Final Project – OS
CSGY 6233

Strace implementation in XV6

Table of Content

1. Getting Familiar with Linux strace

1.1 In Linux, we can use strace to debug a particular piece of software to see what system calls and library calls it is using at a low level. With this we can roughly guess what the program does and what system calls it fail. In linux, **strace echo hello** would tell us what low level functions the binary "echo" uses when it takes in input "hello" and displays it.

```
hex@DESKTOP-AMPIGLE:~$ strace echo hello
execve("/usr/bin/echo", ["echo", "hello"], 0x7ffd0e5ba6f8 /* 24 vars */) = 0
brk(NULL)                               = 0x5584afbb3000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffed7529840) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f371d15f000
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
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=14795, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 14795, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f371d15b000
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\0P\237\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0i8\235HZ\227\223\333\350s\360\352,\223\340."..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2216304, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f371cf33000
mmap(0x7f371cf5b000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f3
mmap(0x7f371d0f0000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f371d0f0000
mmap(0x7f371d148000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7f3
mmap(0x7f371d14e000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f371d14e
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f371cf30000
arch_prctl(ARCH_SET_FS, 0x7f371cf30740) = 0
set_tid_address(0x7f371cf30a10)         = 33
set_robust_list(0x7f371cf30a20, 24)     = 0
rseq(0x7f371cf310e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f371d148000, 16384, PROT_READ) = 0
mprotect(0x5584afad5000, 4096, PROT_READ) = 0
mprotect(0x7f371d199000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f371d15b000, 14795)           = 0
getrandom("\xd7\x7f\x5a\x18\x6c\xfc\x5f\x38", 8, GRND_NONBLOCK) = 8
```

1.2 There were numerous system calls made while running "**strace echo hello**". Four such system calls are:

- **execve** – Commonly used by shells, execve starts a new process and executes programs with different command line options. Here, we can see echo, hello, <address of environment variables> being passed and return value 0 indicates successful run of the program.
- **brk** – It is a system call which is used to adjust the size of a process' heap memory. When brk(NULL) is called, it returns the current address of the heap.
- **mmap** – It is used to map files into memory allowing access to them as if they were a block of memory. It takes in six arguments: start addr, length in bytes, protection flags, additional flags, file descriptor (ignore here since MAP_ANONYMOUS is used) and offset. It returns the newly allocated memory address.

- **openat** – It is used here to open up the file /etc/ld.so.cache that contains a cache of the shared libraries on the system with a read only flag, When an executable or shared library is loaded, the dynamic linker uses this cache to quickly locate its dependencies without having to search the file system each time. On success it gives a file descriptor integer, here 3.

---

2. Building strace in xv6

To implement strace, following approach was taken. Upon inspecting the working of xv6 kernel space, we observed that syscall() in syscall.c seemed like a good area to input our strace code handler. This is because it can be easily used to map out entire system calls currently in use/used for a program in xv6. We designed a few system calls to help us throughout the program strace. To print out the system call names, we utilized the existing array in syscall.c and designed a new one which maps syscall numbers to their following string values. More details about our implementation are in their respective sections.

2.1 Implement "strace on"

To implement strace on, step one was to setup a system call which enables strace flag. This was done by trace() system call in sysfile.c and its definition in syscall.c,user.h,usys.S and supporting files. This turns on the flag trace_on in proc.h. In syscall(), program checks if trace_on is set to use strace. We also setup an array of system call names (syscall_names) and corresponding numeric value to help us trace. Additional check on SYS_trace is put to suppress immediate strace output after typing "strace on".

```
444  int
445  sys_trace(void)
446  {
447    int n;
448    if(argint(0, &n) < 0)
449      return -1;
450    proc->trace_on = n;
451    return 0;
452  }
453
```

sysfile.c

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this p
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[32];               // Process name (debugging)
  int trace_on;

};
```

proc.h

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

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    ret = syscalls[num]();
    proc->tf->eax = ret;
    if(proc->trace_on && num != SYS_trace){
      cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->name, syscall_names[num], ret);
      add_event(proc->pid, proc->name, syscall_names[num], ret);
    }
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

syscall.c

```
$ strace on
$ echo hello
TRACE: pid = 8 | process name = echo | syscall = exec | return = 0
hTRACE: pid = 8 | process name = echo | syscall = write | return = 1
eTRACE: pid = 8 | process name = echo | syscall = write | return = 1
lTRACE: pid = 8 | process name = echo | syscall = write | return = 1
lTRACE: pid = 8 | process name = echo | syscall = write | return = 1
oTRACE: pid = 8 | process name = echo | syscall = write | return = 1

TRACE: pid = 8 | process name = echo | syscall = write | return = 1
$
```

output

2.2 Implement “strace off”

We again used the same methodology as in strace on, but this time we switched trace_on to 0 through the user level strace.c program. This makes syscall() works as in default state.

```
int
main(int argc, char *argv[])
{
  int cur;
  if(argc < 2){
    printf(2, "Usage: strace [on|off|run <command>]\n");
    exit();
  }

  if(strcmp(argv[1], "on") == 0) {
    trace(1);
  } else if(strcmp(argv[1], "off") == 0) {
    trace(0);
  } else if(strcmp(argv[1], "run") == 0) {
    if(argc < 3){
      printf(2, "Usage: strace run <command>\n");
      exit();
    }
```

strace.c

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ strace on
$ echo a
TRACE: pid = 4 | process name = echo | syscall = exec | return = 0
aTRACE: pid = 4 | process name = echo | syscall = write | return = 1

TRACE: pid = 4 | process name = echo | syscall = write | return = 1
$ strace off
TRACE: pid = 5 | process name = strace | syscall = exec | return = 0
$ echo a
a
$
```

output

2.3 Implementing “strace run”

To implement this, we adopted a simple process of forking. The forked process (child) turns on the strace (sets flag trace_on to 1), performs the command supplied (using exec call) and then finally turns sets trace_on to it’s initial state as it was before the command.

