través de una *llamada al sistema*. El sistema operativo vuelve a tomar el control y puede mover el proceso al estado bloqueado hasta que la operación termine.

El acto de quitar un proceso del estado ejecución y reemplazarlo con otro se conoce como *conmutación de contexto (context switch)*. El *contexto* de un proceso es la información sobre el proceso y su ambiente necesaria para continuarlo después de una conmutación de contexto. Es evidente que el ejecutable, la pila, los registros y el contador del programa son parte del contexto, al igual que la memoria utilizada por las variables asignadas estática y dinámicamente. Para poder continuar el proceso de manera transparente, el sistema operativo también mantiene el estado del proceso, el estado de la E/S del programa, la identificación del usuario y el proceso, los privilegios, información sobre la planificación y administración, e información sobre la administración de la memoria. Otra información que también es parte del contexto es la de si un proceso está en espera de un evento o ha atrapado una señal. El contexto también contiene información sobre otros recursos, como los bloqueos mantenidos por el proceso.

La utilería `ps` muestra información sobre procesos.

SINOPSIS

```
ps [-aA] [-G grouplist] [-o format]...[-p proclist]
   [-t termlist] [-U userlist]
```

POSIX.2

De manera preestablecida, `ps` presenta información sobre los procesos asociados con el usuario. La opción `-a` muestra información de los procesos asociados con terminales, mientras que la opción `-A` visualiza información de todos los procesos. La opción `-o` especifica el formato de la salida.

La especificación Spec 1170 difiere un poco de la proporcionada por POSIX.

SINOPSIS

```
ps [-aA] [-defl] [-G grouplist] [-o format]...[-p proclist]
   [-t termlist] [-U userlist] [-g grouplist] [-n namelist]
   [-u userlist]
```

Spec 1170

Muchas de las implantaciones de `ps` que hacen los vendedores no cumplen exactamente con la especificación POSIX o Spec 1170. Por ejemplo, Sun Solaris 2 emplea `-e` en lugar de `-A` para la opción que permite obtener información sobre todos los procesos. La versión de Sun sin ningún argumento presenta información sobre los procesos asociados con la terminal de control más que con el usuario. Las terminales de control serán estudiadas en la sección 7.5.

Ejemplo 2.2

La ejecución de `ps -l` *bajo Sun Solaris 2 produce la salida siguiente:*

```
F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME COMD
8 S   512  4509  4502 80  40 20 fc579000    205 fc5791c8 pts/13   0:00 csh
8 O   512  4627  4509 13  60 20 fc5b1800    151          pts/13   0:00 ps
```

La forma larga del `ps` de Sun del ejemplo 2.2 muestra mucha información interesante sobre los procesos asociados con la terminal de control activa (la cual tiene el nombre de dispositivo `pts/13`). La tabla 2.3 contiene un resumen con el significado de los distintos campos.

Encabezado	Significado
`F`	banderas asociadas con el proceso
`S`	estado del proceso
`UID`	ID del usuario propietario del proceso
`PID`	ID del proceso
`PPID`	ID del padre del proceso
`C`	utilización del procesador empleada para la administración de procesos
`PRI`	prioridad del proceso
`NI`	valor amabilidad (*nice value*)
`ADDR`	dirección en memoria del proceso
`SZ`	tamaño de la imagen del proceso
`WCHAN`	dirección del evento si el proceso está suspendido
`TTY`	terminal de control
`TIME`	tiempo acumulado de ejecución
`COMMAND`	nombre del comando

Tabla 2.3: Campos notificados por la forma larga del comando `ps` de Sun Solaris

2.5 Creación de procesos y el `fork` de UNIX

UNIX crea los procesos a través de una llamada `fork` al sistema, copiando la imagen en memoria que tiene el proceso padre. El nuevo proceso recibe una copia del espacio de direcciones del padre. Los dos procesos continúan su ejecución en la instrucción que está después del `fork`.

SINOPSIS

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

pid_t fork(void);
```

POSIX.1, Spec 1170

La creación de dos procesos totalmente idénticos no es algo muy útil. El valor devuelto por `fork` es la característica distintiva importante que permite que el padre y el hijo ejecuten código distinto. El `fork` devuelve 0 al hijo y el ID del hijo, al padre.

Ejemplo 2.3

En el siguiente fragmento de código tanto el padre como el hijo ejecutan la instrucción de asignación `x = 1` *después del regreso de* `fork`. *Existe un proceso y una sola variable* `x` *antes de la ejecución de* `fork`.

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

x = 0;
fork();
x = 1;
```

Después del `fork` del ejemplo 2.3, existen dos procesos independientes. Cada uno tiene su propia copia de la variable x. Puesto que los procesos padre e hijo son ejecutados de manera independiente, no ejecutan el código que esté bloqueado ni tampoco modifican la misma localidad de memoria. No es posible distinguir los procesos padre e hijo, ya que no se examinó el valor devuelto por `fork`.

Ejemplo 2.4

Después de la llamada a `fork` *en el siguiente fragmento de código, los procesos padre e hijo imprimen sus ID de proceso.*

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

if ((childpid = fork()) == 0) {
    fprintf(stderr, "Soy el hijo, ID = %ld\n", (long)getpid());
        /* el código del hijo va aquí */
} else if (childpid > 0) {
    fprintf(stderr, "Soy el padre, ID = %ld\n", (long)getpid());
        /* el código del padre va aquí */
}
```

El valor de la variable `childpid` en los procesos originales del ejemplo 2.4 es distinto de cero, así que con esto se ejecuta la segunda instrucción `fprintf`. El valor de `childpid` en el proceso hijo es cero, lo que ejecuta la primera instrucción `fprintf`. La salida de `fprintf` puede aparecer en cualquier orden.

Ejemplo 2.5

El siguiente fragmento de código crea una cadena de `n` *procesos.*

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

```
int i;
int n;
pid_t childpid;
for (i = 1; i < n; ++i)
   if (childpid = fork())
      break;
fprintf(stderr, "Éste es el proceso %ld con padre %ld\n",
      (long)getpid(), (long)getppid());
```

Con cada ejecución de `fork` en el ejemplo 2.5, el valor de `childpid` en el proceso padre es distinto de cero, lo que termina el ciclo. El valor de `childpid` en el proceso hijo es cero, con lo que éste se convierte en un padre en la siguiente iteración del ciclo. En caso de que haya un error en la llamada a `fork`, el valor de regreso es -1 y el proceso que hizo la llamada es el que termina el ciclo. Los ejercicios de la sección 2.12 se basan en este ejemplo.

La figura 2.4 muestra una representación gráfica de la cadena de procesos generada por el ejemplo 2.4 cuando `n` es 4. Cada círculo representa un proceso y está etiquetado por el valor de `i` que tiene el proceso correspondiente cuando abandona el lazo. Las aristas representan la relación "is a parente A" $\rightarrow B$ significa que A es el padre del proceso B.

Figura 2.4: Cadena de procesos generada por el fragmento de código del ejemplo 2.5 cuando `n` es 4.

Ejemplo 2.6

El siguiente fragmento de código crea un abanico de procesos.

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

