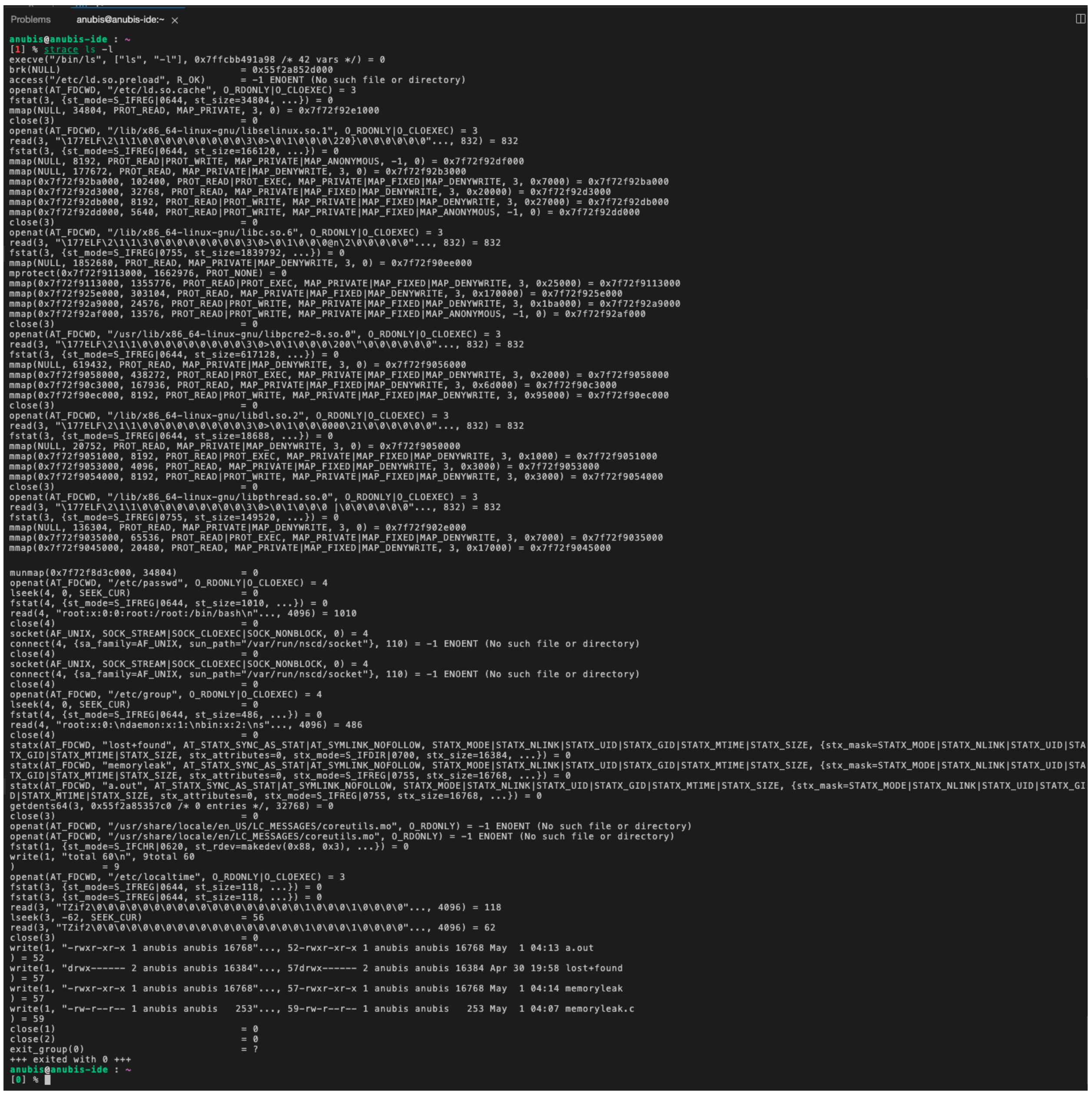
OPERATING SYSTEMS FINAL PROJECT REPORT

Prachika Agarwal : pa2191 **(To be used for grading)**
Shreya Singh : ss14937

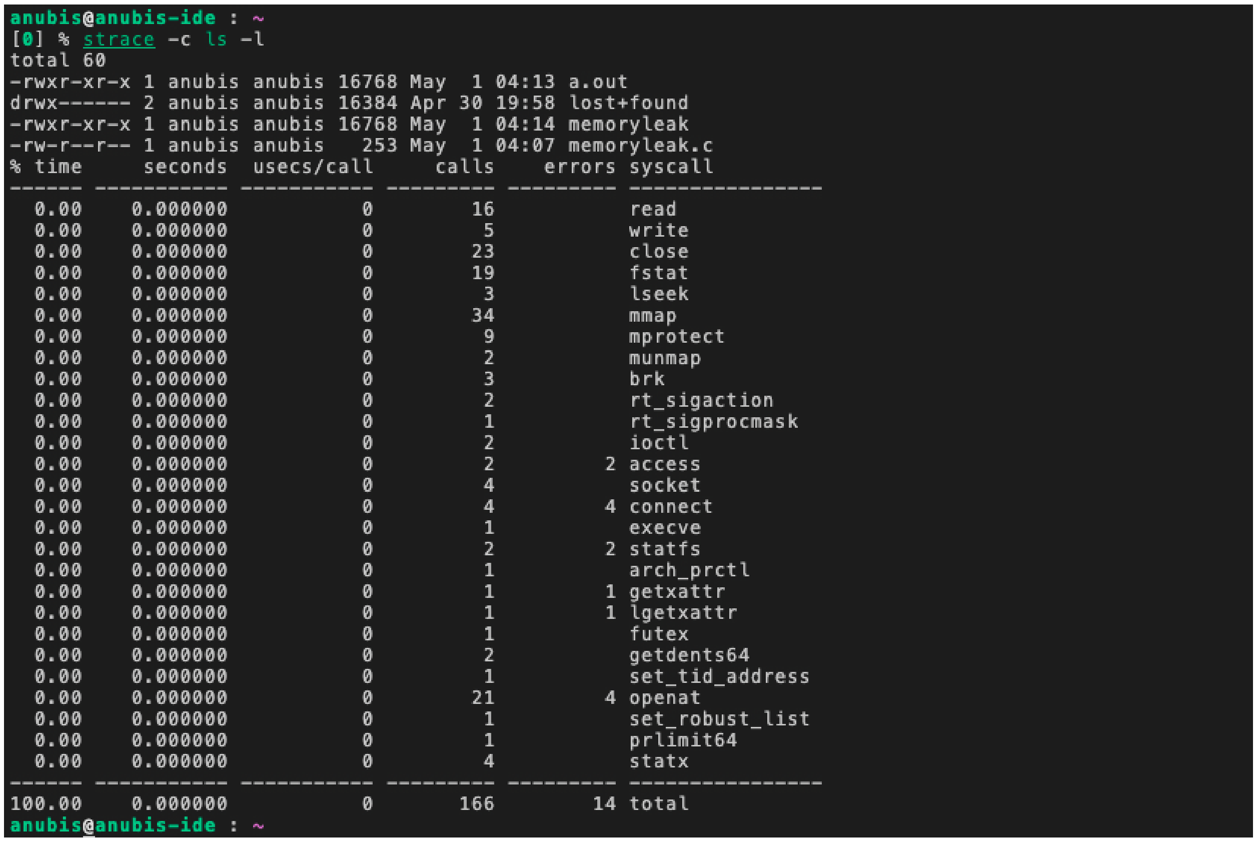
1) GETTING FAMILIAR WITH LINUX STRACE

a) Strace ls -l



```
anubis@anubis-ide : ~
[1] % strace ls -l
execve("/bin/ls", ["ls", "-l"], 0x7ffcbb491a98 /* 42 vars */) = 0
brk(NULL)                               = 0x55f2a852d000
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) = 0x7f72f92e1000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220}\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=166120, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72f92df000
mmap(NULL, 177672, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72f92b3000
mmap(0x7f72f92ba000, 102400, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x7000) = 0x7f72f92ba000
mmap(0x7f72f92d3000, 32768, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x20000) = 0x7f72f92d3000
mmap(0x7f72f92db000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x27000) = 0x7f72f92db000
mmap(0x7f72f92dd000, 5640, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f72f92dd000
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, 1852680, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72f90ee000
mprotect(0x7f72f9113000, 1662976, PROT_NONE) = 0
mmap(0x7f72f9113000, 1355776, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x25000) = 0x7f72f9113000
mmap(0x7f72f925e000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x170000) = 0x7f72f925e000
mmap(0x7f72f92a9000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7f72f92a9000
mmap(0x7f72f92af000, 13576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f72f92af000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/libpcre2-8.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200\"\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=617128, ...}) = 0
mmap(NULL, 619432, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72f9056000
mmap(0x7f72f9058000, 438272, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f72f9058000
mmap(0x7f72f90c3000, 167936, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x6d000) = 0x7f72f90c3000
mmap(0x7f72f90ec000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x95000) = 0x7f72f90ec000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000\21\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=18688, ...}) = 0
mmap(NULL, 20752, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72f9050000
mmap(0x7f72f9051000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f72f9051000
mmap(0x7f72f9053000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x3000) = 0x7f72f9053000
mmap(0x7f72f9054000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x3000) = 0x7f72f9054000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0 |\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=149520, ...}) = 0
mmap(NULL, 136304, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72f902e000
mmap(0x7f72f9035000, 65536, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x7000) = 0x7f72f9035000
mmap(0x7f72f9045000, 20480, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17000) = 0x7f72f9045000

munmap(0x7f72f8d3c000, 34804)           = 0
openat(AT_FDCWD, "/etc/passwd", O_RDONLY|O_CLOEXEC) = 4
lseek(4, 0, SEEK_CUR)                   = 0
fstat(4, {st_mode=S_IFREG|0644, st_size=1010, ...}) = 0
read(4, "root:x:0:0:root:/root:/bin/bash\n"..., 4096) = 1010
close(4)                                = 0
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 4
connect(4, {sa_family=AF_UNIX, sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT (No such file or directory)
close(4)                                = 0
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 4
connect(4, {sa_family=AF_UNIX, sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT (No such file or directory)
close(4)                                = 0
openat(AT_FDCWD, "/etc/group", O_RDONLY|O_CLOEXEC) = 4
lseek(4, 0, SEEK_CUR)                   = 0
fstat(4, {st_mode=S_IFREG|0644, st_size=486, ...}) = 0
read(4, "root:x:0:\ndaemon:x:1:\nbin:x:2:\ns"..., 4096) = 486
close(4)                                = 0
statx(AT_FDCWD, "lost+found", AT_STATX_SYNC_AS_STAT|AT_SYMLINK_NOFOLLOW, STATX_MODE|STATX_NLINK|STATX_UID|STATX_GID|STATX_MTIME|STATX_SIZE, {stx_mask=STATX_MODE|STATX_NLINK|STATX_UID|STA
TX_GID|STATX_MTIME|STATX_SIZE, stx_attributes=0, stx_mode=S_IFDIR|0700, stx_size=16384, ...}) = 0
statx(AT_FDCWD, "memoryleak", AT_STATX_SYNC_AS_STAT|AT_SYMLINK_NOFOLLOW, STATX_MODE|STATX_NLINK|STATX_UID|STATX_GID|STATX_MTIME|STATX_SIZE, {stx_mask=STATX_MODE|STATX_NLINK|STATX_UID|STA
TX_GID|STATX_MTIME|STATX_SIZE, stx_attributes=0, stx_mode=S_IFREG|0755, stx_size=16768, ...}) = 0
statx(AT_FDCWD, "a.out", AT_STATX_SYNC_AS_STAT|AT_SYMLINK_NOFOLLOW, STATX_MODE|STATX_NLINK|STATX_UID|STATX_GID|STATX_MTIME|STATX_SIZE, {stx_mask=STATX_MODE|STATX_NLINK|STATX_UID|STATX_GI
D|STATX_MTIME|STATX_SIZE, stx_attributes=0, stx_mode=S_IFREG|0755, stx_size=16768, ...}) = 0
getdents64(3, 0x55f2a85357c0 /* 0 entries */, 32768) = 0
close(3)                                = 0
openat(AT_FDCWD, "/usr/share/locale/en_US/LC_MESSAGES/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/share/locale/en/LC_MESSAGES/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x3), ...}) = 0
write(1, "total 60\n", 9total 60
)             = 9
openat(AT_FDCWD, "/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=118, ...}) = 0
fstat(3, {st_mode=S_IFREG|0644, st_size=118, ...}) = 0
read(3, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\1\0\0\0\1\0\0\0\0"..., 4096) = 118
lseek(3, -62, SEEK_CUR)                 = 56
read(3, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\1\0\0\0\1\0\0\0\0"..., 4096) = 62
close(3)                                = 0
write(1, "-rwxr-xr-x 1 anubis anubis 16768"..., 52-rwxr-xr-x 1 anubis anubis 16768 May  1 04:13 a.out
) = 52
write(1, "drwx------ 2 anubis anubis 16384"..., 57drwx------ 2 anubis anubis 16384 Apr 30 19:58 lost+found
) = 57
write(1, "-rwxr-xr-x 1 anubis anubis 16768"..., 57-rwxr-xr-x 1 anubis anubis 16768 May  1 04:14 memoryleak
) = 57
write(1, "-rw-r--r-- 1 anubis anubis   253"..., 59-rw-r--r-- 1 anubis anubis   253 May  1 04:07 memoryleak.c
) = 59
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
anubis@anubis-ide : ~
[0] %
```

b) **Strace -c ls -l**



```
anubis@anubis-ide : ~
[0] % strace -c ls -l
total 60
-rwxr-xr-x 1 anubis anubis 16768 May  1 04:13 a.out
drwx------ 2 anubis anubis 16384 Apr 30 19:58 lost+found
-rwxr-xr-x 1 anubis anubis 16768 May  1 04:14 memoryleak
-rw-r--r-- 1 anubis anubis   253 May  1 04:07 memoryleak.c
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0        16           read
  0.00    0.000000           0         5           write
  0.00    0.000000           0        23           close
  0.00    0.000000           0        19           fstat
  0.00    0.000000           0         3           lseek
  0.00    0.000000           0        34           mmap
  0.00    0.000000           0         9           mprotect
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           rt_sigaction
  0.00    0.000000           0         1           rt_sigprocmask
  0.00    0.000000           0         2           ioctl
  0.00    0.000000           0         2         2 access
  0.00    0.000000           0         4           socket
  0.00    0.000000           0         4         4 connect
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         2 statfs
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         1         1 getxattr
  0.00    0.000000           0         1         1 lgetxattr
  0.00    0.000000           0         1           futex
  0.00    0.000000           0         2           getdents64
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0        21         4 openat
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
  0.00    0.000000           0         4           statx
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0       166        14 total
anubis@anubis-ide : ~
```

An example of system calls being made and their functionality are:

1) **Read** - It reads upto count bytes from a file descriptor fd into the buffer bf
2) **Write -** It writes upto count bytes from the buffer starting at buf to the file referred by the file descriptor
3) **Close -** It closes the file descriptor so that it no longer refers to any file and it can be reused
4) **Fstat -** It is used to return information from a file. It is similar to stat() but here the the file to be stat-ed is specified by the file descriptor fd
5) **Lseek -** It repositions file offset of the open file description associated with file descriptor fd to argument offset according to directive
6) **Socket -** It creates an endpoint for communication and returns a file descriptor that refers to that end point

2. BUILDING STRACE IN XV6

a) Strace on and Strace off

For implementation of strace on/off the following changes were made:

i) **trace.h**

In trace.h define T_UNTRACE, T_TRACE and T_FORK as follows:

```
#define T_UNTRACE 0
#define T_TRACE 1
#define T_ONFORK 2
```

ii) **proc.c**

- Include header file trace.h
- In static struct proc *allocproc(void) we set p->traced = T_UNTRACE
- Turn tracing on in int fork(void) as follows:

```
np->traced = (proc->traced & T_ONFORK) ? proc->traced: T_UNTRACE ;
```

iii) **proc.h**

- Add a new integer variable traced in struct proc

```
int traced; // added
```

iv) **sh.c**

- Include header file trace.h
- Set int tracing = 0 at the beginning of the code
- Create two groble character arrays which will be used to compare with user input and pass values to our kernel.

```
char trace_cmd[] = "strace on\n";  //added
char untrace_cmd[] = "strace off\n";  //added
```

- Create a function streq which compares user input buf with the character array as follows:

```
int streq(char *a, char *b) // added
{
   while(1){
       if(*a != *b)
       { return 0;}
       if(*a == '\n')
```

```
        {
            return 1;
        }
        a++;
        b++;
    }
}
```

- In function void runcmd(struct cmd *cmd) turn tracing on in case EXEC as follows:

```
case EXEC:
    ecmd = (struct execcmd *)cmd;
    if (ecmd->argv[0] == 0)
      exit();
    if (tracing) trace(T_TRACE | T_ONFORK); //added
    exec(ecmd->argv[0], ecmd->argv);
    printf(2, "exec %s failed\n", ecmd->argv[0]);
    break;
```

- In while (getcmd(buf, sizeof(buf))>=0) add a nested if statement after if(fork == 0) which compares user input buf with our predefined character arrays trace_cmd and untrace_cmd and turns tracing on/off is both strings are same.

```
if (fork1() == 0){
        if (streq(buf, trace_cmd)) // added
        {
            tracing = 1;
            continue;
        }
        if (streq(buf, untrace_cmd)) // added
        {
            tracing = 0;
            continue;
        }
}
```

v) **syscall.c**

- Include header file trace.h
- Add a system call sys_trace() as follows:

```
int sys_trace(){ // added
    int n;
    argint(0,&n);
    proc->traced = (n & T_TRACE) ? n:0;
    return 0;
}
```

- In static int(*syscalls[])(void) = {} add SYS_trace sys_trace
- Declare a static char *syscall_names = {} with all the names of system calls in order to print the system call names as follows:

```
static char *syscall_names[] = { // added
    [SYS_fork] "fork",    [SYS_exit] "exit",      [SYS_wait] "wait",
    [SYS_pipe] "pipe",    [SYS_read] "read",      [SYS_kill] "kill",
    [SYS_exec] "exec",    [SYS_fstat] "fstat",    [SYS_chdir]
"chdir",
    [SYS_dup] "dup",      [SYS_getpid] "getpid", [SYS_sbrk] "sbrk",
    [SYS_sleep] "sleep", [SYS_uptime] "uptime", [SYS_open] "open",
    [SYS_write] "write", [SYS_mknod] "mknod",    [SYS_unlink]
"unlink",
    [SYS_link] "link",    [SYS_mkdir] "mkdir",    [SYS_close]
"close",
    [SYS_trace] "trace", [SYS_passtokernel] "passtokernel",
[SYS_resetoption] "resetoption", [SYS_stracedump] "tracedump",
};
```

- In void syscall (void) declare three variables int i, int is_traced, a char array procname[16] and a for loop where we store the process names in this character array procname[16] as follows:

```
int num,i;
 int is_traced = (proc->traced & T_TRACE);
 char procname[16];

 for(i =0; proc->name[i]!=0; i++)
 {
     procname[i] = proc->name[i];
 }
 procname[i] = proc->name[i];
```

- Now in void syscall (void) if num == SYS_exit and tracing is on then we will print the following:

```
if(num == SYS_exit && is_traced){
cprintf("\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\e[0m\n", proc->pid, procname,
syscall_names[num]);
}
```