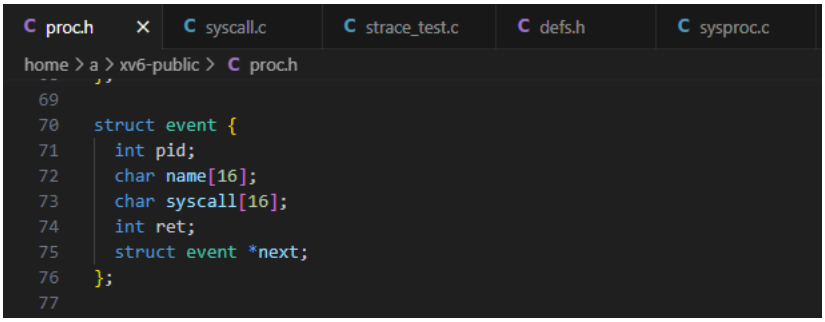
```
18    } else if(strcmp(argv[1], "run") == 0) {
19      if(argc < 3){
20        printf(2, "Usage: strace run <command>\n");
21        exit();
22      }
23
24      int pid = fork();
25      if(pid < 0) {
26        printf(2, "fork failed\n");
27        exit();
28      } else if(pid == 0) { // Child process
29        trace(1); // Enable strace
30        if(exec(argv[2], &argv[2]) < 0) {
31          printf(2, "exec failed for %s\n", argv[2]);
32          exit();
33        }
34      } else { // Parent process
35        wait(); // Wait for the child to finish
36      }
37    }
```

strace.c – added run handler

```
$ strace off
TRACE: pid = 11 | process name = strace | syscall = exec | return = 0
$
$ strace run echo a
TRACE: pid = 14 | process name = echo | syscall = exec | return = 0
aTRACE: pid = 14 | process name = echo | syscall = write | return = 1

TRACE: pid = 14 | process name = echo | syscall = write | return = 1
$ strace run echo hello
TRACE: pid = 16 | process name = echo | syscall = exec | return = 0
hTRACE: pid = 16 | process name = echo | syscall = write | return = 1
eTRACE: pid = 16 | process name = echo | syscall = write | return = 1
lTRACE: pid = 16 | process name = echo | syscall = write | return = 1
lTRACE: pid = 16 | process name = echo | syscall = write | return = 1
oTRACE: pid = 16 | process name = echo | syscall = write | return = 1

TRACE: pid = 16 | process name = echo | syscall = write | return = 1
$ echo a
a
$
```

2.4 Implementing strace dump

To implement this, we configured a dumptrace system call. This would run add_event() in syscall(). We implemented a linked list data structure to store most recent 35 system calls using a function called add_event(). This is defined using a macro “#define N 35” in syscall.c. When strace dump is used, most recent 35 system calls are dumped in LIFO order. We added the supporting structure in proc.h.



proc.h

```c
#define N 35 //for dump as per instructiions, hardcoded value
struct event *event_list = 0;
static int event_count = 0;

void add_event(int pid, char *name, char *syscall, int ret) {
  struct event *new_event = (struct event *)kalloc();
  if (!new_event) {
    cprintf("Failed to allocate memory for event\n");
    return;
  }

  new_event->pid = pid;
  strncpy(new_event->name, name, sizeof(new_event->name) - 1);
  strncpy(new_event->syscall, syscall, sizeof(new_event->syscall) - 1);
  new_event->ret = ret;
  new_event->next = event_list;

  event_list = new_event;
  event_count++;

  // Remove the oldest event if the list size exceeds N
  if (event_count > N) {
    struct event *current = event_list;
    while (current->next->next) {
      current = current->next;
    }
    kfree((char*)current->next);
    current->next = 0;
    event_count--;
  }
}
```

