

Tracing for read()

The read function is defined as follows:-

```
read(int fildes, void *buf, size_t nbyte);
```

The *read()* function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Before any action described below is taken, and if *nbyte* is zero, the *read()* function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the *read()* function shall return zero and have no other results.

On files that support seeking (for example, a regular file), the *read()* shall start at a position in the file given by the file offset associated with *fildes*. The file offset shall be incremented by the number of bytes actually read.

Files that do not support seeking-for example, terminals-always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent *read()* requests is implementation-defined.

If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

When attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, *read()* shall return 0 to indicate end-of-file.
- If some process has the pipe open for writing and O_NONBLOCK is set, *read()* shall return -1 and set *errno* to [EAGAIN].
- If some process has the pipe open for writing and O_NONBLOCK is clear, *read()* shall block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If O_NONBLOCK is set, *read()* shall return -1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is clear, *read()* shall block the calling thread until some data becomes available.
- The use of the O_NONBLOCK flag has no effect if there is some data available.

When we take the example for any command and start the process for *read()* functionality it go through various files and to make the command successful. Let's take the example for cat command

Implementation and working, here is cat.c file where it define the functionality for cat command: -

```
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 char buf[512];
6
7 void
8 cat(int fd)
9 {
10     int n;
11     while((n = read(fd, buf, sizeof(buf))) > 0)
12         write(1, buf, n);
13     if(n < 0){
14         printf(1, "cat: read error\n");
15         exit();
16     }
17 }
18
19 int
20 main(int argc, char *argv[])
21 {
22     int fd, i;
23     if(argc <= 1){
24         cat(0);
25         exit();
26     }
27     for(i = 1; i < argc; i++){
28         if((fd = open(argv[i], 0)) < 0){
29             printf(1, "cat: cannot open %s\n", argv[i]);
30             exit();
31         }
32         cat(fd);
33         close(fd);
34     }
35     exit();
36 }
```

when it start cat command it will read argc and check the command for its validity, if it is valid command it will call cat function with sending 'fd' as argument, which will be used in the read function as file descriptor and it will run read() till n>0, now read function go to its definition which lied in usys.S file. where it simply call SYSCALL(name) assembly code and it will define global variable, in our case it will be read and it will move SYS_read into eax register, then it will generate intrupt T_SYSCALL.

```
22 8: 83 ec 04      sub    $0x4,%esp
23 b: ff 75 f4      pushl  -0xc(%ebp)
24 e: 68 80 0b 00 00 pushl  $0xb80
25 13: 6a 01          pushl  $0x1
26 15: e8 6c 03 00 00 call    386 <write>
27 1a: 83 c4 10      add    $0x10,%esp
28 void
29 cat(int fd)
30 {
31     int n;
32     while((n = read(fd, buf, sizeof(buf))) > 0)
33         write(1, buf, n);
34     if(n < 0){
35         printf(1, "cat: read error\n");
36         exit();
37     }
38 }
```



```
1 #include "syscall.h"
2 #include "traps.h"
3
4 #define SYSCALL(name) \
5     .globl name; \
6     name: \
7     movl $SYS_## name, %eax; \
8     int $T_SYSCALL; \
9     ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
17 SYSCALL(close)
18 SYSCALL(kill)
19 SYSCALL(exec)
20 SYSCALL(open)
21 SYSCALL(mknod)
22 SYSCALL(unlink)
23 SYSCALL(fstat)
24 SYSCALL(link)
25 SYSCALL(mkdir)
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(cgps)
33
```

after calling the read function in cat.c>cat.asm> this will call system call read() through usys.S where we have defined SYS_read where we have defined a number 5 to it so that usys.S file can transfer value 5 to eax register, after that it will go to int(interrupt) t_syscall and assign(64). After that it will move to trapasm.s and run till it call trap and move to another file i.e. trap.c. it will call syscall() and move to file.c

```

trapasm.S
#include "mmu.h"

# vectors.S sends all traps here.
.globl alltraps
alltraps:
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal

# Set up data and per-cpu segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
movw $(SEG_KCPU<<3), %ax
movw %ax, %fs
movw %ax, %gs

# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp

# Return falls through to trapret...
.globl trapret
trapret:
popal
popl %gs
popl %fs
popl %es
popl %ds
addl $0x8, %esp # trapno and errcode
iret

```



```

trap.c
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(proc->killed)
            exit();
        proc->tf = tf;
        syscall();
        if(proc->killed)
            exit();
        return;
    }

    switch(tf->trapno){
        case T_IRQ0 + IRQ_TIMER:
            if(cpu->id == 0){
                acquire(&tickslock);
                ticks++;
                wakeup(&ticks);
                release(&tickslock);
            }
            lapiceoi();
            break;
        case T_IRQ0 + IRQ_IDE:
            ideintr();
            lapiceoi();
            break;
        case T_IRQ0 + IRQ_IDE+1:
            // Bochs generates spurious IDE1 interrupts.
            break;
        case T_IRQ0 + IRQ_KBD:
            kbdintr();
            lapiceoi();
            break;
        case T_IRQ0 + IRQ_COM1:
            uartintr();
            lapiceoi();
            break;
        case T_IRQ0 + 7:
        case T_IRQ0 + IRQ_SPURIOUS:
            cprintf("cpu%d: spurious interrupt at %x\n",

