Exploring the file system

Johan Montelius and Amir H. Payberah

1 Introduction

This is a quite easy exercise but you will learn a lot about how files are represented. We will not look to the actual content of the files but on the so call *meta-data* that we have of each file. In order to understand how much data there is, we will implement a command that you have probably used many times in the shell. We will also gather some statistics on the size of files and present the numbers in some nice graphs.

2 List a directory

The command that we will implement is a version of the well known `ls` command, the command you use to *list* the content of a directory. Our first try will not be very impressive but at least it will do something. Let's call it `myls` so we can use the regular `ls` to verify that we do the right thing. Create a file `myls.c` and start coding.

2.1 Read the directory

To our help we have a library call `opendir()` that will make things a bit easier for us. This procedure will return a pointer to a *directory stream*, i.e., a sequence of directory entries. We can then access each of these entries using the library procedure `readdir()`. You should by now be able to figure out what header files you need to include, so we will not list them in the code sections.

```
int main(int argc, char *argv[]) {

  if (argc < 2 ) {
    perror("usage: myls <dir>\n");
    return -1;
  }

  char *path = argv[1];

  DIR *dirp = opendir(path);

  struct dirent *entry;

  while ((entry = readdir(dirp)) != NULL) {
    printf("type: %u", entry->d_type);
    printf("\tinode %lu",  entry->d_ino);
    printf("\tname: %s\n", entry->d_name);
  }
}
```

Do `man readddir` and you will see what the `dirent` structure looks like and what we could find more than what we have printed (not much). If you compile and run the program you will see your first attempt of mimicking `ls`, it works but we're far from there. This is what my attempt looked like.

```
 > gcc -o myls myls.c
 > ./myls ./
type: 4    inode: 917888        name: .
type: 8    inode: 945912        name: myls
type: 8    inode: 963825        name: myls.c
type  4:   inode: 940418        name: ..
```

So what do we have here: a type, the inode number and the name. We can make it a bit more readable by interpreting the type information.

2.2 The type

The interpretation is found in the man pages for `readdir()` and we can print them out using a `switch` statement.

```
while((entry = readdir(dirp)) != NULL) {
  switch( entry->d_type) {
    case DT_BLK  :  // This is a block device.
      printf("b:");
      break;
    case DT_CHR  :  //This is a character device.
      printf("c:");
      break;
    case DT_DIR  :  //This is a directory.
      printf("d:");
      break;
    case DT_FIFO :  //This is a named pipe .
      printf("p:");
      break;
    case DT_LNK  :  //This is a symbolic link.
      printf("l:");
      break;
    case DT_REG  :  //This is a regular file.
      printf("f:");
      break;
    case DT_SOCK :  //This is a UNIX domain socket.
      printf("s:");
      break;
    case DT_UNKNOWN :  // The file type is unknown.
      printf("u:");
      break;
  }
  printf("\tinode %lu",  entry->d_ino);
  printf("\tname: %s\n", entry->d_name);
}
```

As seen in the code above we are not talking about the *type* in terms of *pdf*, *txt* or a C source file. The type we are talking about is to differentiate directories and symbolic links etc from regular files. To find out if a particular file is a *pdf* file we would have to look at its name and see if ends in `.pdf`; this is only a convention and you're of course free to name a *pdf-file* to `foo.jpg` if you want to; doing so will however make it hard to automatically determine which application to open when you want to view the content of the file.

This is all there is to the directory, it's a mapping from names to inodes. We do not now anything about the file more than the name and remember that the name is just something that is valid in the directory that we are currently looking at.