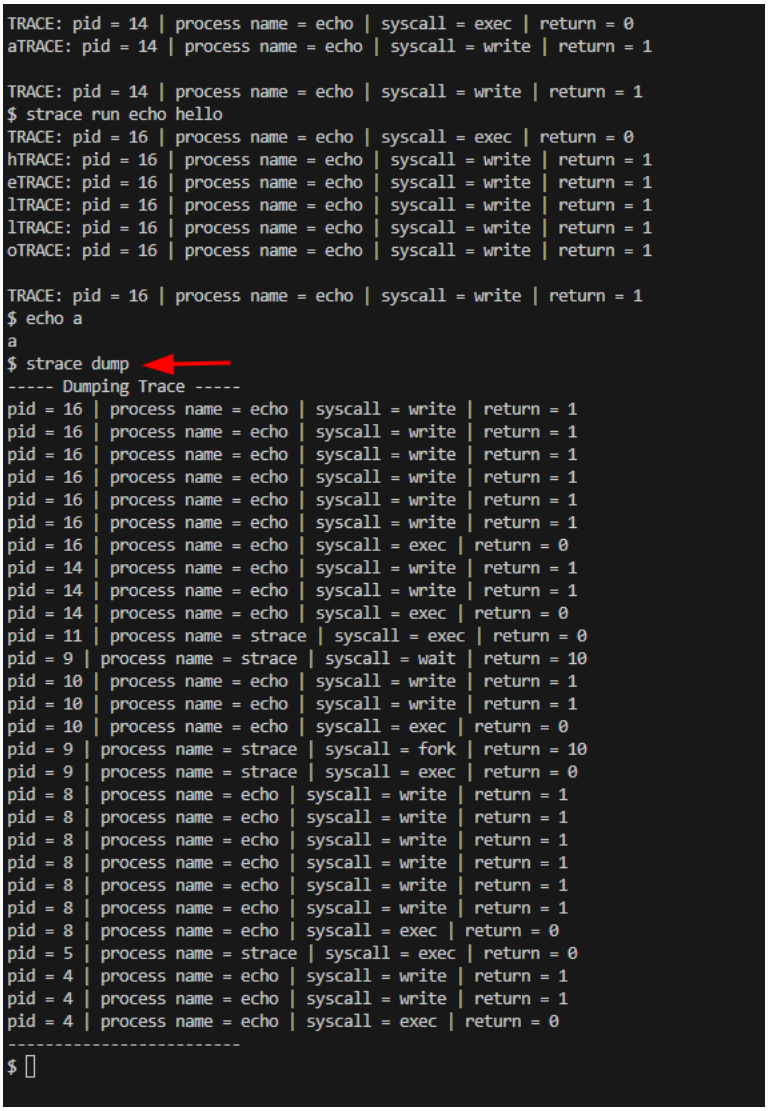
syscall.c – add_event function

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

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    ret = syscalls[num]();
    proc->tf->eax = ret;
    if(proc->trace_on && num != SYS_trace && num!= SYS_sbrk){
      cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->name, syscall_names[num], ret);
      add_event(proc->pid, proc->name, syscall_names[num], ret);
    }
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
```

syscall.c – Added dump handler

```
  else if (strcmp(argv[1], "dump") == 0) {
    cur = gettrace();
    dumptrace();
    trace(cur);
  } else {
    printf(2, "Invalid option, use: strace [on|off|dump|run <command>]\n");
  }

  exit();
}
```

strace.c – Added dump handler



output

2.5 Trace child process

We created a script called strace_test.c. This simple script takes inspiration from Homework 3. Here, 2 child processes are forked. A simple print statement is given for both children and the parent to track when they execute and exit. As we can see, strace is utilizing kernel memory to store strace output of all processes. We can confirm all child processes are being traced. In strace dump, we can see the output being stored too. We can modify variable N (currently 35) to show more lines of output.

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main() {
    int pid = fork();
    if(pid < 0) {   // Child 1
        printf(1, "Error forking child 1.\n");
    } else if (pid == 0) {
        printf(1, "Child 1 Executing\n");
```

```
    } else {
        pid = fork();   // Child 2
        if(pid < 0) {
            printf(1, "Error forking second child.\n");
        } else if(pid == 0) {
            printf(1, "Child 2 Executing\n");
        } else {    // Parent
            printf(1, "Parent Waiting\n");
            wait();
            wait();
        printf(1, "Children Exited\n");
        printf(1, "Parent Executing\n");
        printf(1, "Parent Exiting\n");
        }
    }
    exit();
}
```

```
$ strace on
$ strace_test
TRACE: pid = 21 | process name = strace_test | syscall = exec | return = 0
TRACE: pid = 21 | process name = strace_test | syscall = fork | return = 22
TRACE: pid = 21 | process name = strace_test | syscall = fork | return = 23
PTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
aTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
rTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
eTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
nTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
tTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
 TRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
WTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
aTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
iTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
tTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
Child 2 Executing
iTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
nTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
gTRACE: pid = 21 | process name = strace_test | syscall = write | return = 1

TRACE: pid = 21 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 21 | process name = strace_test | syscall = wait | return = 23
CTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
hTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
iTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
lTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
dTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
 TRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
1TRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
 TRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
ETRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
xTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
eTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
cTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
uTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
tTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
iTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
nTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
gTRACE: pid = 22 | process name = strace_test | syscall = write | return = 1
```

output

```
ETRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
xTRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
iTRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
tTRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
iTRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
nTRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
gTRACE: pid = 4 | process name = strace_test | syscall = write | return = 1

TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
$ strace dump
TRACE: pid = 7 | process name = sh | syscall = sbrk | return = 16384
TRACE: pid = 7 | process name = strace | syscall = exec | return = 0
TRACE: pid = 7 | process name = strace | syscall = sbrk | return = 12288
TRACE: pid = 7 | process name = strace | syscall = gettrace | return = 1
----- Dumping Trace -----
TRACE: pid = 7 | process name = strace | syscall = gettrace | return = 1
TRACE: pid = 7 | process name = strace | syscall = sbrk | return = 12288
TRACE: pid = 7 | process name = strace | syscall = exec | return = 0
TRACE: pid = 7 | process name = sh | syscall = sbrk | return = 16384
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
TRACE: pid = 4 | process name = strace_test | syscall = write | return = 1
```

output – dumping process' strace

2.6 Extra credits - Format a more readable output

To implement a more readable output, we inspected xv6 even further. We observed that when write() is used, sys_write is called and thereafter, a filewrite function is called to write the output to console. We introduced a simple check in this system call which simply returns 1 when write is used instead of calling filewrite, except when file is not writable it returns -1 directly (inspired from filewrite in file.c). This cleaned up our output making strace more clearly visible. This does not violate any of the existing working since xv6, even on invalid commands writes outputs with write return value as 1 (due to their printed output on console) and non-writable files give -1 value. In short, we found a way to suppress command output when strace is on without impairing functionality.

