CSE509: Lab 2

Sanket Goutam

October 19, 2021

1 Warmup

1.1 Get familiar with strace by reading its man page and by using it. What are the system calls made by ls when you run it with no options? What about ls -l? Try the command on directories large and small to have confidence that you have exercised ls well. Can you explain the reasons for differences in system calls made?

The ls program internally calls functions that are defined in the glibc system library. Using strace on any program (like ls) will allow us to view the system calls that glibc library is making.

The following are a small selection of the commands that were run when I executed "strace ls temp/". Note that I created this temp folder which has a random number of files. For demonstration purposes, I am using 3 files inside temp currently but will create more files for testing.

```
sanket@sanket-VirtualBox:~/Documents$ strace ls temp/
execve("/bin/ls", ["ls", "temp/"], 0x7ffc56ef4c58 /* 22 vars */) = 0
brk(NULL)                               = 0x5648643d0000
.
.
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "temp1  temp2  temp3\n", 20temp1  temp2  temp3
)   = 20
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
```

The output above shows all the system calls that were made just to run the "ls" command on a given directory. There are a bunch of system calls that are being made by ls here, which can broadly be classified into the following sections:

- Process management system calls
- File management system calls
- Directory and filesystem management system calls
- Other system calls

Some interesting functions to note from the strace output above are execve. From going through the man pages of execve(), it is clear that execve just executes a program pointed to by a file name. So in this case, execve() is the function which executes "ls" with the "temp" directory as an argument. The **openat** system call opens temp directory, **getdents** system call gets the system directories. Once we get these information, we use **write** system call to write this message to STDOUT.

```
sanket@sanket-VirtualBox:~/Documents$ strace ls -l temp/
.
lstat("temp/temp2", {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
lgetxattr("temp/temp2", "security.selinux", 0x55f6d0c75990, 255) = -1 ENODATA (No data availabl
getxattr("temp/temp2", "system.posix_acl_access", NULL, 0) = -1 ENODATA (No data available)
lstat("temp/temp1", {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
lgetxattr("temp/temp1", "security.selinux", 0x55f6d0c75aa0, 255) = -1 ENODATA (No data availabl
getxattr("temp/temp1", "system.posix_acl_access", NULL, 0) = -1 ENODATA (No data available)
lstat("temp/temp3", {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
lgetxattr("temp/temp3", "security.selinux", 0x55f6d0c75bd0, 255) = -1 ENODATA (No data availabl
getxattr("temp/temp3", "system.posix_acl_access", NULL, 0) = -1 ENODATA (No data available) 198
openat(AT_FDCWD, "/usr/share/locale/en_US.UTF-8/LC_MESSAGES/coreutils.mo", O_RDONLY) = -1 ENOEN
```

With *ls -l*, we see a bunch of additional system calls being made. Some interesting ones are **lstat** and **getxattr**. **lstat** gets the status of a file or a symbolic link, **getxattr** retrieves the value of the extended attribute identified by name and associated with the given path in the filesystem. Since with *ls -l* we need to view all file permissions, ownership information, and any symbolic links as well, there are all these additional system calls that are being made by *ls -l*. Most of the additional system calls seem to have to do with fetching all metadata associated with each file.

1.2 Get familiar with ltrace. Use it to study ls in the same was as the problem above. Use appropriate options to ltrace (or process the output using tools such as grep or awk) so as to leave out calls to utility functions such as malloc, free, getenv, getopt, strcmp, and so on, while only retaining calls that look like system calls. (But do not use the -S option to ltrace.) Comment on the output you get this way, as compared to strace output.

ltrace allows us to check what library calls are being made by a program. It monitors all the libraries that a program is using and shows which functions are getting invoked. Refer to the snippet below which shows a small selection of system calls being made by ls.

A directory called temp/ is being opened by the opendir library function, followed by calls to the readdir function. This **readdir** function is reading the contents of the directory. Finally there is a call to the **closedir** function, which closes the directory that was opened earlier.

