Laboratorio de
Sistemas Operativos

Práctica 7: Estrategias de I/O

Objetivo

En esta práctica nos enfocaremos en los elementos básicos para el desarrollo de aplicaciones de sistemas para ambiente GNU/Linux utilizando el lenguaje nativo del mismo: C.

Índice

Desarrollo de programas usando Programación de sistemas

Programación a nivel-Sistemas Linux (Linux System-Level Programming or Linux System Programming) es el arte de escribir **software del Sistema**.

- **Software del sistema:** es aquel que está en contacto directo con el manejo del kernel y las bibliotecas del sistema. El desarrollo de software de sistema puede requerir que el usuario conozca sobre el hardware en el que va a trabajar el software. El sofware de sistema suele escribirse en el lenguage C, con algunas rutinas de eficiencia muy crítica

directamente en ensamblador

Ejemplos de software de sistema incluyen:
- El sistema operativo
- Los interpretadores de comandos
- Los sistemas de interface gráfica
- Los compiladores, ensambladores, controladores de versión, depuradores, etc.
- Los drivers de dispositivos de entrada/salilda

- **Software de Aplicaciones** (***Application Software***) : pues es todo el resto del software. Un **software de aplicación** es un programa que realiza una función específica directamente con el usuario.
  Ejemplos incluyen:
  - Hojas de cálculo, proceso de textos, presentaciones…
  - Apps en web y en móviles
  - Programas científicos
  - Aplicaciones para empresas como nóminas, inventarios, SAP y similares

La programación a nivel sistema se hace a través de **llamadas al sistema (*system calls or supervisor calls*)**, que pueden invocarse a través de proposiciones en lenguages de más alto nivel, como el lenguaje C

Las **llamadas al sistema** son funciones que se invocan desde nivel usuario hacia el kernel para requerir un servicio o recurso del sistema operativo.

Llamadas al sistema desde C incluyen
- Sistemas de Archivo: **open()**, **read()**, **write()**, **close()**, etc.
- Procesos; **fork()**, **getpid()**, **set_thread_area()**, etc.
- Red: **socket()**, **bind()**, etc.

Cada Sistema Operativo define diversas llamadas al sistema. En el caso de Linux el número de llamadas al sistema son aproximadamente unas 300.

NOTA: (En Windows, pueden consultar una lista de llamadas al sistema en esta liga.)

En esta práctica nos enfocaremos en llamadas al sistema desde el lenguaje C, relacionadas con estrategias de entrada/salida.

---- Algunas llamadas a sistema---

La documentación completa de estas llamadas a está en los manuales del programador (“man pages”). En algunas distribuciones de linux estas páginas podrían no haber sido instaladas automáticamente; se deben instalar los paquetes man-db y manpages-dev

➔ Abrir archivos:

Antes que los datos de un archivo se puedan leer, éste debe de ser abierto utilizando las llamadas a sistema **open()** o **create()**.

```
int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);
```

- Si se puede abrir exitosamente, regresa un *descriptor de archivo --file descriptor o fd*. El descriptor apunta internamente a la posición actual del archivo, inicialmente el principio (posición cero)
- El archivo se abre de acuerdo al contenido del parámetro “flags”.
- Recuerden que en linux cualquier elemento que pueda operarse por medio de inputs y outputs se considera archivo. “todo es un archivo”, no solo los archivos en disco. El teclado es un archivo que sólo admite input; la pantalla es un archivo de “output”; una conexión de red entre dos computadoras también se considera un archivo, asi como un “pipe” y stdin y stdout y un directorio ETC

La variable **“flags”** puede contener diferentes valores dependiendo de la operación que deseamos realizar sobre el archivo: *O_RDONLY*, *O_WRONLY*, or *O_RDWR*.

La variable **modo** puede tomar los siguientes valores: *O_APPEND*, *O_CREAT*, *O_EXCL* ú *O_TRUNC*. La información detallada se encuentra en **man open** .

La definición de estas funciones, así como las constantes utilizadas en los “flags”, se pueden encontrar en los siguientes archivos de encabezados:

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

Lectura mediante read( )

El mecanismo más básico y comúnmente usado para leer, es la llamada a sistema **read()**, se define:

```
#include <unistd.h>
size_t read (int fd, void *buf, ensize_t l);
```

Cada llamada lee hasta una cantidad de *len* bytes a partir de la posición actual referenciada por el descriptor de archivo *fd*, y los guarda en *buf.* Regresa el número de bytes escritos en *buf*, o -1 si ocurre un error. Los tipos de dato “ssize_t” y “size_t” están definidos en unistd.h y son equivalentes a un integer.

Escribiendo con write( )

La llamada a sistema más básica y común es **write()**. Se define:

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

La llamada a **write()** escribe una cantidad de *count* bytes a partir de la posición actual referenciada por *fd*. Regresa el número de bytes escritos, o -1 en caso de error.

Cerrar archivos

Cuando un programa ha terminado de trabajar con un file descriptor, debe desasociar el file descriptor del archivo mediante la llamada a sistema **close()**:

```
#include <unistd.h>
int close (int fd);
```

Regresa un 0 en caso de éxito, o un -1 en caso de error.

--- Un ejemplo de llamadas al sistema ---

Este ejemplo es una codificación simple del comando **cp** para copiar archivos en disco:

```
/* PROGRAMA: copiar.c
FORMA DE USO:
             ./copiar origen destino
VALOR DE RETORNO:
             0: si se ejecuta satisfactoriamente.
             -1: si se da alguna condición de error
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>              /* open() */
#include <sys/stat.h>              /* open() */
#include <fcntl.h>                /* open() */
#include <unistd.h>              /* read() y write() */

char buffer[BUFSIZ]; /* Buffer para manipular datos. BUFSIZ esta definido en alguno de
los archivos de encabezados */

int main(int argc, char *argv[])
{
  int fd_origen;
  int fd_destino;
  int nbytes;

  /*Analisis de los argumentos de la linea de comandos*/
  if (argc != 3) {
    fprintf(stderr,"Forma de uso: %s origen destino\n", argv[0]);
    exit(-1);
  }

  /*Apertura del archivo en modo solo lectura (O_RDONLY). */
  if ((fd_origen=open(argv[1],O_RDONLY))== -1) {
    perror(argv[1]);
    exit(-1);
  }

  /* Apertura o creacion de archivos en modo solo escritura. */
  /* Abrir en modo solo escritura (O_WRONLY). */
  /* Si el archivo existe entonces truncar a cero bytes (O_TRUNC).*/
  /* El archivo es creado en caso de que no exista (O_CREAT). */
  /* El modo que se crea es 0666. */
  if ((fd_destino=open(argv[2],O_WRONLY|O_TRUNC|O_CREAT,0666))== -1) {
    perror(argv[2]);
    exit(-1);
```

```
  }

  /* copiamos el archivo origen en el archivo destino. */
  while ((nbytes=read(fd_origen, buffer, sizeof buffer))> 0)
     write(fd_destino, buffer, nbytes);

  close(fd_origen);
  close(fd_destino);
}
```

A la hora de ejecutar, observe que cuando no hay error, el programa envía al sistema operativo el valor 0, y si hay error el programa envía al sistema operativo el valor de -1. El sistema operativo toma el valor y lo almacena en la variable **$?**. Entonces, el SO puede utilizar esta información en un programa shell para tomar decisiones. A continuación se muestra el ejemplo de un caso con error:

```
user@localhost ~$ ./copia copiar.c copiar2.c
user@localhost ~$ echo $?
0
user@localhost ~$ ./copia copiar3.c copiar2.c
copiar3.c: No such file or directory
user@localhost ~$ echo $?
255
```

File I/O: writeback , Synchronized I/O

Comportamiento del write (writeback)

