Sistemas Operativos. Práctica: "File System, System Calls"

Ricchy Alain Pérez Chevanier
Angel Renato Zamudio Malagón

1. Objetivo

El objetivo de esta práctica es que el alumno implemente las *system calls* de interacción con el *File System*, en particular implementarás el concepto de lista de archivos abiertos por proceso.

En esta práctica no es necesario validar los *pointers* que proveen los procesos de usuario como argumento a las *system calls* que se ejecutan en *kernel mode*.

2. Introducción

Prácticamente todo sistema de cómputo moderno tiene la capacidad de almacenar información aunque se encuentre apagado. Usualmente los sistemas de cómputo cuentan con dos jerarquías de memoria: principal y secundaria. La memoria principal se utiliza para almacenar los programas en ejecución (procesos) y los datos que utilizan, en la mayoría de los casos esta memoria es volátil mientras que la memoria secundaria tiene mayor tamaño que la principal y no es volátil. Los dispositivos de almacenamiento secundario, se conectan al sistema por medio del BUS principal y el procesador puede acceder a ellos por los canales de entrada/salida, es decir, son dispositivos de entrada/salida.

Los dispositivo de almacenamiento secundario utilizan tecnologías muy variadas para implementar el almacenamiento no volátil como: discos flexibles, cintas magnéticas, discos duros, discos de estado sólido. Cada tecnología tiene sus características propias para almacenar la información. Sin embargo, todos los dispositivos de almacenamiento secundario presentan una interfaz de comunicación común con el procesador. La interfaz consiste en que los dispositivos están separados en bloques o sectores de tamaño específico (generalmente 512 bytes), los cuales están indexados por números enteros iniciando en cero. De esta forma, el procesador accede al dispositivo leyendo o escribiendo sectores específicos.

Como se mencionó al inicio, el objetivo de los dispositivos de almacenamiento secundario es que los procesos puedan almacenar datos que permanecerán aunque el sistema se apague. Es decir, los procesos deben tener acceso al almacenamiento secundario, sin embargo, el dilema es si se debe permitir que los procesos de usuario utilicen directamente la interfaz de bajo nivel descrita anteriormente. El principal inconveniente de permitirlo es que los dispositivos son recursos compartidos por todos los procesos, lo cual genera inconvenientes de seguridad como de integridad.

Los sistemas de archivos surgen como una capa que sirve de intermediario entre los dispositivos de almacenamiento y los procesos, generalmente se implementan a nivel *kernel*. La idea es proporcionar a los procesos una interfaz de alto nivel que facilite el uso del almacenamiento secundario y garantice integridad y seguridad.

2.1. El concepto de archivo

Como el nombre lo dice, los sistemas de archivos basan su funcionamiento en el concepto de archivo. La idea general de un archivo es una estructura abstracta la cual se implementa sobre la arquitectura de los dispositivos de almacenamiento secundario. Es decir, los procesos en lugar de manipular sectores del disco, manipularán archivos. Según Tanenmbaum[1]:

> Los archivos son unidades lógicas de información creados por procesos. Los archivos proveen mecanismos de acceso a la información almacenada.

Para el sistema operativo un archivo debe ser una estructura que pueda manipular para poder administrarla. En este caso particular, dicha estructura debe tener la característica de poder almacenarse en un dispositivo de almacenamiento secundario. En ese sentido el sistema operativo debe permitir a los proceso de usuario realizar, al menos, las siguientes acciones:

1. Crear un archivo.
2. Eliminar un archivo.
3. Leer datos de un archivo.
4. Escribir datos en un archivo.

El primer detalle a tener en cuenta es que los archivos deben guardarse en el almacenamiento secundario, y deben contener información para realizar al menos las cuatro acciones mencionadas. Además de los datos que debe almacenar el archivo, usualmente se requiere información extra del archivo que permitirá administrarlo, a esta información se le conoce como *meta-datos* del archivo. En el caso de Pintos, la definición de archivo se encuentra en el archivo *src/filesys/file.c*:

```
1 struct file
2   {
3     struct inode *inode;
4     off_t pos;
5     bool deny_write;
6   };
```

Es necesario señalar que esta estructura representa al archivo en memoria principal. Consta de tres campos y el que de momento nos interesa es el campo de tipo `struct inode`. El concepto *inodo* esta presente en sistemas operativos de tipo UNIX y a grandes rasgos podemos decir que un *inodo* es la unidad mínima de almacenamiento de un sistema de archivos.

