Benjamin Michalowicz
CSE 509 HW2, Part 1

1. Strace Results on ls, ls -l, and directories large and small

To start off with reading strace's documentation, using the "-c" flag appears to be the simplest way of obtaining the list of system calls for a given application (e.g. /bin/ls).

There are exactly ninety-three syscalls made across the twenty-one used in /bin/ls for /home/sekar, when used with no other options. The order in which the syscalls appear in the printout differ between each run, though this is because of the changing times taken by some of the syscalls while in kernel space. In a smaller directory, such as ~/HW2, where there's initially only one file (this report), ninety-one system calls are made, and in a blank directory (e.g. ~/HW2/test), eighty-nine system calls are made. Depending on the directory (e.g. ~ versus /), the number of system calls change when ls uses different flags. For example, doing "strace -c ls -l" on my /home/sekar directory results in 311 system calls. The same action in ~/HW2 results in 196 system calls made without the dummy "test" directory, and 227 system calls with it as part of ~/HW2's contents. Calls to functions within the *stat* family deal with checking for file types, inode numbers, file system, device number, etc. I would venture to guess that mmap2 and brk would deal with managing heap space as more and more files are recognized by /bin/ls.

Across each run of strace, errors tend to show up in the same system calls, namely within statfs64(2) and access(2). Looking at the possible errors for access(2), the main cause would most likely be an incorrectly specified mode, a bad flag, or maybe a pathname pointing outside the given process's address space. There are consistently two errors in statfs64, which is most likely because of trying to access the parent and current directory symbols, which would be outside the directory and irrelevant for anything other than having a reference to oneself.

Attempting the above in a larger directory, in this case /bin, "strace -c ls" returns with 117 syscalls called and nine errors among those calls, and "strace -c ls -l" returns with 986 syscalls called and 319 errors.

When "ls -l is used," over thirty syscalls are used in generating its output. This is mainly because of having to access the read, write, and execution attributes of each file at the user, group, and world levels.

2. Ltrace results

Ltrace has similar flags to that of strace, though one thing I noticed with ltrace was the -e flag; this flag comes with a "filter" argument that takes a bastardized regex pattern and shows only the library calls that match the pattern and its contents. A special variant of this in ltrace is being able to add and remove from the filter using + and -. The man pages show the example "malloc+free-@libc.so*," which refers to monitoring malloc and free calls, but no other libc calls. This logic was used to remove most of the usually Linux utility functions in these demos (*alloc, str*, get*) to keep only the library calls that "look like" system calls. Attempting to use formatting tools such as awk, grep, and sed, and piping output through these tools started taking more time than what I considered necessary to be able to complete these tests.

In using a modified pattern for these tests (e.g ltrace -ce @MAIN-malloc-free-strlen-getenv-getopt*-str*-*alloc-f*-io*-mem* ls) returned 404 library calls that were analyzed when running "ls" on /home/sekar on my HW1 VM.

Compared to the small change in system calls called when moving to a smaller directory, the number of library calls drastically decreases when moving to something small, such as ~/HW2 (404 library calls to 32), where, and then the number of library calls still drops in an empty directory (~/test) to 20. Performing this test in a larger directory, /bin, gives 3,260 library calls used that are not utility functions. Because a system call requires going into kernel space, thus generally taking longer, and libc

functions don’t necessarily have to do this, library calls tend to be favored over syscalls when speed is needed.

Using ls -l on /home/sekar, ~/HW2, ~/test, and /bin, with the same specifications as above, results in 942, 249, 219, and 5365 library calls. The main reason behind these increases is because of also having to go through to collect meta-data such as last access, file size, ownership, and read, write, and execute permissions. Interestingly enough, there are certain library calls (getxattr, readdir64, etc), who do not have man pages associated with them. At the same time, the jump from 32 library calls to 249 does not quite match up, proportionately speaking, with the jump from 404 to 942 (a jump to nearly eight times the number of calls, compared to only a 2.5x increase). One reason why this might be the case is the addition of calls needed to either read files versus directories, as well as having to go through many different hoops and system calls (nested within the library calls) to try and obtain all the information required by the -l flag in ls -l.

3. Strace for file access count

Strace has two main features for being able to access files when running an application. One is using “-e trace=%desc,” which, according to the Linux man pages, traces all file descriptor-related system calls. The otheris “-e trace=%file,” which traces all the system calls which take a file name as an argument. From here, we can use one or both of these system calls to keep track of the number of files accessed by, say, /bin/ls. We will also see what happens when opening an even larger program, such as nano and Mozilla Firefox.

For pure file accessing on its own, we will take a look at only the number of times access(2) and open(2) called in each case through the use of the “-e trace=%file” option. My full command used for each test is “strace -ce trace=%file ___” where ___ refers to a given

In the case of /bin/ls, seven calls to access(2) are made by the application, though in the HW1 VM, and since strace also shows return values and error codes, it’s shown that all seven return with ENOENT. In addition, there are two statfs64 system calls, and both also return with ENOENT. In addition, nine open(2) calls are made in the process of executing /bin/ls, none of which fail.

In the case of /bin/nano, nine calls to access(2) are made, with seven returning ENOENT. In the case of these accesses, /bin/nano is appearing to look for files helping its configuration in multiple spots, and two of the nine access(2) calls succeed upon finding these files. In addition, nano makes fifty-nine calls to open(2), with six of them returning errors. Looking at the trace for many of these calls, nano appears to require more than a few *.nanorc files in order to properly function.

Lastly, I present the case of Mozilla Firefox. In starting Firefox up, 652 calls are made to access(2), with 253 of them returning an error, all with ENOENT. This refers to a similar problem in the previous processes, where a certain file is requested, but cannot be found. Similarly, 879 open(2) calls are made in starting up Firefox, with 246 of them returning with an error. In addition to configuration files, a larger application like Firefox also has to have access to kernel memory to request space for multiple processes and CPU scheduling to ensure that with potentially many tabs, no race conditions occur.

Below is the result of “strace -ce trace=%file” for each of the processes mentioned above: