Strace in XV6

Xiang Mei
xm2146@nyu.edu

April 29, 2022

1 Introduction

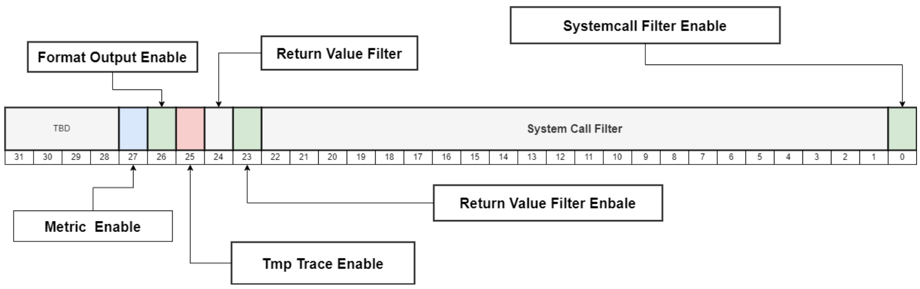


Figure 1: My Strace

Strace is a very useful tool on Linux. It's widely used to perform troubleshooting. But we don't have preinstalled strace on XV6. I'll implement a simple strace on XV6.

It sounds like reinventing a wheel. But for my experience in this course, Intro to OS, I think writing a simple version of the "wheel" could help me to understand the complexity of the "wheel" and help me think like an engineer. We need to consider 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 the 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 section 3.

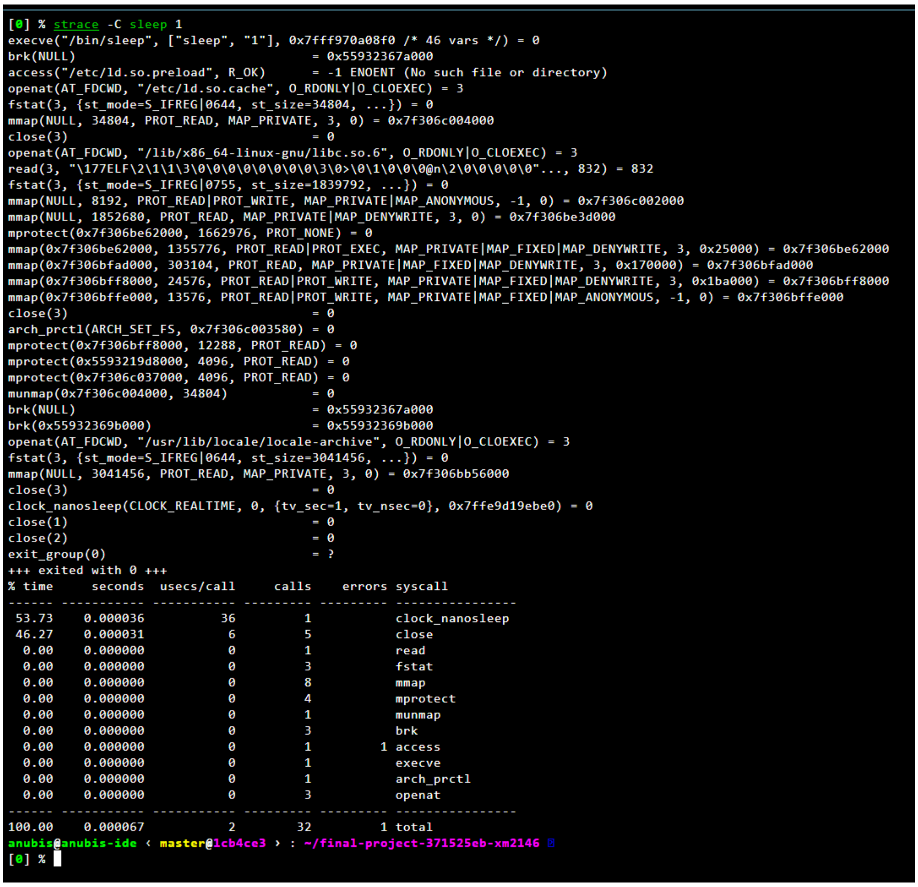
Information for TA team My strace could satisfies all the requirments in the write-up, including "strace on", "strace off", "strace run command", "strace dump", "Trace child process", "Option: -e system-call-name", "Option: -s", "Option: -f", "Option runs only once", "Write output of strace to file", and "Application of strace".

Also, my strace satisfies all the extra credits, including "Extra credits: Formatting more readable output", "Extra credits: Combine options", and "Extra credits: Implement -c options".

All explanations and screenshots are in Section 4 Design; The README file of this assignment is at /README.MD; "A text that state your partner or working alone" is at /FYI.MD; and you can find "Application of strace" content in section 5.

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, and 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 tasks 1-2, I choose "mmap, read, execve, mprotect" as the targets to explain. We can see the procedure clearly in the 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 environment 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 the 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 an arbitrary address and we can also set it to a non-NULL value to get a chunk strat from that address. This is used to allocate more space based on a known chunk. The second parameter is the length we want to allocate while the third argument is the permission of the chunk, such as readable, writeable, and executable.

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 to 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 straightforward. It reads the content from the first parameter's corresponding file and stores the content in the second parameter's correct 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 tool and I have been using it for a long time but I still find something new by reading its 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:

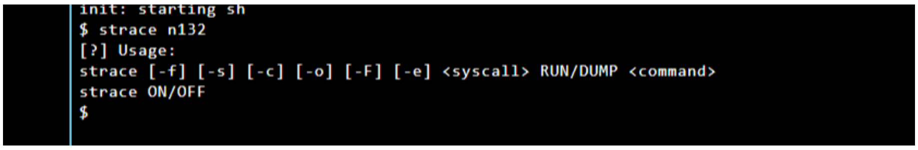


Figure 3: Usage

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 a 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 to keep 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, the "-s/f" flag would force the strace to only record and print successfully/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 for 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 materials 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 the screen in format PID (process id), command name, system call name, return value."

The "ON/OFF" implementation is easy and straightforward. 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 a global variable in kernel space to present the current strace_mode.

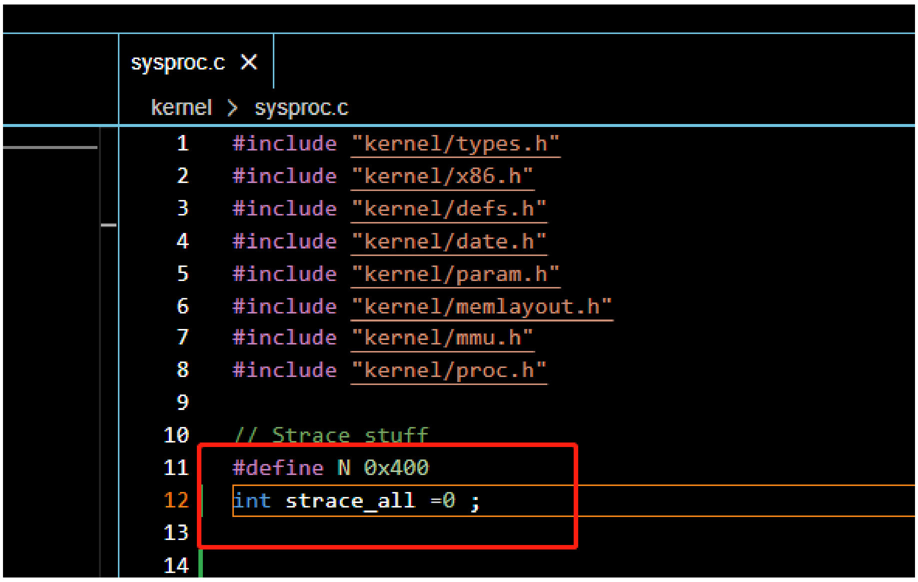


Figure 4: Strace All

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 transfer commands from user space to the kernel.

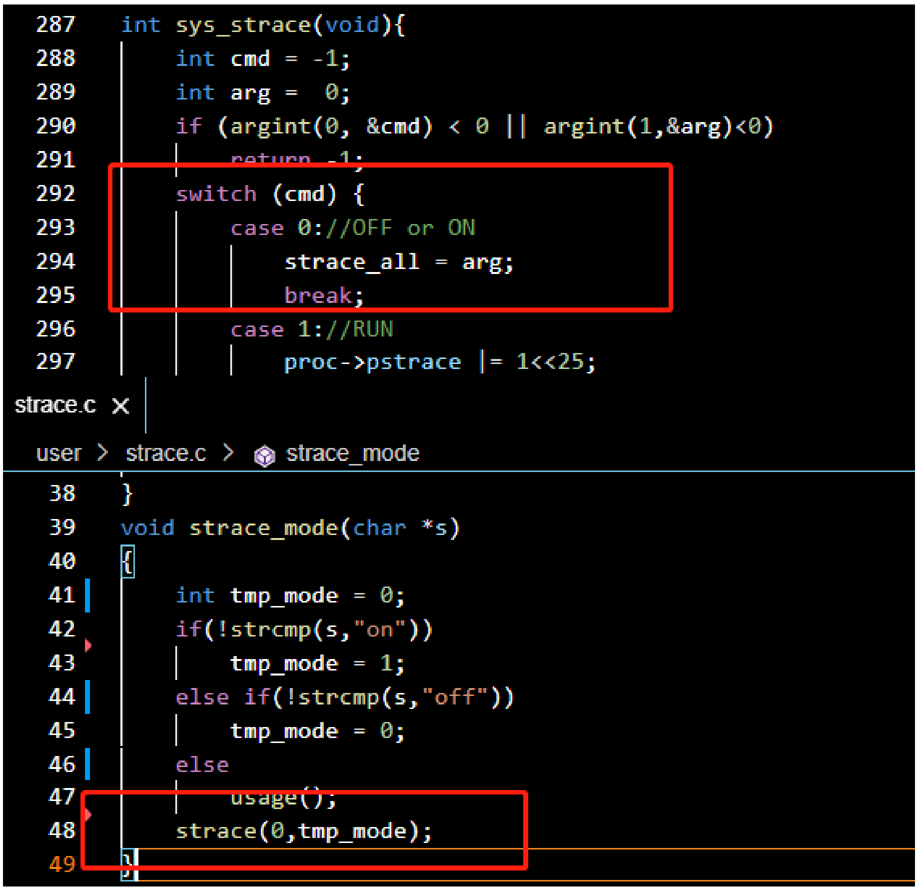


Figure 5: Strace Syscall

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 userspace. 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 into 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 convenient because we may need to modify 21 places for every single change. My experience told me it's horrible. 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 about in this trap section, the syscall in userspace would use create an interrupt to inform the kernel. And the interrupt is handled by the function alltraps in "trapasm.S" it would store the current context in the trap frame and call the function trap which is shown in 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 represents 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.

```c
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 of the syscall but we may lose the "exit" syscall because the process would stop in the "exit" syscall. So I add the same function before the process really exits which is shown in figure 8.

```c
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 nothing more about these two sub-commands but there is a little problem with the global variable. In order for avoiding race conditions, we need to implement a lock mechanism for this variable. However, this is not the requirement for this assignment and there is only one member on my team. I decide to 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 words, it can be hard to code but the way to implement is to use define to declare a variable called N with a certain value. When 'strace dump' command is called, print all events that are saved in kernel memory."

In previous figures, you may notice that I implemented a function to log the syscall. so why do I implement a such complex 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 the 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

The circle buffer for DUMP operation is easier than the 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 are verified 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 another test, 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 information in the correct node and move the pointer like what I did in my previous demo.

```
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 designs I'll mention this function later to introduce other 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.

```
TRACE: pid = 3   |   Command_name = grep   |   syscall = exit   |
$ 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 a complex test case 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 executes the command. For example: when typing 'strace run echo hello' in the terminal, we get the output tracing of echo hello."

For sub-command "RUN", we gonna run a command and strace the syscall of the command. So we can parse the parameter and use the "fork" or "exec" syscall to run the command.

I used the "exec" syscall for my strace and there is a disadvantage compared with the fork version. I know the fork version from my professor: we can use "fork" to create a child process and let the parent process wait for the metadata of outputs from the kernel. So all the output parts would be handled in user space which is much more secure and beautiful. However, I have almost finished my implementation and there is not much time for this huge modification. Therefore, I'll keep my "exec" version and for my implementation, I think they are similar in complexity.

Type	Description	Options
Sub-Command	strace on	-
Sub-Command	strace off	-
Sub-Command	RUN	[-e] [-s] [-f] [-o] [-F] [-c]
Sub-Command	DUMP	[-o]
Option	-F	More readable output
Option	-f	Only record failed syscalls
Option	-s	Only record successful syscalls
Option	-e	Only record specific syscalls
Option	-c	Print out a table of used syscalls
Option	-o	Redirecte the strace output to a file

Figure 15: Supported Opetions

The "RUN" sub-command is the most complex part of my strace, I'll introduce some basic ideas about this sub-command in this parameter and explain the rest parts with the supported options. So how can we strace one command once?

If we "RUN" a command, we are going to use "fork" to create a new process or use "exec" to run an executable file by using current memory space. We must have a way to pass the information that the new process should be traced. In order to implement this, I add a new element in the "proc" struct in proc.

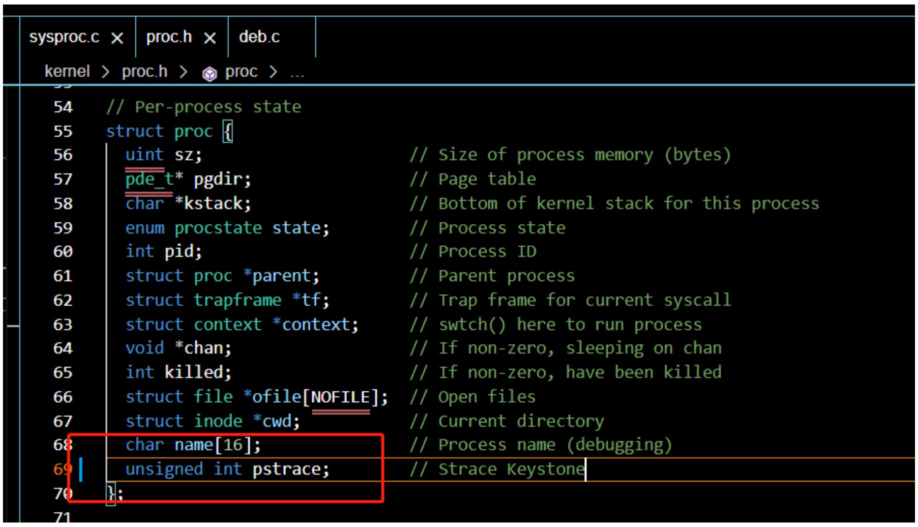


Figure 16: Strace a Process

As you can see in the above figure, I add an unsigned integer to the proc struct. I don't want to waste unnecessary space in the kernel space. so I would use this 4 bytes variable to store all the strace information such as output filter information and output format information. I'll explain the struct of this variable in the option-related paragraph.

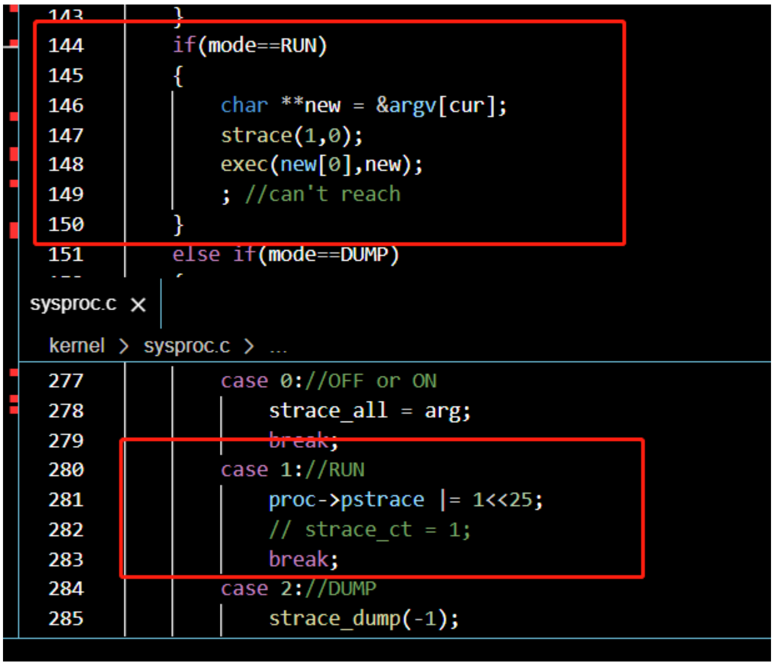
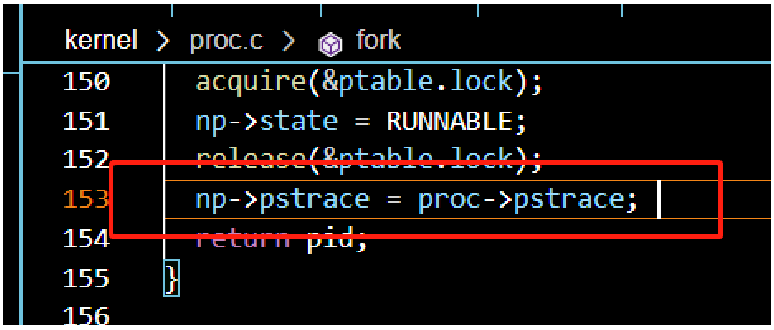


Figure 17: RUN Sub-command in User Space

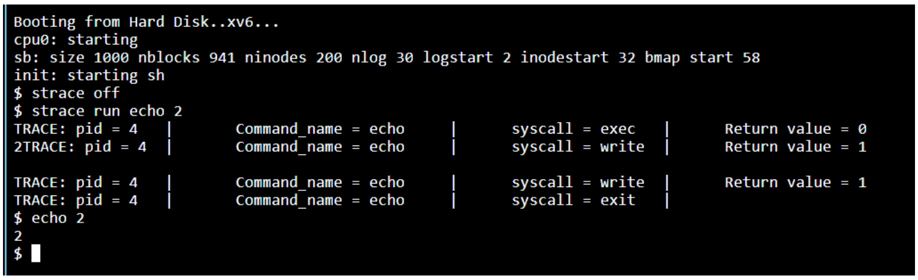
The above figure is my userspace interface and corresponding system call implementation. As you can see I use the 26th bit of the "pstrace" to sign if the kernel should strace the process. Another advantage of having a variable in the "proc" struc is that we can easily follow the subprocess by modifying the "fork" function in "proc.c".



```
kernel > proc.c > fork
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);
  np->pstrace = proc->pstrace;
  return pid;
}
```

Figure 18: Trace the Child Processes

Now we can use strace to run strace any process! There is a simple demo of "RUN" sub-command.



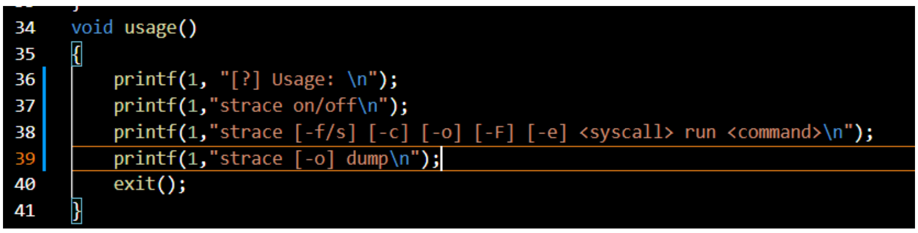
```
Booting from Hard Disk..xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ strace off
$ strace run echo 2
TRACE: pid = 4  |    Command_name = echo   |    syscall = exec  |    Return value = 0
2TRACE: pid = 4 |    Command_name = 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  |
$ echo 2
2
$
```

Figure 19: Strace RUN

Opetions

My strace supports kinds of output filter and format options which could help to eliminate the uninterested syscalls.



```
void usage()
{
    printf(1, "[?] Usage: \n");
    printf(1,"strace on/off\n");
    printf(1,"strace [-f/s] [-c] [-o] [-F] [-e] <syscall> run <command>\n");
    printf(1,"strace [-o] dump\n");
    exit();
}
```

Figure 20: Supported Opetions

I would parse the parameters in the user space and use the strace syscall to pass the operations to the kernel mode. As we talked about in the previous paragraph, the atomic unit of strace is the process so we need a variable in proc struct to tell strace-kernel what should it do. I'll introduce every bit of the "pstrace" variable in the following paragraphs. And you can check the struct of the variable in the following figure:

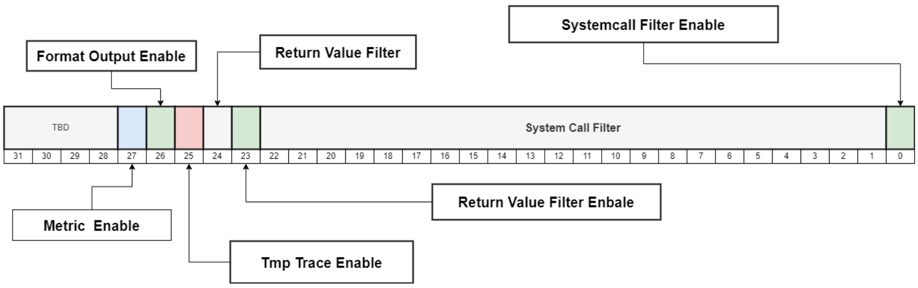


Figure 21: "pstrace" in "proc"

> Option -e [system call name]: When option flag -e is provided followed by a system call name, we will print only that system call. If no such system call is made in the command, print nothing.
>
> Option -s: When option flag -s is provided, print only successful system call.
>
> Option -f: When option flag -f is provided, print only failed system call.
>
> Option -s: Options -c in strace will generate a statistic report of system call regarding the input command such as duration, total call, and failed call. Create a similar report table using option -c.
>
> Option -f: When option flag -f is provided, print more readable output.

We have 21 syscalls on xv6 and 5 different options, so we can store this information in a 32-bit variable. As you can see in the figure, I use 21+1 bit for the "-e" option.

Option -e The 0th bit is the inuse-bit of 1-22 bits. For example, if we don't run strace with the "-e" flag the 0th bit is 0, the strace-kernel would not check the syscall filter.

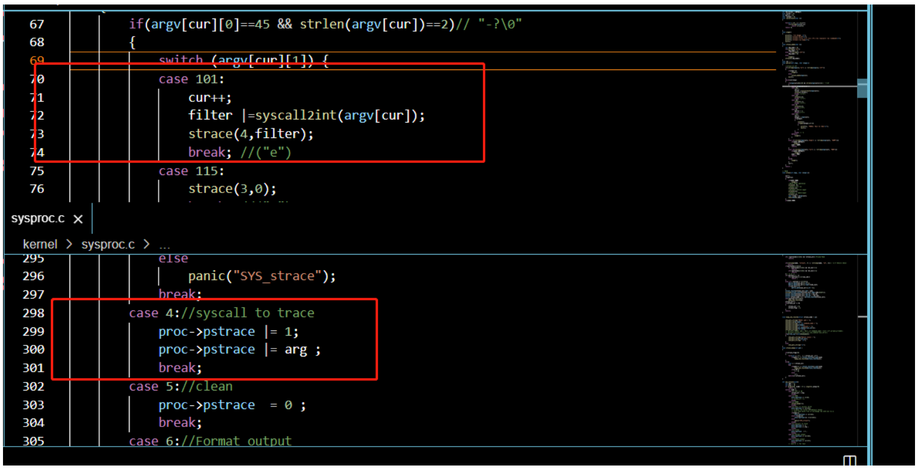


Figure 22: -e Opetion Implementation

The above figure is the user space and the kernel space handler of the "-e" operation. The userspace handler would use a strace syscall to pass the filter to the kernel space which is a whitelist of syscalls while the kernel space would enable the syscall filter and apply the filter.

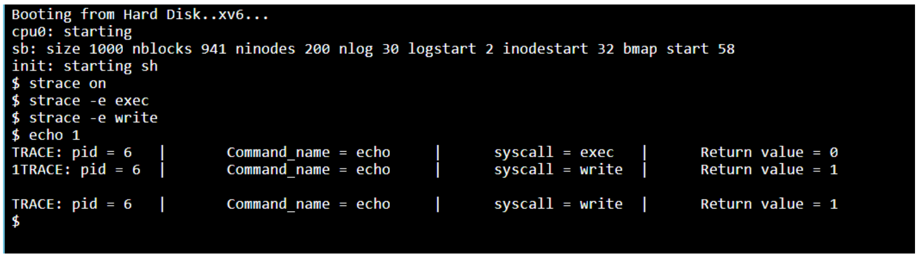
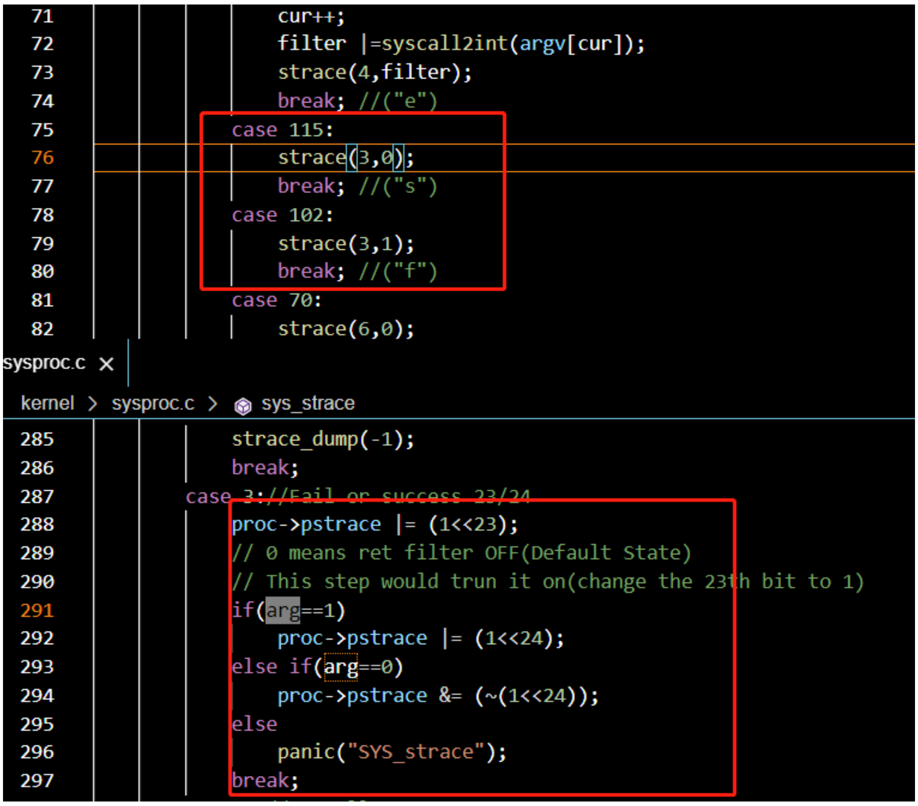


Figure 23: -e Opetion Demo

The above figure is a screenshot of the usage of the "-e" option and as you can see, my implementation could naturally handle the combination of the "-e" option.

Option -s/f The 23rd and 24th bits of pstrace are designed for the "-s/f" flag. The 23rd bit is the inuse-bit of the 24th bit. And if the 24th bit is 0, the kernel would only record successful syscalls while if the 24th bit is 1, the kernel would only record failed syscalls.



```
cur++;
filter |=syscall2int(argv[cur]);
strace(4,filter);
break; //("e")
case 115:
    strace(3,0);
    break; //("s")
case 102:
    strace(3,1);
    break; //("f")
case 70:
    strace(6,0);
```

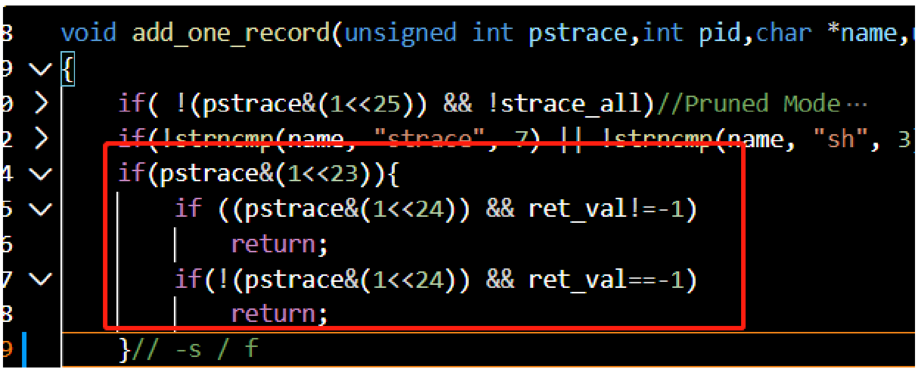
sysproc.c ×

kernel > sysproc.c > sys_strace

```
    strace_dump(-1);
    break;
case 3://Fail or success 23/24
    proc->pstrace |= (1<<23);
    // 0 means ret filter OFF(Default State)
    // This step would trun it on(change the 23th bit to 1)
    if(arg==1)
        proc->pstrace |= (1<<24);
    else if(arg==0)
        proc->pstrace &= (~(1<<24));
    else
        panic("SYS_strace");
    break;
```

Figure 24: -s/f Opetion Implementation(Handler)

The above figure is the user space and the kernel space handler of the "-s/f" operation. The user part would simply parse the arguments and call the kernel to apply the filter. Also, the kernel-mode would apply the operation to the pstrace of the proc. And as you can see in the beneath figure we will check the filter before printing out the syscall in "add_one_record".



```
void add_one_record(unsigned int pstrace,int pid,char *name,
{
    if( !(pstrace&(1<<25)) && !strace_all)//Pruned Mode···
    if(!strncmp(name, "strace", 7) || !strncmp(name, "sh", 3
    if(pstrace&(1<<23)){
        if ((pstrace&(1<<24)) && ret_val!=-1)
            return;
        if(!(pstrace&(1<<24)) && ret_val==-1)
            return;
    }// -s / f
```

Figure 25: -s/f Opetion Implementation(Filter)

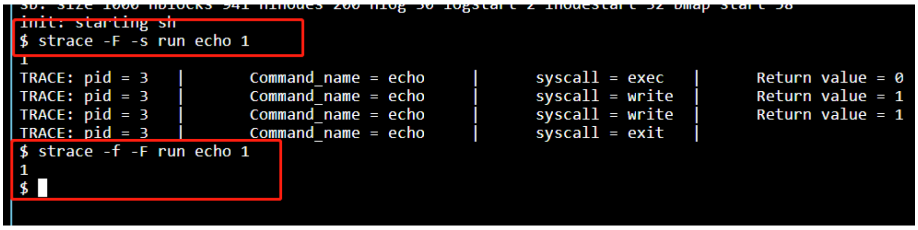


Figure 26: -s/f Opetion Demo

So far we finish the introduction of filter options (-s/f/e), and we'll move to format options in the following paragraphs.

Option -F

> "Depends on your implementation, you might notice your command result is printed along with your strace result which causes tracing to be difficult to read when characters keep mixing up between lines (ex: 'ls' has a very long list result). Choose one of the following approaches to solve this issue:"

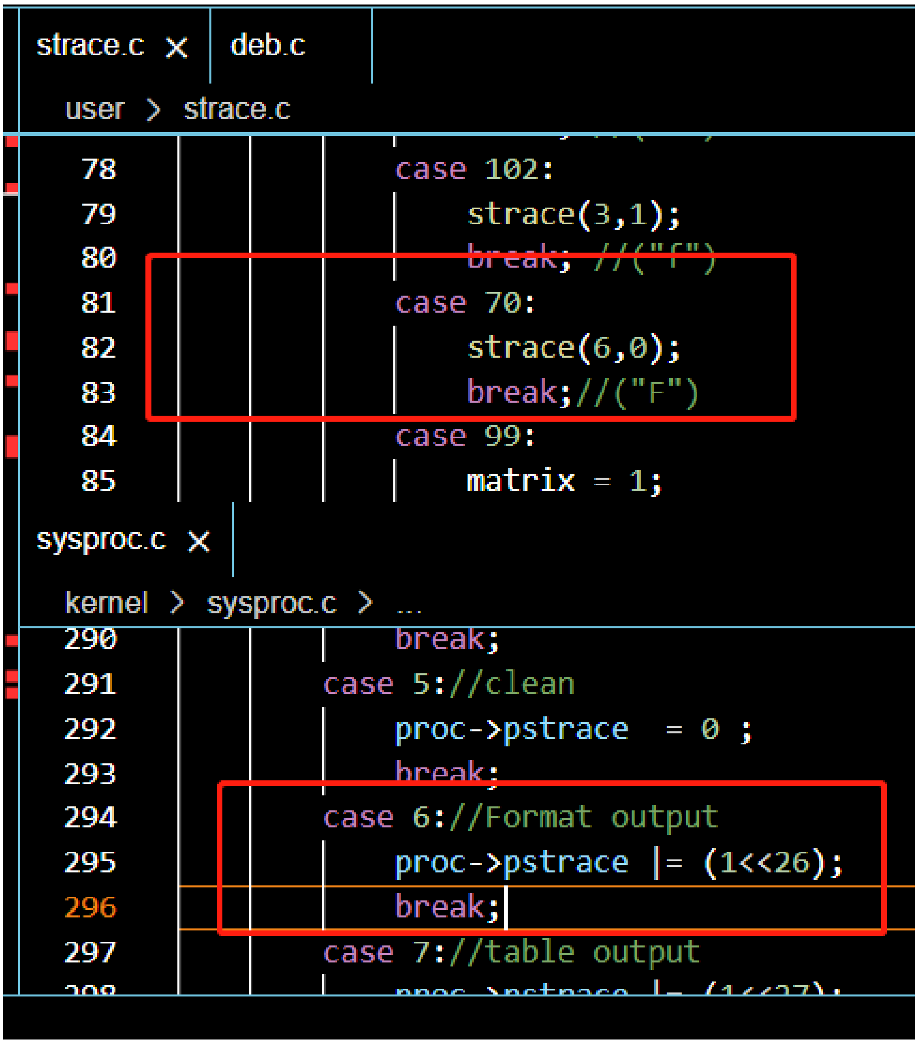


Figure 27: -F Opetion Implementation(Handler)

For this option, we need to "pause" the state output until the process finished its output. So I would store the output in someplace and dump it after the process exits by reusing partial code of the "DUMP" sub-command. This is quite simple and elegant but it has a little flaw: What if the process class tons of syscall and we could only store the latest N records.

