Homework 6: Filesystem Implementation

Assigned:	Oct 27, 2009
Snapshot due:	Nov 3, 2009 (by 12:25pm)
Due:	Nov 10, 2009 (by 12:25pm)

1 Overview

Based on MT's support for strategically important features like efficient semaphores and a proportional share scheduler, a leading digital camera vendor is considering using a version of MT as the operating system that runs on its high-end products. However, to convince this vendor that you are serious about supporting embedded devices you need to quickly develop a simple filesystem that can load and store images on a flash memory card or a tiny hard disk. Since the vendor is having a bit of trouble coming up with a functional sample of its new camera, you will do your development on UNIX using a simulated block device that provides an interface that is identical to the interface provided by the device driver for flash memory cards and tiny disks.

Here are some things to keep in mind:

- Most people should be working in the same teams as Homework 4 — it's okay to switch teams if you want, but you are not permitted to work in groups of more than two people.
- Get started early: this project is more loosely specified and requires more up-front design work than previous projects have. You will **not** be able to write this code the day it's due.
- Read this document completely and carefully before writing any code.

2 The MT Filesystem Specification

This section describes the requirements for the filesystem that you will implement.

2.1 Features and requirements

For the most part you will be implementing UNIX file semantics. Consult the relevant man pages for details. The following are the ways in which the MT filesystem differs from UNIX. Read these carefully, as they will lighten your workload by orders of magnitude.

- The MT filesystem has only a single level. That is, there is only one directory — the root directory — and all files reside there.
- You are required to implement a specific on-disk data format; see Section 2.4 for details.
- You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

- All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.
- Since digital camera files are typically of limited size, file block indexing is direct — you need not implement indirect blocks, double-indirect blocks, etc.
- Each file can be referenced by at most one file descriptor at a time. A single file descriptor, however, can be shared among multiple threads.
- There are no hard links or symbolic links.
- Some filesystem operations are defined to only operate on files that are not open.
- Disk hardware is defined to be free of failures.

2.2 The MT filesystem API

You must implement these functions as specified. The header file for this interface is provided to you as `fs.h`.

```
int MT_fs_open (const char *name, int create);
```

Opens the specified file and returns a file descriptor to it. If the file exists then the `create` parameter is ignored. If it does not exist and create is zero, then return an error code. If it does not exist and create is non-zero, then create a new file with the specified file name, open it, and return the file descriptor. File names are limited to 63 characters and are null-terminated like other C strings.

You must support up to 100 open files at a time — it is an error to call `MT_fs_open` when there are already 100 open files.

Summary: return a file descriptor on success or -1 on error.

```
int MT_fs_close (int fd);
```

Closes the file associated with the file descriptor. Return 0 on success or -1 if `fd` is out of range or is not open.

```
int MT_fs_unlink (const char *name);
```

Delete the closed file with filename `name`. Return 0 on success or -1 if the filename is invalid, the file does not exist, or the file is currently open. On success, all disk blocks associated with the file must be freed.

```
int MT_fs_read (int fd, void *buf, int count);
```

Read `count` bytes from the file referred to by `fd` into the memory buffer `buf`. Quoting from the Linux man pages:

> On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (because we were close to end-of-file). On error, -1 is returned. In this case it is left unspecified whether the file position changes.

```
int MT_fs_write (int fd, void *buf, int count);
```

Write `count` bytes to the file referred to by `fd` from the memory buffer `buf`. Quoting the Linux man pages:

> On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned. If count is zero, 0 will be returned without causing any other effect.

```
int MT_fs_getpos (int fd);
```

Returns the current file position on success or -1 on error. This function will only fail if `fd` is not a valid file descriptor.

```
int MT_fs_truncate (const char *name, int length);
```

On success, the closed file with filename `name` has a length of `length` bytes. If the file was previously longer, it is shortened. If it was previously shorter, it is extended and zeros are written to the bytes between the previous end-of-file and the new end-of-file. Return 0 on success or -1 if the filename is invalid, the file does not exist, the file is currently open, or an allocation error occurs (filesystem runs out of free blocks or free index slots) while the truncate is in progress.

```
int MT_fs_lseek (int fd, int pos);
```

On success, the current position of the open file associated with the descriptor is set to `pos`. If `pos` is -1 then the position is set to the end-of-file. If `pos` is larger than the current end-of-file then the file is extended, with zeros being written between the previous end-of-file and the new end-of-file.