```
int i;
int n;
pid_t childpid;

for (i = 1; i < n; ++i)
   if (childpid = fork()) <= 0)
      break;
fprintf(stderr, "Éste es el proceso %ld con padre %ld\n",
      (long)getpid(), (long)getppid());
```

La figura 2.5 muestra el abanico de procesos generado por el ejemplo 2.6 cuando `n` es 4. Los procesos de la figura están señalados con los valores de `i` que éstos tienen cuando abandonan el ciclo del ejemplo 2.6. El proceso original crea `n - 1` hijos. Los ejercicios de la sección 2.13 se basan en este ejemplo.

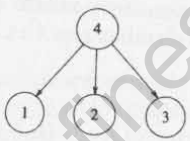

Figura 2.5: Abanico de procesos generado por el código del ejemplo 2.6 cuando `n` es 4.

Ejemplo 2.7

El siguiente código produce un árbol de procesos.

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

int i;
int n;
pid_t childpid;

for (i = 1; i < n; i++)
   if (childpid = fork()) == -1)
      break;
fprintf(stderr, "Éste es el proceso %ld con padre %ld\n",
      (long)getpid(), (long)getppid());
```

La figura 2.6 muestra el árbol de procesos generado por el ejemplo 2.7 cuando `n` es 4. Cada proceso está representado por un círculo y tiene como etiqueta el valor de `i` al momento en que fue creado.

El proceso original tiene la etiqueta 0. Las letras minúsculas distinguen procesos que fueron creados con el mismo valor de `i`. Si bien este código parece ser similar al del ejemplo 2.5, no hace distinción entre el padre y el hijo en el `fork`. Tanto el padre como el hijo crean hijos en la siguiente iteración del ciclo, esto explica la explosión poblacional de procesos.

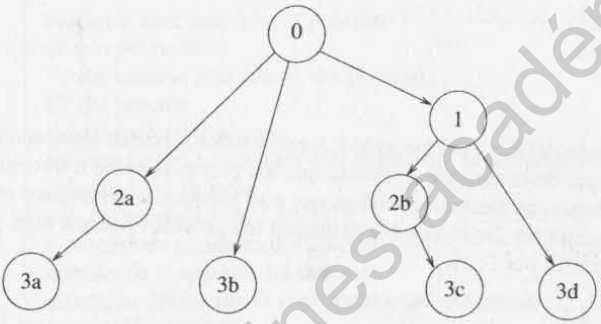

Figura 2.6: Arbol de procesos producido por un ciclo `fork`.

`Fork` crea procesos nuevos haciendo una copia de la imagen del padre en la memoria. El hijo *hereda* la mayor parte de los atributos del padre, incluyendo el ambiente y los privilegios. El hijo también hereda algunos de los recursos del padre, tales como los archivos y dispositivos abiertos. Las implicaciones de la herencia son más complicadas de lo que en principio parecen ser. En la sección 3.3 y el capítulo 4 se explora los aspectos de la herencia de archivos.

No todos los atributos o recursos del padre son heredados por el hijo. Este último tiene un ID de proceso nuevo y, claro está, un ID de padre diferente. Los tiempos del hijo para el uso del CPU son iniciados a 0. El hijo no obtiene los bloqueos que el padre mantiene. Si el padre ha puesto una alarma, el hijo no recibe notificación alguna del momento en que ésta expira. El hijo comienza sin señales pendientes, aunque el padre las tenga en el momento en que se ejecuta el `fork`.

Aunque el hijo hereda la prioridad del padre y los atributos de la administración de procesos, tiene que competir con otros procesos, como entidad aparte, por el tiempo del procesador. Un usuario que utiliza un sistema de tiempo compartido muy concurrido puede obtener un tiempo de acceso mayor mediante la creación de muchos procesos. Por el contrario, el sistema operativo VAX VMS permite la creación de un tipo de procesos en los que todos aquellos que son creados por un solo usuario comparten el tiempo de CPU asignado al usuario. El administrador de un sistema UNIX académico con gran demanda puede restringir la creación de procesos para impedir que un usuario cree procesos con la finalidad de obtener un segmento mayor de los recursos.

2.6 Llamada `wait` al sistema

¿Qué sucede con el proceso padre después de que éste crea un hijo? Tanto el padre como el hijo continúan la ejecución desde el punto donde se hace la llamada a `fork`. Si un padre desea esperar hasta que el hijo termine, entonces debe ejecutar una llamada a `wait` o a `waitpid`.

SINOPSIS

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

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

POSIX.1, Spec 1170

La llamada al sistema `wait` detiene al proceso que llama hasta que un hijo de éste termine o se detenga, o hasta que el proceso que la invocó reciba una señal. `Wait` regresa de inmediato si el proceso no tiene hijos o si el hijo termina o se detiene y aún no se ha solicitado la espera. Si `wait` regresa debido a la terminación de un hijo, el valor devuelto es positivo e igual al ID de proceso de dicho hijo. De lo contrario, `wait` devuelve `-1` y pone un valor en `errno`. Un `errno` igual a `ECHILD` indica que no existen procesos hijos a los cuales esperar, mientras que un `errno` igual a `EINTR` señala que la llamada fue interrumpida por una señal. `stat_loc` es un apuntador a una variable entera. Si quien hace la llamada pasa cualquier cosa distinta a `NULL`, `wait` guarda el estado devuelto por el hijo. POSIX especifica los macros `WIFEXITED`, `WEXITSTUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED` y `WSTOPSIG` para analizar el estado devuelto por el hijo y que permanece guardado en `*stat_loc`. El hijo regresa su estado llamando a `exit`, `_exit` o `return`.

Ejemplo 2.8

En el código siguiente el padre determina el estado de la salida de un hijo.