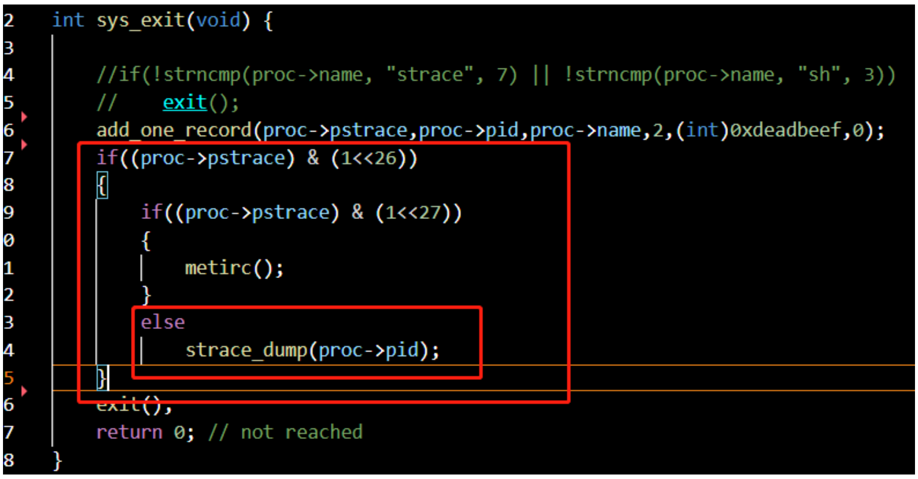


Figure 28: -F Opetion Implementation

For this issue, I can just set a variable to monitor the storage and dump the records if we are in "-F" mode. But I think this method would break XV6's simply so I would rather mention this issue in the README.md. And the following figure shows the output of the "-F" option.

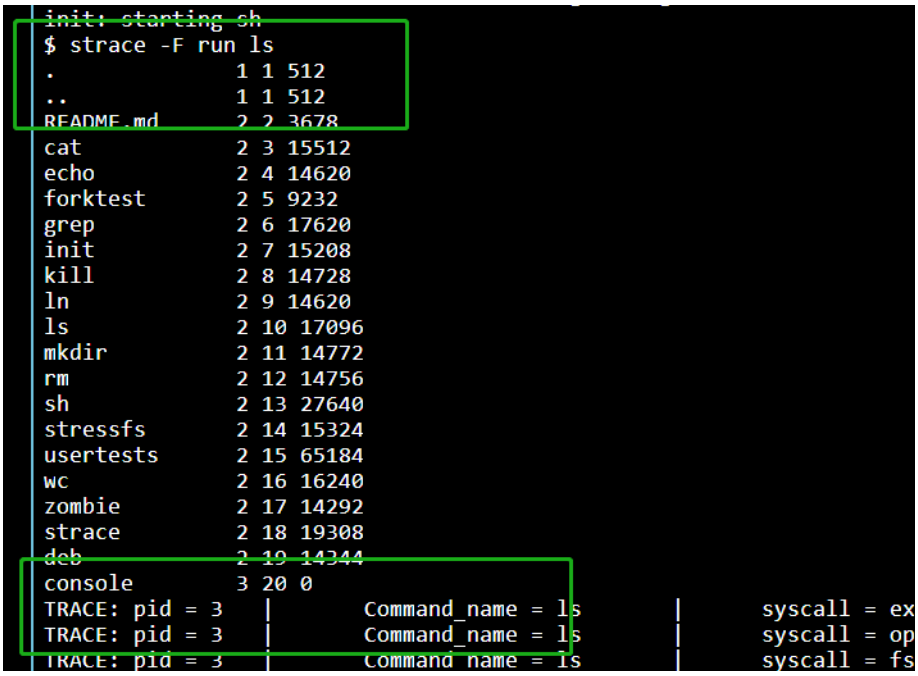


Figure 29: -F Opetion Demo

Option -c

"Options -c in strace will generate a statistic report of system call regarding the input command such as duration, total call, failed call. Create a similar report table using option -c."

```
if(mode==RUN)
{
    if(matrix)
    {
        //Disable runtile ouput
        strace(6,0);
        //Enable -c table-ouput
        strace(7,0);
    }
    char **new = &argv[cur];
    strace(1,0);
    exec(new[0],new);
    ; //can't reach
}
```

Figure 30: -C Option User Space Implementation

The "-c" would show the metrics of the command. We need to store more information about the syscalls so that allocating more space is necessary. For time recording, I would use uptime syscall to calculate the time used for every syscall. And it's easy to count the error number and usage number in function "add_one_record".