2.3 More information

To find more information about a particular file we use the system call `fstatat()`. This procedure will populate a `stat` structure with all the information about the file we are looking for. There is also a `stat()` procedure but this procedure would look-up a file name in the *current directory* which will probably not be the directory that we are looking at. The `fstatat()` procedure allows us to specify in which directory we should do the look-up.

```
struct stat file_st;

fstatat(dirfd(dirp), entry->d_name, &file_st, 0);
```

If we insert this above the `printf()` statement we can write out more information about the file, for example its size.

```
 printf("\tinode: %lu", entry->d_ino);
 printf("\tsize: %lu", file_st.st_size);
 printf("\tname: %s\n", entry->d_name);
```

Take a look in the man pages of `fstatat()` and you will find more information about the file.

3 Things are not that simple

To disturb your world of comfort, where directories map names to inodes and inodes are something that is on some disk we will complicate things a bit. First we will create a dummy directory called `dram` and then mount a `tmpfs` file system using the directory as a mount point. Let's first create a directory and see what it looks like.

```
> mkdir foo
  :
> ./myls .
  :
```

As you see the directory is given a new inode number and we can verify that we do the right thing by looking at the output of the regular `ls` command.

```
> ls -il .
  :
```

Now let's try this - mount a `tmpfs` file system using the `dram` directory as the destination.

```
> sudo mount -t tmpfs tmpfs ./dram
 :
```

Nothing much has changed and this can be verified my looking at the output from `myls`. But check what you see when you try the regular command `ls -li` hmm, what is going on? If you wan to get even more confused look at the output when we look inside the `dram` directory.

```
> ./myls dram
  :
```

```
> ls -ila dram
  :
```

The regular `ls` command does not report what we see with `myls`, but instead collects the information from the `stat` structure. Try the following and things might be a bit more clear.

```
printf("\tinode: %lu", entry->d_ino);
printf("\tdev: 0x%lx", file_st.st_dev);
printf("\tinode: %lu", file_st.ino);
printf("\tsize: %lu", file_st.st_size);
printf("\tname: %s\n", entry->d_name);
```

As you see `ls` uses the *inode* number found in the `stat` structure rather than in the directory listing. Also note that the inode numbers of the mounted directory, `2`, is the same as in your regular root directory. If you try the command `df` you will see a list of all file systems currently mounted in your system and if do some more investigation you will find that all of them have a root directory with inode number `2`.

Inode numbers are local to a file system and the operating system needs to keep track of which file system we are talking about; or, what *device* the file system is found at. The hierarchical name space were directories serves as *mount points* for different file system, provides a seamless name space. You're normally not aware of which file system that is activated.

If you're regular Windows user you might be used to the *driver letters* `C:`, `D:` etc, these are the Windows equivalent of mounted file systems. You might wonder where `A:` and `B:` went but if you attach a `floppy disk drive` you might have it mounted as the `A:` drive.

3.1 The device

We printed the `st_dev` value in hex and we did so for a reason. The value that was printed, let's assume `0x801`, is interpreted as disk 8, partition 1. If you examine the `/dev` directory you probably see that the file system that it is referring to is your main disk drive. This might be different depending on which machine you run but you should be able to work out the details.

```
> ls -l /dev/sda*
```

So the `stat` structure contains a local inode number and which device this inode node number belongs to. This indirection from the “true” inodes that are on disk is called *virtual inodes* or *vnodes*; an abstraction layer that allows your Unix file system to present several different file systems with the same interface. Enough about mounted file systems. If you *unmount* `dram` we shall try to gather some statistics of your files on your machine.

4 Traverse the tree

Let’s write a program that counts the number of files in a directory including all sub-directories. We have most of the components we need and if we can only avoid the most obvious pit-falls we should be done i ten minutes. Create a new file `total.c` and start coding.