Next, if (num < NELEM(syscalls) && syscalls[num]) we print the following:

```
if (num < NELEM(syscalls) && syscalls[num]) {
   proc->tf->eax = syscalls[num]();

       if (is_traced) {
       cprintf((num == SYS_exec && proc->tf->eax == 0) ?
       "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\e[0m\n" :
        "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\treturn val = %d |\e[0m\n",
        proc->pid, procname, syscall_names[num], proc->tf->eax);
```

vi) **syscall.h**

- In syscall.h define the following system call:

```
#define SYS_trace  22 //added
```

vii) **user.h**

- In user.h define the system call trace:

```
int trace(int); //added
```

viii) **usys.S**

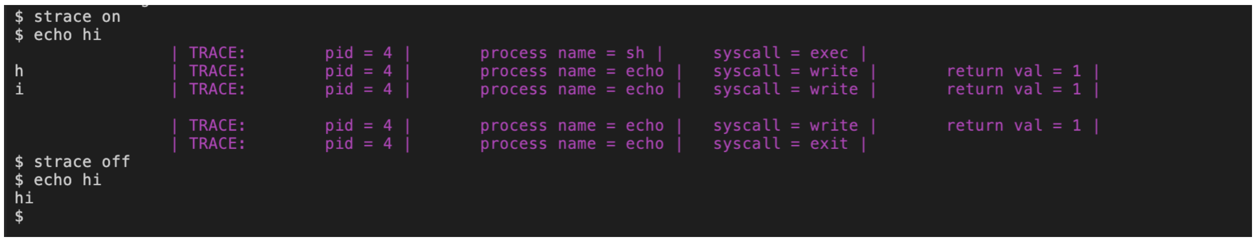
- In usys.S define the system call trace as follows:

```
SYSCALL(trace)
```

<u>How to run test cases for Strace on and Strace off :</u>

1) $Make clean; $make qemu;
2) In the shell type $strace on;
3) Type the system calls you want to trace for example $echo hi;
4) To turn strace off, type $strace off;

<u>OUTPUT:</u>



```
$ strace on
$ echo hi
                | TRACE:        pid = 4 |       process name = sh |     syscall = exec |
h               | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
i               | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |

                | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = echo |   syscall = exit |
$ strace off
$ echo hi
hi
$
```

b) Strace run <command>

To implement strace run <command> we made the following changes in **sh.c** :

1) Add a character variable char trace_run_cmd[] as follows:

```
char tr3ace_run_cmd[] = "strace run";
```

2) Declare a function streqrun which compares trace_run_cmd with a specified no if input characters in the input string buf as follows:

```
int streqrun(char *a, char *b, int n) // added
{
   int i = 0;
   while(i<n){
       if(*a != *b)
       {
           return 0;
       }
       if(*a == '\n')
       {
           return 1;
       }
       a++;
       b++;
       i++;
   }
   return 1;
}
```

3) Declare a structure stracerun as follows:

```
struct stracerun{        //added
   struct cmd *cmd;
};
```

4) In while (getcmd(buf, sizeof(buf))>=0) add a nested if statement after the part where we turned tracing on and off as follows:

```
if (streqrun(buf, trace_run_cmd, 10)) // added
       {   static char bufcommand[100];
           int i = 11, j=0;
           while(i<sizeof(buf)){
               bufcommand[j] = buf[i];
               i++;
               j++;
           }
```

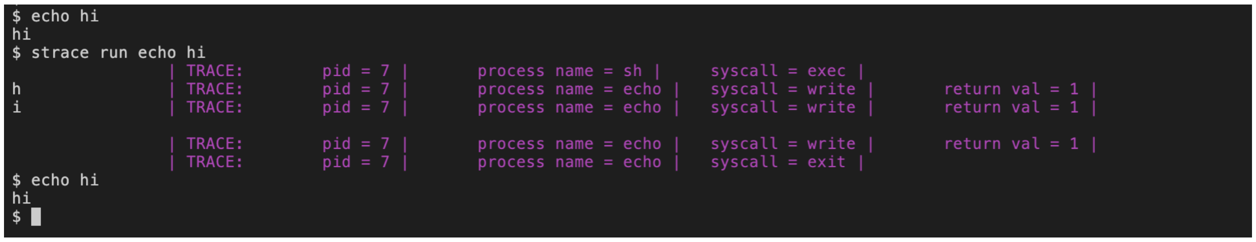
```
            tracing = 1;
            runcmd(parsecmd(bufcommand));
            printf(1,"exec complete");
            tracing = 0;
            continue;
        }
```

Here, we checked if our input string has strace run, if it has we stored the command entered by user in charac array bufcommand[100], after that we turned tracing on, and sent the command that was stored in bufcommand to runcmd(parsecmd([bufcommand])) which executed the command, after execution was complete we turned tracing off.

How to run strace run <command> in shell :

1) $Make clean; $make qemu;
2) In the shell type $strace run <command> for example strace run echo hi ;
3) Once the command is executed tracing is turned off automatically

OUTPUT:



```
$ echo hi
hi
$ strace run echo hi
                | TRACE:        pid = 7 |       process name = sh |     syscall = exec |
h               | TRACE:        pid = 7 |       process name = echo |   syscall = write |       return val = 1 |
i               | TRACE:        pid = 7 |       process name = echo |   syscall = write |       return val = 1 |

                | TRACE:        pid = 7 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 7 |       process name = echo |   syscall = exit |
$ echo hi
hi
$
```

c) Strace dump - stores last 10 commands in its buffer

i) **dump.h**

- Create a header file dump.h as follows:

```
#define MAX 10 //added for dump

struct dump {  //added for dump
   int d_pid;
   char d_procname[20];
   char d_syscallname[20];
   int d_returnval;
};
```

ii) **syscall.h**

- Include header file trace.h
- Declare a function void storedump to store the last 10 values when the print statement is called in void syscall(void) as follows:

```
static struct dump du[MAX];

static int counter_dump=0;

void storedump(int pid, char *procname, char *syscallname, int
returnval)
{
       du[counter_dump % MAX].d_pid = proc->pid;
       copy_dump(du[counter_dump % MAX].d_procname, procname);
      copy_dump(du[counter_dump % MAX].d_syscallname,
syscallname);
       du[counter_dump % MAX].d_returnval = proc->tf->eax;
       counter_dump ++;
}
```

- Pass value to this function so that it is stored in a buffer array as follows:

```
storedump(proc->pid, procname, syscall_names[num], proc->tf->eax);
```

- Create a system call sys_stracedump(void) as follows:

```
int sys_stracedump(void){
   int i;
   for(i=0; i<MAX; i++){
       cprintf("\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\treturn val = %d |\e[0m\n",
       du[i].d_pid, du[i].d_procname, du[i].d_syscallname,
du[i].d_returnval);
   }
   return 1;
}
```

- Declare stracedump as an extern int

```
extern int sys_stracedump(void);
```

- In static int(*syscalls[])(void) = {} add SYS_stracedump sys_stracedump

iii) **user.h**

- In user.h add the system call stracedump as follows:

```
int stracedump(void); //added
```

iv) **defs.h**

- Add system call for stracedump

```
int             stracedump(void);
```

v) **sh.c**

- Include header file trace.h
- Define the char buffer as follows:

```
char trace_dump[] = "strace dump\n"; //added
```

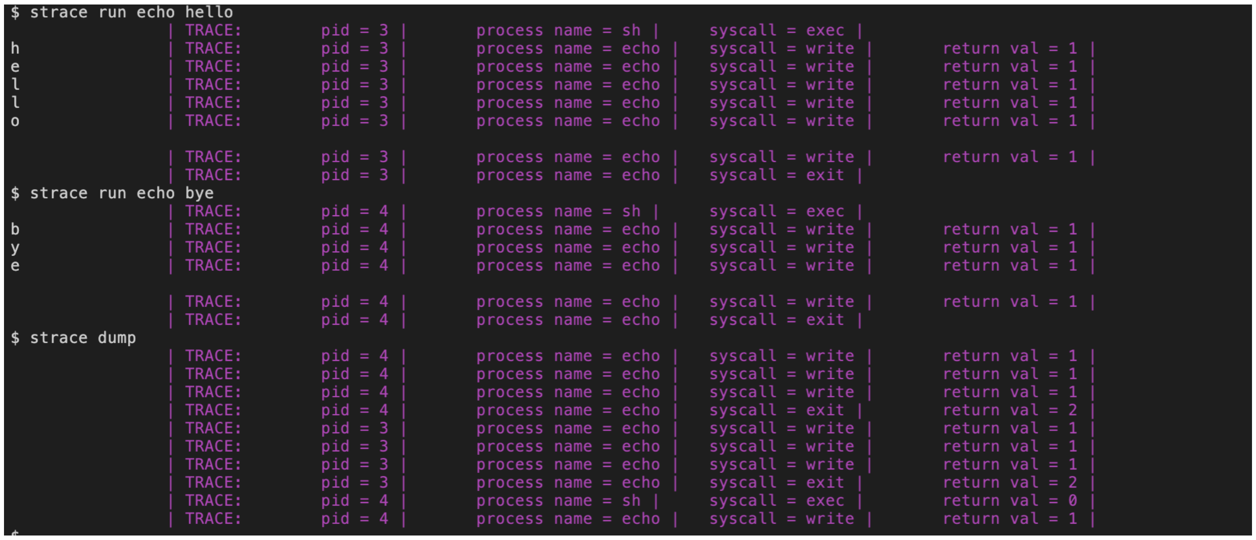
- In while (getcmd(buf, sizeof(buf))>=0) add a nested if statement after the part where we turned tracing on and off as follows:

```
if (streq(buf, trace_dump)) //added
       {
           stracedump();
           continue;
       }
```

<u>How to run strace dump in shell:</u>

1) $Make clean; $make qemu;
2) In the shell run a few strace commands such as
   $strace run echo hello
   $strace run echo bye
3) In the shell type $strace dump
4) Notice the pid and other information gets updated after the buffer reaches it's maximum size it again restarts from 0

