Strace in XV6

Xiang Mei
xm2146@nyu.edu

April 27, 2022

1 Introduction

Strace is a very useful tool on Linux. It's widely used to do the troubleshooting. But we don't have pre-installed strace on XV6. I'll implement a simple strace on XV6.

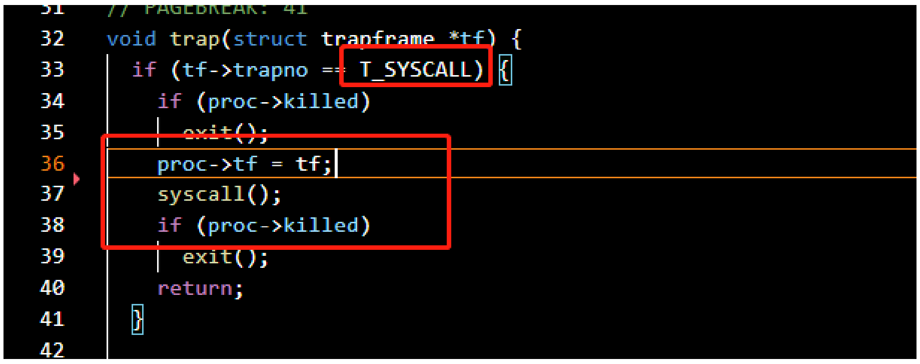
It sounds like reinventing a wheel. But for my expeirence in this course, Intro to OS, I think write a simple version of "wheel" could help me to understand the complesity of the "wheel" and help me think as an engineer. We need to consider about lots of questions during the implementation, such as "why should the wheel be round?". Anyway, I learned a lot and used some skills I learn from previous assignment.

The first section is the introduction of the report and I'll document my design in the second section. The real task-related parts start from the section 3.

2 How to monitor the syscall

At first, a simple plan came up in my mind. I can insert a piece of code in every syscall so that I can print the result and parameters when running the syscall. And I set a global variable "strace_all" to tell the system if we should print the strace info while calling syscalls. But quickly I found there are some serious issues with this solution. We have 21 syscalls and if we need to write different strace_handle for every syscall and it's not convinience because we may need to modify 21 places for every single change.

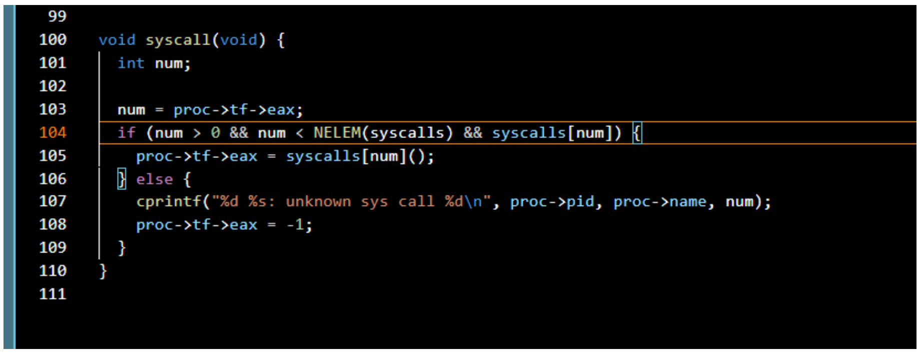
If we need to print the content of the syscall we have to do that because the syscalls would parse the parameters in these syscall handlers. I asked in slack and found we don't need to print the details about the syscall so that we can move the strace code to the syscall_interupt handle or the wrapper function.



```
// PAGEBREAK: 41
void trap(struct trapframe *tf) {
  if (tf->trapno == T_SYSCALL) {
    if (proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if (proc->killed)
      exit();
    return;
  }
```

Figure 1: Syscall Handler in Trap Fcuntion

As we talked in this trap section, the syscall in user space would use create an interupt to inform the kernel. And the interup is handled by the function alltraps in "trapasm.S" it would store current context in the trap frame and call function trap which is shown on the above figure. And we can see the trap would check the process's state and call the function syscall. This function is in file syscall.c.



```
void syscall(void) {
  int num;

  num = proc->tf->eax;
  if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n", proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

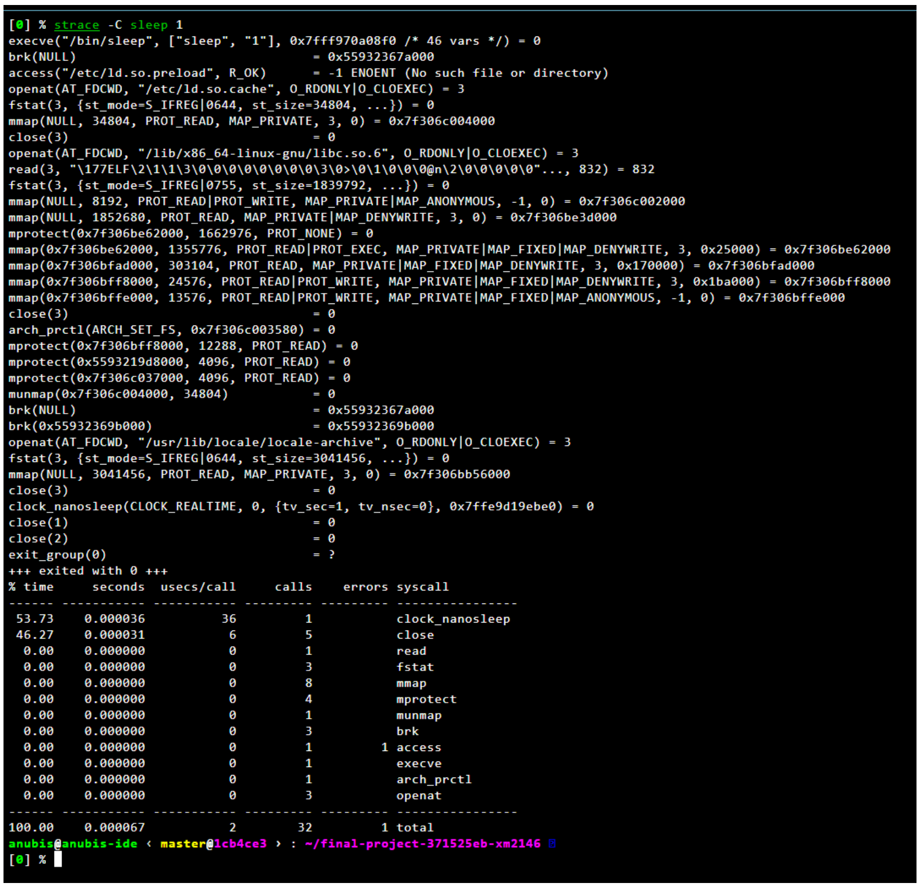
Figure 2: Syscall Wrapper

In this function, we would parse the EAX which represent the syscall index and store the return value in the EAX. So I think this function is a good candidate to insert our strace code.

Because we need to interactive with kernel, I add a syscall named strace to switch between the strace_OFF and strace_ON. In order to prevent "strace traces strace_stscall", I add some check to the strace_logger.

3 Get familiar with Linux strace

In order to get familiar with the real strace, I use it to trace the sleep command. After reading the help page of strace, I use the "-C" flag to show the list of called syscalls, total number of calls, time of running strace on command.