4.1 A recursive solution

Since we know that the directory tree is not very deep we will implement a recursive procedure. The procedure `count()` will open a directory, count the number of files and recursively count the number of files in its sub-directories. We do a quick and dirty implementation where we assume that no directory path is longer than 1024 characters - a proper solution would allocate the required amount of memory on the heap.

```
unsigned long count(char *path) {
  unsigned long total = 0;
  DIR *dirp = opendir(path);
  char subdir[1024];
  struct dirent *entry;
  struct stat file_st;

  while ((entry = readdir(dirp)) != NULL) {
    switch(entry->d_type) {
    case DT_DIR:  //This is a directory.
        :
      sprintf(subdir, "%s/%s", path, entry->d_name);
      total += count(subdir);
      break;
    case DT_REG:  //This is a regular file.
      total++;
      break;
    default:
      break;
    }
  }
  closedir(dirp);
  return total;
}
```

We have to think a bit here - the directory contains two very special entries `".."` and `"."` . If we follow these paths we will for sure wait for a very long time before we receive any results. We need to prevent the `count()` procedure from going into an endless loop so we insert the following check.

```
 if((strcmp(entry->d_name, ".") == 0) | (strcmp(entry->d_name, "..") == 0)) {
   break;
 };
```

A small `main()` procedure and we're done. If you have added all the right include directives you should be able to compile and run your file counter.

```
int main(int argc, char *argv[]) {
  if (argc < 2 ) {
    perror("usage: total <dir>\n");
    return -1;
  }
  char *path = argv[1];
  unsigned long total =  count(path);
  printf("The directory %s contains %lu files\n", path, total);
}
```

4.2 Exceeding you rights

If you try to run you program on for example `/etc` you will probably get the a segmentation fault. Instead of fixing this directly (I know the reason) we can try to use `gdb` to try to find out what happened.

```
./total /etc
segmentation fault (core dumped)
```

We first compile the program with the `-g` flag set. This will produce a binary with some additional debug information that will allow us to use `gdb`.

```
> gcc -g -o total total.c
```

We then start `gdb` giving the program as an argument.

```
> gdb total
 :
 :
```

We now have a `(gdb)` prompt and can start using the debugger. In this small example we will only run the program, let it crash and then try to figure out what happened. You start the program with the `run` command, giving `/etc` as an argument. It will look something like the following.

```
(gdb) run /etc
Starting program: ...../src/total /etc

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7ad58a2 in __readdir (dirp=0x0) at ../sysdeps/posix/readdir.c:44
44        ../sysdeps/posix/readdir.c: No such file or directory.
(gdb)
```

This tells us that the segmentation fault occurred in the system call `__readdir` where `dirp` was a *null pointer*. To see where in our code this happened we step up in the call stack.

```
(gdb) up
#1  0x000000000040084e in count (path=0x7fffffffd9f0 "/etc/...") at total.c:66
66          while((entry = readdir(dirp)) != NULL) {
(gdb)
```

Now we see were we are in our code - the while statement in the `count()` procedure. We can confirm that `dirp` is actually a null pointer by printing its value.

```
(gdb) p dirp
$1 = (DIR *) 0x0
```

The question is why; we received the pointer from the `opendir()` procedure so the question is what value `path` had when we called it. Printing the value of `path` will (in my case) reveal that something strange happened when we tried to open `"/etc/polkit-1/localauthority"`.

```
(gdb) p path
$2 = 0x7fffffffd9f0 "/etc/polkit-1/localauthority"
(gdb)
```

If you look at the directory that caused your problem (if there was one) you might find the reason for the failure.

```
> ls -ld /etc/polkit-1/localauthority
drwx------   7 root  root   4096 2016-04-21 00:11 /etc/polkit-1/localauthority
```

Hmmm, owned by `root` with the access rights `"rwx---"`. No wonder a regular user could not read that directory. Let's fix our code so we take care of the case where we, for some reason, will not be able to read the directory.

```
  if(dirp == NULL) {
    printf("not able to open %s\n", path);
    return 0;
  }
```

Ok, give it a try.

4.3 Double counting

What we're counting is the number of links to files, if a file object is linked to by several links this object will be counted twice. We could of course keep track of which files that we have seen and make sure that we only count them twice. If you tried to implemented this, how would you identify unique files? Would the inode number be enough?