```

```

Find View Goto Tools Project Preferences Help
syscall.c
120 [SYS_write] sys_write,
121 [SYS_mknod] sys_mknod,
122 [SYS_unlink] sys_unlink,
123 [SYS_link] sys_link,
124 [SYS_mkdir] sys_mkdir,
125 [SYS_close] sys_close,
126 };
127
128 void
129 syscall(void)
130 {
131     int num;
132
133     num = proc->tf->eax;
134     if(num > 0 && num < NELEM(syscalls) && syscalls[num]){
135         proc->tf->eax = syscalls[num]();
136     } else {
137         cprintf("%d %s: unknown sys call %d\n",
138             proc->pid, proc->name, num);
139         proc->tf->eax = -1;
140     }
141 }
142

```

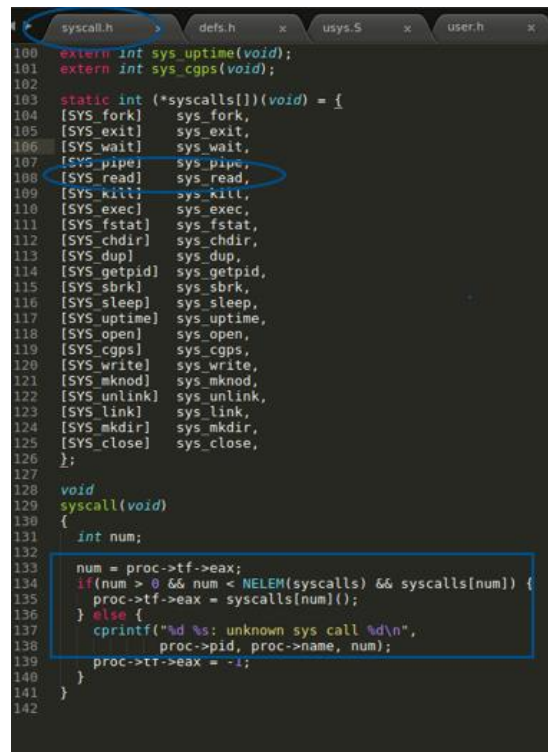
Where num(eax)>syscall trap 64>tvinitt(64vector)(in the initialization time)>return

```

4 // Read from file f.
5 int
6 fileread(struct file *f, char *addr, int n)
7 {
8     int r;
9
10    if(f->readable == 0)
11        return -1;
12    if(f->type == FD_PIPE)
13        return piperead(f->pipe, addr, n);
14    if(f->type == FD_INODE){
15        ilock(f->ip);
16        if((r = readi(f->ip, addr, f->off, n)) > 0)
17            f->off += r;
18        iunlock(f->ip);
19        return r;
20    }
21    panic("fileread");
22 }
23

```

After that it will come back to `usys.S` saved in `eax` (son of `tf`) which is equal to `(-1)>trap.asm>iRET`(final exit door of `syscall`)>`int64` completion and finally it will return from `syscall.c`

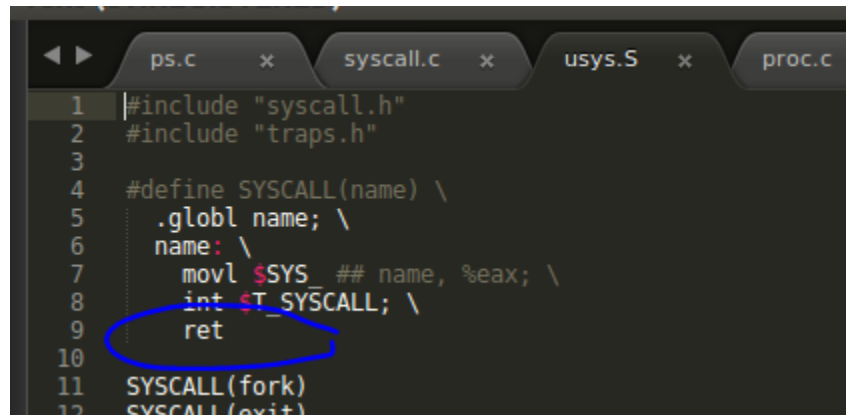


```

100 extern int sys_uptime(void);
101 extern int sys_cgps(void);
102
103 static int (*syscalls[])(void) = {
104     [SYS_fork]    sys_fork,
105     [SYS_exit]    sys_exit,
106     [SYS_wait]    sys_wait,
107     [SYS_pipe]    sys_pipe,
108     [SYS_read]    sys_read,
109     [SYS_kill]    sys_kill,
110     [SYS_exec]    sys_exec,
111     [SYS_fstat]   sys_fstat,
112     [SYS_chdir]   sys_chdir,
113     [SYS_dup]     sys_dup,
114     [SYS_getpid]  sys_getpid,
115     [SYS_sbrk]    sys_sbrk,
116     [SYS_sleep]   sys_sleep,
117     [SYS_uptime]  sys_uptime,
118     [SYS_open]    sys_open,
119     [SYS_cgps]    sys_cgps,
120     [SYS_write]   sys_write,
121     [SYS_mknod]   sys_mknod,
122     [SYS_unlink]  sys_unlink,
123     [SYS_link]    sys_link,
124     [SYS_mkdir]   sys_mkdir,
125     [SYS_close]   sys_close,
126 };
127
128 void
129 syscall(void)
130 {
131     int num;
132
133     num = proc->tf->eax;
134     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
135         proc->tf->eax = syscalls[num]();
136     } else {
137         cprintf("%d %s: unknown sys call %d\n",
138             proc->pid, proc->name, num);
139         proc->tf->eax = -1;
140     }
141 }
142

```

When it comes to syscall.c file and check for syscall function , it will push SYS_READ to eax register (-1) and return. And gives an error.



```
1 #include "syscall.h"
2 #include "traps.h"
3
4 #define SYSCALL(name) \
5     .globl name; \
6     name: \
7     movl $SYS_ ## name, %eax; \
8     int $T_SYSCALL; \
9     ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
```