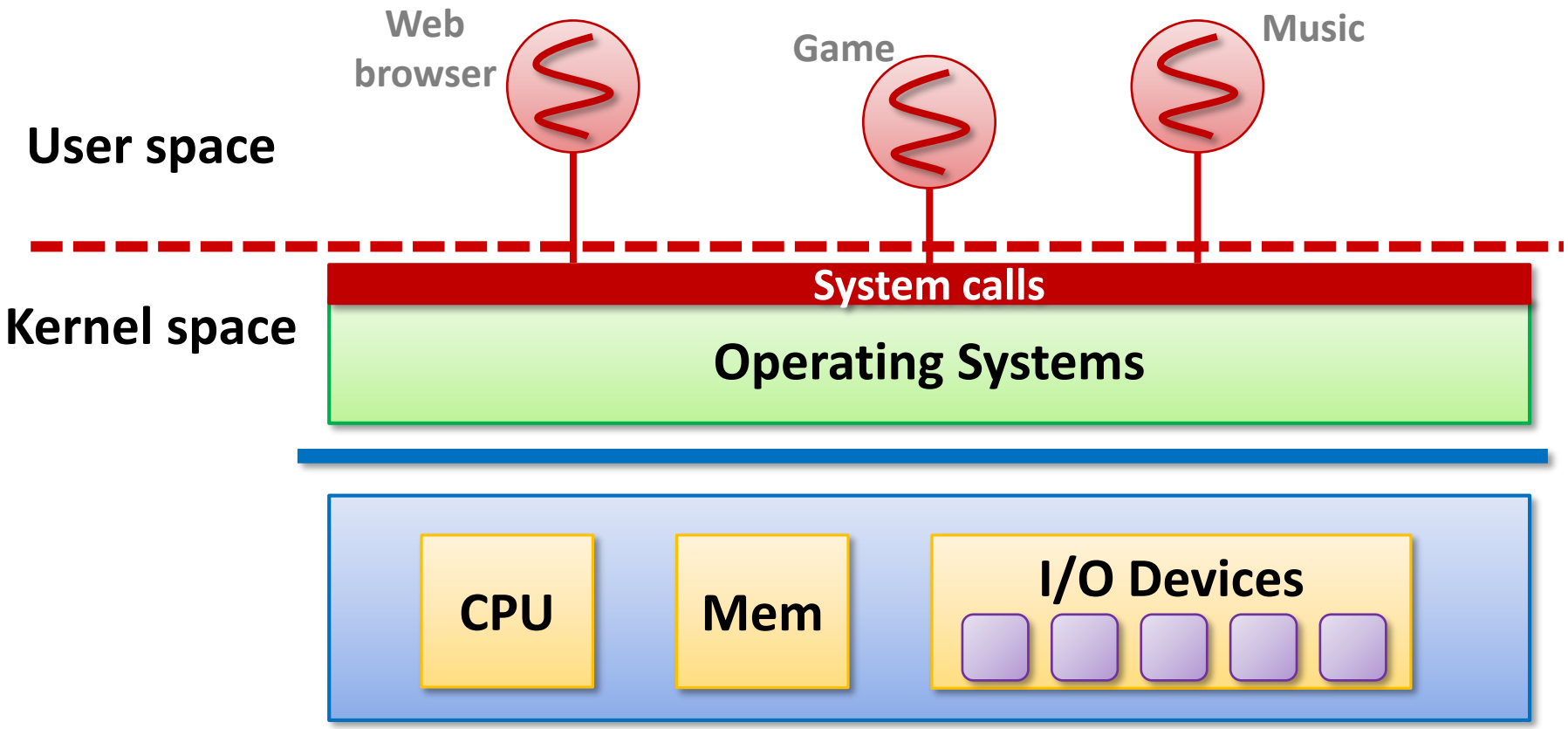
MISYS

Operating Systems Practice

2: System calls

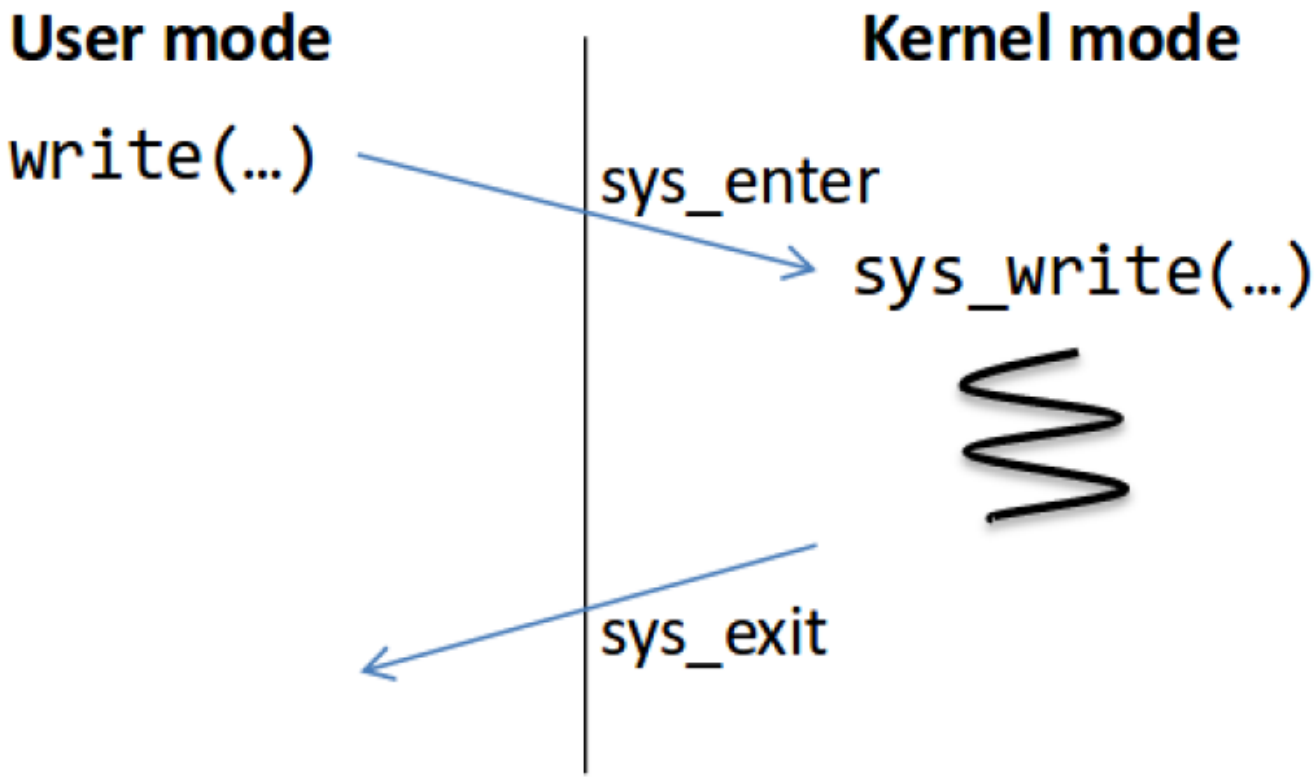
Kilho Lee

Operating system



System Call

- An interface for accessing the kernel from user space



Trap Handling Process

- Intel architecture

Trap Handling Process (Cont'd)

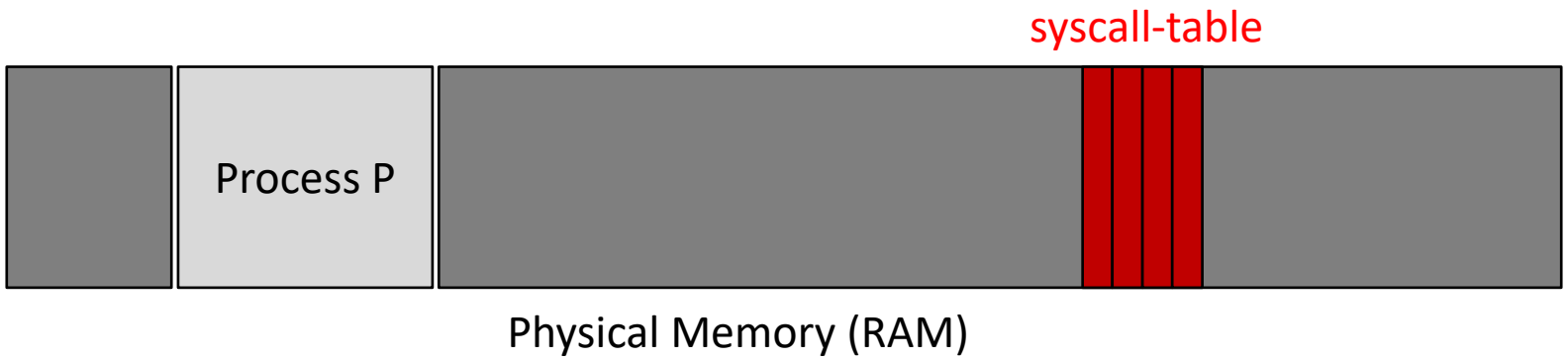