La disparidad en el rendimiento entre procesadores y discos duros haría que para efectos de rendimiento se retrase la escritura. Cuando una aplicación en el espacio del usuario realiza una llamada al sistema **write(),** el kernel de Linux realiza un par de revisiones, y entonces simplemente copia los datos en un buffer interno al sistema operativo (distinto del buffer utilizado en el programa) Después, en background, el kernel toma todos los buffers “sucios” de todos los usuarios (es decir, que no han sido aún escritos a disco), los ordena óptimamente (de tal forma que la cabeza escritora del disco duro minimice su trayectoria), y los escribe en el disco. **Este proceso se conoce como writeback.** Esto permite al kernel aplazar las escrituras a periodos más desocupados y procesar en lote muchas escrituras juntas, optimizando el tiempo de escritura a disco.

Este comportamiento realmente mejora el rendimiento, de la misma forma en que un **read** puede leer de la memoria caché sin tener que ir al disco. Las peticiones de read y write se intercalan según lo previsto, y los resultados son los esperados – eso sí el sistema no tiene una falla antes de que los datos lleguen al disco...

- Aún cuando una aplicación pueda creer que se realizó la escritura con éxito, en caso de falla los datos que están en el bufer interno podrían no haber llegado al disco, por ejemplo si “se fué la luz” y no hay respaldo de batería en el servidor linux.
- Otro problema con las escrituras retrasadas es que pudiera no ser posible cumplir el

orden de escritura en caso de falla

- Un problema final con las escrituras retrasadas tiene que ver con el reporte de ciertos errores de I/O. Cualquier error de I/O que ocurra durante la escritura a disco —por ejemplo, una falla física de disco— no puede ser reportada al proceso que emitió la petición de escritura.

El kernel trata de minimizar los riesgos de estas escrituras diferidas. Para asegurar que los datos son escritos oportunamente, el kernel establece una edad máxima del buffer, y escribe a disco todos los buffers "sucios" antes de que maduren más allá del tiempo dado. Los usuarios pueden configurar este valor en el archivo /proc/sys/vm/dirty_expire_centisecs. El valor está especificado en centésimas de segundo.

Synchronized I/O

Aunque la sincronización de I/O es un tema importante, los problemas asociados con las escrituras retrasadas no deben ser temidos. Las escrituras mediante buffers proporcionan una mejora enorme en el rendimiento, y consecuentemente, cualquier sistema operativo que merezca la etiqueta de "moderno", implementa escrituras diferidas mediante buffers internos. Sin embargo, hay ocasiones en las que las aplicaciones desean controlar de manera precisa el momento en que los datos llegan al disco. Para esos usos, el kernel de Linux proporciona varias opciones que permiten negociar el rendimiento por medio de las operaciones sincronizadas.

fsync() asegura que todos los datos sucios asociados con el archivo mapeado por el file descriptor *fd* sean escritos al disco en el momento en que se ejecuta esa llamada. Menos óptimo, pero de más amplio alcance, la llamada a sistema **sync()** se proporciona para sincronizar todos los buffers al disco.

Otra posibilidad es asignar la bandera *O_SYNC* en la llamada a sistema **open()**, indicando que todas las operaciones I/O deben ser sincronizadas. Por ejemplo, en el programa *copiar.c*, cuando el archivo de escritura es abierto:

```
if ((fd_destino=open(argv[2],O_WRONLY|O_TRUNC|O_CREAT,0666))== -1)
```

podríamos incluir la bandera *O_SYNC* de la siguiente forma:

```
if ((fd_destino=open(argv[2],O_WRONLY|O_TRUNC|O_CREAT|O_SYNC, 0666))== -1)
```

➔ Lecturas predictivas

"Readahead" se refiere a la acción de leer más datos del disco y de la página de caché en la que se hizo la petición de lectura, en efecto, leyendo un poco hacia adelante. Esta acción es automática en el kernel de linux, que asume que en la mayoría de los casos la lectura de un archivo es secuencial.