2.2. Crear un archivo

Como se mencionó anteriormente, las estructuras definidas deben permitir realizar al menos 4 operaciones sobre un archivo: crear, eliminar, leer y escribir.

Pintos ya cuenta con un *sistema de archivos* que implementa las funcionalidades más básicas como crear un archivo, eliminarlo, escribir y leer del mismo. Sin embargo dichas funcionalidades no se encuentran ligadas a las llamadas a sistema correspondientes, por lo cual no pueden ser utilizadas por los procesos de usuario.

El *sistema de archivos* de *pintos* se implementa en el directorio `src/filesys`; en concreto, las funcionalidades que deben ser enlazadas al sistema de archivos se encuentran en los archivos `src/filesys/filesys.c` y `src/filesys/file.c`. En el primer archivo se implementan las funciones:

```
1 bool filesys_create (const char *name, off_t initial_size);
2
3 struct file *filesys_open (const char *name);
4
5 bool filesys_remove (const char *name);
```

Estas tres funciones permiten crear, eliminar y abrir un archivo; el comportamiento específico de abrir se explicará con detalle más adelante. Por otro lado, en el archivo `src/filesys/file.c` se implementan las siguientes funciones:

```
1 off_t file_read (struct file *file, void *buffer, off_t size) ;
2
3 off_t file_write (struct file *file, const void *buffer, off_t size);
4
5 off_t file_length (struct file *file) ;
6
```

```
7 void file_seek (struct file *file, off_t new_pos);
8
9 off_t file_tell (struct file *file);
```

Podemos observar que todas la funciones mostradas arriba utilizan como parámetro un apuntador a una estructura de tipo `struct file`; dicha estructura la obtenemos utilizando la función `filesys_open` que se implementa en el archivo `src/filesys/filesys.c`. La forma más sencilla de enlazar estas funcionalidades con las llamadas a sistema correspondientes sería utilizar directamente apuntadores a la estructura `struct file` para referirnos a un archivo abierto en particular. Sin embargo, esta idea tiene el inconveniente de que los procesos de usuario tendrían acceso a dicha estructura, lo que implicaría que los procesos de usuario no utilizan una abstranción para manipular archivos con respecto a la implementación del *sistema de archivos* del kernel.

La mayoría de los sistemas operativos utilizan el concepto de *descriptor de archivo* (*file descriptor*) para enlazar el sistema de archivos con las llamadas a sistema. Un *descriptor de archivos* es un entero que se encuentra asociado a un archivo abierto en particular. La idea es que cuando un proceso de usuario abre un archivo, en lugar de recibir un apuntador hacia un `struct file`, reciba un *descriptor de archivo* con el cual podrá realizar operaciones sobre el archivo en cuestión.

3. Forma de Trabajo

Pasos a seguir para trabajar en esta práctica:

1. Debes de tener una solución funcional para la práctica anterior.
2. A partir de la rama que contiene la solución de la práctica anterior crea una nueva rama llamada `proyecto02/filesys-syscalls` y posiciónate en ella.
3. Reemplaza el archivo `.github/workflows/run-tests.yml` con el que entregamos junto con este documento.
4. Agrega los archivos `filesys-syscall.*` en el directorio `src/userprog`.
5. Reemplaza el archivo `src/Makefile.build` con el que entregamos junto con este documento.
6. Agrega los archivos `execute-tests-filesys-syscalls` y `execute-tests-filesys` en el directorio `src/userprog`. Dale permiso de ejecución a estos archivos con `chmod +x <PATH_TO_FILE>`.
7. Agrega el archivo `designdoc-filesys-syscalls.md` en el directorio `designdocs/userprog`.
8. Crea un *commit* de estos cambios y sube (*push*) la rama al repositorio.
9. Crea un *Pull Request* para la rama `proyecto02/filesys-syscalls` hacia la rama con la solución de la práctica enterior. Este *Pull Request* debe de contener los archivos que introdujiste en los pasos anteriores en la sección de *Files changed*.
10. En la sección de *Actions* solamente vas a pasar las pruebas de *Argument Passing* y *Syscalls Processes*, pero tienes que implementar las llamadas a sistema del *sistema de archivos* para pasar el resto de las pruebas.
11. Una nota importante es que para pasar las pruebas vamos a utilizar la máquina virtual `qemu`, pues `bochs` puede tener algunos problemas.

