Los file pointers y los file descriptors

20-agosto-2003

Nota:

Las siguientes notas son una traducción de una parte del Apéndice B (Low-Level I/O), del libro Advanced Linux Programming de M. Mitchell, J, Oldham y A. Samuel. Fueron tomadas sin permisos del autor, teniendo en cuenta que se utilizaran para fines académicos.

1. Introducción

El objetivo de este documento es que el lector entienda la diferencia entienda la diferencia que existe entre un file pointer y un file descriptor.

2. Leyendo y escribiendo datos

La primera función de entrada-salida que encontraste cuando aprendiste el lenguaje C fue printf. Esta formatea un texto y después lo imprime en pantalla. La versión generalizada, fprintf, puede imprimir el texto. Una corriente es representada por un file pointer. Tú obtienes un file pointer abriendo un archivo con fopen. Después de haberlo hecho, puedes cerrarlo con fclose. Además de fprintf, puedes usar funciones tales como fputc, fputs, y fwrite para escribir datos a la corriente, o fscanf, fgetc, fgets, y fread para leer datos.

Con las operaciones Linux de bajo nivel de entrada-salida, puedes manejar una herramienta llamada "file descriptor", en vez de un "file pointer". Un "file descriptor" es un valor de número entero que se refiere a un caso particular de un archivo abierto en un proceso. Este puede ser abierto para la lectura, para la escritura, o para ambos (para lectura y la escritura).
Un "file descriptor" no tiene que referirse a un archivo abierto; este puede representar una conexión con otro componente de sistema que sea capaz de enviar o recibir información. Por ejemplo, una conexión a un dispositivo de hardware es representada por un "file descriptor", como es un enchufe abierto o un final de un tubo.

Incluye los archivos de cabecera <fcntl.h>, <sys/types.h>, <sys/stat.h>, y <unistd.h> si utilizas cualquiera de las funciones de bajo nivel de entrada y salida descritas.

3. Abriendo un archivo

Para abrir un archivo y producir un "file descriptor" que pueda accesar a ese archivo, usa la "open call". Esta toma como argumentos el nombre de camino del archivo para abrir, como una cadena de caracteres, y banderas que especifican como abrirlo. Puedes usar "open" para crear un archivo nuevo; si lo haces, pasa un tercer argumento, que especifica los permisos de acceso para el archivo nuevo.

Si el segundo argumento es O_RDONLY, el archivo es abierto únicamente para lectura; un error resultará si posteriormente intentas escribir en el "file descriptor". Asimismo O_WRONLY hace que el "file descriptor" sea sólo de escritura. La especificación O_RDWR produce un "file descriptor" que puede ser usado para la lectura y para la escritura. Nota que no todos los archivos pueden ser abiertos en los tres modos. Por ejemplo, los permisos sobre un archivo podrían prohibir un proceso particular para abrirlo para lectura o para escritura; un archivo de sólo lectura, así como un CD-ROM puede no ser abierto para escritura.

Puedes especificar opciones adicionales usando el bitwise o algo de este valor con una o más banderas. Estos son los valores más comúnmente usados:

- **O_TRUNC** para truncar el archivo abierto, si este existió anteriormente. La información escrita en el "file descriptor" reemplazará contenidos anteriores del archivo.

- **O_APPEND** para añadir a un archivo existente. La información escrita en el "file descriptor" será añadida al final del archivo.

- **O_CREAT** para crear un archivo nuevo. Si el nombre del archivo que proporcionas para abrir, no existe, un nuevo archivo será creado siempre y cuando, el directorio que lo contiene exista y que el proceso tenga permiso de crear archivos en aquel directorio. Si el archivo ya existe, este es abierto.

- **O_EXCL** con **O_CREAT** fuerza la creación de un archivo nuevo. Si el archivo ya existe, la llamada para abrirlo, fallará.

Si haces una llamada para abrir un archivo con O_CREAT, proporciona un tercer argumento adicional especificando los permisos para el nuevo archivo. Por ejemplo, el programa B.1 crea un nuevo archivo con el nombre del archivo especificado en la línea de comando. Esta usa la bandera O_EXCL con open, de manera que si el archivo ya existe, un error ocurre. El nuevo archivo da permisos de lectura y escritura para el grupo propietario y lee los permisos para otros.

