Report

The trace of close()

We shall trace the close() syscall based on the user program. The first step in the process is that the user calls the "close" system call in the user program. The following steps represent the trace step by step what happens after that

1. User calls ***close(fd)***
   1.1. This will then invoke a function defined in ***usys.S***

```
17 SYSCALL(close)
```

   1.2. The following macro will then be executed

```
4 #define SYSCALL(name) \
5   .globl name; \
6   name: \
7     movl $SYS_ ## name, %eax; \
8     int $T_SYSCALL; \
9     ret
```

   1.3. Which will become

```
4 #define SYSCALL(name) \
5   .globl name; \
6   name: \
7     movl $SYS_close  %eax; \
8     int $T_SYSCALL; \
9     ret
```

   1.4. This value of SYS_close comes from the definition in ***syscall.h***

```
22 #define SYS_close  21
```

   1.5. Here, ***eax*** means that the value of SYS_close should be put in the ***eax*** register. Since 16 is the value of SYS_close, the ***eax*** register will have 16
   1.6. The value of **int** becomes 64 as the value of T_SYSCALL is 64 in ***traps.h***

```
27 #define T_SYSCALL       64      // system call
```

2. Trap function call
   2.1. The trap function in **trap.c** is called next

```
35 //PAGEBREAK: 41
36 void
37 trap(struct trapframe *tf)
38 {
39   if(tf->trapno == T_SYSCALL){
40     if(myproc()->killed)
41       exit();
42     myproc()->tf = tf;
43     syscall();
44     if(myproc()->killed)
45       exit();
46     return;
47   }
```

   2.2. Function trap(myproc()->tf), checks if the passed trapno is a valid system call. If the system call is valid, then it calls syscall()
3. Syscall function
   3.1. The syscall function is defined in the ***syscall.c*** file such as:

```
131 void
132 syscall(void)
133 {
134   int num;
135   struct proc *curproc = myproc();
136
137   num = curproc->tf->eax;
138   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num]();
140   } else {
141     cprintf("%d %s: unknown sys call %d\n",
142             curproc->pid, curproc->name, num);
143     curproc->tf->eax = -1;
144   }
145 }
```

   3.2. The function checks the value of the ***eax*** register. If the value is a valid number, then the function calls the function as defined in ***syscall.c*** on line 128

```
107 static int (*syscalls[])(void) = {
108 [SYS_fork]    sys_fork,
109 [SYS_exit]    sys_exit,
110 [SYS_wait]    sys_wait,
111 [SYS_pipe]    sys_pipe,
112 [SYS_read]    sys_read,
113 [SYS_kill]    sys_kill,
114 [SYS_exec]    sys_exec,
115 [SYS_fstat]   sys_fstat,
116 [SYS_chdir]   sys_chdir,
117 [SYS_dup]     sys_dup,
118 [SYS_getpid]  sys_getpid,
119 [SYS_sbrk]    sys_sbrk,
120 [SYS_sleep]   sys_sleep,
121 [SYS_uptime]  sys_uptime,
122 [SYS_open]    sys_open,
123 [SYS_write]   sys_write,
124 [SYS_mknod]   sys_mknod,
125 [SYS_unlink]  sys_unlink,
126 [SYS_link]    sys_link,
127 [SYS_mkdir]   sys_mkdir,
128 [SYS_close]   sys_close,
129 };
```

3.3. The sys_close function is defined in the ***sysfile.c*** file

```
 93 int
 94 sys_close(void)
 95 {
 96   int fd;
 97   struct file *f;
 98
 99   if(argfd(0, &fd, &f) < 0)
100     return -1;
101   myproc()->ofile[fd] = 0;
102   fileclose(f);
103   return 0;
104 }
```

4. Sys_close

4.1. The sys_close function calls argfd() as defined in ***sysfile.c***

```
19 // Fetch the nth word-sized system call argument as a file descriptor
20 // and return both the descriptor and the corresponding struct file.
21 static int
22 argfd(int n, int *pfd, struct file **pf)
23 {
24   int fd;
25   struct file *f;
26
27   if(argint(n, &fd) < 0)
28     return -1;
29   if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
30     return -1;
31   if(pfd)
32     *pfd = fd;
33   if(pf)
34     *pf = f;
35   return 0;
36 }
```

4.2. Here we check for a valid file descriptor. Since the user gave an invalid file descriptor, it returns **-1**. This value is passed up the stack of calls that the user ***close()*** called. This value is passed to disclose() which passes it to syscall() function.

4.3. The syscall function then branches into:

```
138   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num]();
140   } else {
141     cprintf("%d %s: unknown sys call %d\n",
142             curproc->pid, curproc->name, num);
143     curproc->tf->eax = -1;
144   }
```

4.4. Now that the ***eax*** register is set to **-1.**

5. Handle trap error

5.1. Since the ***eax*** register is set to -1, the ***ret*** instruction in the assembly code then returns the value of the ***eax*** register. Since the value is negative, it indicates that an error occurred, so the error is handled and returned to the user.

```
7     movl $SYS_close  %eax; \
8     int $T_SYSCALL; \
9     ret
```

5.2. The error will handle this error and show the user the appropriate message.