- Process P can only see its own memory because of user mode (other areas, including kernel, are hidden)

Trap Handling Process (Cont'd)

- Process P wants to call `kill()` system call

Trap Handling Process (Cont'd)

```
static int(*syscalls[])(void)      (syscall.c)
```

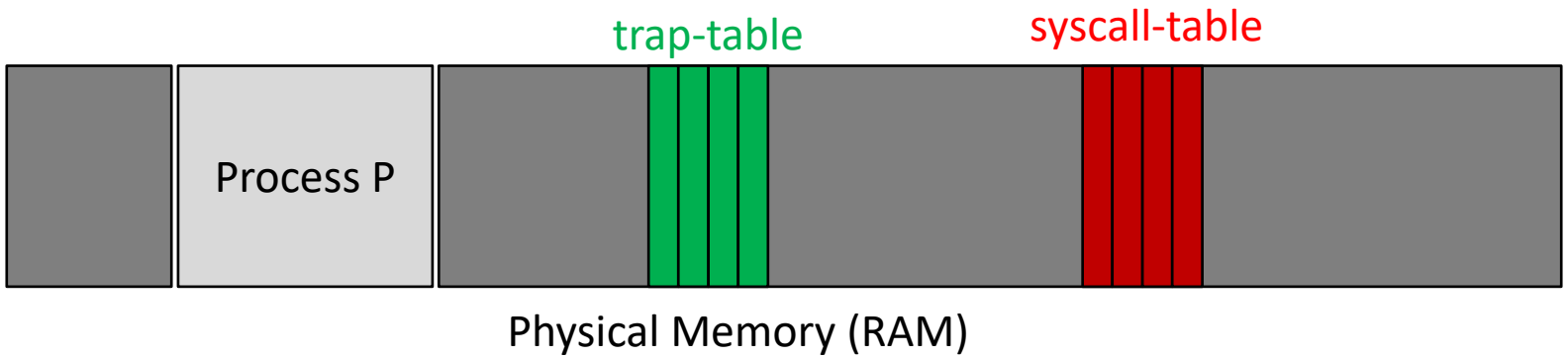



```
movl $6, %eax;        int $64
```

