

SW E3004 Operating Systems, Fall 2023

Project 2. System call

TA)

Hyeonmyeong Lee

Gwanjong Park

Jongseok Kim

ShinHyun Park

Project plan

- Total 6 projects
 - ~~1) Booting xv6 operating system~~
 - 2) **System call**
 - 3) CPU scheduling
 - 4) Virtual memory
 - 5) Page replacement
 - 6) File systems

Project 2: Make system calls

- Make system calls in xv6 kernel
 - getnice
 - setnice
 - ps

Project 2: Make system calls

- Goal : make new three system calls(`getnice`, `setnice`, `ps`)
- Synopsis
 - `int getnice(int pid);`
 - `int setnice(int pid, int value);`
 - `void ps(int pid);`
- Description
 - The **`getnice`** function obtains the nice value of a process.
 - The **`setnice`** function sets the nice value of a process.
 - The **default nice value is 20**. Lower nice values cause more favorable scheduling.
 - It will be necessary to implement the nice value before creating the system call.
 - **The range of valid nice value is 0~39**

Project 2: Make system calls

▪ Description (cont'd)

- In kernel, the **ps** system call prints out process(s)'s information, which includes name, pid, state and priority(nice value) of each process.
- If the pid is 0, print out all processes' information.
- Otherwise, print out corresponding process's information.
- If there is no process corresponding to the pid, print out nothing.

name	pid	state	priority
init	1	SLEEPING	20
sh	2	SLEEPING	20
ps	3	RUNNING	20

▪ Return value

- **getnice** : Return the nice value of target process on success. Return -1 if there is no process corresponding to the pid.
- **setnice** : Return 0 on success. Return -1 if there is no process corresponding to the pid or the nice value is invalid.
- **ps** : No return value.

How to add system call (getpname)

1. Add your syscall to usys.S

```
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(getpname)
```

2. Add syscall number to syscall.h

```
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_getpname 22
```

3. Add extern and syscall element in syscall.c

```
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_getpname(void);
```

```
[SYS_link] sys_link,
[SYS_mkdir] sys_mkdir,
[SYS_close] sys_close,
[SYS_getpname] sys_getpname,
};
```

How to add system call (getpname)

4. Add a sys_function to sysproc.c

```
int
sys_getpname(void)
{
    int pid;

    if(argint(0, &pid) < 0)
        return -1;
    return getpname(pid);
}
```

5. Add a function that performs a real action to proc.c

```
int
getpname(int pid)
{
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            cprintf("%s\n", p->name);
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

How to add system call (getpname)

6. Add a definition to defs.h and user.h

```
void      userinit(void);  
int       wait(void);  
void      wakeup(void*);  
void      yield(void);  
int       getpname(int);
```

```
char* sbrk(int);  
int sleep(int);  
int uptime(void);  
int getpname(int);
```


How to test your system call

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

int main()
{
    int i;
    for (i=1; i<11; i++) {
        printf(1, "%d: ", i);
        if (getpname(i))
            printf(1, "Wrong pid\n");
    }
    exit();
}
```

mytest.c

“mytest.c” is an example code.
Create and use your own test code.

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _mytest\
```

Makefile

Test with user program

```
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive fi
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 13648
echo       2 4 12656
forktest   2 5 8092
grep       2 6 15524
init       2 7 13240
kill       2 8 12708
ln         2 9 12604
ls         2 10 14792
mkdir      2 11 12788
rm         2 12 12764
sh         2 13 23252
stressfs   2 14 13436
usertests  2 15 56364
wc         2 16 14184
zombie     2 17 12428
mytest     2 18 12612
console    3 19 0
$ mytest
1: init
2: sh
3: Wrong pid
4: mytest
5: Wrong pid
6: Wrong pid
7: Wrong pid
8: Wrong pid
9: Wrong pid
10: Wrong pid
$
```

Submission

- This project is to implement only the system calls (getnice, setnice, ps)
 - The user program for testing is irrelevant.
- Use the **submit & check-submission** binary file in Ji Server
 - \$ make clean
 - \$ ~swe3004/bin/submit pa2 xv6-public
 - You can submit several times, and the submission history can be checked through check-submission
 - Only the last submission will be graded

Submission

- PLEASE DO NOT COPY
 - We will run inspection program on all the submissions
 - Any unannounced penalty can be given to **both students**
 - 0 points / negative points / F grade ...
- Due date: 9/27(Wed.), 23:59:59 PM
 - -**25%** per day for delayed submission

Questions

- If you have questions, please ask on i-campus
 - Please use the discussion board
 - Discussion board preferred over messages
- You can also visit Corporate Collaboration Center #85533
 - Please iCampus message TA before visiting
- Reading xv6 commentary will help you a lot
 - <http://csl.skku.edu/uploads/SSE3044S20/book-rev11.pdf>

Appendix. Trap Handling Process on xv6

- Example : **kill** system call