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

pid_t child
int status;

while(((child = wait(&status)) == -1) && (errno == EINTR))
   ;
if (child == -1)
   perror("No fue posible esperar al hijo");
else if (!status)
   printf("El hijo %ld terminó normalmente, el estado devuelto es cero\n",
          (long)child);
else if (WIFEXITED(status))
   printf("El hijo %ld terminó normalmente, el estado devuelto es %d\n",
          (long)child), (WEXITSTATUS(status));
```

```
else if (WIFSIGNALED(status))
   printf("El hijo %ld terminó debido a una señal no atrapada\n",
          (long)child);
```

El programa 2.5 ilustra la llamada `wait` al sistema. El proceso sólo tiene un hijo; así que si el valor devuelto no es el ID de proceso de dicho hijo entonces `wait` regresó tal vez por causa de una señal.

Programa 2.5 : Programa simple que ilustra el empleo de `wait`.

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

void main (void)
{
   pid_t childpid;
   int status;

   if ((childpid = fork()) == -1) {
      perror("fork falló");
      exit(1);
   } else if (childpid == 0)
      fprintf(stderr,
         "Soy el hijo con pid = %ld\n";(long)getpid());
   else if (wait(&status) != childpid)
      fprintf(stderr,"Una señal debió interrumpir la espera\n");
   else
      fprintf(stderr,
         "Soy el padre con pid = %ld e hijo con pid = %ld\n",
      (long)getpid(), (long)childpid);
   exit(0);
}
```

Programa 2.5

Ejercicio 2.3

¿Cuáles son las formas posibles de la salida generada por el programa 2.5?

Respuesta:

Existen varias posibilidades.

- Si `fork` falla (lo que es poco probable a menos que el programa genere un árbol sin fin de procesos), aparece entonces el mensaje `"fork falló"`. De lo contrario, si no existen señales, debe aparecer algo parecido a lo siguiente:

```
Soy el hijo con pid = 3427
Soy el padre con pid = 3426 e hijo pid = 3427
```

- Si una señal llega después de que el hijo ejecute `fprintf` pero antes del `exit`, entonces aparece lo siguiente:

```
Soy el hijo con pid = 3427
Una señal debió interrumpir la espera
```

- Si una señal llega después de que el proceso hijo termine y `wait` regrese, se imprime lo siguiente:

```
Soy el hijo con pid = 3427
Soy el padre con pid = 3426 e hijo pid = 3427
```

- Si la señal llega después de la terminación del hijo, pero antes de que `wait` regrese, entonces cualquiera de los dos resultados es posible, lo que depende del momento en que llegue la señal.
- Si la señal llega antes que el hijo ejecute el `fprintf` y si el padre ejecuta primero su instrucción `fprintf`, entonces aparece lo siguiente:

```
Una señal debió interrumpir la espera
Soy el hijo con pid = 3427
```

- Finalmente, si la señal llega antes de que el hijo ejecute el `fprintf` pero el hijo logra ejecutar éste, entonces se imprime lo siguiente:

```
Soy el hijo con pid = 3427
Una señal debió interrumpir la espera
```

Para que el hijo del programa 2.5 siempre imprima su mensaje primero, el padre debe esperar repetidamente hasta que el hijo termine, antes de imprimir su propio mensaje.

Ejemplo 2.9

El siguiente fragmento de código reinicia el `wait` *hasta que termine un proceso hijo en particular.*

```
#include <sys/types.h>
#include <sys/wait.h>
int status;
pid_t childpid;

while(childpid != wait(&status))
```

El `wait` del ejemplo 2.9 puede fallar en regresar el `childpid` si encuentra un error. La manera de distinguir entre una falla debida a una señal de otros errores, es analizar `errno`. Cuando `wait` falla, devuelve `-1` y establece el valor de `errno`. Un valor de `errno` igual a `EINTR` indica interrupción por una señal.

Ejemplo 2.10

El siguiente segmento de código ejecuta el ciclo hasta que el hijo con ID de proceso `chilpid` *termine o se presente algún error.*

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

int status;
pid_t childpid;

while(childpid != wait(&status))
   if ((childpid == -1) && (errno != EINTR))
      break;
```

La llamada `waitpid` al sistema proporciona métodos más flexibles para esperar a los hijos. Un proceso puede esperar un hijo en particular sin tener que esperar a todos sus hijos. Esta característica es útil para hacer el seguimiento de cierto hijo sin interferencia alguna por parte de los demás hijos cuya ejecución haya terminado. `Waitpid` también tiene una forma que no bloquea, de modo que un proceso pueda verificar de manera periódica las condiciones de no espera de los hijos sin quedarse suspendido indefinidamente.

La llamada a la función `waitpid` tiene tres parámetros: un `pid`, un apuntador que señala hacia una localidad donde guardar el estado, y las banderas de opción. Si `pid` es `-1`, `waitpid` espera a cualquier proceso. Si `pid` es positivo, entonces `waitpid` espera al hijo cuyo ID de proceso es `pid`. La opción `WNOHANG` hace que `waitpid` regrese, incluso si el estado del hijo no está disponible de inmediato. Para una especificación completa de todos los parámetros de `waitpid`, consulte la página del manual correspondiente.

Ejemplo 2.11

El siguiente fragmento de código espera a cualquier hijo, evitando el bloqueo si no hay hijos cuyo estado se encuentre disponible de inmediato. El procedimiento vuelve a iniciarse si `waitpid` *es interrumpida por una señal.*

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
int status;
pid_t waitreturnpid;

while(waitreturnpid = waitpid(-1, &status, WNOHANG))
   if ((waitreturnpid == -1) && (errno != EINTR))
      break;
```

Cuando `waitpid` devuelve 0, significa que existen hijos a los que todavía hay que esperar, pero ninguno de ellos está listo. ¿Qué sucede con un proceso cuyo padre no lo espera? En la terminología de UNIX, éste se convierte en un *zombie*. Los zombies permanecen en el sistema hasta que alguien los espere. Si un padre termina y no espera a uno de sus hijos, el hijo se convierte en un *huérfano* y es adoptado por el proceso `init` del sistema, el cual tiene un ID

de proceso igual a 1. El proceso `init` espera periódicamente a los hijos de modo que, en algún momento, los *zombies* huérfanos desaparecen del sistema.

Ejercicio 2.4

El siguiente segmento de código crea un abanico de procesos. Todos los procesos generados por llamadas a `fork` son hijos del proceso original. ¿Cuál es el orden de los mensajes de salida?

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

int i;
int n;
pid_t childpid;
int status;

for (i = 1; i < n; ++i)
   if (childpid = fork()) <= 0)
      break;
for( ; ; ) {
   childpid = wait(&status);
   if ((childpid == -1) && (errno != EINTR))
      break;
}
fprintf(stderr, "Soy el proceso %ld, mi padre es %ld\n",
                (long)getpid(), (long)getppid());
```

Respuesta:

Dado que ninguno de los hijos generados por `fork` son padres, el `wait` de éstos devuelve `-1` y hace que `errno` sea `ECHILD`. Ellos no son bloqueados por el segundo ciclo `for`. Sus mensajes de identificación pueden aparecer en cualquier orden. El mensaje del proceso original aparecerá al final, después de haber esperado a todos sus hijos.

Ejercicio 2.5

El siguiente fragmento de código crea una cadena de procesos. Sólo uno de los procesos generados por `fork` es un hijo del proceso original. ¿Cuál es el orden en que aparecen los mensajes?

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

int i;
int n;
pid_t childpid;
```

```
int status;
pid_t waitreturn;

for (i = 1; i < n; ++i)
   if (childpid = fork())
      break;
while(childpid != waitreturn = wait(&status)))
   if ((waitreturn == -1) && (errno != EINTR))
      break;
fprintf(stderr, "Soy el proceso %ld, mi padre es %ld\n",
                (long)getpid(), (long)getppid());
```

Respuesta:

Cada hijo generado por `fork` espera a que su hijo termine antes de escribir su propio mensaje de salida. Los mensajes aparecen en orden inverso al de la creación de los procesos.

2.7 Llamada exec al sistema

La llamada `fork` al sistema crea una copia del proceso que la llama. Muchas aplicaciones requieren que el proceso hijo ejecute un código diferente del de su padre. La familia `exec` de llamadas al sistema proporciona una característica que permite traslapar al proceso que llama con un módulo ejecutable nuevo. La manera tradicional de utilizar la combinación `fork-exec` es dejar que el hijo ejecute el `exec` para el nuevo programa mientras el padre continúa con la ejecución del código original.

Las seis variaciones de la llamada `exec` al sistema se distinguen por la forma en que son pasados los argumentos de la línea comando y el ambiente, y por si es necesario proporcionar la ruta de acceso y el nombre del archivo ejecutable. Las llamadas `execl` (`execl`, `execlp` y `execle`) pasan los argumentos de la línea comando como una lista y son útiles si se conoce el número de argumentos de línea comando se conoce al momento de la compilación. Las llamadas `execv` (`execv`, `execvp` y `execve`) pasan los argumentos de la línea comando en un arreglo de argumentos.

El código del programa 2.6 llama al comando `ls` con un argumento de línea comando igual a `-l`. El programa supone que `ls` está localizado en el directorio `/usr/bin`.

Programa 2.6: Programa que crea un proceso para ejecutar `ls -l`.

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

void main (void)
{
   pid_t childpid;
```

```
    int status;

    if ((childpid = fork()) == -1) {
        perror("Error al ejecutar fork");
        exit(1);
    } else if (childpid == 0) {                          /* código del hijo */
        if (execl("/usr/bin/ls", "ls", "-1", NULL) < 0) {
            perror("Falla en la ejecución de ls");
            exit(1);
        }
    } else if (childpid ! wait(&status))        /* código del padre */
        perror("Se presentó una señal antes de la terminación del hijo");
    exit(0);
}
```

Programa 2.6

SINOPSIS

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ...,
          const char *argn, char * /*NULL*/);
int execle (const char *path,char *const arg0[], ... ,
          const char *argn, char * /*NULL*/,
          char *const envp[]);
int execlp (const char *file, const char *arg0, ...,
          const char *argn, char * /*NULL*/);
```

POSIX.1, Spec 1170

El parámetro `path` de `execl` es la ruta de acceso y el nombre del programa, especificado ya sea como un nombre con la ruta completa o relativa al directorio de trabajo. Después aparecen los argumentos de la línea comando seguidos de un apuntador `NULL`. La lista de parámetros de cadena de caracteres corresponde al arreglo `argv` para el comando `exec`. Puesto que `argv[0]` es el nombre del programa, éste es el segundo argumento de `execl`.

Una forma alternativa es `execlp`, la cual tiene los mismos parámetros que `execl` pero utiliza la variable de ambiente `PATH` para buscar el ejecutable. De manera similar, cuando el usuario introduce un comando, el shell intenta localizar el archivo ejecutable en uno de los directorios especificados por la variable `PATH`.

Una tercera forma de `execl` es `execle`, la cual es similar a `execl` con la excepción de que toma un parámetro adicional que representa el nuevo ambiente del programa por ejecutar. En el casop de las otras formas de `execl`, el nuevo programa hereda el ambiente del padre.

`Execv` toma exactamente dos parámetros, un nombre y ruta de acceso para el ejecutable y un arreglo de argumentos. (En este caso, resulta útil la función `makeargv` del programa 1.2.) `Execvp` construye un nombre y ruta de acceso completo a partir del parámetro `file` al utilizar los prefijos de ruta de acceso encontrados en la variable de ambiente `PATH`. La forma `execve`

requiere un tercer parámetro, el arreglo de argumento `envp`, el cual especifica el ambiente para el proceso creado.

SINOPSIS

```
#include <unistd.h>

int execv(const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *path, char *const argv[],
      char *const envp[]);
```

POSIX.1, Spec 1170

El programa 2.7 utiliza a `execvp` para ejecutar el comando pasado por la línea de comando. Suponga que el ejecutable del programa 2.7 es `myexec`. El proceso original crea un hijo y luego espera a que éste termine. El proceso hijo llama a `execvp` con el arreglo de argumento formado por los argumentos de la línea comando del programa original.

Ejemplo 2.12

La línea siguiente comando hace que `myexec` *cree un nuevo proceso para ejecutar el comando* `ls -l`.

```
myexec ls -l
```

El arreglo `argv` original producido en el ejemplo 2.12 contiene apuntadores a tres fichas (*tockens*): `myexec`, `ls` y `-l`. El arreglo de argumento para la función `execvp` comienza en `&argv[1]` y contiene apuntadores a los dos componentes léxicos `ls` y `-l`. El padre espera al hijo. Si el `wait` es interumpido por una señal, `errno` es igual a `EINTR` y el padre vuelve a comenzar la espera.

Programa 2.7: Programa que crea un proceso para ejecutar el comando que es pasado como argumento de la línea comando.

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

void main(int argc, char *argv[])
{
    pid_t childpid;
    int status;

    if ((childpid = fork()) == -1) {
       perror("Error al ejecutar fork");
```

```
        exit(1);
    } else if (childpid == 0) {                    /* código del hijo */
        if (execvp(argv[1], &argv[1]) < 0) {
            perror("Falla en la ejecución del comando");
            exit(1);
            }
    } else                                         /* código del padre */
        while(childpid != wait(&status))
            if ((childpid == -1) && (errno != EINTR))
                break;
    exit(0);
}
```

Programa 2.7

Ejercicio 2.6

Ejecute `myexec` del programa 2.7 con la línea comando `myexec ls -l *.c`. ¿Cuál es el tamaño del arreglo de argumentos pasado como el segundo argumento a `execvp`?

Respuesta:

El tamaño depende del número de archivos con extensión `.c` presentes en el directorio, ya que el shell amplía `*.c` antes de pasar la línea comando a `myexec`.

El programa 2.8 llama a la función `makeargv` del programa 1.2 para crear un arreglo de argumento a partir de la cadena de caracteres pasada como el primer argumento de la línea comando. Luego, se hace un `execvp` del comando representado por dicha cadena de caracteres. El paso de una cadena que contiene muchos componentes léxicos se hace delimitando ésta con comillas dobles (por ejemplo, `myexec "ls -l"`).

Observe que la llamada a la función `makeargv` la hace sólo el proceso hijo del programa 2.8. Si el padre llama a `makeargv` antes del `fork`, entonces el padre tiene un arreglo de argumentos sin utilizar asignado en su *heap*. Una sola llamada a `makeargv` no presenta ningún problema. Sin embargo, en un *shell* donde el paso de asignación puede repetirse cientos de veces, la limpieza de la memoria puede convertirse en un problema.

Programa 2.8: Programa que crea un proceso para ejecutar el comando que es pasado como argumento de la línea comando. El programa 1.2 muestra una implantación de la función `makeargv`.

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

```
int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv[]) {
    char **myargv;
    char delim[] = " \t";
    pid_t childpid;
    int status;

    if (argc != 2) {
        fprintf(stderr, Uso: %s string\n", argv[0];
        exit(1);
    }
    if ((childpid = fork()) == -1) {
        perror("Error al ejecutar fork");
        exit(1);
    } else if (childpid == 0) {                    /* código del hijo */
      if (makeargv(argv[1], delim, &myargv) < 0) {
          fprintf(stderr, "No fue posible construir el arreglo de argumentos\n");
          exit(1);
      } else if (execvp(myargv[0], &myargv[0]) < 0) {
          perror("Falla en la ejecución del comando");
          exit(1);
      }
    } else                                         /* código del padre */
        while(childpid != wait(&status))
            if ((childpid == -1) && (errno != EINTR))
                break;
    exit(0);
}
```

Programa 2.8

Exec copia un ejecutable nuevo en la imagen del proceso. No se sabe con exactitud qué es lo que se conserva del proceso original. Tanto el texto del programa como las variables, la pila y el *heap* son sobrescritos. El nuevo proceso hereda el ambiente (lo que significa la lista de variables de ambiente y sus valores asociados), a menos que el proceso original llame a execle o execve. Normalmente, los archivos que están abiertos antes del exec por lo común permanecen abiertos (como se explica en el capítulo 3.3). El capítulo 5 estudia los efectos de exec sobre las señales y los bloqueos.

La tabla 2.4 presenta un resumen de los atributos heredados por los procesos ejecutados por exec. La segunda columna de la tabla proporciona las llamadas al sistema relacionada con los ítems. Los ID asociados con el proceso quedan intactos después del exec. Si un proceso establece una alarma antes de llamar a exec, ésta generará de todos modos una señal cuando el tiempo termine. Las señales que están pendientes también serán traspasadas al exec, contrariamente a lo que sucede con fork. Los procesos crean archivos con los mismos permisos igual que antes del exec, y la contabilidad del tiempo de CPU continúa sin ser reiniciada.

Atributo	Llamada relevante al sistema
ID del proceso	`getpid( )`
ID del padre del proceso	`getppid( )`
ID de grupo del proceso	`getpgid( )`
membresía de sesión	`getsid( )`
ID real del usuario	`getuid( )`
ID real del grupo	`getgid( )`
ID de grupos complementarios	`getgroups( )`
tiempo que resta en la señal de alarma	`alarm( )`
directorio de trabajo en uso	`getcwd( )`
directorio raíz	
máscara del modo de creación del archivo	`umask( )`
máscara de señal del proceso	`sigprocmask( )`
señales pendientes	`sigpending( )`
tiempo utilizado hasta el momento	`times( )`

Tabla 2.4 : Atributos conservados después de hacer llamadas a la función exec. En la segunda columna aparecen las llamadas al sistema importantes para estos atributos.

2.8 Procesos en plano secundario y demonios

El *shell* es un intérprete de comandos dispuesto a aceptar comandos, leerlos de la entrada estándar, crear hijos para ejecutarlos, y esperar a los hijos terminen. Cuando la entrada y la salida provienen de un dispositivo tipo terminal, el usuario puede terminar la ejecución de un comando presionando el carácter de interrupción. (Éste puede ser configurado, pero lo más común es que sea `ctrl-c`.)

Ejercicio 2.7

Vaya a un directorio que contenga muchos archivos (por ejemplo, `/etc`), y ejecute el siguiente comando.

```
ls -l
```

¿Qué sucede? Ahora ejecute `ls -l` de nuevo pero presione `ctrl-c` tan pronto como aparezca el listado en la pantalla. Compare los resultados con lo ocurrido en el primer caso.

Respuesta:

En el primer caso el resultado se imprime después de que el listado del directorio se ha completado debido a que el *shell* espera al hijo antes de continuar. En el segundo caso, `ctrl-c` termina la ejecución de `ls`.

Muchos *shells* interpretan una línea que termina con un & como un comando que debe ser ejecutado por un proceso en plano secundario, en asincronía. Cuando el *shell* crea un proceso de fondo, no espera a que éste termine para imprimir el indicador de línea comando y aceptar más comandos. Por otra parte, el `ctrl-c` del teclado no termina un proceso en plano secundario. (El capítulo 7 presenta una discusión más técnica de los procesos en plano secundario.)

Ejercicio 2.8

Compare los resultados del ejercicio 2.7 con los que se obtiene al ejecutar el siguiente comando.