syscall-table index

Trap Handling Process (Cont'd)

```
struct gatedesc idt[256]    (trap.c)
```



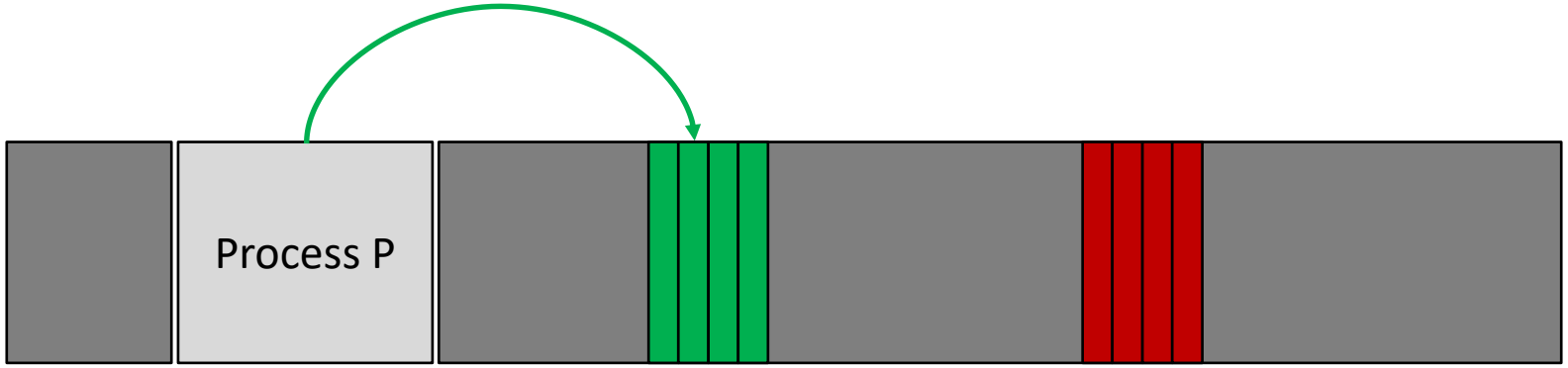

```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



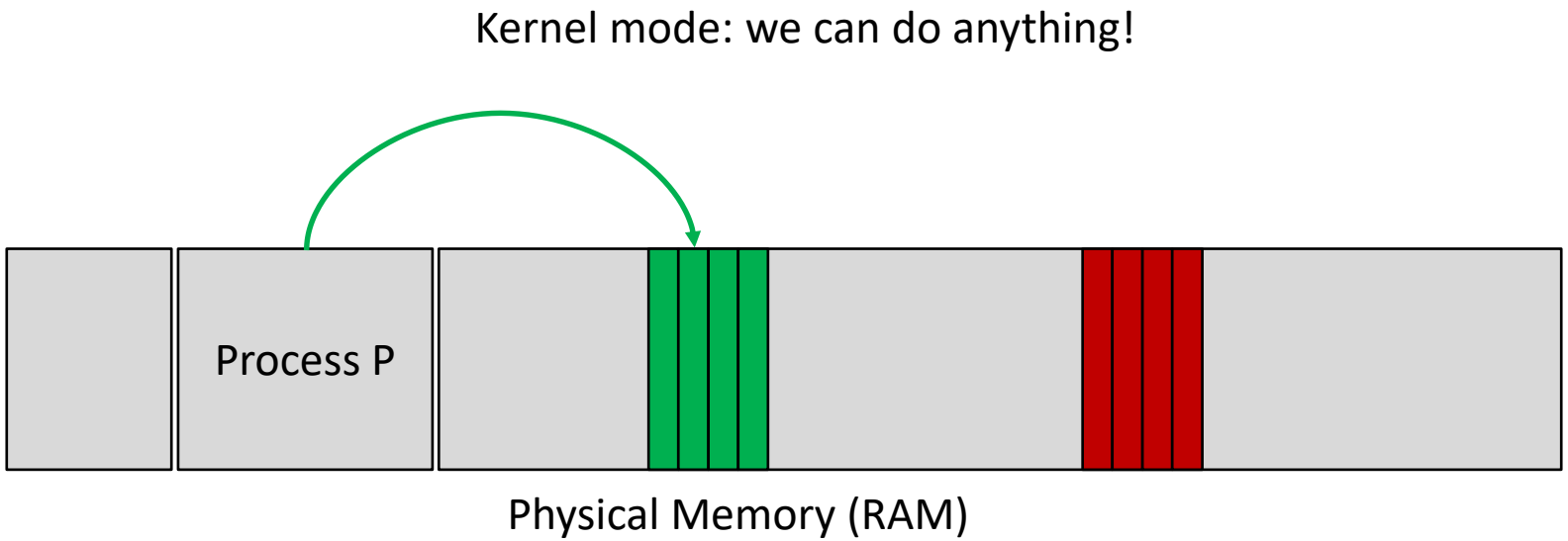