5 How large are files?

If we forget about the problem with double counting, we can implement a program that generates some statistics of file sizes. We start with some assumptions and a global data structure where will store the result. Create a file called `freq.c`, or rather take copy of `total.c` since we're going to reuse most of the code.

A *frequency table* will keep track of how many files we have of certain sizes. We're not particularly interested in the exact distribution so we only keep track of the size in steps of power of two. We do not count files of size 0 and only keep track of files up to the size $2^{\text{FREQ\_MAX}}$. The last entry in the table will contain all larger files and if the machine that you're running on is not also the machine that keeps all your movies, I guess 2^{32} will be fine for our purposes.

```
#define FREQ_MAX 32

unsigned long freq[FREQ_MAX];

void add_to_freq(unsigned long size) {
  if (size != 0) {
    int j = 0;
    int n = 2;
    while (size / n != 0 & j < FREQ_MAX) {
      n = 2*n;
      j++;
    }
    freq[j]++;
  }
}
```

We now change the procedure `count()` so that it will call `add_to_freq()` with the size of each file that it sees. We don't have to return anything so there are only small changes. We check that `fstatat()` succeeds before using `file_st`, it could be a file that we do not have read permission to.

```
void count(char *path) {
 :

 case DT_REG:  //This is a regular file.
   if(fstatat(dirfd(dirp), entry->d_name, &file_st, 0) == 0) {
     add_to_freq(file_st.st_size);
   }

   break;
 :
}
```

Almost done, some small changes to the `main()` procedure and we're done.

```
    :
  printf("#The directory %s: number of files smaller than 2^k:\n", path);
  printf("#k\tnumber\n");

  for (int j= 0; j < FREQ_MAX; j++) {
    printf("%d\t%lu\n", (j+1), freq[j]);
  }
    :
```

Hmm, should work - try with a smaller directory first and then gather some statistics of a larger directory (try `/usr`). If everything works we should have nice printout of a table and we can of course not resist to explore this data using gnuplot.

6 Some nice graphs

Start by saving the histogram in a file called `freq.dat` and then you can generate your first graph in one line of code.

```
>./freq /usr > freq.dat
> gnuplot
  :
  :
gnuplot> plot "freq.dat" using 1:2 with boxes
```

If you graph looks anything like mine you see that most file are between 1K and 2K bytes (in the 2^{11} bucket). There are plenty of smaller files but few that are smaller than 32 bytes; files above one megabyte are also rare. It looks like half of the files are between 512 and $4K$ bytes. An alternative way of presenting these numbers is as a *cumulative frequency diagram*. Try the following:

```
gnuplot> a=0
gnuplot> cumulative_freq(x)=(a=a+x,a)
 :
gnuplot> plot "freq.dat" u 1:(cumulative_freq($2)) w linespoints
```

This diagram adds the frequencies as we go and shows how many files are less than a certain size. If we know that there were 280000 file in the `/usr` directory we could get a nice *y-axis* that would give us the percentage of all files.

```
gnuplot> plot "freq.dat" u 1:(cumulative_freq($2)/280000) w linespoints
```

You could of course wonder how much of the hard drive is taken up by files of what size and we can give a hint of this by multiplying the frequency with the average size of the category. If we adapt the function to take the size into account we have the following:

```
gnuplot> a=0
gnuplot> cumulative_size(k, x)=(a=a+(x*((2**k)-((2**(k-1)))/2)),a)
 :
gnuplot> plot "freq.dat" u 1:(cumulative_size($1,$2)) w linespoints
```

The graphs that we have generated now will give you a quick overview of what the file system looks like. If you would use them in a presentation you would of course do some more work to fix the scales, titles etc. The graph should contain all information needed to interpret it; it should not be open for guessing what the x-axis really means.

7 Summary

So with some simple coding I hope that you have learned some more about the file system and even if it was nothing completely new, at least it gave you some first hand experience of what it looks like. It's one thing to look at the power-point slide, another actually doing it.