En el archivo `readme.md` del repositorio se incluyen instrucciones de cómo compilar el código y de cómo ejecutar algunas pruebas. Para ejecutar todas las pruebas de esta práctica, puedes acceder al contenedor, posicionarte en el directorio `src/userprog` y ejecutar la instrucción
`bash execute-tests-filesys-syscalls` y `bash execute-tests-filesys`.

4. Actividades

4.1. Implementación (7 puntos)

Implementar las siguientes llamadas a sistema para enlazar las funcionalidades del sistema de archivos con los procesos de usuario:

```
bool create(const char* file, unsigned initial_size)
```

Crea un nuevo archivo llamado *file* con un tamaño inicial de *initial_size* bytes. Regresa *true* si el archivo se creó correctamente, *false* en otro caso.

```
bool remove(const char* file)
```

Elimina el archivo con nombre *file*. Regresa *true* si se eliminó correctamente y *false* en otro caso. El archivo debe ser eliminado sin importar si está abierto o cerrado; eliminar un archivo abierto no debe cerrarlo.

```
int open(const char* file)
```

Abre el archivo con nombre *file*. Regresa un entero no negativo llamado descriptor de archivo (*file descriptor*) el cual servirá para identificar y hacer operaciones con el archivo. Si el archivo no se puede abrir se regresa -1. Los descriptores de archivo 0 y 1 están reservados para la entrada estándar (STDIN_FILENO) y para la salida estándar (STDOUT_FILENO) respectivamente, por lo cual no se debe regresar ninguno de los dos valores.

Cada proceso debe tener un conjunto independiente de descriptores de archivos; los descriptores de archivos no se heredan a los procesos hijo. Si un archivo se abre más de una vez, cada nueva llamada a *open* debe regresar un descriptor de archivo diferente a los anteriores.

```
int filesize(int fd)
```

Regresa el tamaño en bytes del archivo representado por el descriptor de archivo *fd*.

```
int read(int fd, void* buffer, unsigned size)
```

Lee *size* bytes del archivo representado por *fd* en el apuntador *buffer*. Regresa el número de bytes leídos, si hubo algún problema con la lectura se debe regresar -1. Si el descriptor de archivo es 0, se debe leer utilizando la función `input_getc()`, esta función lee un caracter a la vez, por lo que tendrás que llamarla *size* veces; esta función está definida en `devices/input.*`.

```
int write(int fd, const void* buffer, unsigned size)
```

Escribe *size* bytes del apuntador *buffer* al archivo representado por *fd*. Regresa el número de bytes escritos que deben ser máximo *size* bytes. Si el descriptor de archivo es `1` se debe escribir en la salida estándar, para ello se puede utilizar la función `putbuf (pointer_to_buffer, buffer_size)` definida en `lib/kernel/stdio.h` e implementada en `lib/kernel/console.c`.

```
void seek(int fd, unsigned position)
```

Cambia el siguiente byte por ser leído o escrito para el archivo representado por *fd* por la posición *position*.

```
unsigned tell(int fd)
```

Regresa la posición del siguiente byte por ser leído o escrito para el archivo representado por *fd*.

```
void close(int fd)
```

Cierra el archivo representado por *fd*. Cuando un proceso termina su ejecución implícitamente debe cerrar todos los archivo abiertos por el proceso.