```
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int
6 main(int argc, char **argv)
7 {
8     int i;
9
10    if(argc < 2){
11        printf(2, "usage: kill pid...\n");
12        exit();
13    }
14    for(i=1; i<argc; i++){
15        kill(atoi(argv[i]));
16    }
17    exit();
18 }
```

kill.c (user level)

```
1 struct stat;
2 struct rtcdate;
3
4 // system calls
5 int fork(void);
6 int exit(void) __attribute__((noreturn));
7 int wait(void);
8 int pipe(int*);
9 int write(int, const void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(const char*, int);
15 int mknod(const char*, short, short);
16 int unlink(const char*);
```

user.h

_kill(user program)'s Build Process

```
ld -m elf_i386 -N -e main -Ttext 0 -o _cat cat.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _echo echo.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o
ld -m elf_i386 -N -e main -Ttext 0 -o _grep grep.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _init init.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _kill kill.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _ln ln.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _ls ls.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _mkdir mkdir.o ulib.o usys.o printf.o umalloc.o
ld -m elf_i386 -N -e main -Ttext 0 -o _rm rm.o ulib.o usys.o printf.o umalloc.o
```

make qemu-nox | grep usys

Functions defined as assembly

```
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)
```

usys.S

```
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
```

syscall.h

```
25 // These are arbitrarily chosen
26 // processor defined exception numbers
27 #define T_SYSCALL 64
28 #define T_DEFAULT 500
```

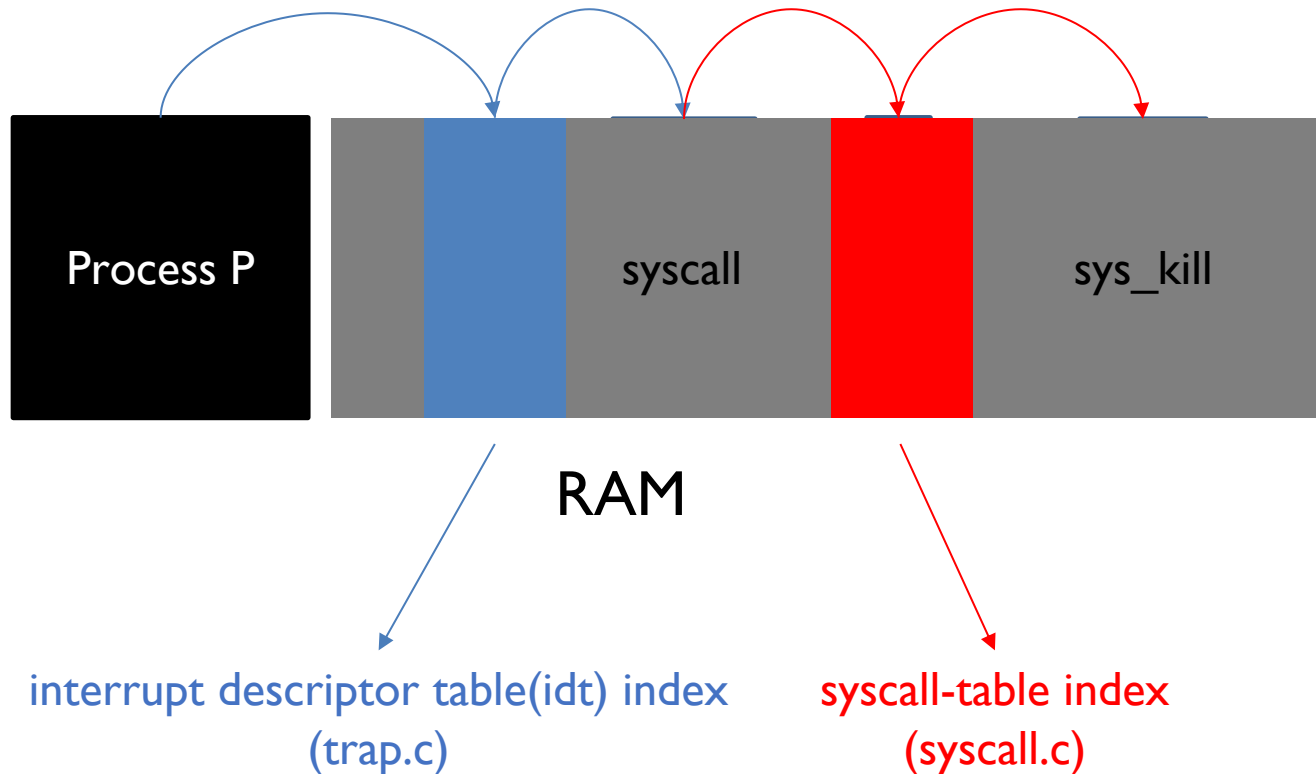
traps.h

After all, what the `_kill` user program calls is three instructions.

Trap Handling Process on xv6

movl **\$6**, \$eax

int **\$64** → INT 64 instruction performs the 64th vector Defined in the **interrupt descriptor table(idt)**.



Interrupt Descriptor Table (IDT)

main.c

