# Strace in XV6

Xiang Mei
xm2146@nyu.edu

April 27, 2022

## 1 Introduction

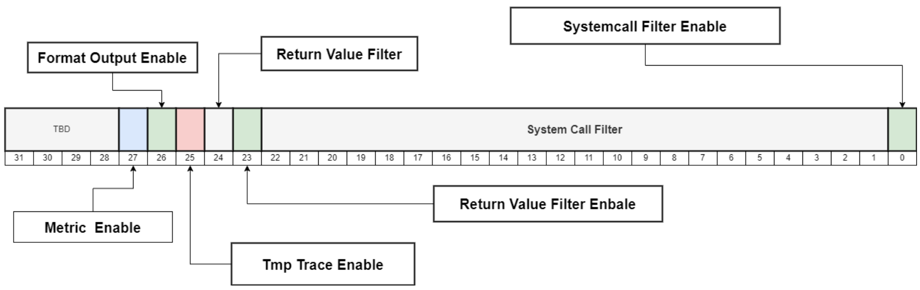



Figure 1: My Strace

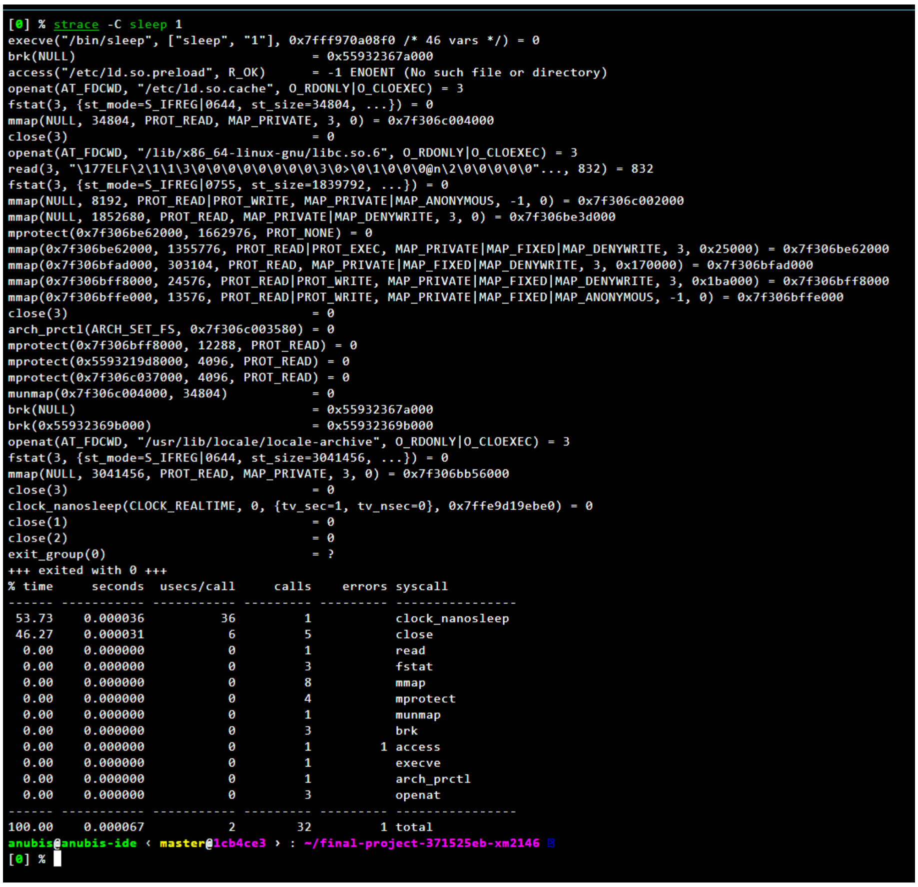
Strace is a very useful tool on Linux. It's widely used to do the troubleshooting. But we don't have pre-installed strace on XV6. I'll implement a simple strace on XV6.

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 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 2: 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.

## 3 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:

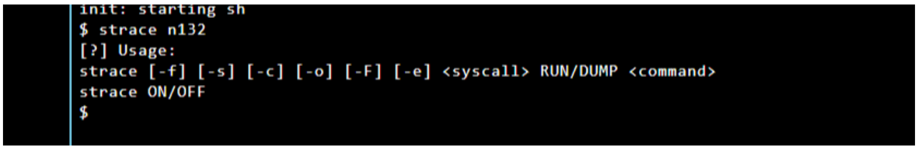



Figure 3: Usage

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.

# 4 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 processes. Obviously, we can't do this task in userspace. I create an global variable in kernel space to present current strace_mode.

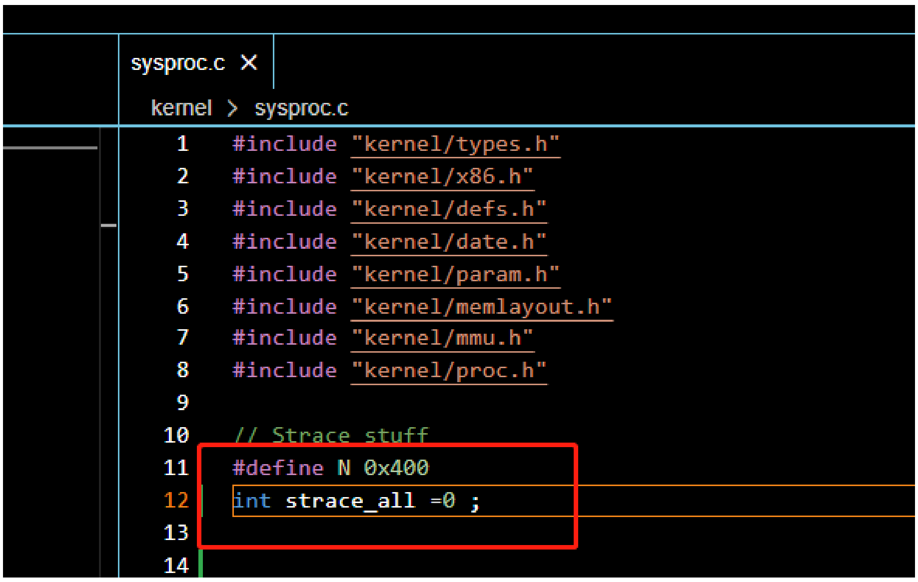
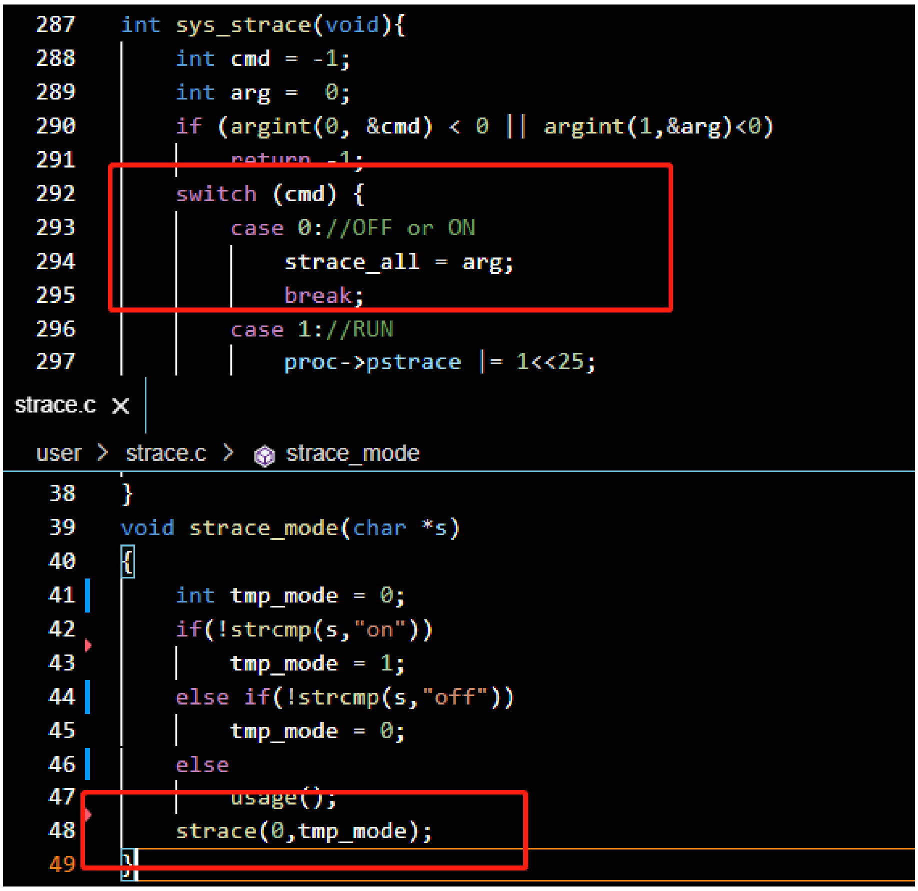



Figure 4: Strace All

Like what shows in the above figure, I set a global variable in "sysproc.c" because I'll later implement a syscall as a bridge to stransfer command from user space to the kernel.



```
int sys_strace(void){
    int cmd = -1;
    int arg =  0;
    if (argint(0, &cmd) < 0 || argint(1,&arg)<0)
        return -1;
    switch (cmd) {
        case 0://OFF or ON
            strace_all = arg;
            break;
        case 1://RUN
            proc->pstrace |= 1<<25;
```

```
strace.c ×
user > strace.c > strace_mode
}
void strace_mode(char *s)
{
    int tmp_mode = 0;
    if(!strcmp(s,"on"))
        tmp_mode = 1;
    else if(!strcmp(s,"off"))
        tmp_mode = 0;
    else
        usage();
    strace(0,tmp_mode);
}
```

Figure 5: Strace Syscall

As you can see in the above figure, I parse the parameters from the command line and use strace syscall to modify the kernel variable. Basically, we can use the code above to control the strace mode from user space. Nevertheless, how do we monitor the syscalls.