Listing B.1

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

int main (int argc, char* argv[ ] )
{
  /* The path at which to create the new file. */
  char* path= argv [1];
 /* The permissions for the new file. */
 mode_t mode = S_IRUSR| S_IWUSR| S_IRGRP| S_IWGRP| S_IROTH;

 /* Create the file.  */
 int fd = open (path, O_WRONLY | O_EXCL| O_CREAT, mode);
 if ( fd== -1) {
   /* An error occurred. Print an error message and bail. */
   perror (“open”)
   return 1;
 }
return 0;
```

Aquí está el programa en acción:

```
 % ./create-file testlife
 % ls –1 testfile
- rw - rw - r - -      1 samuel users                0 Feb 1 22:47 testlife
 %./ create-file testlife
 open: File exists
```

Notar que el tamaño del archivo nuevo es 0 porque el programa no escribió ninguna información.

4. Cerrando File Descriptors

Cuando estés trabajando con un “file descriptor”, ciérralo con close. En algunos casos, así como el programa B1, no es necesario llamar explícitamente a close porque Linux cierra todos los “file descriptors” abiertos cuando un proceso termina (esto es, cuando se termina el programa). Desde luego, una vez que cierras el “file descriptor” no deberías usarlo más.

Cerrar un “file descriptor” puede causar que Linux realice una acción en particular, dependiendo de la naturaleza del “file descriptor”. Por ejemplo, cuando cierras un “file descriptor” para un enchufe de red, Linux cierra la conexión de red entre las dos computadoras que se comunican por el enchufe.

Linux limita el número de “file descriptors” abiertos que un proceso puede abrir al mismo tiempo. Los “file descriptors” abiertos usan recursos del kernel, de manera que es bueno cerrar los “file descriptors” una vez que los hayas usado. Un límite típico es 1,024 “file descriptors” por proceso. Puedes ajustar este límite con el setrlimit system call.

5. Escribiendo Información

Escribe datos a un “File descriptor” usando la llamada a write. Proporcione el “file descriptor”, un indicador de un buffer de datos, y el número de bytes a escribir. El “file descriptor” debe ser abierto para la escritura. Los datos escritos al archivo no necesitan(tienen que) ser una cadena de caracteres; escriba copias de bytes arbitrarios del buffer al “file descriptor”.

El programa B.2. añade el tiempo actual al archivo especificado en el comando line. Si el archivo no existe, este es creado. Este programa también usa las funciones time, localtime, y asctime para obtener y formatear el tiempo actual.

Listing B.2 (timestamp.x) Append a Timestamp to a file

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>

/* Return a character string representing the current date and time. */

char* get_timestamp ()
{
  time_t now = time (NULL);
  return asctime (localtime (&now));
}

int main (int argc, char* argv[ ])
{
  /* The file to which to append the timestamp. */
  char* filename = argv[1];
 /* Get the current timestamp. */
  char* timestamp = get_timestamp ();
  /* Open the file for writing. If it exists, append to it;
  otherwise, create a new file. */
  int fd= open (filename, O_WRONLY| O_CREAT| O_APPEND, 0666);
  /* Compute the length of the timestamp string. */
  size_t length = strlen (timestamp);
  /* Write the timestamp to the file. */
  write (fd, timestamp, length);
  /* All done */
  close (fd);
  return 0;
}
```

Aquí está cómo trabaja el timestamp:

```
% ./ timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
% ./timestam tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
Thu Feb 1 23:25:47 2001
```

Notar que la primera vez que invocamos timestamp, este crea el archivo tsfile, mientras que la segunda vez, se añade a éste.

La llamada a write regresa el número de bytes que fueron realmente escritos, o –1 si un error ocurrió. Para algunos tipos de "file descriptors", el número de bytes realmente escritos podrían ser menos que el número de bytes solicitados. En este caso, es recomendable llamar otra vez a write para escribir el resto de los datos. La función en la lista B.3 demuestra como podrías hacer esto. Nota que para algunas aplicaciones, deberías comprobar para condiciones especiales a la mitad de la operación de escritura. Por ejemplo, si escribes a un enchufe de red, tendrás que aumentar esta función para descubrir cual de las conexiones fue cerrada en la mitad de la operación, y así, esto tiene que reaccionar de manera apropiada.

Listing B.3 (write-all.c) Write All of a Buffer of Data

```
/* Write all of COUNT bytes form BUFFER to file descriptor FD.
    Returns –1 on error, or the number of bytes written. */

Ssize_t write_all (int fd, const void* buffer, size_t count)
{
  size_t left_to_write = count;
  while (left_to_write > 0) {
     size_t written = write (fd, buffer, count);
     if (written == -1 )
           /* An error occurred; bail. */
          return –1;
        else
         /* keep count of how much more we need to write. */
         left_to_write -= written;
 }
/* We should have written no more than COUNT bytes! */
assert (left_to_write ==0);
/* The number of bytes written is exactly COUNT. */
return count;

}
```

6. Leyendo Información

La llamada correspondiente para leer información es "read". Como write, esta toma un "file descriptor", un indicador a un buffer y una cuenta. La cuenta especifica cuantos bytes son leídos del "file descriptor" dentro del buffer. La llamada a read regresa –1 sobre error o el número de bytes realmente leídos. Este puede ser más pequeño que el número de bytes solicitados, por ejemplo, si no hay suficientes bytes dejados en el archivo.

