Práctica 5: Sistema de archivos

2018 – Sistemas Operativos II

Licenciatura en ciencias de la computación

Entrega: fecha a determinar

Nota: debe utilizar el Subversion de la materia creando un subdirectorio por alumno/grupo en: `https://svn.dcc.fceia.unr.edu.ar/svn-no-anon/lcc/R-412/Alumnos/2018/`

1. Introducción

Las prácticas de multiprogramación y memoria virtual hacían uso del sistema de archivos de Nachos. La siguiente fase es construir en efecto este sistema de archivos. Al igual que en algunas de las prácticas anteriores, se le provee cierto código necesario; su trabajo es completar el sistema de archivos y mejorarlo.

El primer paso es leer y entender el sistema de archivos parcial que se ha escrito para usted. En el directorio `filesys`, ejecute el comando `./nachos -f -cp test/small small` para un simple caso de prueba del código – `-f` formatea el disco físico emulado, mientras que `-cp` copia el archivo de Unix `test/small` en ese disco.

Los archivos en los cuales enfocarse son:

`fs_test.cc` – un simple caso de prueba para el sistema de archivos.

`file_system.hh`, `file_system.cc` – interfaz de entrada al sistema de archivos.

`directory.hh`, `directory.cc` – traduce nombres de archivos a cabeceras de archivos en disco; la estructura de datos de directorio se almacena como un archivo.

`file_header.hh`, `file_header.cc` – gestiona la estructura de datos que representa el esquema de los datos de un archivo en disco.

`open_file.hh`, `open_file.cc` – traduce lecturas y escrituras de archivos a lecturas y escrituras de sectores de disco.

`synch_disk.hh`, `synch_disk.cc` – provee acceso síncrono al disco físico asíncrono, de manera que los hilos se bloqueen hasta que sus peticiones se hayan completado.

`disk.hh`, `disk.cc` – emula un disco físico, enviando peticiones para leer y escribir bloques de disco a un archivo de Unix y luego generando una interrupción tras cierto período de tiempo. Los detalles de cómo hacer peticiones de lectura y escritura varían enormemente de un dispositivo de disco a otro; en la práctica, uno desearía esconder estos detalles detrás de algo como la abstracción provista por este módulo.

El sistema de archivos de Nachos tiene una interfaz de estilo Unix, así que puede ser conveniente leer las páginas *man* de Unix para las llamadas a sistema (`creat`, `open`, `close`, `read`, `write`, `lseek` y `unlink`). Por ejemplo, con el comando: `man creat`. Las llamadas de Nachos son similares (pero no idénticas a estas); el sistema de archivos traduce estas llamadas en operaciones del disco físico. Una diferencia importante es que el sistema de Nachos está implementado en C++. `Create` (como el `creat` de Unix), `Open` (`open`) y `Remove` (`unlink`) se definen en la clase `FileSystem`, dado que involucran manipulación de nombres de archivos y directorios. `FileSystem::Open` retorna un puntero a un objeto `OpenFile`, que se usa para operaciones directas sobre el archivo como `Seek` (`lseek`), `Read` (`read`) y `Write` (`write`). Un archivo abierto se cierra (`close`) eliminando el objeto `OpenFile`), es decir que se encarga el destructor; no hay un método específico para cerrar.

Muchas de las estructuras de datos del sistema de archivos se almacenan tanto en memoria principal como en disco. Esto provee cierta uniformidad; todas estas estructuras tienen un método `FetchFrom` que extrae los datos del disco a la memoria y uno `WriteBack` que vuelve a almacenar los datos en disco. Nótese que las representaciones en memoria y en disco no necesariamente son idénticas en un momento dado.

Si bien el código provisto implementa todas las piezas fundamentales de un sistema de archivos, tiene ciertas limitaciones. Su trabajo en esta práctica será corregir estas limitaciones. Tenga en cuenta que hay, por supuesto, interrelaciones entre distintas partes de esta práctica. Por ejemplo, la forma que elija de implementar archivos grandes puede afectar el rendimiento de su sistema para la parte 4.

2. Ejercicios

1. Complete el sistema de archivos básico añadiendo sincronización para que múltiples hilos puedan usar el sistema de archivos de forma concurrente. El código inicial asume que se accede con un solo hilo por vez.

   Además de asegurar que las estructuras de datos internas no se corrompan, las modificaciones deben cumplir las siguientes restricciones (que son las mismas que en Unix):

   a) El mismo archivo puede leerse/escribirse con más de un hilo concurrentemente. Cada hilo abre el archivo de forma separada, lo cual le confiere su propia posición de "seek" privada dentro del mismo.

   b) Todas las operaciones del sistema de archivos deben ser atómicas y serializables. Por ejemplo, si un hilo está en el medio de una escritura, un hilo que concurrentemente lea el archivo verá o bien el cambio completo o ningún cambio. Además, si la operación `OpenFile::Write` finaliza antes de que empiece la llamada a `OpenFile::Read`, entonces `Read` debe reflejar la versión modificada del archivo.

   c) Cuando se borra un archivo, los hilos que lo tengan abierto pueden continuar leyendo y escribiendo hasta que lo cierren. Borrar un archivo (`FileSystem::Remove`) debe impedir próximas operturas del mismo, pero los bloques de disco para el archivo no pueden reclamarse hasta que este haya sido cerrado por todos los hilos que lo tengan ya abierto.

   Consejo: para hacer esta parte, probablemente necesite mantener una tabla de archivos abiertos.