➔ I/O usando buffers (lenguaje C estándar)

El rendimiento de I/O en disco es óptimo cuando las peticiones se realizan entre límites alineados por bloques y de tamaño múltiplo entero del tamaño de bloque. En la práctica, los

tamaños de bloque normalmente son 512, 1,024, 2,048, ó 4,096. En consecuencia, la elección más sencilla es hacer operaciones de I/O usando buffers grandes que sean múltiplos de los tamaños típicos, por ejemplo 4,096 o 8,192 bytes.

El problema, por supuesto, es que los programas rara vez efectúan estas operaciones en términos de bloques. Los programas trabajan con campos, líneas o caracteres, no con abstracciones como los bloques. Para remediar esta situación, los programas pueden utilizar un segundo conjunto de llamadas de I/O que utilizan buffers de usuario declarados por el programador; estas llamadas (**fread** por ejemplo en vez de **read**) invocan a su vez de manera transparente a reads en términos de bloques, en caso de que la lectura sea a disco.

La biblioteca estándar de C provee la librería estándar de I/O (comúnmente llamada *stdio*), la cual a su vez provee una solución de utilización de buffers para usuarios, de forma independiente a la arquitectura sobre la que corra el sistema. Estas rutinas de lectura y escritura utilizando buffers de usuarios pueden utilizarse incluso no solo para escribir a disco, sino para escribir a cualquier dispositivo futuro o incluso para escribir a “sockets” de red y así enviar “writes” y “reads” provenientes de otros programas, posiblemente residentes en otras máquinas de la red.

Que una aplicación use las funciones estándar de I/O o los accesos directos a las llamadas de sistema es una decisión que el desarrollador debe tomar muy cuidadosamente después de sopesar las necesidades de la aplicación y su comportamiento.

Las funciones estándar de I/O no trabajan directamente en los descriptores de archivos fd. En vez de eso, usan su propio identificador único, conocido como "apuntador a archivo". Dentro de la biblioteca de C, el apuntador a archivo es traducido a un descriptor de archivo.

➔ Apuntador a archivo

Es un apuntador al tipo de dato FILE, definido en <stdio.h>, que representa el apuntador a archivo. Hablando del estándar I/O , un archivo abierto es llamado "stream". Estos Streams pueden abrirse para escritura, lectura o ambos.
Para abrir un archivo se utiliza el la función **fopen()** y para cerrarlo **fclose().**

➔ Leyendo Streams

La biblioteca estándar de C implementa diferentes funciones para leer de un stream abierto.

➔ Leyendo un caracter a la vez.

La función **fgetc()** se usa para leer un sólo caracter del stream.

➔ Leer una línea completa.

La función **fgets()** lee una cadena de caracteres (string) del stream.

➔ Leer datos binarios.

Para algunas aplicaciones, leer caracteres individuales o líneas no es suficiente. Algunas veces, los desarrolladores quieren leer y escribir datos binarios complejos, por ejemplo estructuras. Para esto, la biblioteca estándar I/O provee **fread().**

Para mayor referencia sobre estas funciones, consultar los manuales de cada función.

➔ Escribiendo a un stream

La biblioteca estándar de C implementa varias funciones para escribir en un stream abierto.

➔ Escribiendo un carácter a la vez.

La contraparte de fgetc() es fputc():

➔ Escribiendo una cadena de caracteres.

La función **fputs()** es usada para escribir una cadena de caracteres a un stream dado.

➔ Ejemplo de un programa que usa I/O en buffers.

