CS360 Lecture notes

LICENSE

CS360 Lecture notes
Prsize: recursive directory traversal

- James S. Plank
- Directory: **/home/jplank/cs360/notes/Prsize**
- Start Date: Mid 1990's.
- Latest revision: Thu Feb 18 14:03:51 EST 2021
- Git Hash: 79645f157f6c67f0daa11ae07ac9ea10f2bc0c3e

To "follow along" with these lecture notes, if you are on the lab machines, simply do:

```
UNIX> cd /home/jplank/cs360/notes/Prsize/test1
```

If you are on your own machine, then follow these instructions:

Pull the repo, and cd to the Prsize directory. (You may need to go to the **../Libfdr** directory and type **make** first.).

Next, untar the test directories. This is because bitbucket doesn't do a good job of preserving soft links and directory protections:

```
UNIX> tar xpfv test-directories.tar
test1/
test1/d1/
test1/d1/f3
test1/d1/file_with_a_longer_filename.txt
test1/f1
test1/f2
test2/
test2/f4-soft
.....
UNIX>
```

Finally:

```
UNIX> cd test1
```

Now, you're set up.

Starting out: Calculating the size of all files in the current directory

This lecture covers the writing of a command **prsize**. What **prsize** does is return the number of bytes taken up by all files reachable from the current directory (excluding soft links). It is a good program as it illustrates using **opendir/readdir/closedir, stat**, recursion, building path names, and finding hard links.

First, I wrote **src/prsize1.c**. This prints the total size of all files in the current directory. It is a simple use of `stat` and `opendir/readdir/closedir`:

```c
/* This program prints the size of all files in the current directory. */

#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>

int main()
{
  DIR *d;                   /* Return value of opendir(). */
  struct dirent *de;        /* Return value of each readdir() call. */
  struct stat buf;          /* The information about each file returned by stat() */
  int exists;               /* Return value of stat on each file. */
  long total_size;          /* The total size of all files. */

  d = opendir(".");         /* Open "." to list all the files. */
  if (d == NULL) {
    perror(".");
    exit(1);
  }

  total_size = 0;

  /* Run through the directory and run stat() on each file,
      keeping track of the total size of all of the files.  */

  for (de = readdir(d); de != NULL; de = readdir(d)) {
    exists = stat(de->d_name, &buf);
    if (exists < 0) {
      fprintf(stderr, "Couldn't stat %s\n", de->d_name);
    } else {
      total_size += buf.st_size;
    }
  }

  /* Although the closedir call isn't necessary, it will be later... */

  closedir(d);
  printf("%ld\n", total_size);
  return 0;
}
```

Test it out on your current directory (which is `test1`):

```
UNIX> pwd
/home/jplank/cs360/notes/Prsize/test1
UNIX> ../bin/prsize1
357
UNIX>
```

You may get a different value (e.g. on my home machine, I got 12314), but if you do a long listing of all files, your value should equal the sum of all of the file sizes. For example:

```
UNIX> ls -la
total 8
drwxr-xr-x. 3 jplank jplank  36 Feb 11  2018 .
drwxr-xr-x. 9 jplank jplank 240 Feb 18 14:31 ..
drwxr-xr-x. 2 jplank jplank  55 Sep 23  1994 d1
-rw-r--r--. 1 jplank jplank  11 Sep 23  1994 f1
-rw-r--r--. 1 jplank jplank  15 Sep 23  1994 f2
UNIX> echo 36 + 240 + 55 + 11 + 15 | bc
357
UNIX>
```

Making the program recursive to go through subdirectories

Now, the next step we'd like to take is to get the program to sum up the sizes of all files *reachable* from the current directory. To do this, we need to make the program recursive. Instead of putting all our code in the `main()` routine, we'll instead bundle it into a function, and call that function. src/prsize2.c does this. It provides the same functionality as `src/prsize1.c`, except that it makes a call to `get_size()` to find the size. There's no real reason to put the program here – just click on the link if you want to see it.