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;

  if(proc->trace_on)
  {
    if (f->writable == 0)
      return -1;
    else return 1;
  }
  else return filewrite(f, p, n);
}
```

sysfile.c – Added trace_on handler

```
$ strace on
$ echo hello
TRACE: pid = 18 | process name = echo | syscall = exec | return = 0
TRACE: pid = 18 | process name = echo | syscall = write | return = 1
TRACE: pid = 18 | process name = echo | syscall = write | return = 1
TRACE: pid = 18 | process name = echo | syscall = write | return = 1
TRACE: pid = 18 | process name = echo | syscall = write | return = 1
TRACE: pid = 18 | process name = echo | syscall = write | return = 1
TRACE: pid = 18 | process name = echo | syscall = write | return = 1
$ strace off
TRACE: pid = 19 | process name = strace | syscall = exec | return = 0
$ echo hello
hello
$ strace run echo aaa
TRACE: pid = 22 | process name = echo | syscall = exec | return = 0
TRACE: pid = 22 | process name = echo | syscall = write | return = 1
TRACE: pid = 22 | process name = echo | syscall = write | return = 1
TRACE: pid = 22 | process name = echo | syscall = write | return = 1
TRACE: pid = 22 | process name = echo | syscall = write | return = 1
$
```

output

---

3. Building options for strace

The section covers up additional options we need to setup for strace. This part included majority of changes in handling options in kernel level strace functionality. Furthermore, various changes in strace.c to handle these options were made and some additional functions were written. Full implementation of each part is in their respective section.

3.1 Option -e <system call name>

The option should print only the strace output with system call specified in the -e option. For this we introduced a new flag in proc.h under struct flags (int e and it's supporting system call name). We introduced and modified handleflags function (which is also a system call) to include flag e and system call name. We modified syscall() to handle when flag e is set, we'd only display strace output with specified system call name in the next system call. To display output with these options only once, we utilized PID. More details on it is in section 3.4 (Option runs only once). To reset flags after one use, we created a new system call resetflags. Post that, strace would run normally. Screenshots are given below that describe all the changed files. We have highlighted the part related code.

```
  struct inode *cwd;           /
  char name[32];               /
  int trace_on;
  int cur_pid;
  struct{
    int e;
    char syscall[20];
    int s;
    int f;
    int o;
  }flags;
};
```

proc.h

```
462  int
463  sys_handleflags(void)
464  {
465    int e;
466    char *syscall_name;
467
468
469    if (argint(0, &e) < 0 || argstr(1, &syscall_name) < 0)
470      return -1;
471    proc->flags.e = e;
472    strncpy(proc->flags.syscall, syscall_name, sizeof(proc->flags.syscall) - 1);
473
474    return 1;
475  }
476
```

sysfile.c – Added -e handler

```
if (proc->flags.e) {
  if(proc->pid == proc->cur_pid + 1){
    next_trace=1;
  }
  else if (proc->pid == proc->cur_pid + 2){
    //cprintf("resetting flags");
    resetflags();
  }
  //cprintf("inside main E handler\n");
  if(proc->flags.s && next_trace)
  {
    //cprintf("inside s handler\n");
    if(strncmp(proc->flags.syscall, syscall_names[num], sizeof(proc->flags.syscall))==0 && ret!=-1){
      cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->
      add_event(proc->pid, proc->name, syscall_names[num], ret);
    }
  }
  else if(proc->flags.f && next_trace){
    //cprintf("inside F handler\n");
    if(strncmp(proc->flags.syscall, syscall_names[num], sizeof(proc->flags.syscall))==0 && ret==-1){
      cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->
      add_event(proc->pid, proc->name, syscall_names[num], ret);
    }
  }
  else if(strncmp(proc->flags.syscall, syscall_names[num], sizeof(proc->flags.syscall))==0){
    if(next_trace){
      //cprintf("INside onyl E handler\n");
      cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->
      add_event(proc->pid, proc->name, syscall_names[num], ret);
    }
  }
```

syscall.c – Added -e handler

```
else if (argv[1][0] == '-'){
  if(argv[1][1] == 'e' && argc >= 3){
    cur = gettrace();
    e=1;
    if (cur) {
      strcpy(specified_syscall, argv[2]);
      handleflags(e, specified_syscall);
    }
  }
  //else if - add other options here
  else {
    printf(1, "Invalid usage!\nSet strace on first and then use strace -e <sysc
    resetflags();
  }
}
else {
```

strace.c – Added -e option

```
$ strace -e exec
$ echo hello
TRACE: pid = 8 | process name = echo | syscall = exec | return = 0
$ strace -e open
TRACE: pid = 9 | process name = strace | syscall = exec | return = 0
$ cat tmp
TRACE: pid = 10 | process name = cat | syscall = open | return = -1
$
```