```
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    picinit(); // disable pic
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
    ninit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    ideinit(); // disk
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    userinit(); // first user process
    mpmain(); // finish this processor's setup
}
```

https://wiki.osdev.org/Interrupt_Descriptor_Table

IDT entry, Interrupt Gates			
Name	Bit	Full Name	Description
Offset	48..63	Offset 16..31	Higher part of the offset.
P	47	Present	Set to 0 for unused interrupts.
DPL	45..46	Descriptor Privilege Level	Gate call protection. Specifies which privilege level the calling descriptor minimum should have. So hardware and CPU interrupts can be protected from being called out of userspace.
S	44	Storage Segment	Set to 0 for interrupt and trap gates (see below).
Type	40..43	Gate Type 0..3	Possible IDT gate types:
			0b0101 0x5 5 80386 32-bit task gate
			0b0110 0x6 6 80286 16-bit interrupt gate
			0b0111 0x7 7 80286 16-bit trap gate
			0b1110 0xE 14 80386 32-bit interrupt gate
			0b1111 0xF 15 80386 32-bit trap gate
0	32..39	Unused 0..7	Have to be 0.
Selector	16..31	Selector 0..15	Selector of the interrupt function (to make sense - the kernel's selector). The selector's descriptor's DPL field has to be 0 so the iret instruction won't throw a #GP exception when executed.
Offset	0..15	Offset 0..15	Lower part of the interrupt function's offset address (also known as pointer).

In xv6, idt is set in the form shown in the Intel architecture manual.

trap.c

```
17 void
18 tvinit(void)
19 {
20     int i;
21
22     for(i = 0; i < 256; i++)
23         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
24     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
25
26     initlock(&tickslock, "time");
27 }
```

mmu.h

```
// Gate descriptors for interrupts and traps
struct gatedesc {
    uint off_15_0 : 16; // low 16 bits of offset in segment
    uint cs : 16; // code segment selector
    uint args : 5; // # args, 0 for interrupt/trap gates
    uint rsv1 : 3; // reserved (should be zero I guess)
    uint type : 4; // type(STS_{IG32,TG32})
    uint s : 1; // must be 0 (system)
    uint dpl : 2; // descriptor (meaning new) privilege level
    uint p : 1; // Present
    uint off_31_16 : 16; // high bits of offset in segment
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//       the privilege level required for software to invoke
//       this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)
{
    (gate).off_15_0 = (uint)(off) & 0xffff;
    (gate).cs = (sel);
    (gate).args = 0;
    (gate).rsv1 = 0;
    (gate).type = (istrap) ? STS_TG32 : STS_IG32;
    (gate).s = 0;
    (gate).dpl = (d);
    (gate).p = 1;
    (gate).off_31_16 = (uint)(off) >> 16;
}
```

Vector

As a result, int 64 calls vector64, and vector64 executes alltraps.