Note there is no recursion yet – that is for s**rc/prsize3.c**. If you test **bin/prsize2**, you'll see that it does the same thing as **bin/prsize1**.

```
UNIX> ../bin/prsize2
357
UNIX>
```

Now, we want to make `bin/prsize2` recursive. Whenever we encounter a directory, we want to find out the size of everything in that directory, so we will call `get_size()` recursively on that directory. This is done in src/prsize3.c. Here is the relevant code in `get_line()`.

```
/* Run through the directory and run stat() on each file,
keeping track of the total size of all of the files.  */

for (de = readdir(d); de != NULL; de = readdir(d)) {
  exists = stat(de->d_name, &buf);
  if (exists < 0) {
    fprintf(stderr, "Couldn't stat %s\n", de->d_name);
    exit(1);
  }
  total_size += buf.st_size;

  /* If the file is a directory, make a recursive call to get_size(): */

  if (S_ISDIR(buf.st_mode)) {
    total_size += get_size(de->d_name);
  }
}
```

When we try it, we get an odd error:

```
UNIX> ../bin/prsize3
.: Too many open files
UNIX>
```

So, what's happening? Well, to check, I put a print statement into src/prsize3a.c to see when it's making the recursive calls:

```
UNIX> ../bin/prsize3a
Making a recursive call to .
Making a recursive call to .
Making a recursive call to .
Making a recursive call to .
........                                    # Deleting many lines
Making a recursive call to .
.: Too many open files
UNIX>
```

Now you can see what's happening. When enumerating files in ".", you come across the file ".". This is a directory, so you make a recursive call on it. This goes into an infinite loop until you run out of open file descriptors at which point `opendir()` fails. To fix this, you need to check and see whether or not you are trying to make a recursive call to the "." directory. You need to check for ".." as well. We'll do that in the next program:

After checking for "." and ".." we still have problems

We put the checks into src/prsize4.c:

```
  /* If the file is a directory, and not . or .. make a recursive call to get_size(): */

  if (S_ISDIR(buf.st_mode) && strcmp(de->d_name, ".") != 0 && strcmp(de->d_name, "..")
      != 0) {
    total_size += get_size(de->d_name);
  }
}
```

When we run it, the infinite loop bug is fixed, but we have another bug!!

```
UNIX> ../bin/prsize4
Couldn't stat f3
UNIX>
```

Ok, now what's the problem? Where on earth is **f3**?

```
UNIX> find . -name f3 -print          # find is a super-helpful command.  Read the man page.
./d1/f3
UNIX>
```

That gives us the answer: the program is trying to stat `f3` in the directory `d1`, but it's not working in the directory `d1`. In other words, `prsize4` is called from the directory `Prsize/test1`, and makes the call "`exists = stat("f3", &buf)`". Of course, `stat` is going to return -1, because there is no file `f3` in the directory. Instead, we need to look for "`d1/f3`". In other words, our code has a bug – we need to be looking for `fn/de->d_name` in `get_size()`, and not just `de->d_name`.

Building the file names solves one problem and uncovers another

We build the filenames in src/prsize5.c. Here is the code that has changed. It makes you yearn for C++ strings, doesn't it?

```
  int fn_size;            /* This is the length of fn -- so we can build the filename. */
  char *dir_fn;           /* This will be the filename including the directory. */
  int dir_fn_size;        /* This is the bytes in dir_fn is, in case we need to make it
      bigger. */
  int sz;

  /* Skip the initialization. */

  /* ... */

  /* Start building the directory + files.   We'll start by setting dir_fn_size to
      fn_size+10, and we'll make it bigger as we need to.  It will be more efficient to
      use a number bigger than 10 for this, but 10 will let us debug the code if there's
      a problem.

     I'm also setting up dir_fn to hold the directory name and a slash. */

  fn_size = strlen(fn);
  dir_fn_size = fn_size + 10;
  dir_fn = (char *) malloc(sizeof(char) * dir_fn_size);
  if (dir_fn == NULL) { perror("malloc dir_fn"); exit(1); }
  strcpy(dir_fn, fn);
  strcat(dir_fn + fn_size, "/");

  /* Run through the directory and run stat() on each file,
     keeping track of the total size of all of the files.  */

  for (de = readdir(d); de != NULL; de = readdir(d)) {

    /* First, we need to build dir_fn.  First check to see if it's big enough, and if
        not, we'll call realloc() to reallocate space.  Then we put the filename after
        the slash. */

    sz = strlen(de->d_name);
    if (dir_fn_size < fn_size + sz + 2) {       /* The +2 is for the slash and null
        character. */
      dir_fn_size = fn_size + sz + 10;          /* The +10 adds some extra.  Again, I'd
          make this bigger, but I want to debug. */
      dir_fn = realloc(dir_fn, dir_fn_size);
    }
    strcpy(dir_fn + fn_size + 1, de->d_name);   /* I'm starting after the slash, and not
        at the beginning of the string. */

    exists = stat(dir_fn, &buf);                /* Use dir_fn instead of de->d_name. */
    if (exists < 0) {
      fprintf(stderr, "Couldn't stat %s\n", dir_fn);
      exit(1);
    }

    total_size += buf.st_size;

  /* If the file is a directory, and not . or .. make a recursive call to get_size(): */

    if (S_ISDIR(buf.st_mode) && strcmp(de->d_name, ".") != 0 && strcmp(de->d_name, "..")
        != 0) {
      total_size += get_size(dir_fn);
    }
  }
```

```
  closedir(d);
  free(dir_fn);                /* Have to free the memory that we allocated. */
  return total_size;
}
```