```
ls -l &
```

Vuelva a introducir el comando `ls - l &` e intente terminarlo presionando `ctrl-c`.

Respuesta:

En el primer caso el indicador de línea comando aparece antes de que listado esté completo. El `ctrl-c` no afecta al proceso de fondo, de modo que el segundo caso se comporta igual que el primero.

Un *demonio* es un proceso en plano secundario que normalmente se ejecuta por tiempo indefinido. El sistema operativo UNIX depende de muchos procesos demonios para llevar a cabo tareas rutinarias (y otras no tan rutinarias). Bajo Solaris 2, el demonio `pageout` maneja la paginación para la administración de la memoria. El `in.rlogind` maneja las solicitudes remotas de acceso al sistema. Otros demonios se encargan del correo electrónico, ftp, estadísticas, y solicitudes de impresión, para nombrar sólo unas cuantas tareas. Cuando la ejecución del demonio termina, éste no deja ninguna pista de su procedencia.

El programa `runback` del programa 2.9 ejecuta el primer argumento de la línea comando como un proceso en plano secundario. El hijo llama a `setsid` para no ser molestado por ninguna interrupción de tipo `ctrl-c` proveniente de la terminal de control.

Programa 2.9 : Programa `runback` que crea un proceso para ejecutar un comando en plano secundario.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv[]) {
   char **myargv;
   char delim[] = " \t";
   pid_t childpid;

   if (argc != 2) {
```

```
        fprintf(stderr, Uso: %s string\n", argv[0]);
        exit(1);
    }

    if ((childpid = fork()) == -1) {
        perror("Error al ejecutar fork");
        exit(1);
    } else if (childpid == 0) { /* el hijo se convierte en un proceso en plano
                                   secundario */
        if (setsid() == -1)
            perror("No es posible convertirse en un líder de sesión");
        else if (makeargv(argv[1], delim, &myargv) < 0)
            fprintf(stderr, "No fue posible construir el arreglo de argumentos\n");
        else if (execvp(myargv[0], &myargv[0]) < 0)
            perror("Falla en la ejecución del comando");
        exit(1);                                /* el hijo nunca debe regresar */
    }
    exit(0);                                            /* el padre termina */
}
```

Programa 2.9

El programa `runback` utiliza a `setsid` para crear una sesión nueva que no tenga una terminal de control. El ID de sesión determina si el proceso tiene una terminal de control (de modo que pueda recibir una señal proveniente de `ctrl-c`). El capítulo 7 explora estos aspectos más detalladamente.

Ejemplo 2.13

El siguiente comando es similar a la introducción directa en el shell *de* `ls -l &`.

```
runback "ls -l"
```

Algunos sistemas cuentan con un servicio denominado `biff` que permite la notificación de correo. Cuando un usuario está conectado al sistema y recibe correo, `biff` lo notifica de alguna manera, como puede ser una señal audible o la presentación de un mensaje. (En el mundo UNIX, se dice que el autor original de este programa tenía un perro, Biff, que ladraba a todos los carteros.) El programa 2.10 presenta el código de un programa en C, `simplebiff.c`, que emite una señal audible en la terminal a intervalos regulares de tiempo si el usuario, `oshacker`, tiene correo.

El programa notifica al usuario que tiene correo enviando un carácter `ctrl-g` (ASCII 7) al dispositivo de error estándar. Muchas terminales manejan la recepción de un `ctrl-g` produciendo una señal audible breve. El programa continúa produciendo esta señal cada 10 segundos hasta que es terminado o se elimina el archivo de correo.

Ejemplo 2.14

El siguiente comando da inicio a la ejecución de `simplebiff`.

```
simplebiff &
```

Programa 2.10: Programa simple para notificar a `oshacker` que hay correo.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define MAILFILE "/var/mail/oshacker"
#define SLEEPTIME 10

void main(void)
{
   int mailfd;

   for( ; ; ) {
      if ( (mailfd = open(MAILFILE, O_RDONLY)) != -1) {
         fprintf(stderr,"%s", "\007");
         close(mailfd);
      }
      sleep(SLEEPTIME);
   }
}
```

Programa 2.10

El programa 2.10 ilustra la forma en que pueden trabajar los demonios. Por lo común, el correo se guarda en el directorio `/var/mail` o en `/var/spool/mail`. Todo el correo que no ha sido leído por el usuario se halla contenido en un archivo, en alguno de estos directorios, que tiene el mismo nombre de inicio de sesión del usuario. Si `oshacker` tiene correo, entonces la llamada a `open` para abrir `/var/mail/oshacker` tendrá éxito, de lo contrario `open` fallará. Si el archivo existe, entonces el usuario tiene correo que no ha leído y el programa hará que se emita una señal audible. En cualquier caso, el programa duerme y luego repite el proceso indefinidamente.

El programa 2.10 no es muy general porque el nombre del usuario, el directorio del correo y el tiempo durante el cual el programa debe dormir están especificados de antemano. La manera de obtener el nombre del usuario aprobada para POSIX es hacer, primero, una llamada a `getuid` para determinar el ID del usuario y después llamar a `getpwuid` para obtener el nombre de la sesión. La llamada al sistema `stat` proporciona más información sobre un archivo sin todas las complicaciones de `open`.

La estructura de directorio para el correo cambia de un sistema a otro, de modo que el usuario debe determinar la ubicación de los archivos de correo del sistema para poder hacer uso de `simplebiff`. El programa debe permitir que el usuario especifique el directorio en la línea comando, o depender de la información específica del sistema comunicada por las variables de ambiente, si es que esta información se encuentra disponible. El estándar POSIX define diversas variables de ambiente importantes, tales como `MAIL`, `MAILCHECK`, `MAILDIR` y `MAILPATH`. Si estas variables guardan información, entonces el programa debe hacer uso de

ésta. La sección 2.9 estudia la forma en que un programa puede tener acceso a estas variables y la manera en que éstas pueden comunicar información específica del sistema. En la sección 2.14 se analiza un `biff` más conveniente. En la sección 5.5 se modifica el programa `biff` para que haga uso de señales con objeto de habilitar o deshabilitar la notificación.

2.9 Ambiente de los procesos

El biff del programa 2.10 ilustra la importancia que reviste el hacer uso de información dependiente del sistema para hacer una implantación de la aplicación independiente del mismo. Ningún programador experimentado distribuirá un programa que requiera que el usuario cambie las rutas de acceso a los directorios codificadas de antemano para que éste funcione. Las *variables de ambiente* proporcionan un mecanismo para hacer uso de información específica del sistema o del usuario para establecer los valores por omisión dentro del programa.

Una *lista de variables de ambiente* está formada por un arreglo de apuntadores a cadenas de caracteres de la forma *nombre* = *valor*. El *nombre* señala una variable de ambiente. El *valor* especifica una cadena de caracteres. Cada aplicación interpreta la lista de ambiente en una forma que depende de la aplicación. POSIX.2 especifica el significado de las variables de ambiente que aparecen en la tabla 2.5.

Variable	Significado
`HOME`	directorio de trabajo inicial del usuario
`LANG`	localidad cuando no está especificada por `LC_ALL` o `LC_*`
`LC_ALL`	sustitución del nombre de la localidad
`LC_COLLATE`	nombre de la localidad para recopilar información
`LC_CTYPE`	nombre de la localidad para clasificación de caracteres
`LC_MONETARY`	nombre de la localidad para edición monetaria
`LC_NUMERIC`	nombre de la localidad para edición numérica
`LC_TIME`	nombre de la localidad para información fecha/hora
`LOGNAME`	nombre de la contraseña de inicio de sesión asociada con un proceso
`PATH`	prefijos de ruta de acceso para encontrar el ejecutable
`TERM`	tipo de terminal para enviar la salida
`TZ`	información sobre uso horario

Tabla 2.5: Variables de ambiente POSIX.2 y sus significados.

La variable externa environ apunta a la lista de ambiente del proceso cuando comienza la ejecución de éste. La variable environ está definida por

