Practical 3: Filesystem access

Due date: 5pm Thursday 30 October

In this practical, you will implement several system calls and library functions that permit processes to access data stored in a file system. These will all follow the same semantics as those of UNIX, so that code intended to run in a normal UNIX environment that uses these calls can run under the example kernel that we have been using in the course.

Rather than accessing a hard disk, the filesystem support code will simply use a RAM disk, in which all of the data for the filesystem is loaded into memory at boot time. The filesystem can essentially be treated like a normal tree in memory, in that you can traverse it by following pointers and inspecting various fields of the nodes. The system calls that you implement will access this data structure directly. For simplicity, the filesystem data structures are designed for read-only access, so you will not need to implement support for writing to files or creating directories.

The functions you must implement in this assignment are as follows. Each of these are described in their respective man pages.

- `chdir`
- `getcwd`
- `open`
- `close`
- `read`
- `opendir`
- `readdir`
- `closedir`

The `close` and `read` system calls are already present in the sample code, but will need to be extended to support accessing files. Note that it is not necessary to implement `write`, although it will need to return an error code if used on a file handle. The `fscalls.c` file in the sample code includes an implementation of `stat`, which demonstrates how to perform basic filesystem access within a system call. Your implementations of the above functions should all go in `fscalls.c`.

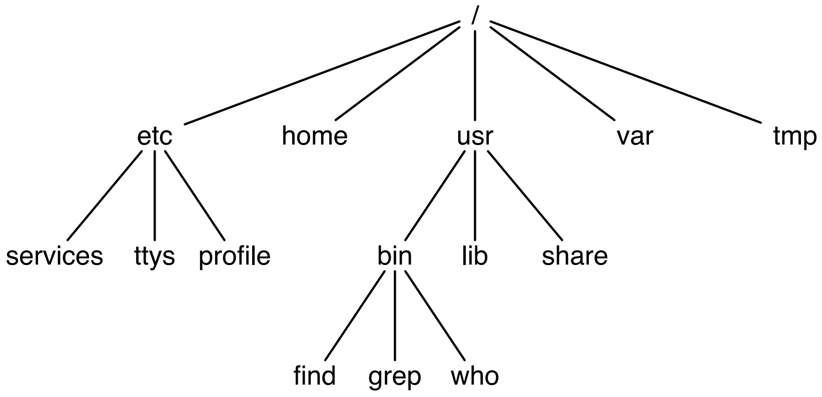

Figure 1: A typical UNIX filesystem tree

The `opendir`, `readdir`, and `closedir` functions are not actually system calls, but instead must be implemented as library functions. The reason is that they need to allocate an object on the heap (a DIR structure), which is used by each of the functions. These rely on another system call named `getdent` which you will also need to implement in the last stage of the assignment. This part of the assignment is discussed further in Section 5.3.

1 Filesystem structure

Version 8 of the kernel contains the data structure definitions and helper functions for accessing the filesystem. The format in which these are arranged is designed to be as simple as possible, and is essentially just a concatenation of all the files together in memory, along with arrays of directory entries specifying the name and location of each file. This format is much simpler than that used by most other file systems, since it is not intended to support modification of files or directories.

The filesystem image is structured as a tree. Each file or directory is represented as a node in the tree, with the root node corresponding to the root directory (/). Directories are branch nodes, and can contain an arbitrary number of files and other directories. Files are leaf nodes, consisting of an array of binary data that corresponds to the contents of the file. A sample filesystem tree, based on files commonly found on UNIX systems, is shown in Figure 1.

Each node within the tree, whether it is a file or directory, is represented by a *directory entry*. This contains various properties such as the type, name, size, and location. For directory entries of type *file*, the location refers to an offset relative to the start of the filesystem image at which the contents of the file may be found. The actual data can be obtained by pointer arithmetic, by taking the address in memory of the filesystem image and adding the location to this address. For directory entries of type *directory*, the location is also an offset relative to the filesystem start, but points to a `directory` structure containing the number of files in that directory, and an array of directory entries. The C structs for directory entries and directories are defined in `filesystem.h` as follows[1]:

```
typedef struct directory_entry {
  unsigned int size; /* size of file/directory in bytes */
```

[1] `__attribute__ ((packed))` just tells the compiler to store these structures directly adjacent to each other in memory, without padding

```
  unsigned int type; /* either TYPE_DIR or TYPE_FILE */
  unsigned int location; /* offset from start of file system */
  unsigned int mode; /* permission bits */
  unsigned int mtime; /* modification time */
  char name[MAX_FILENAME_LEN+1]; /* file or directory name */
} __attribute__ ((packed)) directory_entry;

typedef struct directory {
  unsigned int count;
  directory_entry entries[0];
} __attribute__ ((packed)) directory;
```

Note that although the `entries` field in the `directory` struct is declared to have 0 elements, the `directory` struct is just a way of telling the compiler how to access memory at a particular address. When the filesystem is built, the correct amount of space will be allocated at the location of the `directory` struct to allow for the necessary number of entries. It is thus possible to access elements `entries[n]` for any value of `n` less than `count`.

