El buffer de la entrada estándar en GNU/Linux

Matías Zabaljáuregui (matiasz@info.unlp.edu.ar)
http://linux.linti.unlp.edu.ar

0. Introducción

Se intentará describir en términos sencillos la forma de tratar los datos ingresados por el usuario a través de la entrada estándar de un proceso explicando brevemente el funcionamiento de los buffers de los stream estándar de entrada/salida.
Parte de la información se obtiene del manual de la librería glibc, cuya lectura se recomienda y puede encontrarse en:
http://www.gnu.org/software/libc/manual/

1. Streams Entrada/Salida

Un stream es una abstracción de alto nivel que representa un canal de comunicación hacia o desde un archivo, un dispositivo o un proceso. Este concepto se implementa a través de punteros a estructuras de tipo FILE declaradas en el archivo stdio.h.
Cuando un proceso comienza su ejecución, el sistema operativo abre tres streams: stdin (standard input), stdout (standard output) y stderr (standard error). Generalmente el primero se asocia al teclado y los otros dos al monitor. Ésto puede modificarse al momento de invocar el proceso utilizando las facilidades de redirección y pipes del shell de Linux.

2. Buffers

Los caracteres que escribimos en un stream, normalmente se van acumulando en un buffer (espacio de memoria) para ser transmitidos asincrónicamente hacia el dispositivo o archivo. Por ejemplo, si estamos escribiendo en un archivo de texto, cada vez que enviamos un carácter al stream, este es almacenado temporalmente en un buffer en espacio de usuario, hasta que por algún evento (buffer lleno, carácter de fin de línea, función de limpiado de buffers, etc) se invoca a la llamada al sistema que envía los datos al disco[1].

[1] En realidad, los datos pasarán por toda una jerarquía de buffers antes de ser escritos en el dispositivo. Estos buffers se implementan en los subsistemas de Page Cache, Virtual File System (VFS) y el driver del dispositivo de almacenamiento, todos incluidos en el kernel Linux.

Del mismo modo, cuando ingresamos caracteres del teclado u otro dispositivo de entrada, normalmente estos se van acumulando en un buffer en espacio de usuario (previo buffering en espacio del kernel aunque hay algunas excepciones a esto). Cuando se llama a una función de lectura sobre stdin, en realidad estamos leyendo el contenido de este buffer, como se observa en el siguiente diagrama:

fgets (miBuffer, 15, stdin) (espacio de usuario)
^
^
buffer de stdin (espacio de usuario)
^
^
driver / **subsistema TTY** (kernel)
^
^
dispositivo de entrada (hardware)

3. Estrategias de Buffering

Existen tres estrategias diferentes de buffering:

- Unbeffered Stream: Los caracteres escritos o leídos desde el stream son transmitidos individualmente hacia o desde el dispositivo/archivo tan pronto como sea posible.
- Line Buffered Stream: Los caracteres escritos al stream son transmitidos en bloques al archivo recién cuando se escribe un carácter newline ('\n').
- Fully Buffered Stream: Los caracteres escritos o leídos desde el stream son transmitidos hacia o desde el dispositivo/archivo en bloques de tamaño arbitrario.

Al crear un stream, éste normalmente utiliza una estrategia Fully Buffered. Sin embargo existe una excepción importante: los stream de salida asociados a un dispositivo interactivo como una terminal, los cuales responden a una estrategia Line Buffered. Esto implica que los mensajes de salida que terminen con el carácter newline deberían aparecer inmediatamente en pantalla.

Al principio este proceso suele resultar un tanto engorroso por el hecho de que en realidad se está realizando una operación tan simple como por ejemplo la lectura de caracteres del teclado. Sin embargo debería tenerse en cuenta que la interfaz ofrecida por los streams nos permite el acceso a todo tipo de hardware [2] y la posibilidad de optimizar el mecanismo de buffering subyacente puede implicar mejoras importantes de la performance.

[2] Se suele decir que en la familia de sistemas operativos tipo Unix todo es un archivo. Por ejemplo, el código para enviar una serie de caracteres a un archivo de texto, al puerto serie o al monitor es el mismo.