Como ejemplo, se va a reescribir la función copiar.c usando buffers de I/O.

```
/* PROGRAMA: fcopiar.c
  FORMA DE USO:
  fcopiar origen destino
  VALOR DE RETORNO:
0: si se ejecuta satisfactoriamente.
-1: si se da alguna condicion de error
*/
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char *argv[])
{
	FILE *forigen;
	FILE *fdestino;
	char c;

	/*Analisis de los argumentos de la linea de comandos*/
	if (argc != 3) {
		fprintf(stderr,"Forma de uso: %s origen destino\n", argv[0]);
		exit(-1);
	}

	/*Apertura del archivo en modo solo lectura*/
	if ((forigen=fopen(argv[1],"r"))== NULL) {
		perror(argv[1]);
		exit(-1);
	}

	/* Apertura o creacion de archivos en modo solo escritura*/
	if ((fdestino=fopen(argv[2],"w"))== NULL) {
		perror(argv[2]);
		exit(-1);
	}
```

```
	/* copiamos el archivo origen en el archivo destino. */
	while ((fread(&c, sizeof c, 1, forigen))> 0)
		fwrite(&c, sizeof c, 1, fdestino);

	fclose(forigen);
	fclose(fdestino);
	return 0;
}
```

Después de compilar el archivo, puedes comparar con el comando "time" el tiempo que tarda en ejecutarse el programa fcopiar contra el programa copiar. Nota que en fcopiar, estamos leyendo y escribiendo un caracter a la vez en nuestro programa (sizeof c = 1), por lo cual podría pensar que fcopiar va a ser más lento, Sin embargo eso no es así, pues gracias a la lectura y escritura predictivas, internamente linux está utilizando buffers del tamaño de un bloque de disco!.

Mapeo a Memoria (Memory map)

Los archivos mapeados a memoria son una copia idéntica en memoria de un archivo en disco. La imagen corresponde byte por byte con el archivo en disco. Los archivos mapeados a memoria tienen dos principales ventajas:

- Las operaciones de I/O sobre archivos mapeados a memoria evitan los buffers del kernel, por lo tanto las transferencias I/O son mucho más rápidas. Nota: ya que el kernel tiene un overhead, esta tasa de transferencia es más notoria con archivos grandes.
- Aparte de los posibles fallos de página, leer de y escribir en un archivo mapeado a memoria no incurre en ninguna llamada a sistema o sobrecarga por cambio de contexto. Es tan simple como accesar memoria.
- Los datos del archivo mapeado pueden ser accedidos internamente mediante apuntadores en lugar de las funciones comunes para la manipulación de archivos. Además, buscar en el mapeo involucra manipulaciones triviales de apuntadores. No hay necesidad de la llamada a sistema lseek().

Por estas razones, mmap() es una opción inteligente para muchas aplicaciones.

Hay un par de puntos a mantener en mente cuando se usa mmap():

- Los mapeos a memoria tienen siempre un tamaño que es múltiplo entero de páginas. Así, la diferencia entre el tamaño del archivo y el número entero de páginas es desperdiciado como espacio sobrado. Para archivos pequeños, un porcentaje significativo del mapeo podría ser desperdiciado. Por ejemplo: con páginas de 4KB, un archivo de 7 bytes mapeado desperdicia 4,089 bytes.

- Los mapeos a memoria debe ajustarse dentro del espacio de direcciones del proceso. Con un espacio de direcciones de 32 bits, un número muy grande de mapeos de diferentes tamaños podría resultar en la fragmentación del espacio de direcciones, haciendo difícil encontrar grandes regiones libres contiguas. Este problema, por supuesto, es mucho menos aparente con espacios de direcciones de 64 bits.

- Hay un overhead al crear y mantener los mapeos a memoria y las estructuras de datos asociadas dentro del kernel. Este overhead es obviado generalmente por la eliminación de la doble copia mencionada en la siguiente sección, particularmente para archivos grandes y frecuentemente accedidos.

Por estas razones, los beneficios de **mmap()** son extremadamente notorios cuando el archivo mapeado es grande (y así cualquier espacio desperdiciado es un pequeño porcentaje del mapeo total), o cuando el tamaño total del archivo mapeado es divisible por el tamaño de página (y así no hay desperdicio de espacio).