At very 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 use 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. My expeirence told me it's horriable. We must implement something more elegant.

The advantage of the previous plan is that we can print the arguments of syscalls. 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.

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

Figure 6: 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 7: 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.

```
void syscall(void){
  int num;
  uint time_recorder=0;
  num = proc->tf->eax;
  if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // if(proc->pstrace&(1<<27))
    //      time_recorder= sys_uptime();
    proc->tf->eax = syscalls[num]();
    // if((proc->pstrace)&(1<<27))
    //      time_recorder = sys_uptime() - time_recorder;
    add_one_record(proc->pstrace,proc->pid,proc->name,num,proc->tf->eax,time_recorder);
  } else {
    cprintf("%d %s: unknown sys call %d\n", proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

Figure 8: Monitor

I insert the monitor after the syscall because we need the return value the syscall but we may loss "exit" syscall because the process would stop in "exit" syscall. So I add the same function before the process really exits which is shown in the figure 9.

```
int sys_exit(void) {

    add_one_record(proc->pstrace,proc->pid,proc->name,2,(int)0xdeadbeef,0);
```

Figure 9: Strace Handler in sys_exit

The usage of this sub-command is simple. You can just use "strace on" to turn on strace and "strace off" to turn off strace.

```
$ strace on
$ echo hello
TRACE: pid = 4  |       Command_name = echo      |       syscall = exec   |       Return value = 0
hTRACE: pid = 4 |       Command_name = echo      |       syscall = write  |       Return value = 1
eTRACE: pid = 4 |       Command_name = echo      |       syscall = write  |       Return value = 1
lTRACE: pid = 4 |       Command_name = echo      |       syscall = write  |       Return value = 1
lTRACE: pid = 4 |       Command_name = echo      |       syscall = write  |       Return value = 1
oTRACE: pid = 4 |       Command_name = echo      |       syscall = write  |       Return value = 1

TRACE: pid = 4  |       Command_name = echo      |       syscall = write  |       Return value = 1
TRACE: pid = 4  |       Command_name = echo      |       syscall = exit   |
$ strace off
$ echo hello
hello
$
```

Figure 10: Strace On and OFF

There is no thing more about these two sub-commands but there is little problem about the global variable. In order for avoiding race condition, we need to implement lock mechanism for this variable. Howerver, this not the requirement for this assignment and there is only one member in my team. I decide to do implement that only if I have time left.

**DUMP**

> "Implement a kernel memory that will save N number of latest events. This N number can be configurable by using define in XV6. In order word, it can be hard code but the way to implement is to use define to declare a variable called N with certain value. When 'strace dump' command is called, print all events that saved in kernel memory."

In previosu figures, you may noticed that I implemented a function to log the syscall. so why do I implement a such conplex function "add_one_record".

For implementing the "DUMP" feature, we need to allocate a space in the kernel to store the latest N system calls. These data have to be stored in the kernel space as a global variable because xv6 is a multi-process system. And we need a special circle buffer to store latest N records.

```c
    while(1) // Record Inputs
    {
        read(0,&c,1);
        if(c==0xa) break;
        buf[cur] = c;
        cur++;
        if(cur>=N){
        cur = 0; flag=1;
        }
    }
    //Dump Inputs
    if(flag==0)
        for(i = 0 ;  i < cur ;i++)
            putchar(buf[i]);
    else{
        i = cur;
        do{
            putchar(buf[i]);
            i++;
            if(i>=N) i = 0;
        }while(i!=cur);
    }
    putchar(0xa);
```

Figure 11: Circle Buffer

Circle buffer for DUMP operation is easier than general circle buffer. First, I implemented a C code version for testing. As you can see in the above figure, We read the input to the circle buffer and dump the content when we get an ”enter”. The read-part is simple and the ”flag” variable is important. It decides how many and where to dump. The trick in the code is simple and makes sure we would print the last N records which is veried by my fuzzer. That’s another reason why I write it in C. Also, I attach my fuzz code:

```python
def payload_gen(l):
    res = ''
    for x in range(l):
        res+=table[random.randint(0,len(table)-1)]
    return res
def fuzzer():
    p = process("./main")
    pay = payload_gen(random.randint(1,0x100))
    p.sendline(pay)
    output = p.readline()[:-1].decode()
    if output != pay[-N:]:
        print("Bug Found")
        exit(1)
    p.close()
if __name__ == "__main__":
    while(1):
        fuzzer()
```

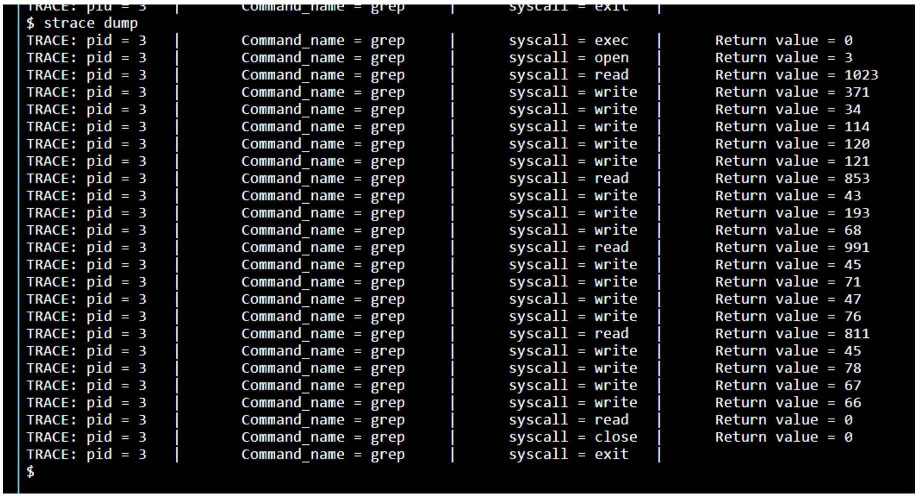
Figure 12: Circle Buffer Fuzzer

After other testing, I implement a similar circle buffer in the kernel to store and dump the syscalls. In function "add_one_record", I record the syscall's inform in the correct node and move the pointer like what I did in my previous demo.

```c
void add_one_record(unsigned int pstrace,int pid,char *name,uint 
{
    if( !(pstrace&(1<<25)) && !strace_all)//Pruned Mode···
    if(!strncmp(name, "strace", 7) || !strncmp(name, "sh", 3))//
    if(pstrace&(1<<23)){···
    }// -e or -c
    if( pstrace & 1 ){···
    }
    else if ( pstrace & (1<<27)){···
    }
    strace_record[strace_cur].pid = pid;
    strncpy(strace_record[strace_cur].name,name,0x10-1);
    strace_record[strace_cur].sys_id = sys_id;
    strace_record[strace_cur].ret_val = ret_val;
    struct strace_node * p = &strace_record[strace_cur];
    if(!(pstrace &(1<<26)))···
    strace_cur++;
    if(strace_cur >= N)
    {
        strace_cur = 0;
        strace_flag = 1;
    }
    return;
}
```

Figure 13: Circle Buffer Fuzzer

That's the key function of all my design I'll metion this function later to introduce otehr features. This function is only called in function syscall and sys_exit so that if we want to modify, delete, or add a feature to strace, we can just modify the code in this function. Another advantage is that we can naturally combine kinds of options.



```
$ strace dump
TRACE: pid = 3  |  Command_name = grep  |  syscall = exec   |  Return value = 0
TRACE: pid = 3  |  Command_name = grep  |  syscall = open   |  Return value = 3
TRACE: pid = 3  |  Command_name = grep  |  syscall = read   |  Return value = 1023
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 371
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 34
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 114
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 120
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 121
TRACE: pid = 3  |  Command_name = grep  |  syscall = read   |  Return value = 853
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 43
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 193
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 68
TRACE: pid = 3  |  Command_name = grep  |  syscall = read   |  Return value = 991
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 45
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 71
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 47
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 76
TRACE: pid = 3  |  Command_name = grep  |  syscall = read   |  Return value = 811
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 45
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 78
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 67
TRACE: pid = 3  |  Command_name = grep  |  syscall = write  |  Return value = 66
TRACE: pid = 3  |  Command_name = grep  |  syscall = read   |  Return value = 0
TRACE: pid = 3  |  Command_name = grep  |  syscall = close  |  Return value = 0
TRACE: pid = 3  |  Command_name = grep  |  syscall = exit   |
$
```

Figure 14: DUMP after Traing ”grep the readme.md”

I did several tests on DUMP and it works well. I attach a simple sample above because the output of complex testcase would be big and hard to recognize. You can also test it with any commands you like and please check the README.md file attached to the assignment submission.

**RUN**

> ”Instead of turning on and off strace, we create ’strace run’ to directly point tracing to the current process that execute the command. For example: when typing ’strace run echo hello’ in the terminal, we get the output tracing of echo hello.”

For run, things are going easier