```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



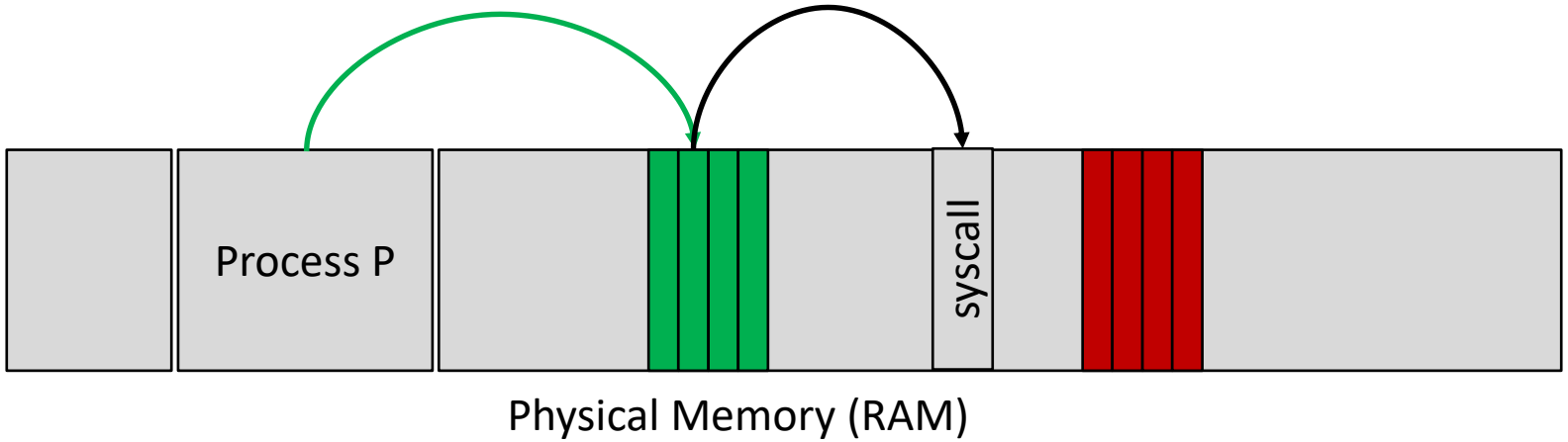

```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



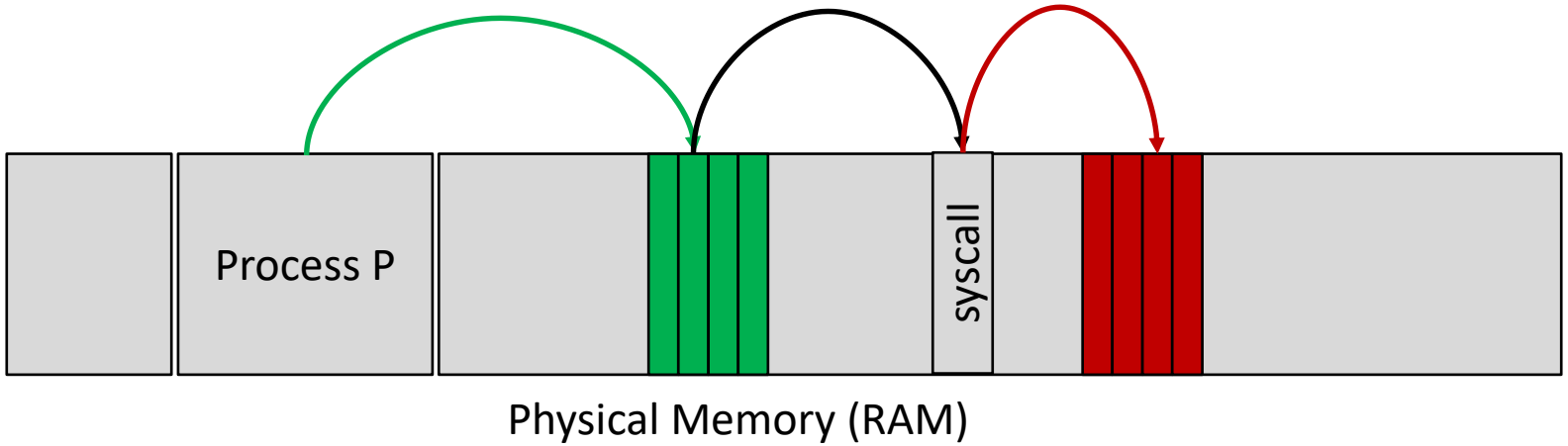

```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



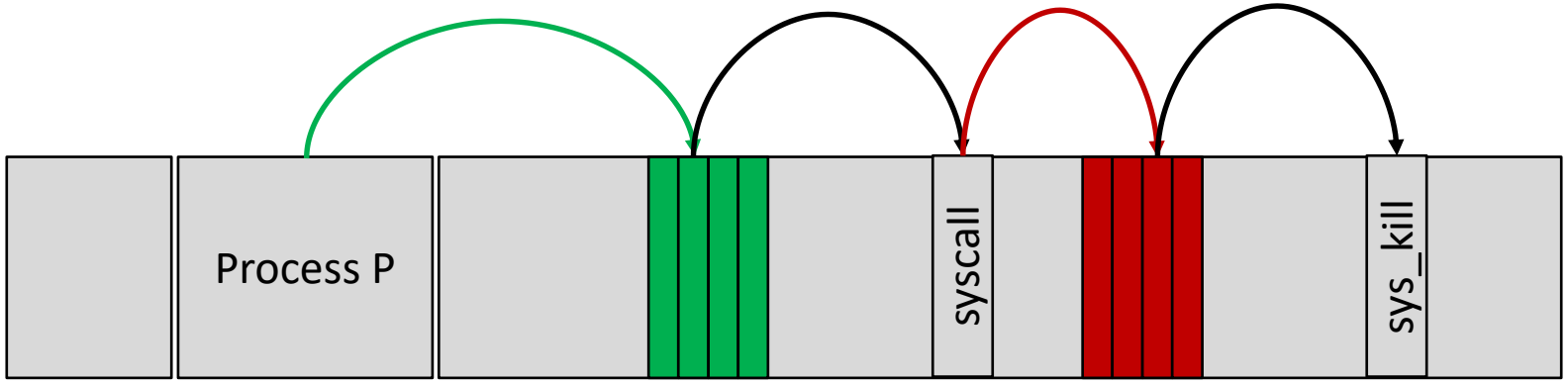

```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)