OUTPUT:



```
$ strace run echo hello
                | TRACE:        pid = 3 |       process name = sh |     syscall = exec |
h               | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
e               | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
l               | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
l               | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
o               | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |

                | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 3 |       process name = echo |   syscall = exit |
$ strace run echo bye
                | TRACE:        pid = 4 |       process name = sh |     syscall = exec |
b               | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
y               | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
e               | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |

                | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = echo |   syscall = exit |
$ strace dump
                | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = echo |   syscall = exit |        return val = 2 |
                | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 3 |       process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 3 |       process name = echo |   syscall = exit |        return val = 2 |
                | TRACE:        pid = 4 |       process name = sh |     syscall = exec |        return val = 0 |
                | TRACE:        pid = 4 |       process name = echo |   syscall = write |       return val = 1 |
```

d) Trace child process

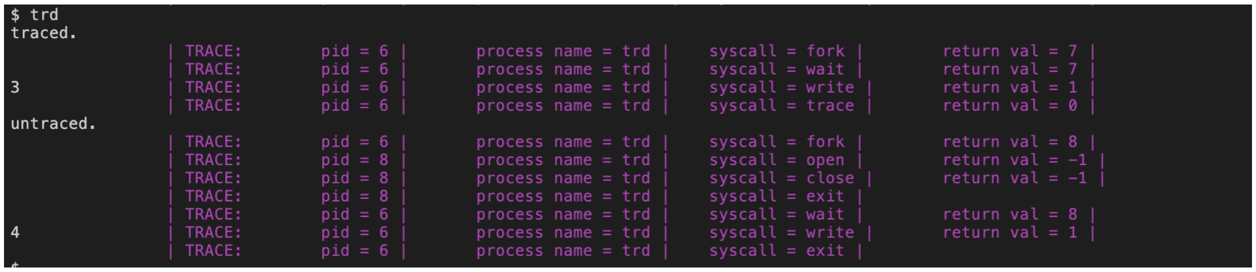
- Create a new file trd.c in user as follows, fork a few times in it and turn tracing on, since we have created the buffer memory for strace dump in kernel, the memory gets updated on its own every time the program runs.
- Write the following code in **trd.c**

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user.h"
#include "kernel/trace.h"

int resource = 2;

void forkrun() {
   int fr = fork();
   if (fr == -1) {
       printf(1, "Fork error!\n");
       return;
   } else if (fr == 0) {
       close(open("README", 0));
       exit();
   } else {
       wait();
       resource++;
       printf(1,"%d",resource);
   }
}
```

```c
int main() {
   printf(1, "traced.\n");
   trace(T_TRACE);
   forkrun();

   trace(T_UNTRACE);
   printf(1, "untraced.\n");
   trace(T_TRACE | T_ONFORK);
   forkrun();
   exit();
}
```

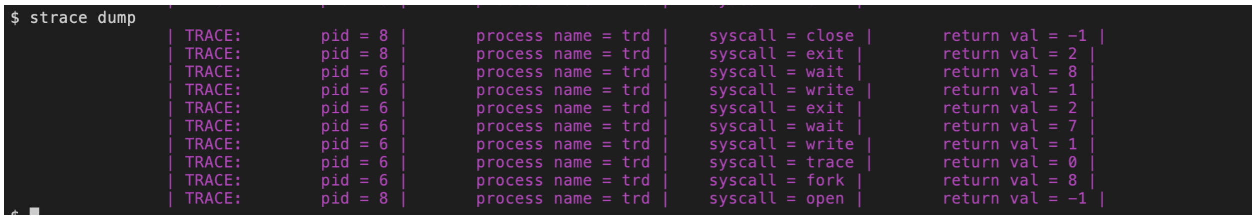
How to run trd.c and it's output

1) Make clean; make qemu;
2) In shell type the following command: $trd



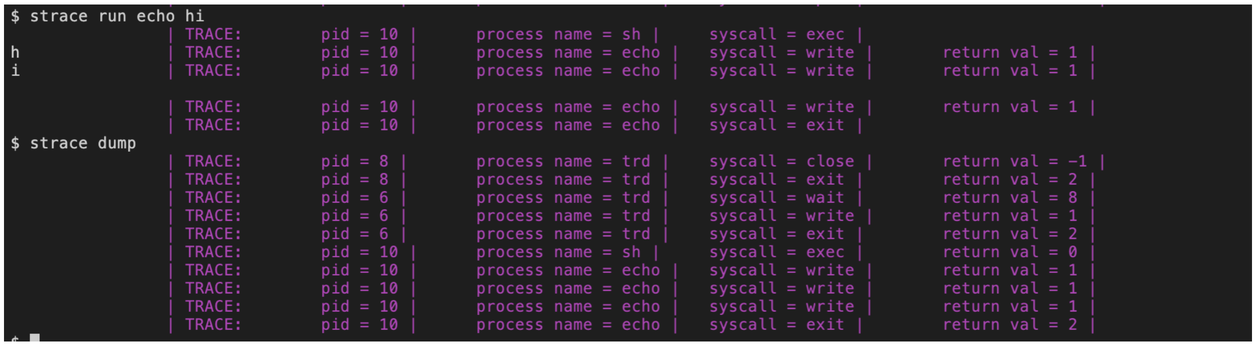
```
$ trd
traced.
                | TRACE:       pid = 6 |      process name = trd |    syscall = fork |       return val = 7 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = wait |       return val = 7 |
3               | TRACE:       pid = 6 |      process name = trd |    syscall = write |      return val = 1 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = trace |      return val = 0 |
untraced.
                | TRACE:       pid = 6 |      process name = trd |    syscall = fork |       return val = 8 |
                | TRACE:       pid = 8 |      process name = trd |    syscall = open |       return val = -1 |
                | TRACE:       pid = 8 |      process name = trd |    syscall = close |      return val = -1 |
                | TRACE:       pid = 8 |      process name = trd |    syscall = exit |
                | TRACE:       pid = 6 |      process name = trd |    syscall = wait |       return val = 8 |
4               | TRACE:       pid = 6 |      process name = trd |    syscall = write |      return val = 1 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = exit |
```

3) Now type
   $strace dump
   Notice that the memory gets updated with the system calls of trd process



```
$ strace dump
                | TRACE:       pid = 8 |      process name = trd |    syscall = close |      return val = -1 |
                | TRACE:       pid = 8 |      process name = trd |    syscall = exit |       return val = 2 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = wait |       return val = 8 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = write |      return val = 1 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = exit |       return val = 2 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = wait |       return val = 7 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = write |      return val = 1 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = trace |      return val = 0 |
                | TRACE:       pid = 6 |      process name = trd |    syscall = fork |       return val = 8 |
                | TRACE:       pid = 8 |      process name = trd |    syscall = open |       return val = -1 |
```

4) In order to check if the buffer gets updated again run the following command
   $strace run echo hi
   $strace dump
   Now you will see the memory got updated, now including system calls from echo hi



```
$ strace run echo hi
                | TRACE:        pid = 10 |      process name = sh |     syscall = exec |
h               | TRACE:        pid = 10 |      process name = echo |   syscall = write |       return val = 1 |
i               | TRACE:        pid = 10 |      process name = echo |   syscall = write |       return val = 1 |

                | TRACE:        pid = 10 |      process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 10 |      process name = echo |   syscall = exit |
$ strace dump
                | TRACE:        pid = 8 |       process name = trd |    syscall = close |       return val = -1 |
                | TRACE:        pid = 8 |       process name = trd |    syscall = exit |        return val = 2 |
                | TRACE:        pid = 6 |       process name = trd |    syscall = wait |        return val = 8 |
                | TRACE:        pid = 6 |       process name = trd |    syscall = write |       return val = 1 |
                | TRACE:        pid = 6 |       process name = trd |    syscall = exit |        return val = 2 |
                | TRACE:        pid = 10 |      process name = sh |     syscall = exec |        return val = 0 |
                | TRACE:        pid = 10 |      process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 10 |      process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 10 |      process name = echo |   syscall = write |       return val = 1 |
                | TRACE:        pid = 10 |      process name = echo |   syscall = exit |        return val = 2 |
$
```

EXTRA CREDIT FORMATTING:

- Formatted the output to look a little clean and becomes easily readable by the user as it can be seen in the screenshots above

3. BUILDING OPTIONS FOR STRACE + EXTRA CREDIT EXECUTING MULTIPLE FLAGS:

i) **trace.h**

- Add a structure tracecall in trace.h which will store the option and syscallname from sh.c as follows:

```
struct tracecall{
   char option[2];
   char syscallname[20];
};
```

ii) **sh.c**

- Create a groble character array for all flags which will be used to compare with user input and pass values to our kernel.

```
char trace_option_e[] = "strace -e";
char trace_option_s[] = "strace -s\n";
char trace_option_f[] = "strace -f\n";
char trace_option_s_e[] = "strace -s -e"; //passed as option d
char trace_option_f_e[] = "strace -f -e"; //passed as option p
struct tracecall tc;
```

- Add string compare functions similar to what we did for strace run <command> in while (getcmd(buf, sizeof(buf)>=0) for all the flags as follows:

```
if (streqrun(buf, trace_option_e, 9)) // added
       {
           static char bufcommand[100];
           int i = 10, j=0;
           while(i<sizeof(buf) && buf[i]!=0){
               bufcommand[j] = buf[i];
               i++;
               j++;
           }
           *(char*)tc.option = 'e';
           memmove(tc.syscallname, bufcommand, sizeof(tc.syscallname));
           passtokernel(&tc); //added
           tracing = 1;
           continue;
       }
       if (streqrun(buf, trace_option_s, 10)) // added
```

```
        {
            *(char*)tc.option = 's';
            passtokernel(&tc); //added
            tracing = 1;
            continue;
        }
        if (streqrun(buf, trace_option_f, 10)) // added
        {
            *(char*)tc.option = 'f';
            passtokernel(&tc); //added
            tracing = 1;
            continue;
        }
        if(streqrun(buf,trace_option_s_e,12))
        {
            static char bufcommand[100];
            int i = 13, j=0;
            while(i<sizeof(buf) && buf[i]!=0){
                bufcommand[j] = buf[i];
                i++;
                j++;
            }
            *(char*)tc.option = 'd';
            memmove(tc.syscallname, bufcommand, sizeof(tc.syscallname));
            passtokernel(&tc); //added
            tracing = 1;
            continue;
        }
        if(streqrun(buf,trace_option_f_e,12))
        {
            static char bufcommand[100];
            int i = 13, j=0;
            while(i<sizeof(buf) && buf[i]!=0){
                bufcommand[j] = buf[i];
                i++;
                j++;
            }
            *(char*)tc.option = 'p';
            memmove(tc.syscallname, bufcommand, sizeof(tc.syscallname));
            passtokernel(&tc); //added
            tracing = 1;
            continue;
```

```
       }
```
This fetches the option and syscallname variable from user and passes it to the structure tracecall that we defined in trace.h file using a systemcall passtokernel

iii) **proc.c**

- In proc.c we define a character array kernel_opt of type struct tracecall where we initially set the option and syscallname value to null

```
struct tracecall kernel_opt={
   .option = " \0",
   .syscallname = "                 \0",
};
```

- After that we define function: passtokernel which copies the option and syscall name from struct tracecall to our character array kernel_opt