```
$ strace on
$ echo hello
TRACE: pid = 4 | process name = echo | syscall = exec | return = 0
TRACE: pid = 4 | process name = echo | syscall = write | return = 1
TRACE: pid = 4 | process name = echo | syscall = write | return = 1
TRACE: pid = 4 | process name = echo | syscall = write | return = 1
TRACE: pid = 4 | process name = echo | syscall = write | return = 1
TRACE: pid = 4 | process name = echo | syscall = write | return = 1
TRACE: pid = 4 | process name = echo | syscall = write | return = 1
$ strace -e write
TRACE: pid = 5 | process name = strace | syscall = exec | return = 0
TRACE: pid = 5 | process name = strace | syscall = gettrace | return = 1
$ echo hello
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
$ echo hello
TRACE: pid = 7 | process name = echo | syscall = exec | return = 0
TRACE: pid = 7 | process name = echo | syscall = write | return = 1
TRACE: pid = 7 | process name = echo | syscall = write | return = 1
TRACE: pid = 7 | process name = echo | syscall = write | return = 1
TRACE: pid = 7 | process name = echo | syscall = write | return = 1
TRACE: pid = 7 | process name = echo | syscall = write | return = 1
TRACE: pid = 7 | process name = echo | syscall = write | return = 1
$
```

output

3.2 Option -s

The option should only display successful system calls' strace output for the next command typed. Taking inspiration from 3.1 implementation, we introduced a new flag in proc.h (int s) and modified handleflags() to support this additional flag. Additionally, in syscall() we implemented a check to print strace output if return value is not -1. To implement this only on the next call, we implemented the same PID comparison method as in part 3.1.

```
  int cur_pid;
  struct{
    int e;
    char syscall[20];
    int s;
    int f;
    int o;
  }flags;
};
```

proc.h

```
int
sys_handleflags(void)
{
  int e;
  char *syscall_name;
  int s;
  int f;
  int o;

  if (argint(0, &e) < 0 || argstr(1, &syscall_name) < 0 || argint(2, &s) < 0 ||
    return -1;
  proc->flags.e = e;
  strncpy(proc->flags.syscall, syscall_name, sizeof(proc->flags.syscall) - 1);
  proc->flags.s = s; // Set the s flag
  proc->flags.f = f;
  proc->flags.o = o;
  proc->cur_pid = proc->pid;
  return 1;
}
```

sysfile.c – Added s flag

```
else if(proc->flags.s){
    if(proc->pid == proc->cur_pid + 1){
      next_trace=1;
    }
    else if (proc->pid == proc->cur_pid + 2){
      //cprintf("resetting flags");
      resetflags();
    }
    if(next_trace){
      if(ret != -1){
        //cprintf("INside S handler\n");
        cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\
        add_event(proc->pid, proc->name, syscall_names[num], ret);
      }
    }
  }
```

syscall.c – Added -s handler

```
TRACE: pid = 4 | process name = sh | syscall = write | return = 1
$ strace -s
TRACE: pid = 5 | process name = strace | syscall = exec | return = 0
TRACE: pid = 5 | process name = strace | syscall = gettrace | return = 1
TRACE: pid = 5 | process name = strace | syscall = handleflags | return = 1
$ abcd
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
$ abcd
TRACE: pid = 7 | process name = sh | syscall = exec | return = -1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
TRACE: pid = 7 | process name = sh | syscall = write | return = 1
```

output

3.3 Option -f

Implementation of -f is the same as 3.2 except we print strace output if return value is -1. We added one more variable in proc.h and modified handleflags system call to include f option too. Finally we added another option in strace.c for -f. We can see in the output that our implementation is working.

```
} else if (proc->flags.s && ret != -1) {
  proc->flags.cs++;
  cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->name, syscall_names[num], ret);
  add_event(proc->pid, proc->name, syscall_names[num], ret);
} else if (proc->flags.f && ret == -1) {
  proc->flags.cf++;
  cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->name, syscall_names[num], ret);
  add_event(proc->pid, proc->name, syscall_names[num], ret);
} else if (!proc->flags.e && !proc->flags.s && !proc->flags.f) {
```

syscall.c – Added -f condition

```
  int cur_pid;
  struct{
    int e;
    char syscall[20];
    int s;
    int f;
    int o;
  }flags;
};
```

proc.h

```
else if (proc->flags.f) {
  if(proc->pid == proc->cur_pid + 1){
    next_trace=1;
  }
  else if (proc->pid == proc->cur_pid + 2){
    //cprintf("resetting flags");
    resetflags();
  }
  if(next_trace && ret == -1){
    //cprintf("INside F handler\n");
    cprintf("TRACE: pid = %d | process name = %s | syscall = %s | retur
    add_event(proc->pid, proc->name, syscall_names[num], ret);
  }
}
```

sysfile.c – Added -f handler

```
init: starting sh
$ strace on
$ strace -f
TRACE: pid = 4 | process name = strace | syscall = exec | return = 0
TRACE: pid = 4 | process name = strace | syscall = gettrace | return = 1
$ xyz
TRACE: pid = 5 | process name = sh | syscall = exec | return = -1
$ xyz
TRACE: pid = 6 | process name = sh | syscall = exec | return = -1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
TRACE: pid = 6 | process name = sh | syscall = write | return = 1
```

output

3.4 Option runs only once

By this time we realized we needed some extra piece of code to make our option run only once. This was tricky since a simple call like echo hello, comprises of multiple system calls (7 calls). And there is no saying how many calls each and every command would run. So we couldn't set up a simple counter. We resorted to a simpler approach. Every time handleflags is called to set flags, we would store the current process ID into a variable called cur_pid. Then we'll use a flag next_trace in syscall, which will only be active if process id is cur_pid +1. Then the option specific logic would run. Furthermore, if PID becomes cur_pid +2, we'd resetflags and everything reverts. This helped us in our implementation. There is a drawback however, user can't press enter after a flag command, let's say -e. Pressing enter would nullify our entire logic. Here is a small example of our implementation code in the case of -f.