```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process on Xv6

- `user.h`

```
 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, void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(char*, int);
15 int mknod(char*, short, short);
16 int unlink(char*);
17 int fstat(int fd, struct stat*);
18 int link(char*, char*);
19 int mkdir(char*);
20 int chdir(char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
```

Trap Handling Process on Xv6

- usys.S

```
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)
```

```
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
```

```
.globl kill;
kill:
    movl $6, %eax;
    int $64;
    ret
```

Trap Handling Process on Xv6

- syscall.h
  - system call number

```
1 // System call numbers
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
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
10 #define SYS_chdir   9
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
14 #define SYS_sleep  13
15 #define SYS_uptime 14
16 #define SYS_open   15
17 #define SYS_write  16
18 #define SYS_mknod  17
19 #define SYS_unlink 18
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
```

Trap Handling Process on Xv6

- traps.h
  - trap number

eunji — ejlee@ejlee-lecture: ~/os20s_lab/xv6-ssu/xv6_ss

```
// x86 trap and interrupt constants.

// Processor-defined:
#define T_DIVIDE         0      // divide error
#define T_DEBUG          1      // debug exception
#define T_NMI            2      // non-maskable interrupt
#define T_BRKPT          3      // breakpoint
#define T_OFLOW          4      // overflow
#define T_BOUND          5      // bounds check
#define T_ILLOP          6      // illegal opcode
#define T_DEVICE         7      // device not available
#define T_DBLFLT         8      // double fault
// #define T_COPROC      9      // reserved (not used since 486)
#define T_TSS           10      // invalid task switch segment
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
// #define T_RES        15      // reserved
#define T_FPERR         16      // floating point error
#define T_ALIGN         17      // aligment check
#define T_MCHK          18      // machine check
#define T_SIMDERR       19      // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64      // system call
#define T_DEFAULT      500      // catchall

#define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ

#define IRQ_TIMER        0
#define IRQ_KBD          1
#define IRQ_COM1         4
#define IRQ_IDE         14
#define IRQ_ERROR       19
#define IRQ_SPURIOUS    31

```

Trap Handling Process on Xv6

- `trap.c`
  - Interrupt Descriptor Table initialization

```
// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;

void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

Trap Handling Process on Xv6

- vectors.S

```
# generated by vectors.pl - do not edit
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
  jmp alltraps
.globl vector2
vector2:
  pushl $0
  pushl $2
  jmp alltraps
```

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

```
# vector table
.data
.globl vectors
vectors:
  .long vector0
  .long vector1
  .long vector2
  .long vector3
  .long vector4
  .long vector5
  .long vector6
  .long vector7
  .long vector8
  .long vector9
  .long vector10
```

Trap Handling Process on Xv6

- `trapasm.S`

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

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

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

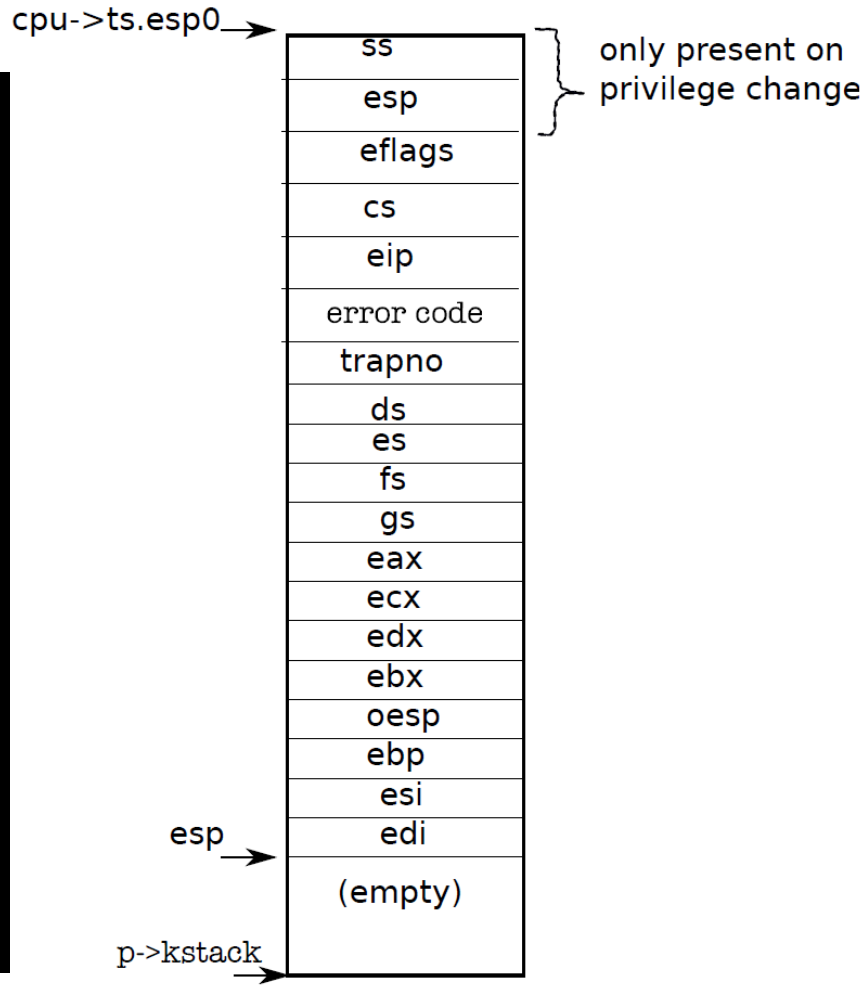


Figure 3-2. The trapframe on the kernel stack

Trap Handling Process on Xv6

- `trap.c`