```
int passtokernel(struct tracecall *tc) //added
{
   memmove(kernel_opt.option, tc->option, 2);
   memmove(kernel_opt.syscallname, tc->syscallname, 20);
   return 0;
}
```

- After that, we define another function resetoption which resets the value of option and syscallname after our code has been executed for the first time, this function is called later in syscall.c

```
void resetoption(struct tracecall tc)
{
   memmove(tc.option, " \0", 2);
   memmove(tc.syscallname, "                 \0", 20);
   memmove(kernel_opt.option, " \0", 2);
   memmove(kernel_opt.syscallname, "                 \0", 20);
}
```

iv) **sysproc.c**

- In sysproc.c we declare kernel_opt as follows:

```
extern struct tracecall kernel_opt;
```

- In sys_passtokernel we fetch the value from struct tracecall in out kernel and it is later passed to proc.c where the value is copied in kernel_opt as follows:

```
int sys_passtokernel(void){ //added
   struct tracecall *tc;
   argstr(0,  (void *)&tc);
  return passtokernel(tc);
}
```

- In sys_resetoption we pass the values in struct tracecall to our function resetoption as follows:

```
 void sys_resetoption(void){
   struct tracecall tc;
   argstr(0,  (void *)&tc);
   resetoption(tc);
}
```

v) **usys.S**

- In usys.S add the system calls passtokernel and resetoption as follows:

```
SYSCALL(passtokernel)
SYSCALL(resetoption)
```

vi) **syscall.h**

- In syscall.h add syscall numbers for passtokernel and resetoption as follows:

```
#define SYS_passtokernel 23 //added
#define SYS_resetoption 24 //added
```

vii) **syscall.c**

- In syscall.c declare a variable for the structure tracecall, and declare the system calls resetoption and passtokernel as extern int

```
struct tracecall *tc;
extern int sys_resetoption(void); //added
extern int sys_passtokernel(void);
```

- In static int(*syscalls[])(void) = {} add [SYS_passtokernel] sys_passtokernel, [SYS_resetoption] sys_resetoption,

- Declare kernel_opt as follows:

```
extern struct tracecall kernel_opt;
```

- In void syscall(void) if num == SYS_exit and tracing is on then we will check if kernel_opt.option we create a switch case for kernel_opt.option[0] as follows:

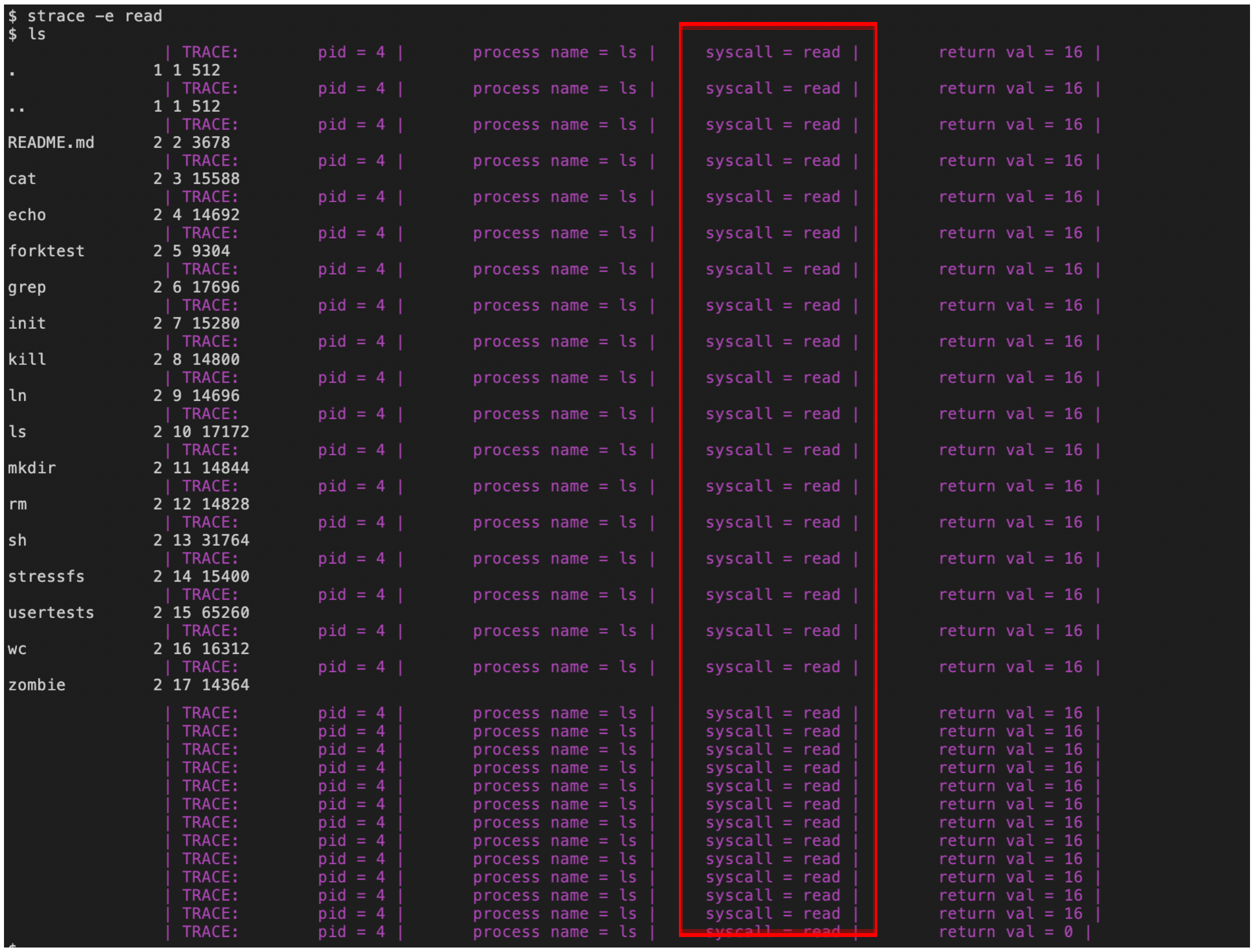
```
if(kernel_opt.option){
         switch (kernel_opt.option[0]) {
           case 'e':
           {
if(streq(syscall_names[num],kernel_opt.syscallname)){
                   cprintf("\e[35m\t\t| TRACE:\tpid = %d
|\tprocess name = %s |\tsyscall = %s |\e[0m\n",
                   proc->pid, procname, syscall_names[num]);
               }
               resetoption(kernel_opt);
               break;
           }

           case 's':
           {
               resetoption(kernel_opt);
               break;
           }

           case 'f':
           {
               resetoption(kernel_opt);
               break;
           }
           case 'd': //strace -s -e <syscallname>
           {
               resetoption(kernel_opt);
               break;
           }
           case 'p': //strace -f -e <syscallname>
           {
               resetoption(kernel_opt);
               break;
           }

           default:
          {
              cprintf("\e[35m\t\t| TRACE:\tpid = %d |\tprocess
name = %s |\tsyscall = %s |\e[0m\n", proc->pid, procname,
syscall_names[num]);
           }
         }
       }
```

- Next, if (num < NELEM(syscalls) && syscalls[num]) we add the code for all our options as follows:

```
if (num < NELEM(syscalls) && syscalls[num]) {
   proc->tf->eax = syscalls[num]();

if (is_traced) { //added
 switch (kernel_opt.option[0]) {
    case 'e':
    {if(streq(syscall_names[num],kernel_opt.syscallname))
       {cprintf((num == SYS_exec && proc->tf->eax == 0) ?
             "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\e[0m\n" : "\e[35m\t\t| TRACE:\tpid = %d
|\tprocess name = %s |\tsyscall = %s |\treturn val = %d |\e[0m\n",
          proc->pid, procname, syscall_names[num], proc->tf->eax);
                }
          break;
     }

   case 's':
   { if(proc->tf->eax != -1){
       cprintf((num == SYS_exec && proc->tf->eax == 0) ?
       "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\e[0m\n" : "\e[35m\t\t| TRACE:\tpid = %d
|\tprocess name = %s |\tsyscall = %s |\treturn val = %d |\e[0m\n",
       proc->pid, procname, syscall_names[num], proc->tf->eax);
               }
         break;
       }