2. Permita que el tamaño máximo de un archivo sea tan grande como el disco (128 KiB). En el sistema de archivos básico, cada archivo está limitado por un tamaño de algo menos de 4 KiB. Cada archivo tiene una cabecera (clase `FileHeader`) que es una tabla de punteros directos a los bloques de disco para ese archivo. Dado que la cabecera se almacena en un sector de disco, el tamaño máximo de un archivo está limitado por el número de punteros que entran en un sector. Para aumentar el límite a 128 KiB probablemente (aunque no necesariamente) se requiera que se implementen bloques con doble indirección.

3. Implemente archivos extensibles. En el sistema básico, el tamaño de un archivo se especifica al crearlo. Una ventaja de esto es que la estructura `FileHeader`, una vez creada, nunca cambia. En el de Unix y la mayoría de los otros sistemas de archivos, un archivo inicialmente se crea con tamaño 0 y luego se expande cada vez que se haga una escritura al final del mismo.

   Como caso de prueba, permita que el archivo de directorio se expanda más allá de su límite actual de diez archivos.

   Al hacer esta parte, procure que los accesos concurrentes a las cabeceras permanezcan apropiadamente sincronizados.

4. **Opcional pero interesante**
   Mejore el rendimiento del sistema de archivos. Hay distintos enfoques que uno podría tomar para esto:

   a) Optimizar la latencia de búsqueda y rotación en el disco. Para leer o escribir un sector, la cabecera del disco debe moverse a la pista correspondiente y luego el sistema debe esperar a que el sector esperado rote bajo la cabecera del disco. En sistemas acotados por disco, estos retardos pueden ser una causa de rendimiento pobre. Se puede solucionar (i) poniendo bloques

de disco del mismo archivo en la misma pista (léase el artículo de McKusick y compañía) o (ii) cuando se encolan múltiples peticiones, efectuando primero aquellas que tengan el retardo más corto, al tiempo que se evita la inanición de peticiones de disco.

b) Modificar el sistema de archivos para matener una caché de bloques de archivo. Cuando se hace una petición para leer un bloque, se comprueba si este está almacenado en la caché, y de ser así, no se necesita acceder a disco y se puede retornar una copia inmediatamente. Una mejora es que en lugar de siempre escribir de forma inmediata datos modificados a disco, los bloques sucios se mantengan en la caché y se escriban en algún momento posterior – esto se llama escritura retardada (en inglés, "write-behind"). Otra mejora es que, cuando se lea un bloque de un archivo, se traiga automáticamente el siguiente bloque a la caché, por si acaso de que ese bloque esté por ser leído – esto se llama lectura adelantada ("read-ahead").

   La caché está limitada a un tamaño de 64 sectores de disco (aproximadamente 6 % del tamañó del disco); este límite de espacio debe incluir la memoria que se use para archivos abiertos, el mapa de bits, etc., si uno mantiene todo esto en memoria.

Para cada optimización que implemente, explique qué mejora de rendimiento espera en posibles "benchmarks", y por qué las otras alternativas (las que no haya implementado) devolverían menos ganancia por el mismo esfuerzo.

5. Implemente un espacio de nombres jerárquico. En el sistema básico, todos los archivos viven en un único directorio; modifique esto para permitir que los directorios apunten tanto a archivos como a otros directorios. Para hacerlo necesitará métodos que conviertan rutas en secuencias de directorios donde buscar los archivos. También tendrá que implementar métodos para cambiar el directorio de trabajo actual (vea el comando `cd` de Unix) y para imprimir los contenidos del directorio actual.

   Para mejor rendimiento, permita múltiples actualizaciones a distintos directorios, pero asegúrese de que las actualizaciones a un mismo directorio se realicen de forma atómica (por ejemplo, para asegurarse de que un archivo se borre solo una vez).

6. **Opcional pero interesante**
   Haga que el sistema de archivos sea robusto. El disco de Nachos puede corromperse si el sistema no termina limpiamente (por ejemplo, luego de colapsar por un error de software o si se sale de GDB sin terminar el programa). Esto se debe a que algunas operaciones del sistema de archivos requieren más de una escritura al disco físico. Un ejemplo es la creación de un archivo nuevo: se debe escribir un `FileHeader` nuevo en disco y también actualizar el directorio y el mapa de bits de bloques libres. Si el sistema termina abruptamente después de haber hecho algunas pero no todas estas actualizaciones, el disco puede quedar en un estado inconsistente.

   Puede encarar este asunto por dos caminos. Uno es registrar cuándo empieza y termina una secuencia de escrituras lógicamente atómicas (vea el artículo de Gray). Otra es estructurar el sistema de archivos de forma que cada actualización tenga efecto con la escritura de un único bloque de disco. Por ejemplo, para modificar un archivo, puede escribir las modificaciones a nuevos bloques de disco, sobreescribir el `FileHeader` para que apunte a estos bloques y entonces devolver los bloques viejos a lista de libres. Note que la lista de libres puede ser redundante; puede ser posible reconstruirla después de un fallo abrupto a partir de otros datos en el disco.

   En caso de servir, puede asumir (de forma algo irrealista) que los colapsos ocurren ya sea antes o después, pero no durante las escrituras a disco. En otras palabras, un sector de disco contendrá o bien el viejo valor o el nuevo, pero no una parte de cada uno.

   Para esta parte, debería testear el sistema haciéndolo fallar de forma abrupta en un momento inoportuno (por ejemplo, en medio de una actualización de directorio) y luego reconstruyendo el disco tras el fallo.