When we run it, it looks pretty good. I put a longer filename into **d1** so that the **realloc()** code is tested:

```
UNIX> ../bin/prsize5
1322
UNIX> ls d1
f3  file_with_a_longer_filename.txt
UNIX>
```

Before moving on, we should sanity check our output:

```
UNIX> ../bin/prsize5
1322
UNIX> ls -la
total 8
drwxr-xr-x. 3 jplank jplank  36 Feb 18 15:00 .
drwxr-xr-x. 9 jplank jplank 240 Feb 18 15:45 ..
drwxr-xr-x. 2 jplank jplank  55 Feb 18 15:44 d1
-rw-r--r--. 1 jplank jplank  11 Sep 23  1994 f1
-rw-r--r--. 1 jplank jplank  15 Sep 23  1994 f2
UNIX> ls -l d1
total 8
-rw-r--r--. 1 jplank jplank  17 Sep 23  1994 f3
-rw-r--r--. 1 jplank jplank 857 Feb 18 15:44 file_with_a_longer_filename.txt
UNIX> echo 36 + 240 + 55 + 11 + 15 + 17 + 857 | bc
1231
UNIX>
```

Hold on – 1322 does not equal 1231. Might not be a bad time to put a print statement in front of the **stat()** call. I'm not going to do that, because I know the problem – I didn't consider "d1/." and "d1/.." when I calculated 1231 above:

```
UNIX> ls -la d1
total 8
drwxr-xr-x. 2 jplank jplank  55 Feb 18 15:44 .
drwxr-xr-x. 3 jplank jplank  36 Feb 18 15:00 ..
-rw-r--r--. 1 jplank jplank  17 Sep 23  1994 f3
-rw-r--r--. 1 jplank jplank 857 Feb 18 15:44 file_with_a_longer_filename.txt
UNIX> echo 36 + 240 + 55 + 11 + 15 + 55 + 36 + 17 + 857 | bc
1322
UNIX>
```

Do you think it's right to count the size of both "./d1" and "d1/."? Or both "." and "d1/.."? I don't. If we move to **../test2**, we'll reveal a related problem:

```
UNIX> cd ../test2
UNIX> ls
f4  f4-hard-link
UNIX> ls -lai
total 8
     7876 drwxr-xr-x. 2 jplank jplank  36 Feb 11  2018 .
402660801 drwxr-xr-x. 9 jplank jplank 240 Feb 18 15:45 ..
     7877 -rw-r--r--. 2 jplank jplank  11 Sep 23  1994 f4
     7877 -rw-r--r--. 2 jplank jplank  11 Sep 23  1994 f4-hard-link
UNIX>
```

As you can see, **f4** and **f4-hard-link** are links to the same file. When we run **bin/prsize5**, it of course counts both of them:

```
UNIX> ../bin/prsize5
298
UNIX> echo 36 + 240 + 11 + 11 | bc
298
UNIX>
```

Time for our next fix:

Keeping track of inodes so we only count each file once

What we need is for **prsize** to be able to recognize hard links, and only count them once.