Nota importante: Debes de sincronizar las *llamadas a sistema* de tal forma que cualquier cantidad de *procesos de usuario* pueda llamarlas al mismo tiempo. En particular no es seguro funciones del *sistema de archivos* desde múltiples *threads* al mismo tiempo. Tu implementación de llamadas a sistema debe de tratar el código del sistema de archivos como una sección crítica. Por ahora no recomendamos modificar código en el directorio `filesys`; en su lugar utiliza un único `struct lock` (definido en `src/threads/synch.c`) para invocar funciones definidas en `src/filesys/filesys.c` y `src/filesys/file.c`.

4.1.1. Pruebas

- `create`: create-normal create-empty create-exists create-long create-bound create-null
- `open`: open-normal open-missing open-twice open-empty open-boundary open-null
- `close`: close-normal close-stdin close-stdout close-bad-fd close-twice
- `read`: read-normal read-zero read-bad-fd read-stdout read-boundary
- `write`: write-normal write-zero write-bad-fd write-stdin write-boundary
- `open/close`: multi-child-fd
- El resto de las *system calls*: sm-create sm-full sm-random sm-seq-block sm-seq-random
- El resto de las *system calls*: lg-create lg-full lg-random lg-seq-block lg-seq-random
- `synchronization`: syn-read syn-write syn-remove

4.2. Documento de Diseño (3 pts)

Contestar todas las secciones del documento de diseño `designdocs/userprog/designdoc-filesys-syscalls.md`.

5. Consejos

Completa la definición de `struct process_open_file` en `filesys-syscall.h` e implementa las funciones que hacen falta en `filesys-syscall.c`; por ejemplo `filesys_syscall_create` se vería así:

```
1  bool
2  filesys_syscall_create (const char* file_name, unsigned initial_size)
3  {
4    if (file_name == NULL) {
5      // terminate this process with exit status -1
6    }
7    bool success = false;
8    lock_acquire (&mutex);
9    // this is executed as a critical section in mutual exclusion context
10   success = filesys_create (file_name, initial_size);
11   lock_release (&mutex);
12   return success;
13 }
```

Luego `syscall.c` puedes crear una envolvente para dicha función y agregarla de la siguiente forma:

```
1 static bool file_create_wrapper (int32_t* esp);
2
3 ....
4
5
```

```
6  static void
7  syscall_handler (struct intr_frame *f )
8  {
9    ...
10   switch(syscall) {
11   ...
12     case SYS_CREATE: {
13       f->eax = file_create_wrapper (esp);
14       break;
15     }
16   ...
17   }
18   ...
19 }
20
21 ...
22
23 static bool
24 file_create_wrapper (int32_t* esp)
25 {
26   char* file_name = (char*) *esp++;
27   unsigned initial_size = (unsigned) *esp;
28   return filesys_syscall_create (file_name, initial_size);
29 }
```

La implementación de `remove` y `open` son análogas.

Implementa con cuidado las instrucciones `TODO` del archivo `filesys-syscall.c`. En particular necesitas un `struct lock` global en este archivo para poder acceder a las funciones del sistema de archivos en contexto de exclusión mutua; en particular debes de inicializar dicho `lock` en la función `filesys_syscall_init`, y luego llamar dicha función en `syscall_init`.

Para manejar los archivos abiertos se puede utilizar una lista en donde la entradas asocien el descriptor de archivo con la estructura `struct file`, debes usar una lista por proceso. En el momento en el que se utilice la llamada `open` se puede obtener espacio para una nueva estructura `struct process_open_file` con `malloc`, después se debe utilizar la función `filesys_open` para obtener el apuntador hacia el `struct file` correspondiente. Para obtener un nuevo descriptor de archivo se puede utilizar un contador inicializado en 2 para cada proceso. La llamada a sistema `close` realiza la operación contraria a `open`, en particular debe de liberar la memoria reservada mediante `malloc` usando la función `free`.

Para las llamadas *read*, *write*, *seek*, *tell* y *filesize*, simplemente se debe utilizar las funciones análogas del archivo *src/filesys/file.c*

6. Entrega

En Google Classroom agrega el enlace del `Pull Request` que contiene tu solución y en donde se deben de observar los resultados de Github Actions al correr las pruebas. También marca la tarea como entregada en Google Classroom.