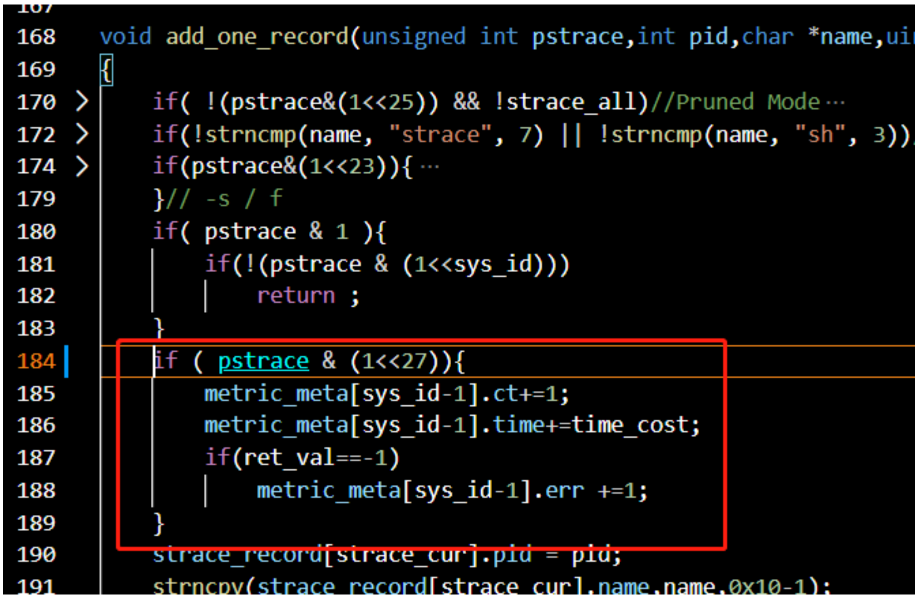


Figure 31: -C Option Implementation

And just before the exit, the "metric" function would be triggered and dump the "-c" related informations as the following figure.

```c
void metirc()
{
    std_puts_string("time\t\tcalls\t\terrors\t\tsyscall\n");
    std_puts_string("-------------------------------------------------------------\n");
    for(int i=0;i<21;i++)
    {
        if(metric_meta[i].ct!=0)
        {
            std_puts_int(metric_meta[i].time);
            std_puts_string("\t\t");
            std_puts_int(metric_meta[i].ct);
            std_puts_string("\t\t");
            std_puts_int(metric_meta[i].err);
            std_puts_string("\t\t");
            std_puts_string(sys_name[i]);
            std_puts_string("\n");
        }
    }
    metric_clean();
}
```

Figure 32: -c Option Demo

```
$ strace -c run ls
.              1 1 512
..             1 1 512
README.md      2 2 3678
cat            2 3 15512
echo           2 4 14620
forktest       2 5 9232
grep           2 6 17620
init           2 7 15208
kill           2 8 14728
ln             2 9 14620
ls             2 10 17096
mkdir          2 11 14772
rm             2 12 14756
sh             2 13 27640
stressfs       2 14 15324
usertests      2 15 65184
wc             2 16 16240
zombie         2 17 14292
strace         2 18 19308
deb            2 19 14344
console        3 20 0
time           calls          errors         syscall
-------------------------------------------------------------
0              1              0              exit
0              33             0              read
1              1              0              exec
0              22             0              fstat
1              22             0              open
2              526            0              write
0              22             0              close
$
```

Figure 33: -c Option Demo

And it's great to know "even" costs such much (33 times more than reading)!

Option -o

> "Find an implementation of choice to write strace output to file. For example: by providing option: -o 'filename' or editing content of README."

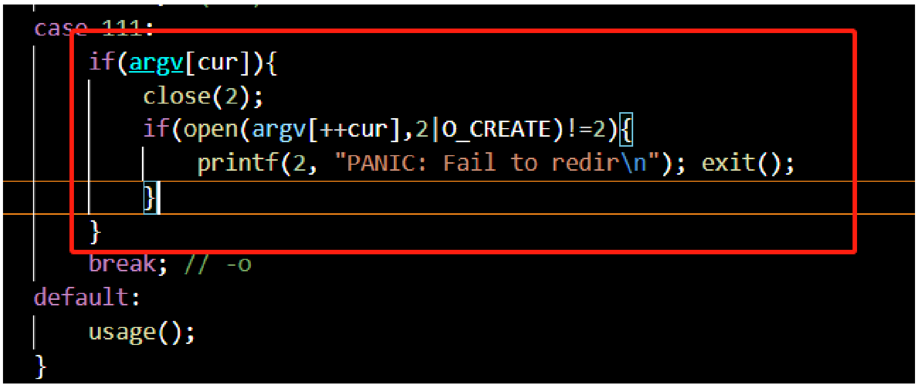


Figure 34: -o Option Userspace Handler

This one is the hardest one for my implementation because I use "exec" to run the command rather than the fork. So I need to handle all the things in the kernel which is not secure. As you see in the above figure, I can easily finish the userspace interface. Similar to what we learned in "sh.c", I just redirect the stderr to the given file.

```c
void std_puts_string(char *s)
{
    filewrite(proc->ofile[2], s, strlen(s));
}
void std_puts_int(int n)
{
    if(n<0){
        std_puts_string("-");
        n = -n;
    }
    uint idx = 0 ;
    char s[0x20]={0};
    //only 32 are used,
    //but it's better for alignment to use 32;
    do{
        s[idx++] = (char)((n%10)+0x30);
        n = n/10;
    }while(n!=0);
    //Out put
    while(idx!=0)
        filewrite(proc->ofile[2], &s[--idx], 1);
}
```

Figure 35: Format Print in Kernel

But unluckily, we don't have "printf" in the kernel to store the data into stderr. So I need to implement a limited "printf" to store the output to the stdin out. I use the read file to achieve that as the above code shows. I replace all output functions with my "format print" functions so that "-o" could naturally work with other commands and options, which is shown in the beneath figure.

```
init: starting sh
$ strace -o n132 run echo n132
n132
$ cat ./n132
TRACE: pid = 3   |        Command_name = echo      |       syscall = exec   |      Return value = 0
TRACE: pid = 3   |        Command_name = echo      |       syscall = write  |      Return value = 1
TRACE: pid = 3   |        Command_name = echo      |       syscall = write  |      Return value = 1
TRACE: pid = 3   |        Command_name = echo      |       syscall = write  |      Return value = 1
TRACE: pid = 3   |        Command_name = echo      |       syscall = write  |      Return value = 1
TRACE: pid = 3   |        Command_name = echo      |       syscall = write  |      Return value = 1
TRACE: pid = 3   |        Command_name = echo      |       syscall = exit   |
$ 
```

Figure 36: -o Opetion Demo

5 Application to strace

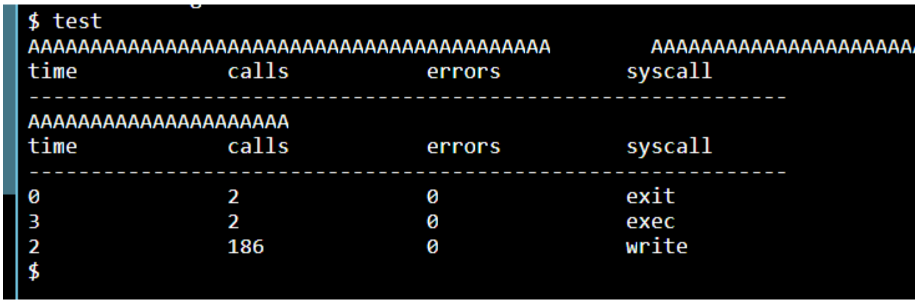
"Write a small program that produces an unexpected behavior such as race condition, delay output, crash on condition, memory leak, or your choice of implementation. Run strace on this program."

```
int main()
{
    malloc(0x1234); // Mem Leak
    int res = fork();
    if(res==0){
        char *cmd[] = {"strace","-c", "run", "echo",\
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};
        run(cmd);
        exit();
    }
    else if(res>0){
        res = fork();
        if(res==0){
        char *cmd[] = {"strace","-c", "run", "echo",\
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};
        run(cmd);
        exit();
        }
    }
    else {
```

Figure 37: Race Condition TestCase

I wrote a simple test case.c to see if there are some race condition problems without the "-c" option. And we could clearly see the demo results in race conditions. It's unexcepted. The except result should be two same tables for each subprocess. I'll fix it in this section.