```
extern char **environ
```

Las cadenas de caracteres en la lista de ambiente pueden aparecer en cualquier orden. Si el proceso ha sido iniciado por `execl`, `execlp`, `execv` o `execvp`, entonces éste hereda la lista de ambiente que tiene el proceso justo antes de la ejecución de `exec`. Las llamadas a `execle` y `execve` permiten especificar la lista de ambiente.

Ejemplo 2.15

El siguiente programa en C imprime el contenido de su lista de ambiente y finaliza después de hacerlo.

```
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

void main(int argc, char *argv[])
{
    int i;

    printf("La lista de ambiente para %s es\n", argv[0];
    for (i = 0; environ[i] != NULL; i++)
        printf("environ[%d]: %s\n", i, environ[i]);
    exit(0);
}
```

Para determinar si cierta variable de ambiente está definida haga uso de `getenv`.

SINOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
```

POSIX.1, Spec 1170

El nombre de la variable de ambiente debe ser pasado como una cadena de caracteres. La función `getenv` devuelve `NULL` si dicha variable no está definida. Si tiene un valor, `getenv` devuelve un apuntador a la cadena de caracteres que contiene el valor. En caso de usar `getenv` varias veces debe tenerse cuidado de copiar en un *buffer* la cadena devuelta por la función. Algunas implantaciones de `getenv` emplean un *buffer* estático para poner en él las cadenas devueltas por la función, y sobrescriben el *buffer* en cada llamada.

POSIX.2 indica que el *shell* sh debe utilizar la variable de ambiente `MAIL` como ruta de acceso a los buzones en lo que respecta al correo que llega, siempre y cuando la variable `MAILPATH` no tenga ningún valor. (Vea la sección 2.14 para más información sobre `MAIL` y `MAILPATH`.)

Ejemplo 2.16

El siguiente fragmento de código pone en `mailp` *el valor de* `MAIL` *si éste se encuentra definido, de lo contrario hace que* `MAILP` *sea igual a* `MAILPATH`. *En cualquier otro caso, pone en* `mailp` *un valor preestablecido.*

```
#include <stdlib.h>
#define MAILDEFAULT "/var/mail"
char *mailp = NULL;

if (getenv("MAILPATH") == NULL)
   mailp = getenv("MAIL");
if (mailp == NULL)
   mailp = MAILDEFAULT;
```

La primera llamada a `getenv` del ejemplo 2.16 sólo verifica la existencia de `MAILPATH`, de modo que no es necesario copiar el valor devuelto en un *buffer* aparte antes de volver a llamar a `getenv` otra vez.

No confunda las variables de ambiente con constantes predefinidas como `MAX_CANON`. Estas constantes están definidas en los archivos de encabezado con un `#define`. Sus valores son constantes y se los conoce al momento de la compilación. Para saber si existe la definición de alguna constante, utilice la directiva del compilador `#ifndef`, como se hace en el programa 2.3. Por el contrario, las variables de ambiente son dinámicas y sus valores no son conocidos al momento de la compilación.

POSIX.2 especifica una utilería `env` para examinar el ambiente y modificarlo con el fin de ejecutar otro comando.

SINOPSIS

```
env [-i] [name=value] ... [utility [argument ...]]
```

POSIX.2, Spec 1170

Cuando es invocado sin argumentos, el comando `env` muestra el ambiente en uso. Los argumentos opcionales `[name=value]`, indican las variables de ambiente que serán modificadas. El argumento opcional `utility` señala el comando que será ejecutado bajo el ambiente modificado. El argumento opcional `-i` indica que el ambiente especificado por los argumentos deberá reemplazar al ambiente en uso para los fines que la ejecución de `utility` implique. La utilería `env` no modifica el ambiente del *shell* que la ejecuta.

Ejemplo 2.17

El siguiente es un listado de la salida generada por la ejecución de `env` *en una máquina que corre bajo Sun Solaris 2.3. El guión (-) indica la continuación de una línea larga.*

```
DISPLAY=:0.0
FONTPATH=/home/robbins/vttool/crttool-2.0/fonts:-
/usr/local/lib/font/85dpi
HELPPATH=/usr/openwin/lib/locale:/usr/openwin/lib/help
HOME=/data1/robbins
HZ=100
LD_LIBRARY_PATH=/usr/openwin/lib:/opt/tex/lib
LOGNAME=robbins
MAIL=/var/mail/robbins
MANPATH=/usr/openwin/share/man:/opt/SUNWspro/man:/usr/man
OPENWINHOME=/usr/openwin
PATH=/usr/openwin/bin:/opt/SUNWspro/bin:/usr/bin:/usr/sbin:-
/usr/ccs/bin:/usr/bin/X11:/opt/gnu/bin:/opt/tex/bin:-
/opt/bin:/data1/robbins/bin:/usr/local/bin:/usr/ccs/lib:.
PROCDIR=/usr/local/bin
PWD=/data1/robbins
SHELL=/bin/csh
TERM=sun-cmd
```

```
TEXFONTS=.:/usr/local/tex/fonts/tfm:/usr/local/src/dvips/PStfms:-
/usr/local/src/dlx/benchmarks/tex/latex:
/data/src/tex/unix3.0/ams/amsfonts/pk/pk300
TZ=US/Central
USER=robbins
XDVIFONTS=/usr/local/tex/fonts/pk
XENVIRONMENT=/data1/robbins/.Xdefaults
XFILESEARCHPATH=/usr/openwin/lib/locale/%L/%T/%N%S:-
/usr/openwin/lib/%T/%N%S
WINDOW_TERMIOS=
TERMCAP=sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
```