Returns the new file position on success or -1 on error (for example, if `fd` is invalid or an allocation error occurs while extending the file).

2.3 Additional requirements

- As in UNIX, each valid file descriptor should refer to an in-kernel (the MT kernel, not the OS kernel) data structure that represents the state associated with the open file, such as the current file position.

- All MT filesystem functions are required to operate atomically with respect to the associated file. This means that if, for example, two threads take turns writing to a file (sharing the same file descriptor) then there is no interleaving between chunks of data written by individual write commands issued by the threads.
- The process containing MT threads should be able to terminate at an arbitrary time — as long as no threads are currently executing an MT filesystem function — without leaving the filesystem in an inconsistent state.
- The filesystem must be left in a consistent state after "normal" allocation errors occur. For example, consider a situation where the filesystem runs out of blocks while extending a file to accommodate an lseek beyond the previous end of the file. In this case the lseek fails, but the file is left in a usable state: the end-of-file and current file position should be set to the last byte of the last block that was successfully allocated.

2.4 The on-disk format

Disk blocks are 512 bytes. A valid filesystem must contain at least three blocks. The maximum size of a filesystem is the number of blocks that can be described by a signed 32-bit integer (i.e. $2^{31} - 1$). The first block on the disk (called the zero block from here on) is special — it contains the directory. Its format is as follows, working on the assumption that the zero block has been read into a C array declared as `int zero_block[128]`:

zero_block[0]	int magic1 == 0xabbaabba;
zero_block[1]	int magic2 == 0xf00bf00b;
zero_block[2]	int block_size == 512;
zero_block[3]	int block_cnt;
zero_block[4]	int free_list_head;
zero_block[5]	int file0;
zero_block[6]	int file1;
zero_block[7]	int file2;
...	
zero_block[127]	int file122;

The two magic numbers are used to implement a crude validity check on MT filesystems: the filesystem code should abort with an error if they do not have the specified values. For purposes of this assignment, `block_size` is always set to 512. `block_cnt` stores the number of blocks that the filesystem contains and `free_list_head` contains the block number of the first block on the free list. The first 4 bytes of each free block are required to store an integer pointing to the next block on the list, with -1 indicating the end-of-list.

Each of the 123 file entries either contains -1 for a free slot, or the disk block number of the inode of a file in the filesystem. An inode is a disk block containing the meta-data associated with a single file. Its format is as follows, working on the assumption that the zero block has been read into a C array declared as `int inode[128]`:

inode[0]	int magic == 0xdeadbeef;
inode[1–16]	char filename[64];
inode[17]	int size;
inode[18]	int block0;
inode[19]	int block1;
inode[20]	int block2;
...	
inode[127]	int block110;

The magic number is used to implement a crude validity check on MT filesystems: the filesystem code should abort with an error if an inode is found to not contain the magic value. The `filename` is an array of 64 bytes that contains up to 63 characters of filename plus a terminating null byte. `size` indicates the size of the file in bytes. Finally, the block pointers describe the disk block numbers of data blocks allocated to the file. It is always the case that block 0 stores bytes 0–511 of a file, block 1 stores bytes 512–1023, etc. It can easily be computed that the maximum size of a file in the MT filesystem is about 56 KB (the digital camera in question apparently supports cutting-edge image compression...).

Every disk block must belong to exactly one of the four categories described above: the zero block, free blocks, inodes, and data blocks. The MT block device driver performs a filesystem validity check at startup time to ensure that this is the case.

2.5 The block device interface

You should include the file `block_dev.h` to use the block device interface. The block device interface is **asynchronous**, meaning that asking it to do something is a non-blocking operation. The disk is defined to be **busy** between when a read or write is initiated and when the associated interrupt arrives, and idle at all other times. The interface is as follows:

```
int MT_init_block_dev (const char *fn,
                       disk_int_t int_routine,
                       int *block_cnt,
                       int *block_size);
```

Initialize the block device driver. This should be called at the very end of `MT_init`, once everything else has been set up. Its argument `fn` is the file name of the (actual) disk file that is to serve as a source of (simulated) disk blocks. The file should be created using the `new_mt_fs` utility that is provided to you.