POSIX.1 estandariza—y Linux implementa—la llamada a sistema **mmap()** para mapear objetos en memoria. Una llamada a **mmap()** pide al kernel mapear **len** bytes del objeto representado por el file descriptor **fd** , iniciando a partir de **Offset** bytes en el archivo, en memoria. Si se incluye **addr** , se indica la preferencia de usar una dirección de inicio en memoria. Los permisos de acceso son dictados por **prot** , y el comportamiento adicional puede ser dado por **flags** (Consultar el manual de mmap).

A continuación se reescribe el programa copiar utilizando las funciones **mmap()** y **unmap()**.

```
/* PROGRAMA: copiar_mm.c (utilizando mapeo de memoria)
  FORMA DE USO:
./copiar_mm origen destino
  VALOR DE RETORNO:
0: si se ejecuta satisfactoriamente.
-1: si se da alguna condicion de error
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
   int fd_origen;
   int fd_destino;
   int nbytes;
   void *src;
   struct stat statbuf;

   /*Analisis de los argumentos de la linea de comandos*/
   if (argc != 3) {
          fprintf(stderr,"Forma de uso: %s origen destino\n", argv[0]);
         exit(-1);
   }

   /*Apertura del archivo en modo solo lectura*/
   if ((fd_origen=open(argv[1],O_RDONLY))== -1) {
         perror(argv[1]);
         exit(-1);
   }

   /* Apertura o creacion de archivos en modo solo escritura*/
   if ((fd_destino=open(argv[2],O_WRONLY|O_TRUNC|O_CREAT, 0666))== -1) {
          perror(argv[2]);
         exit(-1);
   }
```

```
    /* Obtiene la longitud del archivo de lectura. */
    fstat(fd_origen, &statbuf);

    /*Mapea el archivo de entrada. */
    if((src = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED, fd_origen, 0))<0)
           err_quit("mmap");

    nbytes = statbuf.st_size;

    /* Escribe el archivo de memoria a disco. */
    write(fd_destino, src, nbytes);

    close(fd_origen);
   munmap(src, statbuf.st_size);
    close(fd_destino);
    return 0;
}
```

Note que este programa utiliza las mismas funciones read y write ya conocidas para leer y escribir archivos mapeados a memoria.

Con el propósito de comparar usaremos un archivo >1GB llamado 100wrds. A continuación se presentan, en la *Tabla 1,* los tiempos de ejecución de 100 corridas de **fcopiar** y **copiar_mm**, así como el tiempo promedio. También se incluyen los resultados cuando se ejecuta el comando **cp**, el código que utiliza llamadas al sistema, **copiar**, y el código cuando se fuerza escribir a disco duro, **copiar_fw**. Se puede observar que el doble buffering puede tener efectos peores que forzar la escritura en disco.

```
user@localhost ~ $ time ./copiar_mm ./100wrds ./wrds
```

corrida	fcopiar	copiar_mm	copiar_fw	copiar	cp
1	27.27	1.46	27.09	1.03	1.23
2	26.93	1.18	25.21	1.39	1.34
3	27.25	1.64	25.13	1.37	1.27
4	27.06	0.79	26.34	1.86	1.25
5	27.15		1.125.09	1.54	1.45
6	26.82	0.81	25.34	0.95	1.29
7	26.88	1.25	25.09	0.65	1.26
8	27.29	1.21	25.67		11.21
9	27.83	1.02	25.18	1.31	0.99
10	27.29	1.26	25.64	1.32	1.51
Promedio	**27.177**	**1.172**	**25.578**	**1.242**	**1.28**

Tabla 1. Comparación entre los tiempos de ejecución de las variantes del programa copiar.c

Críticas a la Biblioteca I/O Estándar.