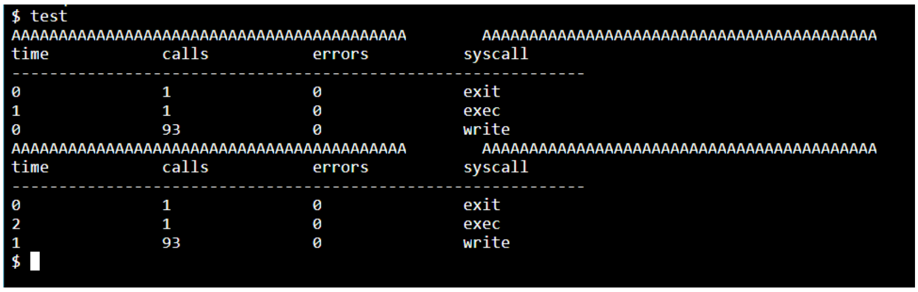
I used to store the "-c" option-related data in a global variable. And if there are two processes using it, the output may be hard to accept because the first exited process would clean the variable.



```
$ test
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA        AAAAAAAAAAAAAAAAAAAAA
time                calls               errors              syscall
----------------------------------------------------------------
AAAAAAAAAAAAAAAAAAAA
time                calls               errors              syscall
----------------------------------------------------------------
0                   2                   0                   exit
3                   2                   0                   exec
2                   186                 0                   write
$
```

Figure 38: Race Condition Found

How to Fix We could let the process possess the struct so different processes would have different variables to store their syscall records. So I change the global variable to a process-owned variable and disable the interrupts while dumping the output.



```
$ test
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
time                calls               errors              syscall
----------------------------------------------------------------
0                   1                   0                   exit
1                   1                   0                   exec
0                   93                  0                   write
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
time                calls               errors              syscall
----------------------------------------------------------------
0                   1                   0                   exit
2                   1                   0                   exec
1                   93                  0                   write
$
```

Figure 39: Race Condition Fixed

As you can see in the above figure, the output keeps the correct order and the number of syscalls is correct!

```
anubis@anubis-ide ‹ master@45c3550 ● › : ~/final-project-371525eb-xm2146
[0] % ./test2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0         5           close
  0.00    0.000000           0         4           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    0.000000           0        33         1 total
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0         5           close
  0.00    0.000000           0         4           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    0.000000           0        33         1 total
anubis@anubis-ide ‹ master@45c3550 ● › : ~/final-project-371525eb-xm2146
[1] %
```

Figure 40: Same Program on Linux

Besides, I implemented a functionally same program in test2.c. The Linux starts surely provides one thing than our process, the time ratio. It could show the ratio of time cost by different syscalls. And both strace can't help me find the data leak because it's a dynamic issue and the strace would not trace all the allocated memory in the process.

```
int main()
{
    malloc(0x1234); // Mem Leak
    int res = fork();
    if(res==0){
        char *cmd = "strace -c echo\
```

Figure 41: Memory Leak

More Testing Also, I write lots of test cases to test my strace in /Test_Log.MD.

6 Todo

During the implementation and the testing, I noticed there are several flaws in my implementation. And I decide to leave these flaws in my code because I didn't have time to improve these issues and this is an educational assignment rather than a product.

Strace RUN For strace run, it's better to use a fork because for some options we need to store the data until the process exits. So we need a parent process to wait for the child to process as a daemon and finish the output task after the child process exits, as it's more secure to write a file in userspace.

-F Option This is a special issue with the -F option which aims at printing more readable output. To protect the simplicity of the xv6 kernel, I decide not to implement a mechanism to deal with infinity syscall records. So if there are more than N records when running the command, I would only print out the last N records rather than all the records.

Race Condition We should be very paranoid about the kernel global variables. If there is more than one process, we may have race conditions! I didn't think much about this and only did some simple testing on that. More paranoid code reviews should be taken for any code in the kernel.

7 Structure of Strace

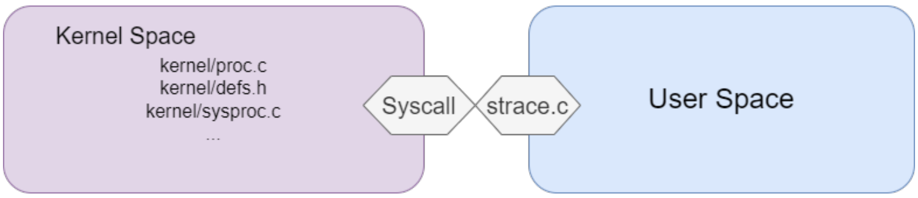


Figure 42: Structure

There are two main parts of this implementation: Kernel Space Part and User Space Part.

In the user space, I use the binary strace as the interface to transfer data to the kernel. And the main handler is the file "kernel/proc.c". I implement a strace syscall and deal with kinds of parameters.

In the kernel space, I use the metadata got from the strace syscall to set the output format/filter options. Also, I use the kernel variables to implement "strace on/off". And you can check all the supported features in the above figure.

Type	Description	Options
Sub-Command	strace on	-
Sub-Command	strace off	-
Sub-Command	RUN	[-e] [-s] [-f] [-o] [-F] [-c]
Sub-Command	DUMP	[-o]
Option	-F	More readable output
Option	-f	Only record failed syscalls
Option	-s	Only record successful syscalls
Option	-e	Only record specific syscalls
Option	-c	Print out a table of used syscalls
Option	-o	Redirecte the strace output to a file

Figure 43: Supported Features

8 Summary

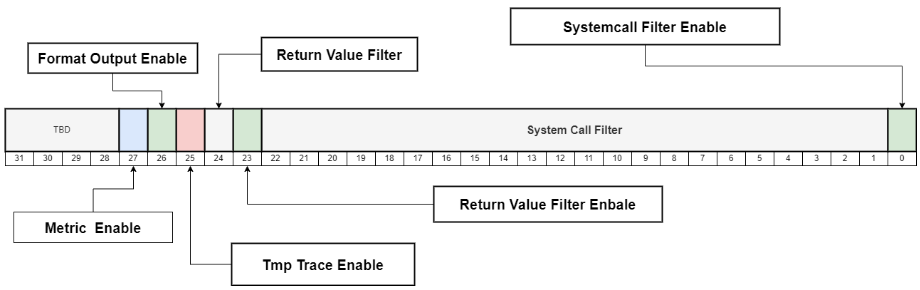


Figure 44: My Strace

The above figure is my pstrace's structure I spent lots of time on it to make it elegant. I am really happy I didn't waste the opportunity to push myself. As a result, I learned a lot during this assignment and have a more complete view of the Unix kernel. And I could feel that the knowledge I learned on the course and the xv6 could be used to understand really Linux kernel.