Listing B.4 proporciona una demostración simple de read. El programa imprime vertederos hexadecimales de los contenidos del archivo especificado en el comando line. Cada línea despliega la compensación en el archivo y los siguientes 16 bytes.

Listing B.4 (hexdump.c) Print a Hexadecimal Dump of a File

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

int main (int argc, char* argv [ ])
{
  unsigned char buffer [16];
  size_t offset =0;
  size_t bytes_read;
  int i;

  /* Open the file for reading. */
  int fd= open (argv[1], O_RDONLY);

 /* Read from the file, one chunk at a time. Continue until read
   "comes up short", that is, reads less than we asked for.
    This indicates that we have hit the end of the file. */
Do{
/* Read the next line's worth of bytes. */
bytes_read = read (fd, buffer, sizeof (buffer));
/* Print the offset in the file, followed by the bytes themselves. */
printf(" 0x%06x : ", offset);
for (i=0; i< bytes_read; ++1)
   printf ("%02x ", buffer[i]);
 printf ("/n");
/* keep count of our position  in the file. */
offset += bytes_read;
}

while (bytes_read == sizeof (buffer));

/* All done. */
close (fd);
return 0;
}
```

Aquí está hexdump en acción.

```
%./hexdump hexdump

0x000000 :  7f  45  4c  46  01  01  01  00  00  00  00  00  00  00  00  00
0x000010 :  02  00  03  00  01  00  00  00  c0  83  04  08  34  00  00  00
0x000020 :  e8  23  00  00  00  00  00  00  34  00  20  00  06  00  28  00
0x000030 :  1d  00  1ª  00  06  00  00  00  34  00  00  00  34  80  04  08
```

Tu despliegue puede ser diferente, dependiendo del compilador usado para compilar hexdump y las banderas de compilación que especificaste.

7. Moviendote Alrededor del Archivo

Un "file descriptor" recuerda su posición en un archivo. Conforme vas leyendo o escribiendo al "file descriptor" su posición avanza de acuerdo al número de bytes que lees o escribes. Algunas veces, sin embargo, necesitarás mover alrededor de un archivo sin leer o escribir información. Por ejemplo, podrías querer escribir sobre la mitad de un archivo sin modificar el principio, o podrías querer regresar al principio del archivo y releerlo sin reabrirlo.

La llamada lseek te impide reposicionar un "file descriptor" en un archivo. Pasa esto al file descriptor y dos argumentos adicionales especificando la nueva posición.

- Si el tercer argumento es SEEK_SET, lseek interpreta el segundo argumento como una posición, en bytes, desde el principio del archivo.

- Si el tercer argumento es SEEK_CUR, lseek interpreta el segundo argumento como un offset, el cual puede ser positivo o negativo, de la posición actual

- Si el tercer argumento es SEEK_END, lseek interpreta el segundo argumento como un offset del final del archivo. Un valor positivo indica una posición más allá del final del archivo.

La llamada a lseek regresa la nueva posición, como un offset del principio del archivo. El tipo del offset es off_t. Si un error ocurre, lseek regresa –1 . No puedes usar lseek con algunos tipos de file descriptors, como los socket file descriptors.

Si quieres encontrar la posición de un file descriptor en un archivo sin cambiarlo, especifica un 0 offset de la posición actual, por ejemplo:

```
Off_t position = lseek (file_descriptor, 0, SEEK_CUR);
```

Linux te impide usar lseek para posicionar un file descriptor más allá del final del archivo. Normalmente, si un file descriptor es posicionado al final del archivo y escribes al file descriptor, Linux automáticamente expande el archivo para hacer espacio para la nueva información. Si posicionas un file descriptor más allá del final de un archivo y después escribes, Linux expande el archivo para acomodar el "gap" que creaste con la operación lseek y después escribes al final de este. Este gap, sin embargo, no ocupa realmente espacio en el disco; Linux solo hace una nota de que tan largo es este. Si más tarde intentas leer del archivo, este aparece a tu programa que el gap es llenado con 0 bytes.

Usando este comportamiento de lseek, es posible crear archivos extremadamente grandes que casi no ocupan espacio en el disco. El programa lseek-huge en Listing B.5 hace esto. Toma como argumentos de comandos de línea el nombre de un archivo y un tamaño de archivo de llegada, en megabytes. El programa abre un archivo nuevo, avanza más allá el final del archivo usando lseek, y luego escribe el 0 byte antes del cierre del archivo.

Listing B.5 (lseek-huge.c) Create Large Files with lseek

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

int main (int argc, char* argv[ ])
{
  int zero=0;
  const int megabyte = 1024 * 1024;

  char* filename = argv[1];
  size_t length = (size_t) atoi (argv[2]) * megabyte;

 /* Open a new file. */
int fd = open (filename, 0_WRONLY | O_CREAT| O_EXCL, 0666);
/* Jump to 1 byte short of where we want the file to end. */
lseek (fd, length –1, SEEK_SET);
/* Write a single 0 byte. */
write (fd, &zero, 1);
/* All done. */
close (fd);

return 0;

}
```