```
else if (proc->flags.f) {
  if(proc->pid == proc->cur_pid + 1){
    next_trace=1;
  }
  else if (proc->pid == proc->cur_pid + 2){
    //cprintf("resetting flags");
    resetflags();
  }
  if(next_trace && ret == -1){
    //cprintf("INside F handler\n");
    cprintf("TRACE: pid = %d | process name = %s | syscall =
    add_event(proc->pid, proc->name, syscall_names[num], ret
  }
}
```

syscall – run only once

```
sys_handleflags(void)
{
  int e;
  char *syscall_name;
  int s;
  int f;
  int o;

  if (argint(0, &e) < 0 || argstr(1, &syscall_name) < 0
    return -1;
  proc->flags.e = e;
  strncpy(proc->flags.syscall, syscall_name, sizeof(proc-
  proc->flags.s = s; // Set the s flag
  proc->flags.f = f;
  proc->flags.o = o;
  proc->cur_pid = proc->pid;
  return 1;
}
```

sysfile.c – run only once

3.5 Combining options

We observed that -s and -f are to be run in conjunction with -e, so we modified the existing -e handler in syscall to accommodate multiple options. We also modified the user level strace program to handle multiple options. But only -s or -f with -e else it would give an error and exit. Also, order should always be -s/f and then -e. Our existing handleflags() syscall was able to handle multiple flags too. Here, 1=e handler; 2=e and s both; 3= e and f both, 4 = just e.

```
if(proc->trace_on && num != SYS_trace){
  if (proc->flags.e) {                                                    1
    if(proc->pid == proc->cur_pid + 1){
      next_trace=1;
    }
    else if (proc->pid == proc->cur_pid + 2){
      //cprintf("resetting flags");
      resetflags();
    }
    //cprintf("inside main E handler\n");
    if(proc->flags.s && next_trace)                                       2
    {
      //cprintf("inside s handler\n");
      if(strncmp(proc->flags.syscall, syscall_names[num], sizeof(proc->flags.syscall))==0 && ret!=-1){
        cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->name,
        add_event(proc->pid, proc->name, syscall_names[num], ret);
      }
    }
    else if(proc->flags.f && next_trace){                                 3
      //cprintf("inside F handler\n");
      if(strncmp(proc->flags.syscall, syscall_names[num], sizeof(proc->flags.syscall))==0 && ret==-1){
        cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->name,
        add_event(proc->pid, proc->name, syscall_names[num], ret);
      }
    }
    else if(strncmp(proc->flags.syscall, syscall_names[num], sizeof(proc->flags.syscall))==0){   4
      if(next_trace){
        //cprintf("INside onyl E handler\n");
        cprintf("TRACE: pid = %d | process name = %s | syscall = %s | return = %d\n", proc->pid, proc->name,
        add_event(proc->pid, proc->name, syscall_names[num], ret);
      }
    }
  }
```

syscall.c – combining options

```
else if (argv[1][0] == '-'){
  if(argv[1][1] == 'e' && argc == 3){
    e=1;
    strcpy(specified_syscall, argv[2]);
    handleflags(e, specified_syscall,0,0,0);
  }
  else if(argv[1][1]=='s' && argv[2][0]=='-' && argv[2][1]=='e'){
    e=1;
    s=1;
    strcpy(specified_syscall, argv[3]);
    handleflags(e,specified_syscall,s,0,0);
  }
  else if(argv[1][1]=='f' && argv[2][0]=='-' && argv[2][1]=='e'){
    f=1;
    e=1;
    strcpy(specified_syscall, argv[3]);
    handleflags(e,specified_syscall,0,f,0);
  }
```

strace.c – combining options

```
$ strace on
$ strace -s -e exec
TRACE: pid = 4 | process name = sh | syscall = sbrk | return = 16384
TRACE: pid = 4 | process name = strace | syscall = exec | return = 0
TRACE: pid = 4 | process name = strace | syscall = sbrk | return = 12288
$ cat tmp
TRACE: pid = 5 | process name = cat | syscall = exec | return = 0
$ cat tmp
TRACE: pid = 6 | process name = cat | syscall = exec | return = 0
TRACE: pid = 6 | process name = cat | syscall = open | return = -1
TRACE: pid = 6 | process name = cat | syscall = write | return = 1
TRACE: pid = 6 | process name = cat | syscall = write | return = 1
TRACE: pid = 6 | process name = cat | syscall = write | return = 1
TRACE: pid = 6 | process name = cat | syscall = write | return = 1
TRACE: pid = 6 | process name = cat | syscall = write | return = 1
TRACE: pid = 6 | process name = cat | syscall = write | return = 1
TRACE: pid = 6 | process name = cat | syscall = write | return = 1
```

output – s & e