    case 'f':
  { if(proc->tf->eax==-1){
    cprintf( "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\treturn val = %d |\e[0m\n",
      proc->pid, procname, syscall_names[num], proc->tf->eax);
                 }
           break;
      }
    case 'd': //strace -s -e <syscallname>
   {if(streq(syscall_names[num],kernel_opt.syscallname) &&
proc->tf->eax != -1)
       {cprintf((num == SYS_exec && proc->tf->eax == 0) ?
          "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s
|\tsyscall = %s |\e[0m\n" : "\e[35m\t\t| TRACE:\tpid = %d
|\tprocess name = %s |\tsyscall = %s |\treturn val = %d |\e[0m\n",
         proc->pid, procname, syscall_names[num], proc->tf->eax);
                }
      break;
     }
    case 'p': //strace -f -e <syscallname>
   { if(streq(syscall_names[num],kernel_opt.syscallname) &&
proc->tf->eax==-1)
        { cprintf( "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name =
%s |\tsyscall = %s |\treturn val = %d |\e[0m\n",
         proc->pid, procname, syscall_names[num], proc->tf->eax);
           }
      break;
    }
    default:{
    cprintf((num == SYS_exec && proc->tf->eax == 0) ?
    "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name = %s |\tsyscall
= %s |\e[0m\n" : "\e[35m\t\t| TRACE:\tpid = %d |\tprocess name =
%s |\tsyscall = %s |\treturn val = %d |\e[0m\n",
      proc->pid, procname, syscall_names[num], proc->tf->eax);
               }
           }
       }
   }
```

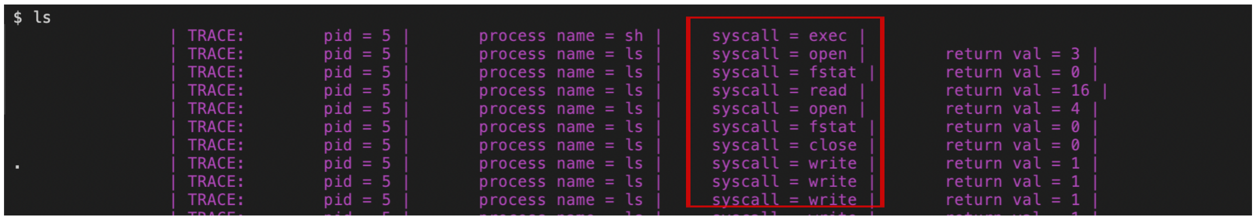
<u>How to run commands in xv6 and their OUTPUT:</u>

1. **For -e flag** type strace -e <syscallname> example $strace -e read $ls gives the following output



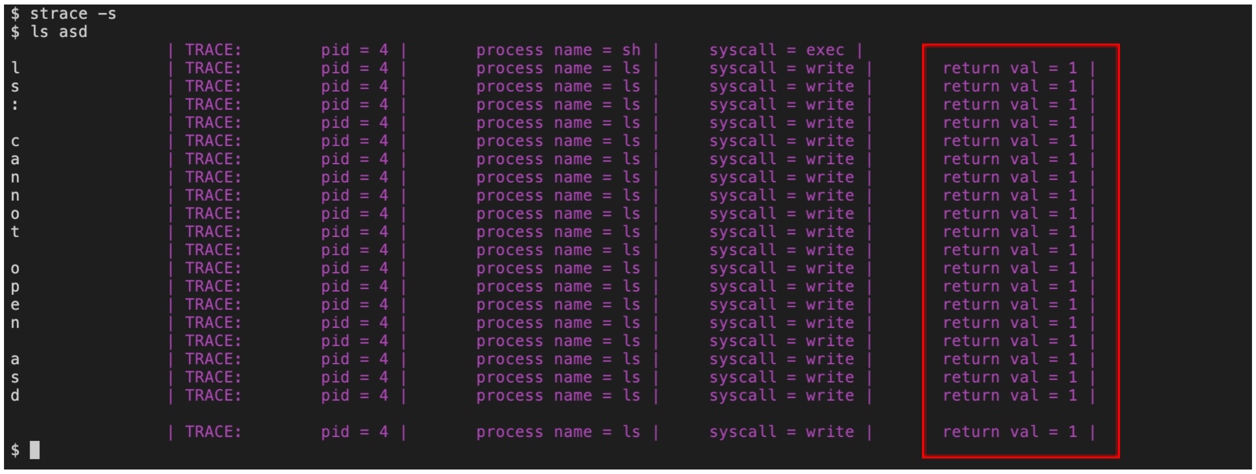
```
$ strace -e read
$ ls
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
.              1 1 512
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
..             1 1 512
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
README.md      2 2 3678
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
cat            2 3 15588
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
echo           2 4 14692
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
forktest       2 5 9304
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
grep           2 6 17696
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
init           2 7 15280
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
kill           2 8 14800
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
ln             2 9 14696
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
ls             2 10 17172
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
mkdir          2 11 14844
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
rm             2 12 14828
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
sh             2 13 31764
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
stressfs       2 14 15400
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
usertests      2 15 65260
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
wc             2 16 16312
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
zombie         2 17 14364

                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = read |        return val = 0 |
```

2. **After executing the above command the value for option and syscallname is reset** automatically so the next time we execute ls trace for all syscalls are printed. This reset option has been added for all flags



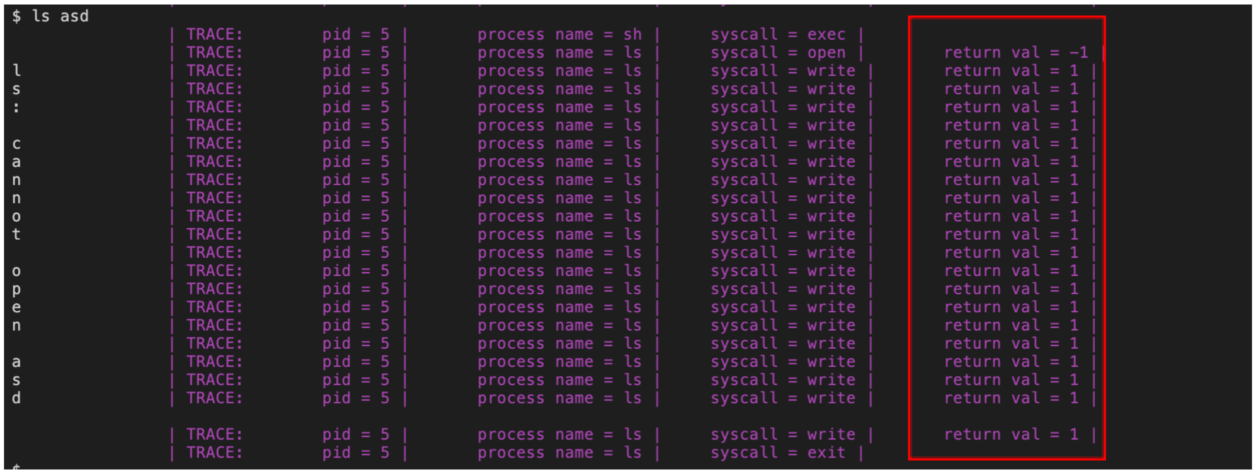
```
$ ls
                | TRACE:        pid = 5 |       process name = sh |     syscall = exec |
                | TRACE:        pid = 5 |       process name = ls |     syscall = open |        return val = 3 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = fstat |       return val = 0 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = read |        return val = 16 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = open |        return val = 4 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = fstat |       return val = 0 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = close |       return val = 0 |
.               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
```

3. **For -s flag** we try to run a systemcall where we know it will fail and see if the call with a return value of -1 is printed or not. For example: ls asd is an invalid system call so when it tries to open asd it should return a value of -1 and that system call should not be printed for -s as this only prints trace for successful system calls i.e. calls with a return value >=0.



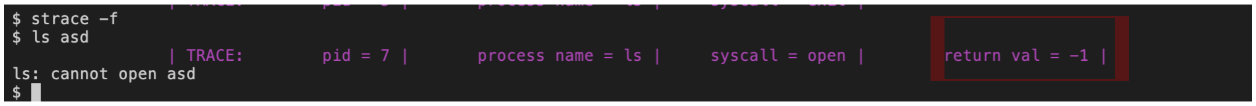
```
$ strace -s
$ ls asd
                | TRACE:        pid = 4 |       process name = sh |     syscall = exec |
l               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
s               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
:               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
c               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
a               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
o               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
t               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
o               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
p               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
e               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
a               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
s               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
d               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |

                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
$ 
```

4. **Now, after command is executed option value is reset** and when we run ls asd we should see all the return all system calls both successful and failed ones



```
$ ls asd
                | TRACE:        pid = 5 |       process name = sh |     syscall = exec |
                | TRACE:        pid = 5 |       process name = ls |     syscall = open |        return val = -1 |
l               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
s               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
:               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
c               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
a               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
o               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
t               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
o               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
p               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
e               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
a               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
s               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
d               | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |

                | TRACE:        pid = 5 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 5 |       process name = ls |     syscall = exit |
```

5. **For -f flag**, the command should only return only return open syscall where the return value is -1 as follows:

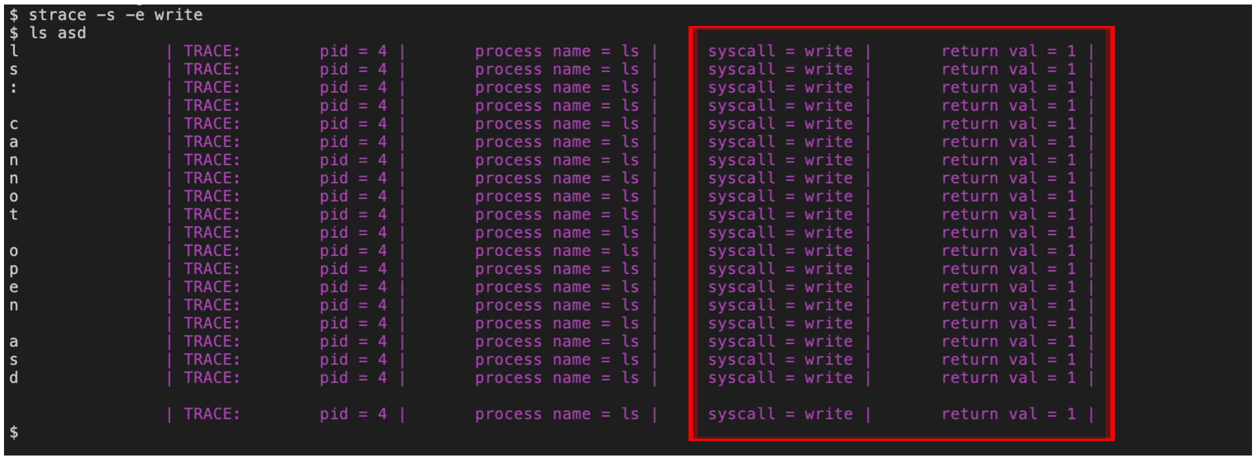


```
$ strace -f
$ ls asd
                | TRACE:        pid = 7 |       process name = ls |     syscall = open |        return val = -1 |
ls: cannot open asd
$ 
```

6. **After execution of -f flag** the option value is again reset automatically and now if we run ls asd we will get the same output as in pt. 5 of this section.

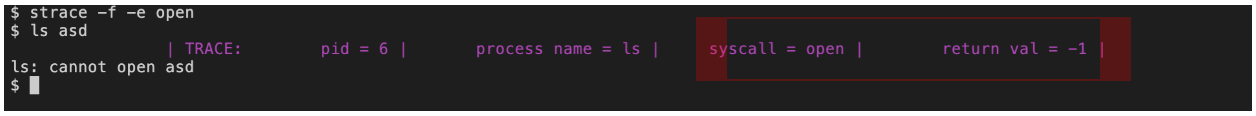
EXTRA CREDIT (COMBINING FLAGS)

7. **Strace -s -e <syscall name>**
   For this we execute the example: $strace -s -e write $ls asd and see among all system calls only successful write calls are printed