```
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   }
```

```
150 struct trapframe {
151   // registers as pushed
152   uint edi;
153   uint esi;
154   uint ebp;
155   uint oesp;      // usel
156   uint ebx;
157   uint edx;
158   uint ecx;
159   uint eax;
160
161   // rest of trap frame
162   ushort gs;
163   ushort padding1;
164   ushort fs;
165   ushort padding2;
166   ushort es;
167   ushort padding3;
168   ushort ds;
169   ushort padding4;
170   uint trapno;
```

x86.h

Trap Handling Process on Xv6

- `syscall.c`

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

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

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
```

```
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
```

```
.globl kill;
kill:
    movl $6, %eax;
    int $64;
    ret
```

Trap Handling Process on Xv6

- sysproc.c

```
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 }
```

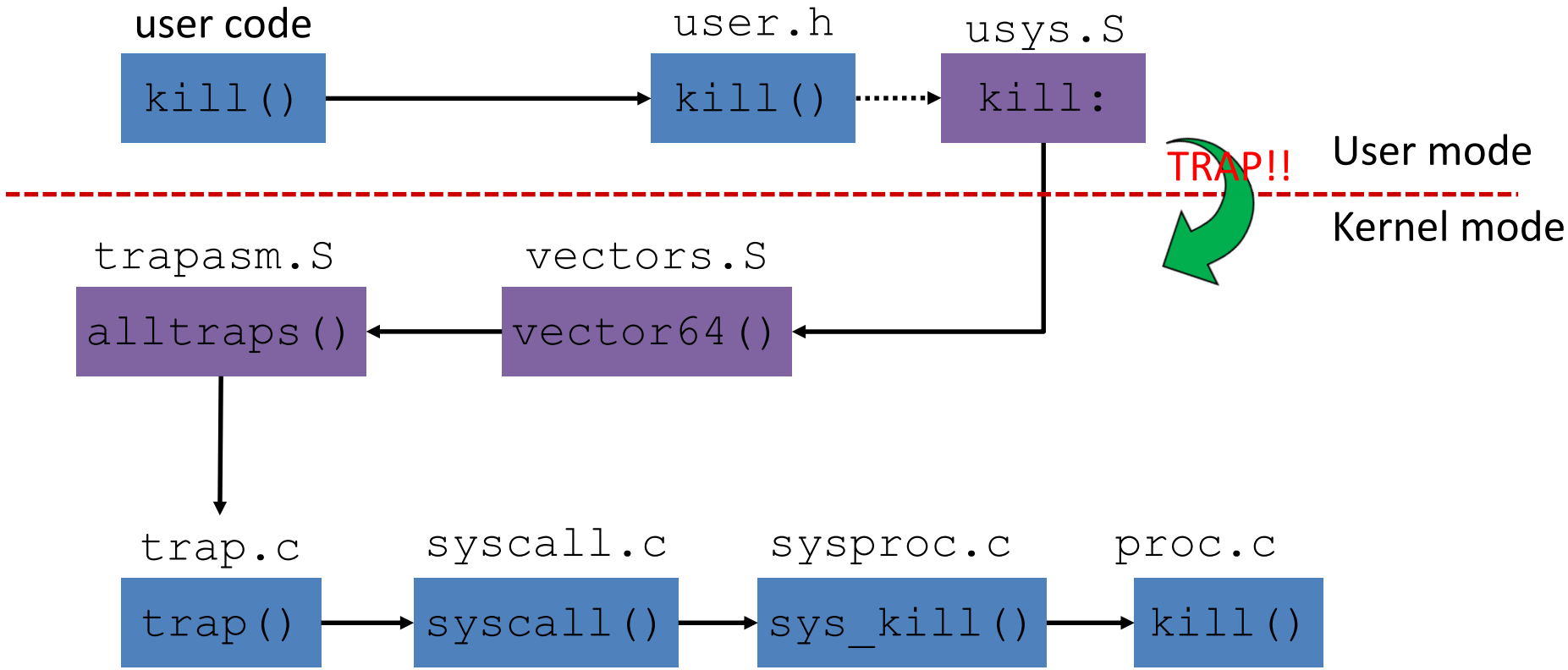
- proc.c

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

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
```

Trap Handling Process on Xv6

- `kill` system call



Test with User Program

- Example: `kill` system call
- `user/kill.c`

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

int
main(int argc, char **argv)
{
  int i;

  if(argc < 2){
    printf(2, "usage: kill pid...\n");
    exit();
  }
  for(i=1; i<argc; i++)
    kill(atoi(argv[i]));
  exit();
}
```