```
[0] % strace -C sleep 1
execve("/bin/sleep", ["sleep", "1"], 0x7fff970a08f0 /* 46 vars */) = 0
brk(NULL)                               = 0x55932367a000
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
fstat(3, {st_mode=S_IFREG|0644, st_size=34804, ...}) = 0
mmap(NULL, 34804, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f306c004000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@n\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1839792, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f306c002000
mmap(NULL, 1852680, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f306be3d000
mprotect(0x7f306be62000, 1662976, PROT_NONE) = 0
mmap(0x7f306be62000, 1355776, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x25000) = 0x7f306be62000
mmap(0x7f306bfad000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x170000) = 0x7f306bfad000
mmap(0x7f306bff8000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7f306bff8000
mmap(0x7f306bffe000, 13576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f306bffe000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f306c003580) = 0
mprotect(0x7f306bff8000, 12288, PROT_READ) = 0
mprotect(0x5593219d8000, 4096, PROT_READ) = 0
mprotect(0x7f306c037000, 4096, PROT_READ) = 0
munmap(0x7f306c004000, 34804)           = 0
brk(NULL)                               = 0x55932367a000
brk(0x55932369b000)                     = 0x55932369b000
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3041456, ...}) = 0
mmap(NULL, 3041456, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f306bb56000
close(3)                                = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffe9d19ebe0) = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 53.73    0.000036          36         1           clock_nanosleep
 46.27    0.000031           6         5           close
  0.00    0.000000           0         1           read
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         8           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         3           openat
------ ----------- ----------- --------- --------- ----------------
100.00    0.000067           2        32         1 total
anubis@anubis-ide ‹ master@1cb4ce3 › : ~/final-project-371525eb-xm2146
[0] %
```

Figure 3: Task 1-1

For task 1-2, I choose ”mmap, read, execve, mprotect” as the targets to explain. We can see the procedure clearly on above figure. The ”execve” call is called once and that’s the first syscall we called. The ”sleep” is an executable file in our system and the execve syscall could run it. It’s worthy to mention the executed binary would use the caller’s memory space and proc struct. By the way, the execve syscall has three 3 parameters, the first one is the path of the executable binary. And the second one would store the arguments while the third one would store the enviroment parameters.

The mmap syscall is used to allocate a large chunk of memory. In our command, it’s used to allocate memory to store the shared libraries, the linker and other needed files. As we can see in above figure, mmap is usually called after openat. The first parameter of mmap syscall is the address of the allocated chunk. You can set it NULL to represent arbitrary address and we can also set it a non-NULL value to get a chunk strat from that adddress.

This is used to allocate more space based on known chunk. The second parameter is the length we want to allocate while the thrid arguments is the permission of the chunk, such as readable, writeable, and executeable.

And the mprotect is used to change the permission of memory. The mprotect can't allocate new memory. It could only change the permission of the memory chunks. In our command, it's used to make mmaped memory not writeable. For example, we allocate a chunk for the shared libraries and the memory must be writeable because we need copy the bytecodes to the memory. But you know it's dangerous to make writeable memory executable. So we need mprotect to make it un-writeable after copying.

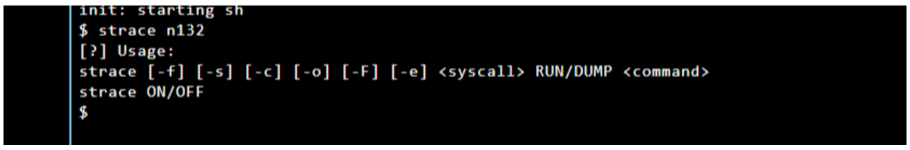
The syscall read is kind of straitforward. It reads the content from the first parameter's corresponding file and store to the second parameter's correctly memory while the third parameter is the max length of content the read syscall could read. It's used once to read the header of our glibc.

So far, we go through the usage and the shown information of strace on linux. Strace is a useful tools and I have been using it for a long time but I still find something new by reading it's help page. And we are going to implement out strace.

4 What features do we need

In this assignment, we gonna implement a simple strace. I'll go through and features needed.

For my implementation, the command should be like:



```
init: starting sh
$ strace n132
[?] Usage:
strace [-f] [-s] [-c] [-o] [-F] [-e] <syscall> RUN/DUMP <command>
strace ON/OFF
$
```

Figure 4: Usage

We have 4 sub-commands("RUN", "DUMP", "ON", and "OFF"), 3 filter options("-e","-s",and "-f"), and 3 output control options("-F","-c", and "-o").

I'll introduce these options and sub-commands in later section. And there are a short version introduction: The sub-command "RUN" could strace the following command and trace the syscall until the following command exits and the "DUMP" would dump the syscall records in the kernel. Also we can use "strace on" to ask the kernel keeps recording syscalls while the "strace off" could get the kernel back to the normal mode.

If we add "-e" options, the strace would only record and print specific syscalls. Besides, "-s/f" flag would force the strace to only record and print successful/failed syscalls.

Moreover, we have output control options. The ”-F” flag would print more readable output and the ”-c” would print a table of syscall metrics. As for ”-o”, it can set an output file and the strace’s output would be stored in that file.

5 Design

This section will tell you the reasons of my design. This time, not like the ”uniq” assignment, our implementation is different from the real sample because we don’t have pthread syscall in xv6. Also, the strace is a big project I don’t have much time to read its code. So during the implementation, I try to solve the problem myself and look up the matrials when I don’t have an elegant solution.

ON/OFF

> ”When typing ’strace on’ in the terminal, the mode of strace is on and therefore the next type in command will be traced. The system call list will be printed on screen in format pid (process id), command name, system call name, return value.”

The ”ON/OFF” implementation is easy and straitforward. In the requirements, we need to print the pid, name, and syscall of

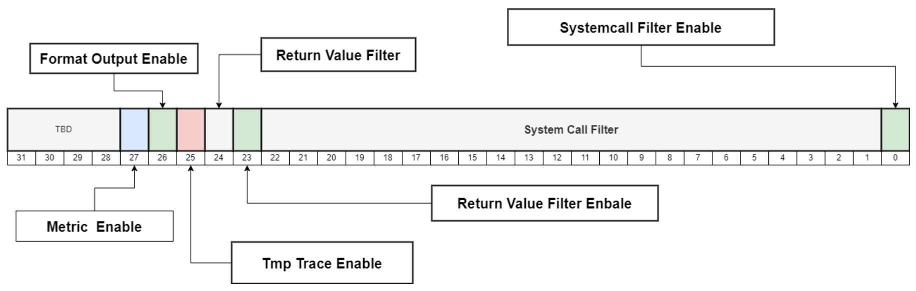


Figure 5: Structure of pstrace in proc struct