Using the argument `int_routine`, you pass a function pointer to the block device driver that serves as the disk interrupt handler. In other words, once you initiate an asynchronous operation, at some point in the future this function will be called. At that point (and only at that point) is the operation guaranteed to be completed.

The number of blocks in the initialized filesystem is returned to you in the location pointed to by `block_cnt`, and the size of each disk block is returned in the location pointed to be `block_size`. Again, this size will always be 512 bytes for this project.

```
int MT_initiate_block_get (int block_num, void *buf);
```

The effect of this routine is to initiate a disk operation that will eventually load a block into memory. It has this effect (and returns 0) only if the disk is idle when it is called. If the disk is busy the function returns -1 and the operation is not initiated — it must be retried later.

```
int MT_initiate_block_put (int block_num, void *buf);
```

This routine acts the same as the get initiator in all respects except that it initiates a put operation. The contents of `buf` must be left undisturbed until the associated interrupt has arrived; only then can the buffer be reused.

```
void __MT_raw_disk_handler__ (void);
```

This function is not really part of the block device interface — its role is to help simulate an asynchronous block device. All you need to do is ensure that it is called close to the start of the timer signal handler routine. For example, my implementation contains this code:

```
static void timer_handler (int ignored)
{
  enter_MT_kernel ();
  __MT_raw_disk_handler__ ();
  check_for_expired_timers ();
  if (current) {
    update_current_vt ();
    make_ready (current);
    ...
```

You **must** call this routine in this manner or the block device driver will not operate properly. However, you should otherwise totally ignore its presence.

3 Implementation and Testing Hints

Previous sections should have given you enough information to complete this project. This section contains completely optional advice that you may find helpful.

1. The asynchronous disk interface is inconvenient (to say the least). One way to deal with it is to implement a synchronous disk interface on top of the asynchronous interface, and then to implement your filesystem in terms of these synchronous operations. The synchronous operations should block the calling thread until the associated operation has completed. You

will need to implement a disk request scheduler as part of this layer — first-come first-serve is a fine scheduling policy since it's fair and because the disk simulator you are using is not sophisticated enough to make a better scheduler worthwhile.

Also helpful are synchronous `free_block()` and `find_free_block()` functions.

2. Although you are not required to implement any caching, it may be helpful to keep a cached copy of the zero block. This can increase performance and avoid the need for a semaphore to provide atomic access to the zero block. If you do this, be careful to always write out the zero block every time the cached copy is changed — you never know when the user program is going to exit, and the filesystem should always be in a consistent state between operations.

3. Some minimal test programs have been provided. For example, `fs_create.c` creates a file and `fs_write.c` performs a few disk writes. You can use these commands in sequence like this:

```
./new_mt_fs 1024 default.img
./fs_create
./fs_write
./fs_delete
./fs_delete
```

   The effect of this sequence is as follows. First, create a new disk image with no files stored on it. Second, run `fs_create`, which creates a file. Third, perform several writes to the file. Fourth, delete the file. Finally, try to delete the non-existent file — this command fails.

4. You should implement additional test programs on your own. Besides testing the obvious functionality, you should ensure that your implementation correctly handles:

   - Invalid and out-of-range file descriptor arguments
   - File names that are too long
   - Running out of inode slots in the zero block
   - Running out of free disk blocks
   - Running out of data block slots in an inode
   - Writes past the end of a file — these are legal but require data blocks to be allocated
   - Reads and writes that span many disk blocks — these are legal as well, but the operation must implemented in terms of smaller operations that work on blocks or parts of blocks

5. If the code you handed in for Homework 4 is unstable, then you should consider backing off and using the version of `mt.c` that was distributed with Homework 4 as the basis for your Homework 6 code. This project does not specifically rely on any features that you implemented for Homework 4.

4 The Assignment

Your assignment is to implement the MT filesystem functions. Unlike previous projects, for this project you must `handin` an interim snapshot indicating your progress one week before the project deadline — see the next section for details.