```
$ strace -s -e write
$ ls asd
l               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
s               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
:               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
c               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
a               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
o               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
t               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
o               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
p               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
e               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
n               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
a               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
s               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
d               | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |

                | TRACE:        pid = 4 |       process name = ls |     syscall = write |       return val = 1 |
$
```

8. **Strace -f -e <syscall name>**
   For this we execute the example: $strace -f -e open $ls asd and see among all system calls only failed open calls are printed



```
$ strace -f -e open
$ ls asd
                | TRACE:        pid = 6 |       process name = ls |     syscall = open |        return val = -1 |
ls: cannot open asd
$
```

5) MEMORY LEAK

- Write a memory leak program as below in user and call it memorkleak.c
- In this code. We are allocating 10MB of memory to our process 30 times and exiting without deallocating it. This is leading to memory leak.

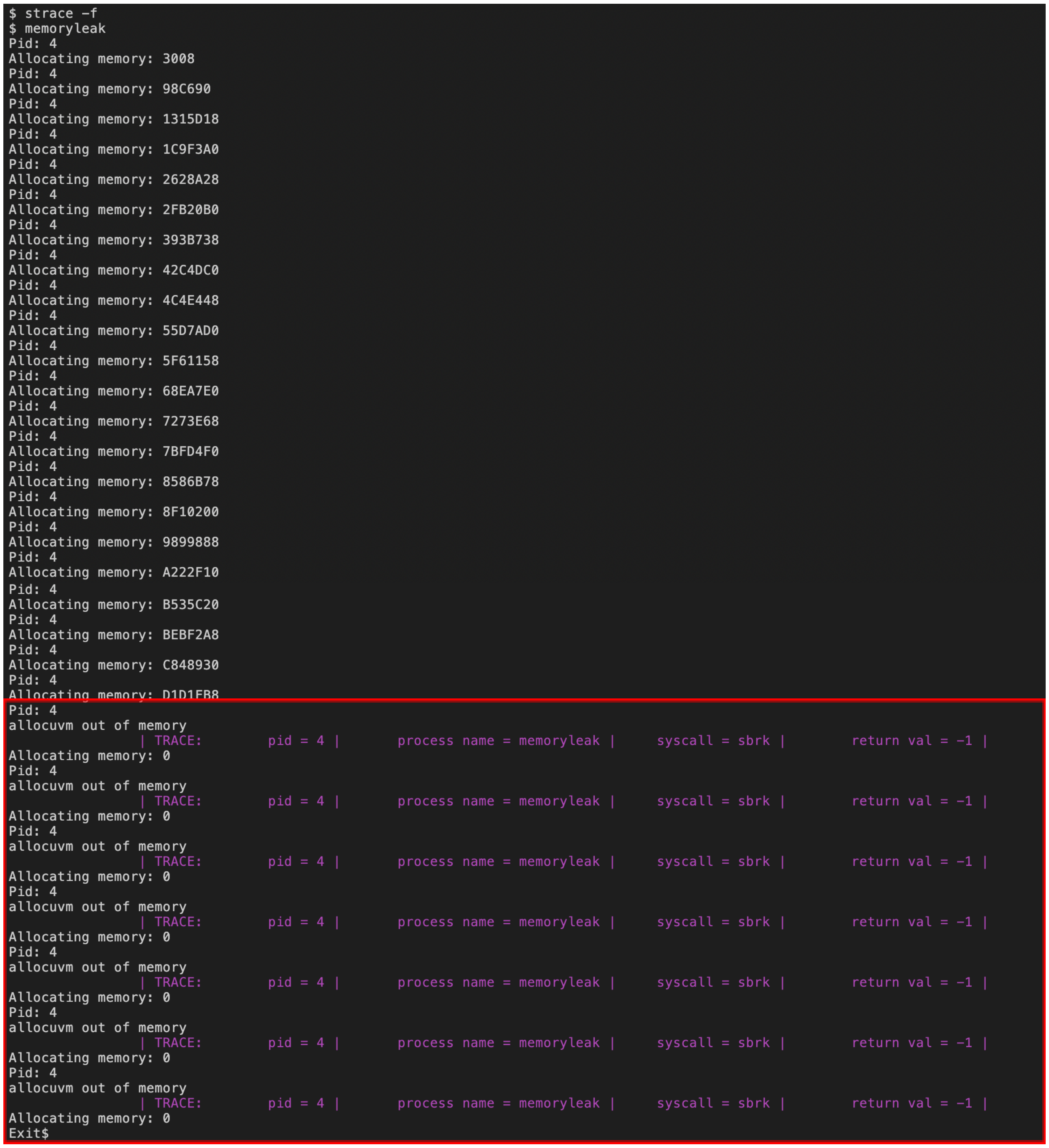
```
#include "kernel/types.h"
#include "user/user.h"

int main(){
   int i;
   for(i=0; i<30; i++){
       printf(1,"Pid: %d\n",getpid());
       char* mem = malloc(10000000);
       printf(1,"Allocating memory: %p\n",mem);
       sleep(100);
   }
   printf(1, "Exit");
   exit();
}
```

Running this program:

1) Make clean; make qemu;
2) Enter the following commands:
   $strace on
   $memoryleak
   You will notice when the memoryleak file is executes it allocates memory to the process until the allocated memory is full and then it shows “allocvm out of memory” error
3) To visualize this run the following command:
   $strace -f
   $memoryleak

OUTPUT:

Here when allocated memory is exhausted it throws an error allocvm out of memory and the return value of this error is -1