2 Helper functions

Only two functions are needed to implement the back-end logic of the filesystem, both of which are provided for you in `filesystem.c`:

```
int get_directory_entry(const char *filesystem, const char *path,
                        directory_entry **entry);
void relative_to_absolute(char *abs, const char *base,
                          const char *rel, int max);
```

`get_directory_entry` retrieves the directory entry associated with a given path name. The first parameter is the address in memory at which the filesystem image begins. The `path` parameter must be an absolute path, i.e. one that starts with the root directory (/). The `entry` parameter is an output parameter in which a pointer to the directory entry will be placed, provided that it exists. This function returns 0 on success, or a negative error code if the requested file or directory could not be found. Upon success, the entry pointer can then be used to examine the contents of the directory or file, depending on its type.

`relative_to_absolute` resolves a relative path to its absolute version, according to the specified base path. For example, if the base path is `/home/peter` and the relative path is `../cruz`, then the absolute path returned by this function would be `/home/cruz`. The `abs` parameter is a fixed size buffer of length `max` supplied by the caller, in which the resulting absolute path will be stored by the function. Since the system calls you are to implement need to work relative to the process's current directory, you will need to use this function to obtain absolute path names for use with `get_directory_entry`.

3 The `fstool` program

Version 8 of the sample code includes a program called `fstool`, which serves two purposes. The first is to demonstrate how to use the filesystem routines to access files, get directory listings, and traverse the tree. Reading through this code will give you an understanding of what your system calls will need to do in order to access the filesystem. The second purpose of `fstool` is to allow you to build filesystem images for testing. It does this by constructing a filesystem image in memory based on the contents of a specified directory, and then saving this to disk. `fstool` can be run directly under Linux.

The program can run in four different modes, depending on the command-line arguments used to invoke it:

- `fstool -build <imagefile> <directory>`

  This creates a file system image in the file called `imagefile`, consisting of all of the files and subdirectories contained within `directory` on the local file system. The image file can then be copied on to the GRUB boot disk image, so that it will be loaded at boot time into memory. This can be done as follows:

  ```
  mcopy -i grub.img <imgfile> ::
  ```

- `fstool -dump <imagefile>`

  Prints out the full directory tree contained in the filesystem image `imagefile`

- `fstool -get <imagefile> <filename>`

  Extracts a particular file from a filesystem image

- `fstool -shell <imagefile>`

  Implements a basic UNIX-like shell with which you can browse the filesystem image. The commands `ls`, `cat`, `cd`, and `pwd` are supported, and work in much the same way as their UNIX counterparts, but do not handle command-line switches.

Examining the source code for `fstool` will give you several of examples of how to use the `get_directory_entry` and `relative_to_absolute` functions. The way in which your system calls make use of these functions will be in a similar manner to how they are used in `fstool`.

4 The shell

In order to test out your system calls, you are given a basic shell which runs on kernel startup and allows the user to browse the file system. It works identically to the shell in `fstool`, and thus supports the `ls`, `cat`, `cd`, and `pwd`, and `find` commands. The difference between the two is that this shell runs under the example kernel, and uses system calls to access the filesystem data. This shell is implemented in `sh.c`. You should not need to modify this file in order to get your code to work, though it may be useful to add debugging code here if you need to during development.

5 What you need to do

All of your system calls must be implemented in the file `fscalls.c`. This is already present in version 8 of the sample code, and contains an example system call called `stat`, which retrieves information about a file. Function prototypes in `user.h` for all of the system calls, and their assembly language stubs in `calls.s` have also been provided to you already. For stage 3 of the assignment, you should add the user space library functions to `libc.c`, and the `getdent` system call to `fscalls.c`.

When adding a new system call, you will need to add the appropriate function to `fscalls.c`, as well as edit the `syscall` function to add an additional case which calls your function to the switch statement.

5.1 Stage 1: Current working directory (20%)

Under most operating systems, each process has a property called the *current working directory*. You will be familiar with this concept from the UNIX shell, where you can move about the filesystem by changing your current working directory using the `cd` command, and print it out to the terminal using the `pwd` command. This property of processes makes it easier to reference files, since you can use relative path names instead of specifying the full absolute path name every time.

For this stage, you must do the following:

- Add a field to the `process` structure that records the current working directory of the process.
- Implement the `chdir` system call to change the current working directory of the process. This should check that the directory actually exists before changing to it, and return an error if it cannot be found. It should also return an error if the pathname refers to a file instead of a directory. Note that the path name accepted by `chdir` is relative to the existing working directory of the process.
- Implement the `getcwd` system call, which allows processes to query their current working directory. You must be careful to respect the size limit specified for the output string, to avoid writing beyond the end of memory that a process may have allocated.
- Modify the `stat` system call so that the path name it accepts is treated as being relative to the current working directory.

You may consult the UNIX man pages to get further details about how these and other system calls are supposed to work. Note that for some calls you may need to explicitly specify the system calls section of the manual (section 2), e.g. `man 2 chdir`.

5.2 Stage 2: Reading files (50%)

Files are accessed via *file handles*, which we have already seen in version 7 of the kernel, when pipes were introduced. The `read` and `write` system calls both take a *file descriptor* as their first parameter, which refers to a file handle within the kernel. This enables a process to have different files, pipes etc. open at the same time, and selectively choose which one it wants to read from or write to.