```
316  jmp alltraps
317  .globl vector64
318  vector64:
319  pushl $0
320  pushl $64
321  jmp alltraps
322  .globl vector65
323  vector65:
```

vectors.S

```
5  alltraps:
6  # Build trap frame.
7  pushl %ds
8  pushl %es
9  pushl %fs
10 pushl %gs
11 pushal
12
13 # Set up data segments.
14 movw $(SEG_KDATA<<3), %ax
15 movw %ax, %ds
16 movw %ax, %es
17
18 # Call trap(tf), where tf=%esp
19 pushl %esp
20 call trap
21 addl $4, %esp
22
23 # Return falls through to trapret
```

trapasm.S

Trap

```
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     }
```

trap.c

By vector 64:
trapno = 64

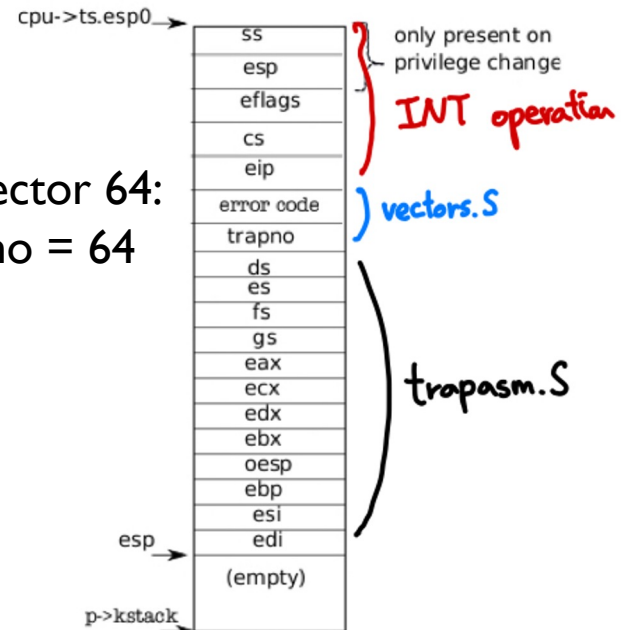


Figure 2-2. The trapframe on the kernel stack

Syscall

```
139 void
140 syscall(void)
141 {
142     int num;
143     struct proc *curproc = myproc();
144
145     num = curproc->tf->eax;
146     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
147         curproc->tf->eax = syscalls[num]();
148     } else {
149         cprintf("%d %s: unknown sys call %d\n",
150             curproc->pid, curproc->name, num);
151         curproc->tf->eax = -1;
152     }
153 }
```

syscall.c

movl \$6, %eax
int \$64

```
111 static int (*syscalls[])(void) = {
112     [SYS_fork]    sys_fork,
113     [SYS_exit]    sys_exit,
114     [SYS_wait]    sys_wait,
115     [SYS_pipe]    sys_pipe,
116     [SYS_read]    sys_read,
117     [SYS_kill]    sys_kill,
118     [SYS_exec]    sys_exec,
119     [SYS_fstat]   sys_fstat,
```

Kill

```
29 int
30 sys_kill(void)
31 {
32     int pid;
33
34     if(argint(0, &pid) < 0)
35         return -1;
36     return kill(pid);
37 }
```

sysproc.c

```
479 int
480 kill(int pid)
481 {
482     struct proc *p;
483
484     acquire(&ptable.lock);
485     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
486         if(p->pid == pid){
487             p->killed = 1;
488             // Wake process from sleep if necessary.
489             if(p->state == SLEEPING)
490                 p->state = RUNNABLE;
491             release(&ptable.lock);
492             return 0;
493         }
494     }
495     release(&ptable.lock);
496     return -1;
497 }
```

proc.c