2.10 Terminación de procesos en UNIX

Cuando termina un proceso, el sistema operativo recupera los recursos asignados al proceso terminado, actualiza las estadísticas apropiadas y notifica a los demás procesos la defunción. La terminación puede ser *normal* o *anormal*. Las actividades realizadas durante la terminación del proceso incluyen la cancelación de los temporizadores y señales pendientes, la liberación de los recursos de memoria virtual así como la de otros recursos del sistema retenidos por el proceso, tales como los bloqueos, y el cierre de archivos abiertos. El sistema operativo registra el estado del proceso y el uso de los recursos, y notifica al padre en respuesta a una llamada `wait` al sistema. Cuando un proceso termina, sus hijos huérfanos son adoptados por el proceso `init`, cuyo ID de proceso es 1. Si el padre no espera a que el proceso termine, entonces éste se convierte en un *zombie*. El proceso `init` espera de manera periódica a los hijos para librarse de los *zombies* huérfanos.

La terminación normal se presenta cuando existe un `return` de *main*, un regreso implícito de *main* (el procedimiento *main* no es capaz de terminar), una llamada a la función de C `exit`, o una llamada a la función del sistema `_exit`. La función `exit` de C llama a los manejadores de la salida del usuario y puede proporcionar tareas de limpieza adicionales antes de invocar la llamada al sistema `_exit`.

SINOPSIS

```
#include <unistd.h>

void _exit(int status);
```

POSIX.1, Spec 1170

SINOPSIS

```
#include <stdlib.h>

void exit(int status);
```

ISO C, POSIX.1, Spec 1170

Tanto `exit` como `_exit` toman un parámetro entero, `status`, que indica el estado de terminación del programa. Para indicar una terminación normal, haga uso de un valor 0 para `status`. Los valores de `status` distintos de 0 y definidos por el programador indican errores. El ejemplo 2.8 de la página 48 ilustra la manera en que el padre puede determinar el valor de `status` cuando espera por los hijos. El valor es devuelto por una llamada a `exit` o `_exit` en cualquier parte del programa, o por el empleo de un `return` en el programa principal. `exit` y `_exit` devuelven un valor al padre incluso si la declaración de *main* indica que el valor devuelto es de tipo `void`.

La función `atexit` de C instala un manejador de terminación definido por el usuario. Cuando se llama a `exit` estos manejadores son ejecutados sobre la base de que el último en ser instalado es el primero en ser ejecutado. Para instalar varios manejadores, haga uso de varias llamadas a `atexit`. Cuando la terminación es normal, el proceso llama a los manejadores como parte de la terminación, siendo el primero instalado el último en ser ejecutado.

El programa 2.11 tiene un manejador de salida denominado `show_times`, el cual hace que se imprima en el dispositivo de error estándar, las estadísticas sobre el tiempo utilizado por el programa y sus hijos antes de que el programa termine. La función `times` devuelve información sobre el tiempo con un número de pulsos del reloj. La función `show_times` convierte el tiempo en segundos dividiendo éste entre el número de pulsos registrados por segundo, dato que se obtiene al llamar a `sysconf`. (El capítulo 6 proporciona un estudio más completo del tiempo en UNIX.)

Programa 2.11: Programa con manejador de salida que imprime el uso de la CPU.

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

static void show_times(void)
{
   status tms times_inf;
   double ticks;

   if ((ticks = (double) sysconf(_SC_CLK_TCK)) < 0)
      perror("No es posible determinar el número de pulsos por segundo");
   else if (times(&times_info) < 0)
      perror("No es posible obtener información sobre la hora")
   else {
      fprintf(stderr, "Tiempo de usuario:              %8.3f segundos\n",
         (times_info.tms_utime/ticks);
      fprintf(stderr, "Tiempo del sistema:             %8.3f segundos\n",
         (times_info.tms_stime/ticks);
      fprintf(stderr, "Tiempo de usuario del hijo:     %8.3f segundos\n",
         (times_info.tms_cutime/ticks);
```

```
    fprintf(stderr, "Tiempo del sistema del hijo:  %8.3f segundos\n",
      (times_info.tms_cstime/ticks);

}

void main(void)
{
    if (atexit(show_times)) {
       fprintf(stderr, "No es posible instalar el manejador de salida
                        show_times\n");
       exit(1);
    }

     /* el resto del programa principal va aquí */
}
```

Programa 2.11

Un proceso también puede ser terminado anormalmente ya sea invocando a la llamada `abort` o procesando una señal que cause la terminación. La señal puede ser generada por un evento externo (como un `ctrl-c` proveniente del teclado) o por un error interno, por ejemplo un intento por tener acceso a una localidad de memoria ilegal. Si un proceso aborta anormalmente, tal vez se produzca un vaciado de memoria (*core dump*), y no se llamará a los manejadores de terminación instalados por el usuario.

2.11 Secciones críticas

Imagine el siguiente escenario. Suponga que un sistema de cómputo cuenta con una impresora a la que tienen acceso directo todos los procesos del sistema. Cada vez que un proceso desea imprimir algo, lo hace con un `write` al dispositivo de impresión. ¿Qué apariencia presentará la salida si más de un proceso intenta escribir en la impresora al mismo tiempo? Recuerde que a cada proceso sólo se le permite un quantum fijo del tiempo del procesador. Si el proceso comienza a escribir, pero lo que debe imprimir es mucho y su quantum termina, entonces se escoge otro proceso. La impresión resultante tendrá mezclada la salida de muchos procesos, lo que constituye una característica poco deseable.

El problema con el escenario anterior es que los procesos intentan simultáneamente acceder a un recurso compartido –un recurso que debe ser utilizado sólo por un proceso a la vez. Esto es, la impresora requiere el *acceso exclusivo* por los procesos del sistema. La parte del código donde cada proceso intenta tener acceso a un recurso compartido recibe el nombre de *sección crítica*. Se debe hacer algo para asegurar la *exclusión mutua* de los procesos mientras éstos ejecutan sus secciones críticas.

Un método para alcanzar la exclusión mutua es utilizar un mecanismo de bloqueo en el que el proceso adquiere un candado que excluye a todos los demás procesos antes de comenzar