```
TRACE: pid = 11 | process name = ls | syscall = read | return = 0
TRACE: pid = 11 | process name = ls | syscall = close | return = 0
$ strace -f -e open
TRACE: pid = 12 | process name = sh | syscall = sbrk | return = 16384
TRACE: pid = 12 | process name = strace | syscall = exec | return = 0
TRACE: pid = 12 | process name = strace | syscall = sbrk | return = 12288
$ cat tmp
TRACE: pid = 13 | process name = cat | syscall = open | return = -1
$ cat tmp
TRACE: pid = 14 | process name = cat | syscall = exec | return = 0
TRACE: pid = 14 | process name = cat | syscall = open | return = -1
TRACE: pid = 14 | process name = cat | syscall = write | return = 1
TRACE: pid = 14 | process name = cat | syscall = write | return = 1
TRACE: pid = 14 | process name = cat | syscall = write | return = 1
TRACE: pid = 14 | process name = cat | syscall = write | return = 1
TRACE: pid = 14 | process name = cat | syscall = write | return = 1
TRACE: pid = 14 | process name = cat | syscall = write | return = 1
```

output – f & e

3.6 Writing output to a file

To implement writing output to a file, we implemented a simple functionality where a user could type in strace -o “filename” and all the previous strace information will be saved in a file. We utilized the same functionality as in dumptrace(). We passed an empty buffer (dynamically allocated heap) to kernel function and syscall writebuffer. We use the previously used linked list to write all strace information in LIFO order in this buffer we passed. Finally, on the user end, we wrote this buffer in the file specified by the user. A supporting itoa function was required to convert PID and return value in string to save to this buffer, but since it wasn’t available, we wrote our own integer_to_string() function in lmao.c and imported it in sysfile.c. Necessary screenshots are as follows.

```c
int
sys_writebuffer(void)
{
  char *buffer;
  if(argstr(0, &buffer) < 0){
    return -1;
  }
  struct event *current = event_list;
  cprintf("----- Writing to file -----\n");
  while (current) {
    char temp[MAX_TRACE_STR];
    char pid_str[MAX_INT_CHARS + 1];
    char ret_str[MAX_INT_CHARS + 1];

    // Convert int to string manually for pid and ret
    integer_to_string(pid_str, sizeof(pid_str), current->pid);
    integer_to_string(ret_str, sizeof(ret_str), current->ret);

    // Construct the formatted trace string manually
    safestrcpy(temp, "TRACE: pid = ", MAX_TRACE_STR);
    safestrcpy(temp + strlen(temp), pid_str, MAX_TRACE_STR - strlen(temp));
    safestrcpy(temp + strlen(temp), " | process name = ", MAX_TRACE_STR - strlen(temp));
    safestrcpy(temp + strlen(temp), current->name, MAX_TRACE_STR - strlen(temp));
    safestrcpy(temp + strlen(temp), " | syscall = ", MAX_TRACE_STR - strlen(temp));
    safestrcpy(temp + strlen(temp), current->syscall, MAX_TRACE_STR - strlen(temp));
    safestrcpy(temp + strlen(temp), " | return = ", MAX_TRACE_STR - strlen(temp));
    safestrcpy(temp + strlen(temp), ret_str, MAX_TRACE_STR - strlen(temp));
    safestrcpy(temp + strlen(temp), "\n", MAX_TRACE_STR - strlen(temp));

    // Concatenate temp to buffer
    safestrcpy(buffer + strlen(buffer), temp, MAXBUF - strlen(buffer));

    current = current->next;
  }
  cprintf("File save successful!\n--------------");
  return 0;
}
```

sysfile.c – writebuffer

```c
#define MAX_INT_CHARS 10  // max number of digits in an int
#define MAX_TRACE_STR 128  // max length of the formatted trace string
#define MAXBUF 25600

int integer_to_string(char *buf, int bufsize, int n) {
  char *start;
  if (n < 0) {
  if (!bufsize)
    return -1;
  *buf++ = '-';
  bufsize--;
  }
  start = buf;
  do {
    int digit;
    if (!bufsize)
      return -1;
    digit = n % 10;
    if (digit < 0)
      digit *= -1;
    *buf++ = digit + '0';
    bufsize--;
    n /= 10;
  } while (n);
  if (!bufsize)
    return -1;
  *buf = 0;
  --buf;
  while (start < buf) {
    char a = *start;
    *start = *buf;
    *buf = a;
    ++start;
    --buf;
  }
  return 0;
}
```

itoa locally implemented

```
  else if (argv[1][1] == 'o' && argc ==3){
    o=1;
    handleflags(0, "",0,0,o);
    if(writebuffer(buf) < 0){
      printf(2, "writebuffer syscall failed\n");
      exit();
    }