There is also an option in ltrace to print the frequency of each call to a function. There are a very high number of str related calls being made, but these are only for string matching of the files and the directories.

```
opendir("temp/")          = 0x55956c68fa10
readdir(0x55956c68fa10)   = 0x55956c68fa40
readdir(0x55956c68fa10)   = 0
closedir(0x55956c68fa10)  = 0
```

```
sanket@sanket-VirtualBox:~/Documents$ ltrace -c ls temp/

temp1  temp2  temp3
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
16.16    0.008646         480        18 strlen
14.65    0.007839         489        16 __ctype_get_mb_cur_max
10.15    0.005430         543        10 __errno_location
8.19    0.004380         486         9 getenv
6.05    0.003236         539         6 readdir
1.81    0.000969         969         1 opendir
1.11    0.000595         595         1 closedir
```

With the *ls -l* there are many more functions being invoked. The most frequently invoked function seems to be **getpwuid**. This function returns a pointer to a structure containing the fields of the record in the password database for a user ID. I am not entirely sure why this function gets invoked

repeatedly, but my assumption would be that it has something to do with getting the file permissions for a user.

```
sanket@sanket-VirtualBox:~/Documents$ ltrace -c ls -l temp/
total 0
-rw-rw-r-- 1 sanket sanket 0 Oct 11 10:27 temp1
-rw-rw-r-- 1 sanket sanket 0 Oct 11 10:27 temp2
-rw-rw-r-- 1 sanket sanket 0 Oct 11 10:27 temp3
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
15.91    0.047429       47429         1 getpwuid
13.63    0.040642         564        72 iswprint
2.47    0.007359         566        13 getenv
1.27    0.003786        1893         2 setlocale
```

1.3 Use strace to count the number of files accessed by applications when they are started up. Answer this question for ls, nano, Open office and Firefox or Chromium. Note that complex applications may create multiple child processes. Use appropriate options to strace so that it captures the system calls made by all of these processes. Note that applications such as Firefox open thousands of files on start up; if you don't see this, then you are making some mistake.

For this exercise, my understanding is that we want to identify how many files are being accessed by a program during startup. Now, based on the previous exercises, there are a handful of system calls which can handle files. Functions like *open, openat, chmod, unlink, statfs, etc.* all include an argument to a filename. strace does provide a way to *trace* for specific system calls when attaching to a process, so technically we could run strace with a process and filter for specific "file-handling" system calls.

```
strace -fy -e trace=open,openat,read,write,access -o ls_strace.txt ls temp/
```

This command above does exactly that. This command will filter the system calls made by ls for only the specific system calls specified in the "trace" parameter. However, this is not the optimal way. The man pages of strace suggest another command line option which includes all of these system calls that take a "filename" as an argument. The command below is used for this part of the question.

```
strace -f -e trace=file -o ls_strace.txt ls temp
```

```
execve("/bin/ls", ["ls", "temp"], 0x7ffe07378790 /* 22 vars */) = 0
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
statfs("/sys/fs/selinux", 0x7ffdf33f1ff0) = -1 ENOENT (No such file or directory)
statfs("/selinux", 0x7ffdf33f1ff0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/proc/filesystems", O_RDONLY|O_CLOEXEC) = 3
access("/etc/selinux/config", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
stat("temp", {st_mode=S_IFDIR|0775, st_size=4096, ...}) = 0
openat(AT_FDCWD, "temp", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3
```

For "ls" there were a total of 15 files being accessed. Some of these function calls are accessing the filesystem, like statfs which checks a mounted filesystem. But we can consider this as a "file" related operation as well.

One thing to note here, is that I am not accounting for one specific file access for the purposes of this assignment. The file *"/etc/ld.so.nohwcap"* is used to disable the loading of optimized libraries. So when any program is run, they will repeatedly check for the presence of this file using an *access* system call. When running strace against any given program, I saw repeated function calls for this file. I am assuming that this file is checked before any library gets loaded by the program. The repeated entries were creating some confusion, so I decided to exclude this particular file from the final tally. So the following tally of "file accesses" is offset by 1 file.

NOTE: Assignment asked us to use *openoffice* to get the number of files accessed by the program. I didn't have openoffice installed in my VM, and trying to install it was taking a lot of time (I was using minimal Ubuntu install), so I decided to test the same with *LibreOffice* instead.

Process name	Files accessed
ls	15
nano	179
libreoffice	3168
firefox	12203

The commands used to run the tests are below, and I later processed these files to remove duplicate filenames in order to get unique files being opened/accessed at startup.

```
strace -f -e trace=file -o firefox_strace.txt firefox
```

```
strace -f -e trace=file -o libreoffice_strace.txt libreoffice --writer
strace -f -e trace=file -o nano_strace.txt nano
strace -f -e trace=file -o ls_strace.txt ls temp/
```

2 Transparent Application Functionality Extension

I have attached the tarball for the "url2file" program along with the makefile. You will need to perform the following commands to run test cases:

```
make
./url2file wc www.google.com
./url2file wc -l www.google.com
./url2file gedit www.google.com
./url2file bash www.google.com
./url2file nano www.google.com
./url2file vim www.google.com
./url2file grep href www.google.com
./url2file cat www.google.com | head -5
./url2file cat http://google.com | grep http
```

Team Members: Sanket Goutam, Sai Bhavana Ambati