Tan ampliamente usado como es el I/O Estándar, algunos expertos apuntan a fallas en él. Algunas de las funciones, como **fgets()**, son ocasionalmente inadecuadas. La queja más grande con I/O estándar es el impacto en el rendimiento originado por la doble copia. Cuando se leen datos, el I/O estándar emite una llamada a sistema **read()** para el kernel, copiando los

datos del kernel al buffer I/O estándar. Cuando una aplicación emite una petición de lectura mediante I/O estándar—por decir, usando **fget()**—los datos son copiados de nuevo, en esta ocasión del buffer I/O estándar al buffer proporcionado. Las peticiones de escritura trabajan en dirección contraria: los datos son copiados una vez del buffer proporcionado al buffer I/O estándar, y entonces después del buffer I/O estándar al kernel mediante **write()**.

Una implementación alternativa podría evitar la doble copia al hacer que cada petición de lectura regrese un apuntador al buffer I/O estándar. Los datos podrían entonces ser leídos directamente, dentro del buffer I/O estándar, sin la necesidad de una copia extraña. En el evento de que la aplicación quisiera que los datos se almacenarán en su propio buffer local—quizás escribir a él—podría siempre realizar la copia manualmente. Esta implementación proveería una interfaz "libre", permitiendo a las aplicaciones indicar cuando se apropian de un pedazo del buffer de lectura. Las escrituras serían un poco más complicadas, pero aún se evitaría la doble copia. Cuando se emite una petición de escritura, la implementación almacenaría el puntero. Al final, cuando se está listo para vaciar los datos al kernel, la implementación podría recorrer su lista de apuntadores almacenados, escribiendo a disco los datos. Esto podría ser hecho usando I/O de dispersión-reunión, vía **writev()**, y por lo tanto solo una llamada a sistema. Existen bibliotecas user-buffering altamente óptimas, que resuelven el problema de la doble copia con implementaciones similares a las que se han discutido. Alternativamente, algunos desarrolladores eligen implementar sus propias soluciones. Pero a pesar de estas alternativas, I/O estándar permanece popular.

Implicaciones del *buffering* del Estándar I/O:

- Mejores que las llamadas a sistema para peticiones pequeñas o des-alineadas—debido al buffering de flujo.
- Cuando se leen caracteres use bibliotecas estándar.
- Cuando se lean bloques use llamadas a sistema (por la doble copia).
- El ancho de banda es limitado al copiar datos al buffer de flujo.
- Para peticiones grandes, **fwrite** evita el buffer.
- **fread** no evita el buffer.

= = = = Laboratorio = = = =

- Actividades a realizar en esta práctica se encuentran descritas en este documento, sus respuestas deben registrarse en :

E**nlace al formulario:** Práctica 7: Estrategias de I/O .

En una compañía de desarrollo de software libre se tiene un programa escrito C denominado fusionar.c, que sirve para fusionar archivos, es decir que agregue el contenido de un archivo al contenido de otro, y guarde el resultado en un archivo de salida. La sintaxis para su uso debe ser la siguiente:

```
./fusionar <archivo1> <archivo2> <salida>
```

Sin embargo, por tragedias que involucran un CPU abierto durante el mantenimiento y malas

prácticas sobre dónde colocar un refresco de cola, el disco duro que contenía el código se daño y solo se consiguió recuperar fragmentos del código fuente original. Es su deber reescribir el código partiendo de lo que se recuperó y que se despliega en la tabla de más abajo. **Como buen desarrollador**, entregará un archivo de parche diferencial con los cambios en las líneas, es decir, utilizará la herramienta **diff**. Para indicar cuales son las líneas restauradas y su nuevo valor.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/____.h>                    /* open() */
#include <sys/____.h>                    /* open() */
#include <____.h>                      /* open() */
#include <____.h>                      /* read() y write() , close() */

char buffer[BUFSIZ]; /*Buffer para manipular datos. */

int main(int argc, char *argv[])
{
   int fd_origen;  int fd_destino;  int nbytes;
   int i;