How do you recognize whether two files are links to the same disk file? You use the inode number. This is held in **buf.st_ino**.

The way we check for duplicate inodes is to maintain a rb-tree of inodes that we have seen so far. Before adding in the size of any file, we check to see if its inode is in the rb-tree. If so, we do nothing. Otherwise, we add in the size, and put the inode into the rb-tree. It is an unfortunate matter that on some systems, inodes are longs rather than ints, so to store them in a JRB, we use the ".l" field of the jval, and add a custom comparison function.

The code is in **src/prsize6.c**, and as usual, I'll just highlight the changes.

```
int compare(Jval v1, Jval v2)           /* Adding a comparison function for inodes. */
{
if (v1.l < v2.l) return -1;
if (v1.l > v2.l) return 1;
return 0;
}

long get_size(const char *fn, JRB inodes)    /* get_size now passes the tree of inodes.
    */

  /* A lot of code deleted. */

  /* Check the inodes tree to check if we've seen this file before.
      If so, ignore.  If not, then add in its size. */

  if (jrb_find_gen(inodes, new_jval_l(buf.st_ino), compare) == NULL) {
    jrb_insert_gen(inodes, new_jval_l(buf.st_ino), new_jval_i(0), compare);
    total_size += buf.st_size;
  }

  /* If the file is a directory, and not . or .. make a recursive call to get_size(): */

  if (S_ISDIR(buf.st_mode) && strcmp(de->d_name, ".") != 0 && strcmp(de->d_name, "..")
      != 0) {
    total_size += get_size(dir_fn, inodes);    /* I add the inode tree to get recursion.
       */
  }
}

closedir(d);
free(dir_fn);
return total_size;
}

int main()
{
long total_size;
JRB inodes;              /* I create the inode tree in main and pass it to the initial
    get_size() call.  */

  inodes = make_jrb();
  total_size = get_size(".", inodes);
  printf("%ld\n", total_size);
  return 0;
}
```

This fixes our previous problems:

```
UNIX> ../bin/prsize6
287                          # This is 11 less than before, so it's correct.
UNIX> cd ../test1
UNIX> ../bin/prsize6
1231                         # This matches our first calculation above, so it's correct.
UNIX>
```

Should we worry about soft links?

Look at the **test3** directory.

```
UNIX> cd ../test3
UNIX> ls -lai
total 4
134269623 drwxr-xr-x. 2 jplank jplank  58 Sep 24  1996 .
402660801 drwxr-xr-x. 9 jplank jplank 240 Feb 18 16:16 ..
134345832 -rw-r--r--. 1 jplank jplank  11 Sep 23  1994 f5
134345834 lrwxrwxrwx. 1 jplank jplank   2 Aug  1  2014 f5-soft-link -> f5
134345833 lrwxrwxrwx. 1 jplank jplank   1 Aug  1  2014 soft-link-to-. -> .
UNIX>
```

There are a couple of soft links here – let's see what they do to **bin/prsize6**:

```
UNIX> ../bin/prsize6
Couldn't stat ./soft-link-to-./soft-link-to-./soft-link-to-./soft-link-to-. ...
UNIX>
```

The code is in **src/prsize6.c**, and as usual, I'll just highlight the changes. So, what has happened? Since we're using **stat()**, **bin/prsize6** doesn't recognize soft links, and thus we have the same infinite loop problem as before. It should be clear what we want – instead of traversing the link to ".", we want **prsize** to count the size of the link itself (2 bytes for **f5-soft-link** and 1 byte for **soft-link-to-.**). Thus, all we need to do in [**prsize7.c**](https://web.eecs.utk.edu/ jplank/plank/classes/cs360/360/notes/Prsize/prsize7.c) is use **lstat()** instead of **stat()**. This gives information about the soft link itself, instead of the file to which the link points:

```
UNIX> ../bin/prsize7
312
UNIX> echo 58 + 240 + 11 + 2 + 1 | bc
312
UNIX>
```

We're not done yet – there's a subtle bug with open files.