Test with User Program

- Makefile

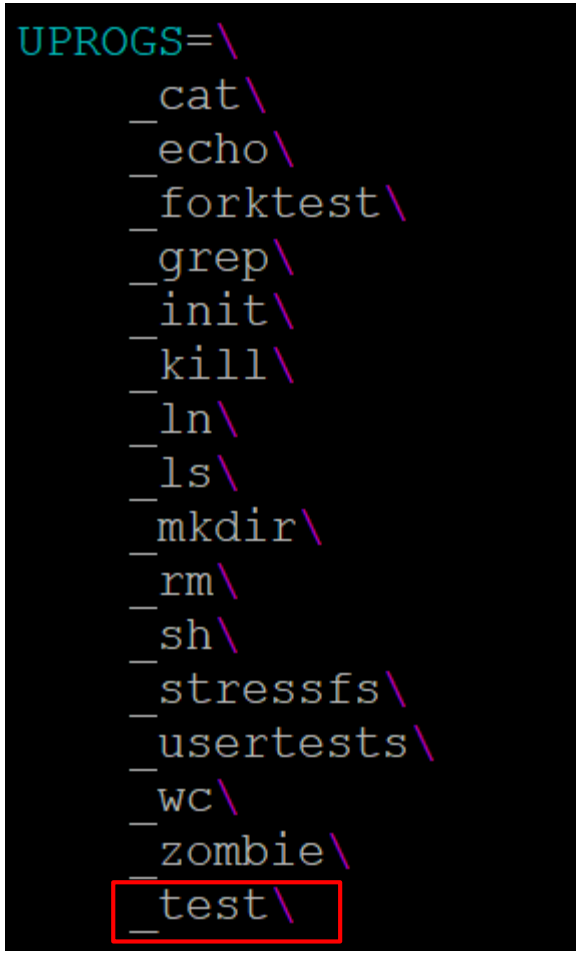
```
167 # http://www.gnu.org/software/make/manual/
168 .PRECIOUS: %.o
169
170 U=user
171 UPROGS=\
172     $U/_cat\
173     $U/_echo\
174     $U/_forktest\
175     $U/_grep\
176     $U/_init\
177     $U/_kill\
178     $U/_ln\
179     $U/_ls\
180     $U/_mkdir\
181     $U/_rm\
182     $U/_sh\
183     $U/_stressfs\
184     $U/_usertests\
185     $U/_wc\
186     $U/_zombie\
187     $U/_test_sys\
188
189 fs.img: mkfs README $(UPROGS)
190     ./mkfs fs.img README $(UPROGS)
191
```

- xv6

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 1973
cat            2 3 14000
echo           2 4 12961
forktest       2 5 8473
grep           2 6 15924
init           2 7 13862
kill           2 8 13093
ln             2 9 12995
ls             2 10 15859
mkdir          2 11 13126
rm             2 12 13103
sh             2 13 25923
stressfs       2 14 14081
usertests      2 15 68544
wc             2 16 14582
zombie         2 17 12727
console        3 18 0
$
```

(How to) Add user program

- Write your `.c` code and add it's name to "Makefile"
  - If you write `test.c` you have to add '`_test\`' to Makefile.
  - Then, you can execute '`test`' program on xv6 after booting it



References

- Xv6-books
  - https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev10.pdf
- Contents
  - **Ch.0: Operating system interface**
  - Ch.1: Operating system organization
  - Ch.2: Page tables
  - Ch.3: Traps, interrupts, and drivers
  - Ch.4: Locking
  - Ch.5: Scheduling
  - Ch.6: File system
  - Ch.7: Summary
  - Appendix A: PC hardware
  - Appendix B: The boot loader

P1. System call Implementation

- Implement your own system calls as follows:
  - getnice
  - setnice
  - ps

setnice / getnice

- A system call that sets/gets a priority of process by using nice value (0~30)
- `int` **`setnice`**`(int pid, int nice)`
  - Set the process priority
  - Argument: process ID, nice value
  - Return value: failure -1 / success 0

- `int` **`getnice`**`(int pid)`
  - Get the process priority
  - Argument: process ID
  - Return value: failure -1 / success process' nice value

ps

- A system call that prints out the process information
  - pid
  - state: RUNNING, RUNNABLE, SLEEPING
  - priority (nice value)
  - runtime (as of now, keep it 0 for future use)

```
$ ps
name     pid     state           priority        runtime         tick 146
init     1       SLEEPING        20              2
sh       2       SLEEPING        20              0
ps       3       RUNNING         20              0
```

Test 1

- user/test_nice.c

  - Get nice of init process (must be 5)
  - Get nice of non-existing process (wrong pid) – should return -1
  - Set nice to current process
  - Set nice to non-existing process (wrong pid)
    – should return -1
  - Set wrong nice(<0 or >30) on current process
    – should return -1
  - Get nice of forked process
    – Forked process inherits parent process priority

Test 1

- user/test_nice.c
  - You should get “OK” messages for all test cases.