4. Flush y Purge

Hacer un flush de la salida en el buffer de un stream significa transmitir al archivo/dispositivo todos los caracteres acumulados en el buffer. Existen varios eventos que disparan una operación de flush automáticamente, pero cuando necesitamos invocarla explícitamente, se utiliza la función *fflush* declarada en *stdio.h:*

int ***fflush*** *(FILE *stream)*

> *This function causes any buffered output on stream to be delivered to the file. If stream is a null pointer, then* `fflush` *causes buffered output on all open output streams to be flushed.*
>
> *This function returns EOF if a write error occurs, or zero otherwise.*

En algunas situaciones donde la información en el buffer no es útil y simplemente se puede desechar es posible utilizar una función que fue introducida en Solaris y que si bien **NO es estándar**, se incluye en la librería glibc. Notar que es necesario incluir el archivo *stdio_ext.h*, en cuyo caso ya no será necesario incluir *stdio.h*.

void ***__fpurge*** *(FILE *stream)*

> *The* `__fpurge` *function causes the buffer of the stream stream to be emptied. If the stream is currently in read mode all input in the buffer is lost. If the stream is in output*

mode the buffered output is not written to the device (or whatever other underlying storage) and the buffer the cleared.

This function is declared in `stdio_ext.h`.

5. Ejemplos de cómo limpiar el buffer de teclado

Ejecutando el siguiente código:

```
#include <stdio.h>
int main()
{
     char nombre[50];
     printf("Nombre: ");
     scanf("%s", nombre);
     printf("%s", nombre);
     printf("-newline\n");
     return 0;
}
```

Se puede observar que scanf deja el carácter newline en el buffer de stdin en lugar de copiarlo a nombre. Este es uno de los casos en que en las próximas llamadas a scanf (u otra función) tendremos comportamientos extraños. Pero no todas las funciones de lectura se comportan igual, por ejemplo:

```
#include <stdio.h>
int main()
{
     char nombre[50];
     printf("Nombre: ");
     fgets(nombre,sizeof(nombre),stdin);
     printf("%s", nombre);
     printf("-newline\n");
```

```
	return 0;
}
```

Este código demuestra que fgets no deja el carácter newline en el buffer de stdin, en cambio lo copia a nombre. Sin embargo, si ingresamos por teclado un string de longitud mayor o igual a 50 tendremos problemas si luego leemos nuevamente de stdin ya que fgets lee caracteres del buffer hasta uno menos que la cantidad indicada en su segundo argumento (en este caso lee 49 y en la última posición coloca el carácter '\0', o fin de string).

Por ejemplo, si se compila el siguiente código

```
#include <stdio.h>
int main()
{

	char nombre[5];
	printf("Nombre: ");
	fgets(nombre,sizeof(nombre),stdin);
	printf("%s", nombre);
	printf("-newline\n");
	fgets(nombre,sizeof(nombre),stdin);
	printf("%s", nombre);
	printf("-newline\n");
	return 0;
}
```

y se ingresa un string de 10 caracteres en la primer oportunidad que nos da el programa, el segundo fgets toma el remanente dejado en el buffer por el primer fgets.

A continuación se presentan dos ejemplos de cómo se puede limpiar el buffer del stream

stdin en GNU/Linux. Es importante recordar que *fflush()* no está definida para stdin, sino para buffers de salida como el de stdout.

En el primer ejemplo se utiliza un loop para terminar de leer caracteres que pueden haber quedado en el buffer por ingresos previos de caracteres. Este es el código

```
#include <stdio.h>
int main()
{
     char nombre[5];
     printf("Nombre: ");
     fgets(nombre,sizeof(nombre),stdin);
     printf("%s", nombre);
     printf("-newline\n");
     printf("Apellido: ");
     while(getchar() != '\n');/*LEE DEL BUFFER HASTA QUE ENCUENTRA
     fgets(nombre,sizeof(nombre),stdin);    UN CARACTER NEWLINE*/
     printf("%s", nombre);
     printf("-newline\n");
     return 0;
}
```

Es posible definir una macro que implemente el loop, como se hace a continuación:

```
#define FLUSH  while(getchar() != '\n')
```

El segundo ejemplo utiliza la función *__fpurge()* explicada en el punto 4. Debe destacarse que __fpurge no es estandar y que si la portabilidad del código que estamos generando es un factor importante, se debería utilizar la solución anterior, sin embargo este código funciona sin problemas en GNU/Linux (asi como en otras variantes de Unix).

```
#include <stdio_ext.h>
{
     char nombre[5];
     printf("Nombre: ");
     fgets(nombre,sizeof(nombre),stdin);
     printf("%s", nombre);
     printf("-newline\n");
     printf("Apellido: ");
     __fpurge(stdin);   /*BORRA EL CONTENIDO DEL BUFFER DE STDIN*/
     fgets(nombre,sizeof(nombre),stdin);
     printf("%s", nombre);
     printf("-newline\n");
     return 0;
}
```