Usando lseek-huge, haremos un archivo de 1GB (1024MB). Nota el espacio libre en el disco antes y después de la operación.

```
% df –h .
Filesystem                Size      used     avail      use%      mounted on.
/dev/hda5                 2.9G      2.1G     655M       76%       /
%./lseek-huge bigfile     1024
% ls –1 bigfile
- rw-r----------------     1 Samuel      Samuel          1073741824 Feb   5  16:29 bigfile.
% df –h
Filesystem                Size      used     avail      use%      mounted on.
/dev/hda5                 2.9G      2.1G     655M       76%       /
```

Poco espacio en el disco es ocupado, comparado con el enorme tamaño del bigfile. Todavía, si abrimos figfile y lo leemos, parece ser llenado con 1GB. Por ejemplo, podemos examinar sus contenidos con el programa hexdump de Listing B.4.

```
%./hexdump hexdump

0x000000 :  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
0x000010 :  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
0x000020 :  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
0x000030 :  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
0x000040 :  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
0x000050 :  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
```

Si lo corres, probablemente querrás pararlo con Ctrl+C en lugar de observar a que imprima 2 30 0 bytes.
Nota que estas gaps mágicas en los archivos son un rasgo especial del sistema ext2file que típicamente es usado para discos GNU/LINUX. Si intentas usar lseek - huge para crear un archivo sobre algún otro tipo de sistema de archivos, como los sistemas fat o vfat usados para montar el DOS y particiones de Windows, encontrarás que el archivo resultante ocupa en realidad todo el espacio del disco.
Linux no te permite rewind antes del inicio de un archivo con lseek.

8. ¿Cuál es la diferencia entre un File Descriptor y un File Pointer?

(Nota: esta sección es una traducción de parte del Apéndice, Secrets of Programmer Job Interviews, del libro Expert C Programming Deep Secrets de Peter Van Der Linden)

Naturalmente esta pregunta sigue a la anterior. Todas las rutinas de UNIX que manipulan archivos usan tanto un file pointer como un file descriptor para identificar el archivo en el cual están operando. ¿Qué son y cuándo se usan? La respuesta nos da una idea de qué tan familiarizada está una persona con UINIX I/O y las varias compensaciones.

Todas las “System Calls” que manipulan archivos toman (o regresan) un “file descriptor” como argumento. “File Descriptor” es un nombre poco apropiado; el “file descriptor” es un entero pequeño (0-255) usado en la implementación Sun para introducirnos al proceso de tabla de archivos abiertos. Las “System IO calls” son create(), open(), read(), write(), close(), ioctl(), etc. Pero estas no son parte de ANSI C, no existirán en desarrollos UNIX y destruirán la portabilidad del programa si se usan.

De ahí, un juego de “Standard I/O calls” fueron especificadas, de las cuales ANSI C requiere todos los hosts para soportar.

Para asegurar la portabilidad del programa, usa las I/O calls fopen(), fclose(), putc(), fseek(). Todos los nombres de estas rutinas empiezan con “f”. Estas calls toman un indicador para una estructura de archivo (algunas veces llamado indicador de corriente) como argumento. El “file pointer” indica una estructura de corriente, definida en <stdio.h>. Los contenidos de la estructura varian de implementación a implementación, y en UNIX es típicamente una entrada en el proceso de tabla de archivos abiertos. Generalmente este contiene el buffer de corriente, todas las variables necesarias para indicar cuantos bytes en el buffer son datos de archivo reales, banderas para indicar el estado de la corriente (como ERROR y EOF).

- De manera que un “file descriptor” es la compensación (offset) en el proceso de tabla de archivos abiertos. Este es usado en un “system call” de UNIX para identificar el archivo.
- Un “file pointer” contiene la dirección de una estructura de archivo que es usada para representar la corriente abierta de entrada-salida. Esta es usada en una “library call” stdio ANSI C para identificar el archivo.

La función de la librería C fdopen() puede ser usada para crear una nueva estructura de archivo y asociarla con el “file descriptor” especificado (convirtiendo entre un número entero de descriptor de archivo y el indicador de corriente correspondiente, aunque esto genere una entrada adicional nueva en la mesa de archivos abiertos).