```
$ strace -f
$ memoryleak
Pid: 4
Allocating memory: 3008
Pid: 4
Allocating memory: 98C690
Pid: 4
Allocating memory: 1315D18
Pid: 4
Allocating memory: 1C9F3A0
Pid: 4
Allocating memory: 2628A28
Pid: 4
Allocating memory: 2FB20B0
Pid: 4
Allocating memory: 393B738
Pid: 4
Allocating memory: 42C4DC0
Pid: 4
Allocating memory: 4C4E448
Pid: 4
Allocating memory: 55D7AD0
Pid: 4
Allocating memory: 5F61158
Pid: 4
Allocating memory: 68EA7E0
Pid: 4
Allocating memory: 7273E68
Pid: 4
Allocating memory: 7BFD4F0
Pid: 4
Allocating memory: 8586B78
Pid: 4
Allocating memory: 8F10200
Pid: 4
Allocating memory: 9899888
Pid: 4
Allocating memory: A222F10
Pid: 4
Allocating memory: B535C20
Pid: 4
Allocating memory: BEBF2A8
Pid: 4
Allocating memory: C848930
Pid: 4
Allocating memory: D1D1FB8
Pid: 4
allocuvm out of memory
                | TRACE:        pid = 4 |       process name = memoryleak |     syscall = sbrk |        return val = -1 |
Allocating memory: 0
Pid: 4
allocuvm out of memory
                | TRACE:        pid = 4 |       process name = memoryleak |     syscall = sbrk |        return val = -1 |
Allocating memory: 0
Pid: 4
allocuvm out of memory
                | TRACE:        pid = 4 |       process name = memoryleak |     syscall = sbrk |        return val = -1 |
Allocating memory: 0
Pid: 4
allocuvm out of memory
                | TRACE:        pid = 4 |       process name = memoryleak |     syscall = sbrk |        return val = -1 |
Allocating memory: 0
Pid: 4
allocuvm out of memory
                | TRACE:        pid = 4 |       process name = memoryleak |     syscall = sbrk |        return val = -1 |
Allocating memory: 0
Pid: 4
allocuvm out of memory
                | TRACE:        pid = 4 |       process name = memoryleak |     syscall = sbrk |        return val = -1 |
Allocating memory: 0
Pid: 4
allocuvm out of memory
                | TRACE:        pid = 4 |       process name = memoryleak |     syscall = sbrk |        return val = -1 |
Allocating memory: 0
Exit$
```

The -f flag prints only unsuccessful trace calls and makes it easier for us to visualize which trace calls were unsuccessful.

Hence with the use of our stace call -f, we can easily identify when out program starts failing.

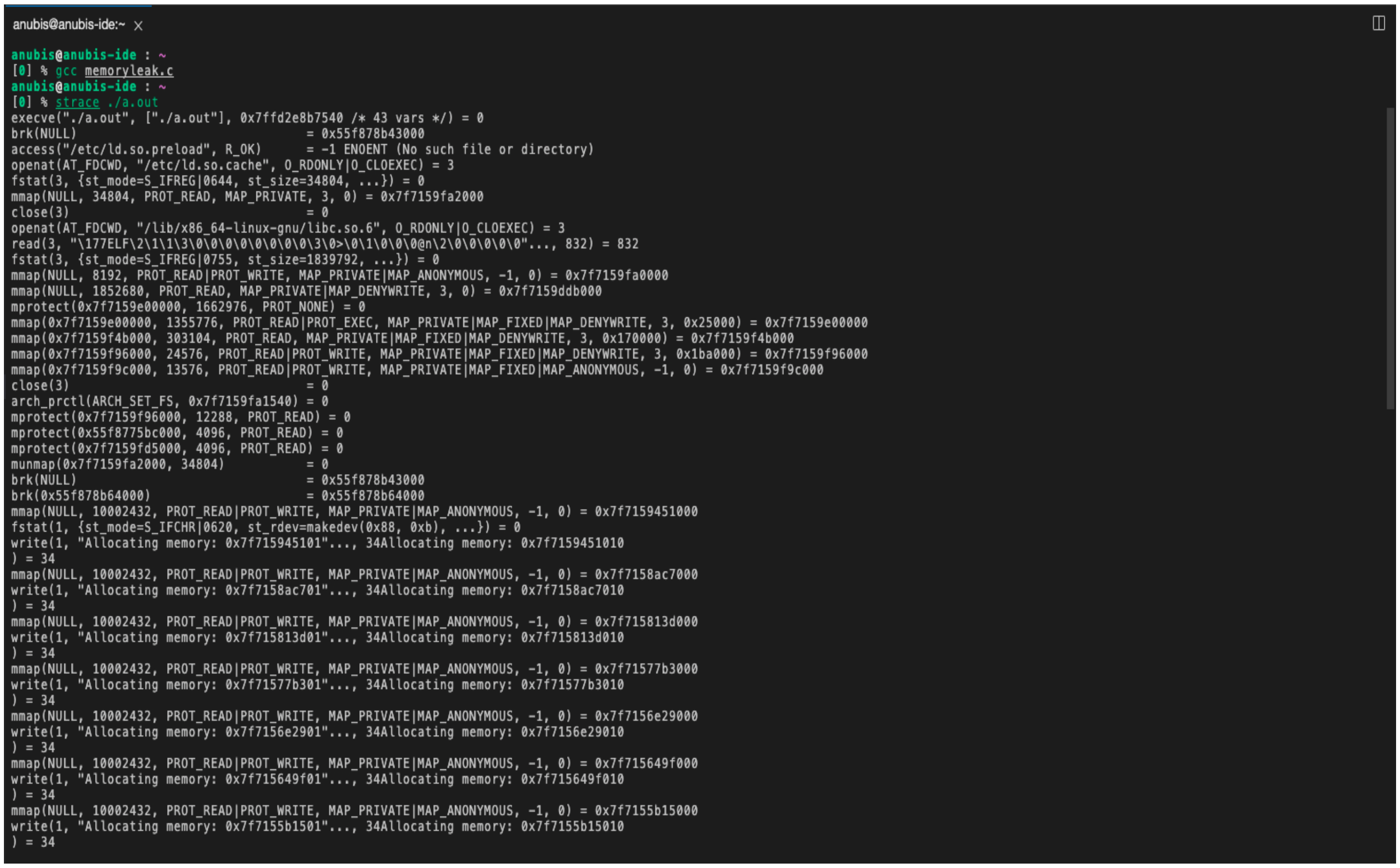
The output is included in the next page

RUNNING memorkleak.c in LINUX:

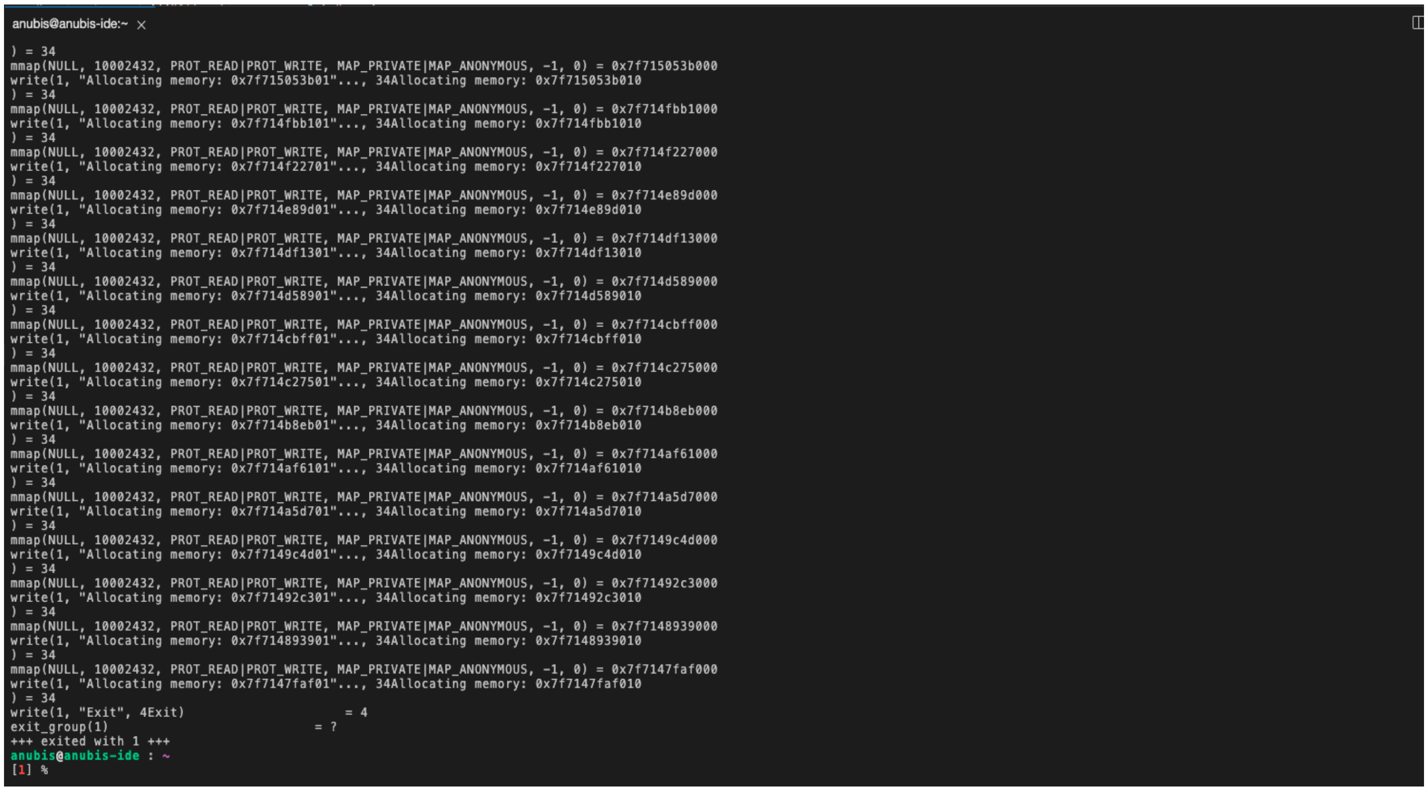
```
#include "stdio.h"
#include "stdlib.h"
#include "unistd.h"
int main(){
  int i;
  for(i=0; i<30; i++){
      char* mem = malloc(10000000);
      printf("Allocating memory: %p\n",mem);
  }
  printf("Exit");
  exit(1);
}
```

To run memoryleak.c in LINUX run the following commands in terminal:
$ gcc memoryleak.c
$ strace ./a.out

OUTPUT:



```
anubis@anubis-ide:~ ×

anubis@anubis-ide : ~
[0] % gcc memoryleak.c
anubis@anubis-ide : ~
[0] % strace ./a.out
execve("./a.out", ["./a.out"], 0x7ffd2e8b7540 /* 43 vars */) = 0
brk(NULL)                               = 0x55f878b43000
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) = 0x7f7159fa2000
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) = 0x7f7159fa0000
mmap(NULL, 1852680, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7159ddb000
mprotect(0x7f7159e00000, 1662976, PROT_NONE) = 0
mmap(0x7f7159e00000, 1355776, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x25000) = 0x7f7159e00000
mmap(0x7f7159f4b000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x170000) = 0x7f7159f4b000
mmap(0x7f7159f96000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7f7159f96000
mmap(0x7f7159f9c000, 13576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7159f9c000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f7159fa1540) = 0
mprotect(0x7f7159f96000, 12288, PROT_READ) = 0
mprotect(0x55f8775bc000, 4096, PROT_READ) = 0
mprotect(0x7f7159fd5000, 4096, PROT_READ) = 0
munmap(0x7f7159fa2000, 34804)           = 0
brk(NULL)                               = 0x55f878b43000
brk(0x55f878b64000)                     = 0x55f878b64000
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7159451000
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0xb), ...}) = 0
write(1, "Allocating memory: 0x7f715945101"..., 34Allocating memory: 0x7f7159451010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7158ac7000
write(1, "Allocating memory: 0x7f7158ac701"..., 34Allocating memory: 0x7f7158ac7010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f715813d000
write(1, "Allocating memory: 0x7f715813d01"..., 34Allocating memory: 0x7f715813d010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f71577b3000
write(1, "Allocating memory: 0x7f71577b301"..., 34Allocating memory: 0x7f71577b3010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7156e29000
write(1, "Allocating memory: 0x7f7156e2901"..., 34Allocating memory: 0x7f7156e29010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f715649f000
write(1, "Allocating memory: 0x7f715649f01"..., 34Allocating memory: 0x7f715649f010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7155b15000
write(1, "Allocating memory: 0x7f7155b1501"..., 34Allocating memory: 0x7f7155b15010
) = 34
```



```
anubis@anubis-ide:~
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f715053b000
write(1, "Allocating memory: 0x7f715053b01"..., 34Allocating memory: 0x7f715053b010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714fbb1000
write(1, "Allocating memory: 0x7f714fbb101"..., 34Allocating memory: 0x7f714fbb1010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714f227000
write(1, "Allocating memory: 0x7f714f22701"..., 34Allocating memory: 0x7f714f227010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714e89d000
write(1, "Allocating memory: 0x7f714e89d01"..., 34Allocating memory: 0x7f714e89d010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714df13000
write(1, "Allocating memory: 0x7f714df1301"..., 34Allocating memory: 0x7f714df13010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714d589000
write(1, "Allocating memory: 0x7f714d58901"..., 34Allocating memory: 0x7f714d589010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714cbff000
write(1, "Allocating memory: 0x7f714cbff01"..., 34Allocating memory: 0x7f714cbff010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714c275000
write(1, "Allocating memory: 0x7f714c27501"..., 34Allocating memory: 0x7f714c275010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714b8eb000
write(1, "Allocating memory: 0x7f714b8eb01"..., 34Allocating memory: 0x7f714b8eb010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714af61000
write(1, "Allocating memory: 0x7f714af6101"..., 34Allocating memory: 0x7f714af61010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f714a5d7000
write(1, "Allocating memory: 0x7f714a5d701"..., 34Allocating memory: 0x7f714a5d7010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7149c4d000
write(1, "Allocating memory: 0x7f7149c4d01"..., 34Allocating memory: 0x7f7149c4d010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f71492c3000
write(1, "Allocating memory: 0x7f71492c301"..., 34Allocating memory: 0x7f71492c3010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7148939000
write(1, "Allocating memory: 0x7f714893901"..., 34Allocating memory: 0x7f7148939010
) = 34
mmap(NULL, 10002432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7147faf000
write(1, "Allocating memory: 0x7f7147faf01"..., 34Allocating memory: 0x7f7147faf010
) = 34
write(1, "Exit", 4Exit)                     = 4
exit_group(1)                           = ?
+++ exited with 1 +++
anubis@anubis-ide : ~
[1] %
```

EXPLANATION:

- In Linux, strace call output also, we can see that the nmap system call starts giving -1 return value once the memory leak happens. It is similar to our strace implementation on xv6.
- However, we can also observe that linux's strace has much more information about the calls being made including the memory address of the memory being allocated. It might be handy for some debugging scenarios.