Finally, there's one more bug in this program, and it's really subtle. It has to do with open file descriptors. First, go to the directory **test4**. Below, I use the `find` command to show that it is composed of 10 nested directories. You can see that **bin/prsize7** works just fine on it:

```
UNIX> cd ../test4
UNIX> find . -print
.
./1
./1/2
./1/2/3
./1/2/3/4
./1/2/3/4/5
./1/2/3/4/5/6
./1/2/3/4/5/6/7
./1/2/3/4/5/6/7/8
./1/2/3/4/5/6/7/8/9
UNIX> ../bin/prsize7
381
UNIX>
```

The reason that it works is that our defaults typically allow for a ton of open files per process. We can see this by running the BASH command **ulimit** (if your shell doesn't recognize this command, try **limit** instead):

```
UNIX> ulimit -a | grep open
open files                      (-n) 1024
UNIX>
```

On my Macintosh, this is even bigger – 2560? It's like system administrators *want* you to program wastefully! Let's set this number to ten instead of 1024. Now, **bin/prsize7** fails because of too many open files:

```
UNIX> ulimit -n 10
UNIX> ulimit -a | grep open
open files                      (-n) 10
UNIX> ../bin/prsize7
./1/2/3/4/5/6/7: Too many open files
UNIX>
```

What's happening is that the recursive calls to **get_size()** are made in between the **opendir()** and **closedir()** calls. That means that each time we make a recursive call, we add one to the number of open files. With only ten open files (and three open to start the process), we run out of file descriptors when we try to open "./1/2/3/4/5/6/7".

The solution to this is to make sure that there are no open files when we make the recursive call. How do we do this? When enumerating the files in a directory, we put all directories into a dllist, and then after closing the directory file, we traverse the list and make the recursive calls. We need to do a **strdup()** when we put the directories into the dllist. Why? Think it over, or see what happens when you don't do it... The correct and final version of prsize is in **src/prsize8.c**. Here are the changes:

```
long get_size(const char *fn, JRB inodes)
{
  /* Other variable declarations are deleted. */

  Dllist directories, tmp;  /* Dllist of directory names, for doing recusion after
      closing. */

  /* Initialize (other code deleted). */

  directories = new_dllist();

  for (de = readdir(d); de != NULL; de = readdir(d)) {

    /* Other code deleted */

    /* Don't make the recursive call, but instead put the directory into the dllist. */

    if (S_ISDIR(buf.st_mode) && strcmp(de->d_name, ".") != 0 && strcmp(de->d_name, "..")
        != 0) {
      dll_append(directories, new_jval_s(strdup(dir_fn)));
    }
  }

  /* Make the recursive calls after you've closed the directory.  */

  closedir(d);

  dll_traverse(tmp, directories) {
    total_size += get_size(tmp->val.s, inodes);
  }
```

```
  /* Clean up.  You need to free the strings inside the dllist, because you
      allocated them with strdup(), and they'll be a memory leak otherwise. */

  dll_traverse(tmp, directories) free(tmp->val.s);
  free_dllist(directories);
  free(dir_fn);

  return total_size;
}
```

Now it works even with 10 file descriptors:

```
UNIX> ulimit -n 10
UNIX> ../bin/prsize8
381
UNIX>
```

As an aside, it's 2021, and **find** still has the same bug as **prsize7.c**:

```
UNIX> ulimit -a | grep open
open files                      (-n) 10
UNIX> find . -print
.
./1
./1/2
./1/2/3
./1/2/3/4
find: â€˜./1/2/3/4â€™: Too many open files
UNIX>
```

On the flip side, **tar** handles it correctly:

```
UNIX> tar cvf ~/junk.tar .
./
./1/
./1/2/
./1/2/3/
./1/2/3/4/
./1/2/3/4/5/
./1/2/3/4/5/6/
./1/2/3/4/5/6/7/
./1/2/3/4/5/6/7/8/
./1/2/3/4/5/6/7/8/9/
UNIX>
```

When I first wrote this lecture, in the mid-1990s, I made **test4** have 257 subdirectories, rather than 10. That way, I didn't have to mess with the **ulimit** command. Within a day, I had an email from our system administrator, complaining that the directory broke the system backup program. It also broke **tar**. So, I changed the directory to its current structure. I like to think that the good folks who write system tools fixed **tar** because they stumbled upon my lecture notes. A man can dream, can't he?