   /*Analisis de los argumentos de la linea de comandos*/
   if (argc <= ____) {
     fprintf(stderr,"Forma de uso: %s  ______ \n", argv[____]);
         exit(____);
   }

   /* Apertura o creacion de archivos en modo solo escritura. */
   /* Abrir en modo solo Lectura (O_WRONLY). */
   /* Si el archivo existe entonces truncar a cero bytes (O_TRUNC)*/
   /* El archivo es creado en caso de que no exista (O_CREAT). */
   /* El modo que se crea es 0666. */
   if ((fd_destino= ____(argv[____-1],  O_WRONLY|O_TRUNC|O_CREAT,  0666))==-1) {
          fprintf(stderr,"Error al crear el archivo de salida \n");
         perror(argv[3]);
         exit(____);
   }

   for(____;i<3;i++)  {

   /*Apertura del archivo 1 en modo solo lectura (O_RDONLY). */
   if ((fd_origen=open(____,O_RDONLY))== -1) {
          fprintf(stderr,"Error al abrir el archivo de entrada: %s ¥n", ____);
          perror(argv[1]);
          exit(____);
   }

   /* copiamos el archivo 1 en el archivo destino. */
   while ((nbytes=read(____, buffer, sizeof buffer))> 0)
         write(____, buffer, nbytes);
         close(____);
   }

   close(____);
}
```

➔ Puedes ver un ejemplo de la ejecución de este programa:

```
user@localhost ~$ ls
```

```
body.c       fusionar.c       fusionar       headers.c
user@localhost ~$ cat headers.c
#include <stdio.h>
user@localhost ~$ cat body.c
const char *msg = "Hello Earthlings!";

int main ()
{
   printf ("Message from Mars: %s\n", msg);

   return 0;
}
user@localhost ~$ ./fusionar headers.c body.c secretProgram.c
user@localhost ~$ ls
body.c       fusionar.c       fusionar       headers.c
       secretProgram.c
user@localhost ~$ gcc -Wall -o secretProgram secretProgram.c
user@localhost ~$ ./secretProgram
Message from Mars: Hello Earthlings!
```

Además de este código original se pretende recuperar archivos que se perdieron de forma permanente los cuales tienen las siguientes características:

1. Una versión modificada de fusionar.c que fusione cualquier número de archivos que le des como argumento, por ejemplo: que fusione todos los archivos con extensión .c en un solo archivo:

   ./fusionar_todo *.c todo.c

2. Una modificación de fusionar.C que se llame **fusionar_sync.c** cuyo objetivo sea forzar la escritura al disco (Synchronized I/O). Si esto les suena a lengua klingon le sugerimos LEA LA PRÁCTICA (Como todo buen programador que primero reúne herramientas antes de tocar código).

* Reporta los cambios realizados en cada código utilizando la herramienta diff:
  diff -Naur fusionar_orig.c fusionar.c

Referencias

[1] Love, Robert. *Linux System Programming*, O'REILLY, 2007. Biblioteca: QA76.76.O63 L69 2007
[2] Marquez, Fernando. *Programación Avanzada en UNIX*, 3a. edicion, 2004. Biblioteca QA, 76.76, .O63, M37, 2006 (Reserva)

Fuentes adicionales

- Secciones del capítulo 1 del libro de Marquez[2]: Introducción (Temas: *Estructura del Sistema, llamadas al sistema, etapas de compilaciòn,* y *Arquitectura del Sistema Operativo UNIX, Dispositivos tipo bloque y tipo caracter, Gestion de Memoria e Interfaz de llamadas al sistema*).

- Puedes leer más sobre system calls en http://en.wikipedia.org/wiki/System_call.
- Encontraras la implementación GNU de libc en: http://www.gnu.org/software/libc/manual/html_mono/libc.html.
- Hay una lista de las llamadas a sistema más comunes con el comando **man syscalls** y en: http://linux.die.net/man/2/syscalls . Usaremos esta última referencia.