    //printf(1, "%s\n", buf); //Writing for now. If successful we'll save it
    int fd = open(argv[2], O_CREATE | O_WRONLY);
    if(fd < 0){
      printf(2, "Failed to open file\n");
      exit();
    }
    if(write(fd, buf, strlen(buf)) < 0){
      printf(2, "Failed to write to file\n");
      close(fd);
      exit();
    }
  close(fd);
  }
  else {
    printf(1, "Invalid usage!\n");
    resetflags();
    exit();
  }
}
```

strace.c – saving output

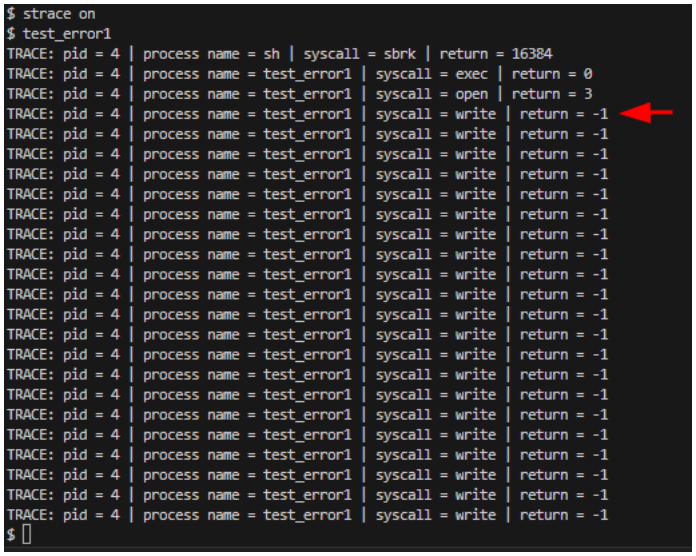
```
$ strace on
$ strace -f
TRACE: pid = 4 | process name = sh | syscall = sbrk | return = 16384
TRACE: pid = 4 | process name = strace | syscall = exec | return = 0
TRACE: pid = 4 | process name = strace | syscall = sbrk | return = 12288
$ cat tmp
TRACE: pid = 5 | process name = cat | syscall = open | return = -1
$ echo aaa
TRACE: pid = 6 | process name = echo | syscall = exec | return = 0
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
$ strace off
TRACE: pid = 7 | process name = sh | syscall = sbrk | return = 16384
TRACE: pid = 7 | process name = strace | syscall = exec | return = 0
TRACE: pid = 7 | process name = strace | syscall = sbrk | return = 12288
$ strace -o trace.txt
----- Writing to file -----
File save successful!
--------------------------
$ cat trace.txt
TRACE: pid = 7 | process name = strace | syscall = sbrk | return = 12288
TRACE: pid = 7 | process name = strace | syscall = exec | return = 0
TRACE: pid = 7 | process name = sh | syscall = sbrk | return = 16384
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = write | return = 1
TRACE: pid = 6 | process name = echo | syscall = exec | return = 0
TRACE: pid = 5 | process name = cat | syscall = open | return = -1
TRACE: pid = 4 | process name = strace | syscall = sbrk | return = 12288
TRACE: pid = 4 | process name = strace | syscall = exec | return = 0
TRACE: pid = 4 | process name = sh | syscall = sbrk | return = 16384
$
```

output

4. Application of strace

To understand the utility of our strace implementation, we simply created a C program which opened a pre-existing file README in read only mode and tried to write in it. Unexpected condition here is that a read only file can't be written into. We ran strace with this program to see what we could infer.

```
#include "types.h"
#include "user.h"
#include "fcntl.h"
int main(void){

    char *reading_file = "README";
    int fd = open(reading_file, O_CREATE|O_RDONLY); //opening file in read mode and then
writing
    printf(fd, "Could we write here?\n");
    exit();

}
```



output

Here, we can see that when xv6 tries to open a file in read only mode, it succeeded with a return value 3 (file descriptor). But when we tried to write in it, it gave a return value of -1. This told us that xv6 couldn't write into the file. We then created a similar program in high level C. When we ran the very same program in Linux (modified for linux: strace_error.c) we observed the following in 4.1 and 4.2.

```
#include <stdio.h>
int main(void){
       FILE * fp;
       fp = fopen ("README", "r");
       fprintf(fp, "Could we write here?\n");
       fclose(fp);
       return 0;
}
```

4.1 Running program normally in Linux

We see that when program ran, no information was given on the console, program exited normally. We read the file and there is nothing in there telling us that the program couldn't write in the file.

```
root@hex:~# ./strace_error
root@hex:~# cat README
root@hex:~# cat strace_error.c
#include <stdio.h>
int main(void){
        FILE * fp;
        fp = fopen ("README", "r");
        fprintf(fp, "Could we write here?\n");
        fclose(fp);
        return 0;
}
root@hex:~#
```

4.2 Running program with strace in Linux

```
root@hex:~# touch README
root@hex:~# strace ./strace_error
execve("./strace_error", ["./strace_error"], 0x7ffd2ae3fa10 /* 45 vars */) = 0
brk(NULL)                               = 0x55c886294000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd4ffcecb0) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f98202b2000
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
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=59059, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 59059, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f98202a3000
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\0P\237\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0i8\235HZ\227\223\333\350s\360\352,\223\340."..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2216304, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9820000000
mmap(0x7f9820028000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f9820028000
mmap(0x7f98201bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f98201bd000
mmap(0x7f9820215000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7f9820215000
mmap(0x7f982021b000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f982021b000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f98202a0000
arch_prctl(ARCH_SET_FS, 0x7f98202a0740) = 0
set_tid_address(0x7f98202a0a10)         = 3093
set_robust_list(0x7f98202a0a20, 24)     = 0
rseq(0x7f98202a10e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f9820215000, 16384, PROT_READ) = 0
mprotect(0x55c884c82000, 4096, PROT_READ) = 0
mprotect(0x7f98202ec000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f98202a3000, 59059)           = 0
getrandom("\x1d\xde\x2c\xac\xf0\x0b\x91\x29", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55c886294000
brk(0x55c8862b5000)                     = 0x55c8862b5000
openat(AT_FDCWD, "README", O_RDONLY)    = 3
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

From the Linux output above, we can clearly see that strace reported a bunch of data about system calls. "openat(AT_FDCWD, "README", O_RDONLY) = 3" shows that system tried to open a file in read mode. But it doesn't show any further information about it trying to write in a file. From this

information it is hard to infer what the system is doing, whereas, in our implementation we see clearly that strace reports -1 value when we try to write in a file opened in read mode.

Conclusion: In our implementation, it is easier to infer for a user with less OS knowledge to understand that the program is not performing write function properly. We clearly see a file being opened, then written into with value -1, meaning unsuccessful. However, in Linux strace, we see a bunch of information about memory addresses and system calls which, while it gives a better overall view of what the system is doing at the time we run this program, makes it a little harder to infer the expected behavior of this program. In conclusion, a layman would prefer our implementation for the simplicity but for a deeper knowledge of the kernel during debugging, Linux strace may be more useful to a user with greater OS knowledge.