CS360 Lecture notes -- Stat

- Jim Plank
- Directory: **/blugreen/homes/plank/cs360/notes/Stat**
- Lecture notes: **http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Stat/lecture.html**

This lecture covers things that you can do with **stat**, **lstat**, and the **opendir** family of calls. Chapter 4 of the book has a nice description of both of these things -- read it to solidify your understanding.

Stat

Read the man page for **stat** (or chapter 4 of the book). **Stat** is a command that you can use to get information about files -- information that is contained in the file's inode. Here I'll go over a simple motivating example. Suppose you don't have the **stat** system call, and you'd like to write a program that for each argument that is a file, lists the file's size and name. Something like the following (in **ls1.c**) would work:

```
#include < stdio.h >
#include < fcntl.h >
#include < sys/types.h >
#include < unistd.h >

main(int argc, char **argv)
{
  int i;
  int fd;
  off_t size;

  for (i = 1; i < argc; i++) {
    fd = open(argv[i], O_RDONLY);
    if (fd < 0) {
      printf("Couldn't open %s\n", argv[i]);
    } else {
      size = lseek(fd, (off_t) 0, SEEK_END);
      printf("%10ld %s\n", size, argv[i]);
      close(fd);
    }
  }
}
```

What it does is attempt to open each file, and then seek to the end of the file to figure out the size. This works ok, but copy and compile it and then try it on the following:

```
UNIX> ls1 /blugreen/homes/plank/cs360/notes/Stat/m*
```

You'll note that it returns the following:

```
       603 /blugreen/homes/plank/cs360/notes/Stat/makefile
Couldn't open /blugreen/homes/plank/cs360/notes/Stat/myfile
```

Yet, when you try a regular **ls -l**, you get:

```
UNIX> ls -l /blugreen/homes/plank/cs360/notes/Stat/m*
-rw-r--r--  1 plank         603 Sep 17 15:02 /blugreen ... Stat/makefile
```

```
----------  1 plank           3 Sep 17 15:05 /blugreen ... Stat/myfile
UNIX>
```

Since **ls1** couldn't open "**myfile**", it couldn't print out its size. This is unfortunate, but it also points out why we need the "**stat**" function -- there are things about a file that it would be nice to know, even if we're not allowed to access the file itself.

In particular, the stat function gives you information about a file's inode. It can do this as long as the user has permission to get to the directory that contains the file.

Read the man page for **stat**. The **stat** struct is defined in **/usr/include/sys/stat.h**, and is roughly:

```
struct stat {
        mode_t    st_mode;     /* File mode (see mknod(2)) */
        ino_t     st_ino;      /* Inode number */
        dev_t     st_dev;      /* ID of device containing */
                               /* a directory entry for this file */
        dev_t     st_rdev;     /* ID of device */
                               /* This entry is defined only for */
                               /* char special or block special files */
        nlink_t   st_nlink;    /* Number of links */
        uid_t     st_uid;      /* User ID of the file's owner */
        gid_t     st_gid;      /* Group ID of the file's group */
        off_t     st_size;     /* File size in bytes */
        time_t    st_atime;    /* Time of last access */
        time_t    st_mtime;    /* Time of last data modification */
        time_t    st_ctime;    /* Time of last file status change */
                               /* Times measured in seconds since */
        long      st_blksize;  /* Preferred I/O block size */
        long      st_blocks;   /* Number of 512 byte blocks allocated*/
};
```

The confusing types are mostly ints, longs, and shorts. I.e. from **/usr/include/sys/types.h**:

```
typedef unsigned long   ino_t;
typedef short           dev_t;
typedef long            off_t;
typedef unsigned short  uid_t;
typedef unsigned short  gid_t;
```

And from **/usr/include/sys/stdtypes.h**:

```
typedef unsigned short  mode_t;         /* file mode bits */
typedef short           nlink_t;        /* links to a file */
typedef long            time_t;         /* value = secs since epoch */
```

Once you have read this man page, it should be trivial to change **ls1.c** to work correctly using **stat** instead of **open/lseek**. This is in **ls2.c**:

```
#include < stdio.h >
#include < sys/types.h >
#include < sys/stat.h >

main(int argc, char **argv)
{
  int i;
  struct stat buf;
  int exists;
```

```
  for (i = 1; i < argc; i++) {
    exists = stat(argv[i], &buf);
    if (exists < 0) {
      fprintf(stderr, "%s not found\n", argv[i]);
    } else {
      printf("%10ld %s\n", buf.st_size, argv[i]);
    }
  }
}
```

```
UNIX> ls2 /blugreen/homes/plank/cs360/notes/Stat/m*
       603 /blugreen/homes/plank/cs360/notes/Stat/makefile
         3 /blugreen/homes/plank/cs360/notes/Stat/myfile
UNIX>
```

Next, we'd like to have our "**ls**" work like the real "**ls**" -- have it accept no arguments, and list all files in the current directory. To do this, we need to use the "**opendir/readdir/writedir**" calls. Read the man page (or chapter 4 in the book). Note that these are C library calls, and not system calls. This means that they call **open/close/read/write**, and interpret the format of directory files for you. This is convenient. The "**struct dirent**" structure is defined in **/usr/include/sys/dirent.h**:

```
struct  dirent {
        off_t           d_off;          /* offset of next disk dir entry */
        unsigned long   d_fileno;       /* file number of entry */
        unsigned short  d_reclen;       /* length of this record */
        char            *d_name;        /* name */
};
```