The `open` system call creates a new file handle which refers to a file. This file handle is referenced by a file descriptor, which is passed back to the process for use in later calls to `read` or `write`. For files, each file handle keeps track of the current position in that file, such that when `read` or `write` is called, the position is advanced by the appropriate amount. When a process first opens a particular file, its position is initialised to 0. If it then reads, say, 1024 bytes from the file, then the position will be advanced by 1024, and the next read will begin at that position.

To add support for accessing files, you will need to do the following:

- Add a function to create a new type of file handle, used for accessing files. This will be similar to those used for creating pipe handles - it must set the appropriate function pointers for `read`, `write`, and `destroy`, so that the respective system calls will be able to call these when the file handle is passed to them. These file handles should have some way to record information about which file they are accessing, and the current position within that file.
- Implement the `open` system call. This takes a relative path name as a parameter, and returns either a file descriptor that can be used to access the file, or a negative value indicating an error. In the latter case, the error code should be one of the `E*` macros defined near the bottom of `constants.h`, such as `ENOENT`.
- Implement the `read` function to be used with your file handles. Note that you will need to correctly handle the case where the end of file is reached; consult the man page for an explanation of what must happen in this case.
- Implement the `write` function for your file handles, which just returns `-EBADF` to indicate that the file is not writable.
- Implement the `destroy` function to get rid of the file handle. This should free all kernel memory that has been allocated for the file handle.

You can test your code by using the `cat` command in the shell, which allows you to view the contents of a particular file.

5.3 Stage 3: Reading directories (20%)

UNIX provides a set of three functions for reading the contents of directories. These are:

```
DIR *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

Notice that these are similar to the `open/read/close` functions used for reading from files, except that instead of using file descriptors, they use an object of type `DIR`. The main reason for that is so that `readdir` is able to return a structure containing the properties of a file, without requiring the programmer to explicitly allocate memory for it like they have to for obtaining file data from `read`. These are *not* implemented as system calls, because system calls cannot use `malloc` to allocate data within a process's heap.

The `opendir`, `closedir`, and `readdir` functions are all just simple wrappers around the `open`, `close`, and `getdent` system calls. `open` and `close` are used exactly as they are for files; the `opendir` wrapper simply opens a directory, gets a file descriptor back, and stores it in a newly allocated `DIR` object. `closedir` closes the file descriptor and frees the memory. The `getdent` system call, which you will also need to implement, operates very similarly to `read`, except that instead of returning an arbitrary amount of binary data, it stores the information about a directory entry in the supplied dirent structure.

When implementing `getdent`, keep in mind that the file handle will need to keep track of its position within the *directory*, not a file. This is best recorded as an index into the array of directory entry objects associated with the directory. `getdent` should return 1 whenever there was another directory entry that it was able to place the name of into the supplied `dirent` structure, and 0 when there are no more directory entries available. `getdent` should return the error code `-ENOTDIR` if invoked with a file handle that refers to a file, instead of a directory.

You can test your implementation of these functions with the `ls` and `find` commands from within the shell.

5.4 Code well (10%)

10% of the marks for this assignment are given for coding style. It is expected that you will neatly structure and properly comment the code, so that it easy to read and understand. You should use the existing sample code as a guideline as to how you should structure your code. Additionally, you should also make sure that you have paid attention to detail in your code, with respect to things like properly handling error conditions and validating input data.

5.5 Testing

There are no automated tests provided for this assignment. You should perform your own testing using the provided `cd`, `pwd`, `ls`, `find`, and `cat` commands, and using one or more other programs that you write to ensure the proper error codes are returned when appropriate.

6 Submission

Note: Your submission for this assignment should be based on the supplied code for version 8 of the kernel, *not* your code from assignment 2. You should download the code again from the web site, as it has recently been updated with a bug fix.

Your code should be checked into your SVN repository, and submitted via the web submission system in the same manner as for assignments 1 and 2.

To create a directory in your SVN repository for the prac, type the following command, all on one line (replacing `aXXXXXXX` with your own username):

```
svn mkdir -m "OS prac 3"
  https://version-control.adelaide.edu.au/svn/aXXXXXXX/os-08-s2-prac3/
```

Then type:

```
svn co https://version-control.adelaide.edu.au/svn/aXXXXXXX/os-08-s2-prac3/
```

This will place a checkout of the (empty) `os-08-s2-prac3` directory into your current working directory. Next, copy all of the supplied files from version 8 of the kernel into this directory, and run the following commands:

```
cd os-08-s2-prac3
svn add *
svn commit -m "Initial import of supplied code for assignment 3"
```

Then work on your the code from this directory, and commit as appropriate. To submit the assignment, go to `https://cs.adelaide.edu.au/for/students/automark/` and follow the prompts to get to OS practical 3, then click on "Make A New Submission For This Assignment". Read through the declaration form, and click on "I Agree" to indicate that you understand the policies relating to plagiarism. Your assignment will be marked manually once the due date has arrived.

Peter Kelly & Cruz Izu, October 2008