Extra files that have been mentioned in this assignment can be found in this directory in the CADE filesystem:

```
~cs5460/hw6
```

Here is a summary of changes you must make to `mt.c`:

- Add the call to `MT_init_block_dev` at the end of `MT_init`. This initializes the block device. You will, of course, need to add some filesystem initialization code as well.
- Add the call to `__MT_raw_disk_handler__` from the timer signal handler
- Change the type of `MT_init` so that it takes a `char *` as an argument to pass along to the block device driver. This makes it convenient to specify a block file on the command line of an MT program (the example programs included with this assignment show how this is done).
- Add code for all of the functions in `fs.h`. It's okay to add new code in the main body of `mt.c`, but the bulk of your new filesystem functions should be placed at the end of this file.

If you have time, consider earning some extra credit by implementing one of the following (you can implement more than one if you want, but it will not help your grade since there is a cap on the maximum possible amount of extra credit per project).

Extra Credit 1 — Caching

Implement caching in your filesystem. The ideal place to put this code is behind the synchronous read/write interface — the rest of the filesystem should change little or not at all.

You should implement a read cache and a write cache, each containing 1/3 of the number of blocks in the currently running filesystem. The read operation should first search the cache for the block and then, if not found, initiate a blocking disk read. The read cache should have a replacement policy such as LRU approximation.

A disk write should copy the written block into the write cache if there is room, and do a synchronous write otherwise. Here you do not need a replacement policy, but rather a mechanism for writing out dirty disk blocks in the background. You may create a new function, `MT_fs_sync()` that flushes the write cache — programs should call this prior to exiting. A buffered/cached filesystem, then, provides weaker consistency semantics than an uncached one (and this might be unacceptable in some cases — notice that when Windows uses a floppy disk there is no buffering of writes).

You should arrange so that your filesystem code can be compiled to run with and without caching. You should also provide a program that performs both reads and writes, whose operation is greatly sped up when the cache is turned on.

Extra Credit 2 — Indirect Blocks

Support files that are larger than the MT filesystem currently supports by making the final 10 slots in an inode indirect — they point to disk blocks that contain lists of block numbers of data blocks. This should increase the size of the largest file that can be stored by roughly 10 times. Provide a test program that demonstrates this.

Extra Credit 3 — Directories

Implement a true hierarchical filesystem. In general, you should follow UNIX semantics and use the `/` character to divide paths into a sequence of directories. Provide a test program.

You should implement a directory block format that closely resembles the zero block format. As in UNIX, each directory should contain special files "`.`" and "`..`" that respectively refer to the current and parent directories.

5 Logistics and Grading

Grading for this project is primarily on correctness: your improved version of MT should implement the specified functionality and should not crash. However, we may award partial credit if you have taken a good approach but messed up some details. Good comments in your code are crucial for helping us understand your solution — partial credit will not be awarded when we cannot understand what you have done.

For this project you will be handing in a snapshot of your code a week before the actual deadline. This is designed to encourage early work on the project and to give you a chance to get feedback on your code before the deadline. You do not have to be at any particular point on the assignment by this point, but if you do not handin a snapshot you will lose a small portion of the credit for this project.

What to handin (BOTH for the snapshot and on the final due date):

1. A file called `README` that provides the names and userids of all group members. (If handing in the snapshot, also say what works at this point and what doesn't work.)

2. All source files required to build your improved version of MT.

3. A makefile that builds all programs you handin. This is **required** — we will not manually invoke the compiler while grading your project.

4. Source files for any extra credit work as well as a file `extracredit.txt` saying which problem(s) you are solving, outlining your approach, and providing other information that might be useful to the graders.

Although you may do your work wherever you want, your project will be graded on a CADE lab Linux machine, so **you must make sure that everything compiles and runs properly there**. If it doesn't you will not get credit for that part of the assignment.

This is a group assignment — you and your partner will hand in only one set of files and you will both receive the same grade. Either of you can handin the files (but not both, please).

Your handin command line on the snapshot due date will have this form:

```
handin cs5460 hw6-snap file1 file2 ...
```

Your handin command line on the final due date will have this form:

```
handin cs5460 hw6 file1 file2 ...
```