That's a little different from the book, but functionally equivalent. **Ls3.c** tweaks **ls2.c** to read from the current directory ("**.**"), and print out all files and their sizes:

```
#include < stdio.h >
#include < sys/types.h >
#include < sys/stat.h >
#include < dirent.h >

main()
{
  struct stat buf;
  int exists;
  DIR *d;
  struct dirent *de;

  d = opendir(".");
  if (d == NULL) {
    fprintf(stderr, "Couldn't open \".\"\n");
    exit(1);
  }

  for (de = readdir(d); de != NULL; de = readdir(d)) {
    exists = stat(de->d_name, &buf);
    if (exists < 0) {
      fprintf(stderr, "%s not found\n", de->d_name);
    } else {
      printf("%s %ld\n", de->d_name, buf.st_size);
    }
  }
}
```

```
UNIX> ls3
. 512
.. 1024
ls1.c 634
ls2.c 625
ls3.c 734
ls4.c 819
ls5.c 828
ls5a.c 997
ls6.c 772
makefile 603
myfile 3
ls1.o 3576
ls2.o 3252
...
```

Now, note two things when you run **ls3** -- first, the output is not formatted. Second, the files aren't sorted. This is because **readdir()** makes no guarantees about the ordering of files in a directory. The next two programs solve each of these problems.

First, formatting output. What we'd like to see is somthing like:

```
.            512
..          1024
ls1.c        634
ls2.c        625
ls3.c        734
ls4.c        819
ls5.c        828
ls5a.c       997
ls6.c        772
makefile     603
myfile         3
ls1.o       3576
ls2.o       3252
...
```

In order to do this, we need to know how long the longest filename is that we're going to print before we print out *any* filenames. So, what we do is read all the directory entries into a linked list, calculating the maximum length one along the way. After doing that, we traverse the list, and print the output in a nice format. Look closely at the **printf** statement, and read the man page on printf so that you can figure out why it works. This is **ls4.c**.

```
#include < stdio.h >
#include < string.h >
#include < sys/types.h >
#include < sys/stat.h >
#include < dirent.h >
#include "dllist.h"

main(int argc, char **argv)
{
  struct stat buf;
  int exists;
  DIR *d;
  struct dirent *de;
  Dllist files, tmp;
```

```
  int maxlen;

  d = opendir(".");
  if (d == NULL) {
    fprintf(stderr, "Couldn't open \".\"\n");
    exit(1);
  }

  maxlen = 0;
  files = new_dllist();

  for (de = readdir(d); de != NULL; de = readdir(d)) {
    dll_append(files, new_jval_s(strdup(de->d_name)));
    if (strlen(de->d_name) > maxlen) maxlen = strlen(de->d_name);
  }
  closedir(d);

  dll_traverse(tmp, files) {
    exists = stat(tmp->val.s, &buf);
    if (exists < 0) {
      fprintf(stderr, "%s not found\n", tmp->val.s);
    } else {
      printf("%*s %10ld\n", -maxlen, tmp->val.s, buf.st_size);
    }
  }
}
```

Why did I use **strdup** in the **dll_append()** call instead of **de->d_name**? The answer is subtle. The man page doesn't tell you anything about how the struct that **readdir()** returns is allocated. All that you can really assume is that until you make the next call to **readdir()** or **closedir()**, the value of what **readdir()** returns is ok. If we knew that **readdir()** mallocs space for the "**struct dirent**" that it returns, and that that space isn't free'd until the user calls **free()**, then we could readily put **de->d_name** into our dlist, and not worry about anything. However, with no such assurances from the man page, we have to call **strdup**. For example, **opendir/readdir/closedir** could be implemented as follows:

- **opendir()** opens the directory file, and mallocs one "struct dirent".
- **readdir()** reads the next directory entry into that one "struct dirent", and returns a pointer to it.
- **closedir()** closes the directory file, and free's the "struct dirent".

You should be able to reason why such an implementation necessitates that we call **strdup()** in the **dll_append()** statement. If we just put **de->d_name** there, then we're going to get into all sorts of problems with memory. This is subtle, but an important point. Make sure you understand it.

Now, to print out the directory files sorted, you just need to insert the entries into a red-black instead of a dllist. The code is in **ls5.c**. It is a very simple change.

Next, we'd like to get rid of the %10d in the **printf** statement. In other words, we want one space between the last column of the file names and the first column of the file sizes. We do this by finding the maximum size of the file size while traversing the directory, and using that in the **printf** statement. This takes a **sprintf** and a **strlen** -- see **ls5a.c**.

Finally, **ls6.c** performs the same function as "**ls -F**". That is, it prints directories with a "/" at the end, symbolic (soft) links with a "@", and executable files with a "*". We are able to do this by interpreting the "**st_mode**" field of the "**struct buf**". Look over the code.