```c
void test_nice()
{
    int pid, nice;

    printf(1, "case 1. get nice value of init process: ");
        if (getnice(1) == 5)
            printf(1, "OK\n");
        else
            printf(1, "WRONG\n");

        printf(1, "case 2. get nice value of non-existing process: ");
        if (getnice(100) == -1)
            printf(1, "OK\n");
        else
            printf(1, "WRONG\n");

        printf(1, "case 3. set nice value of current process: ");
        pid = getpid();
        setnice(pid, 3);
        if (getnice(pid) == 3)
            printf(1, "OK\n");
        else
            printf(1, "WRONG\n");

        printf(1, "case 4. set nice value of non-existing process: ");
        if (setnice(100, 3) == -1)
            printf(1, "OK\n");
        else
            printf(1, "WRONG\n");

        printf(1, "case 5. set wrong nice value of current process: ");
        if (setnice(pid, -1) == -1 && setnice(pid, 11) == -1)
            printf(1, "OK\n");
        else
            printf(1, "WRONG\n");

        printf(1, "case 6. get nice value of forked process: ");
        nice = getnice(pid);

        pid = fork();
        if (pid == 0) { //child
            if (getnice(getpid()) == nice) {
                printf(1, "OK\n");
                exit();
            }
            else {
                printf(1, "WRONG\n");
                exit();
            }
        }
        else                        //parent
            wait();
}
```

Test 2

- user/ps.c

  • Comment out the 8th line (printf ()) and uncomment the 7th line (ps()), and then run the user program.

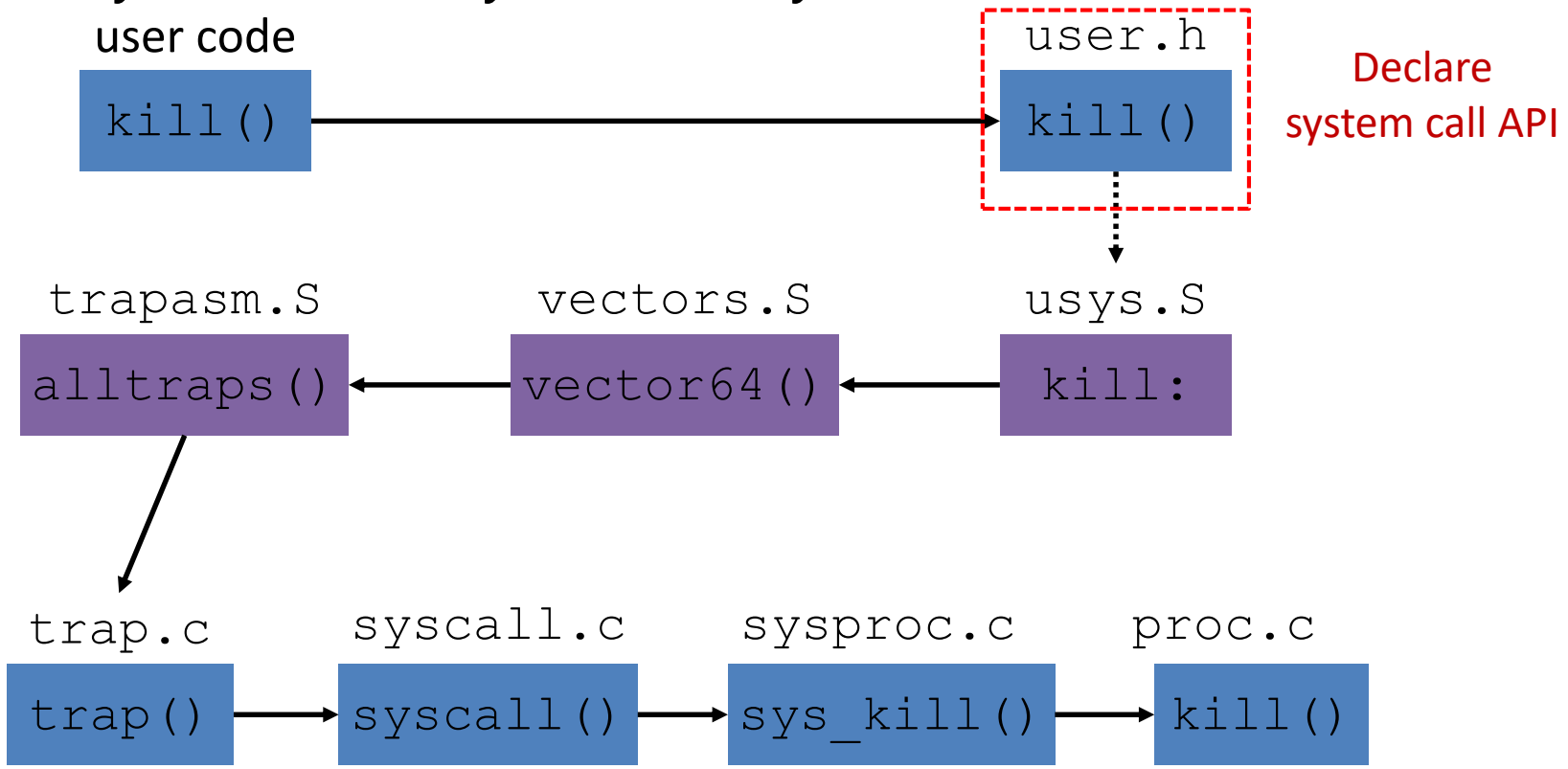
```
 1 #include "types.h"
 2 #include "stat.h"
 3 #include "user.h"
 4
 5 int main(int argc, char **argv)
 6 {
 7     //ps();
 8     printf(1, "ps\n");
 9     exit();
10 }
11
```

  • You should get the process information as follows (specific numbers may vary):

```
$ ps
name     pid     state           priority        runtime         tick 146
init     1       SLEEPING        20              2
sh       2       SLEEPING        20              0
ps       3       RUNNING         20              0
```

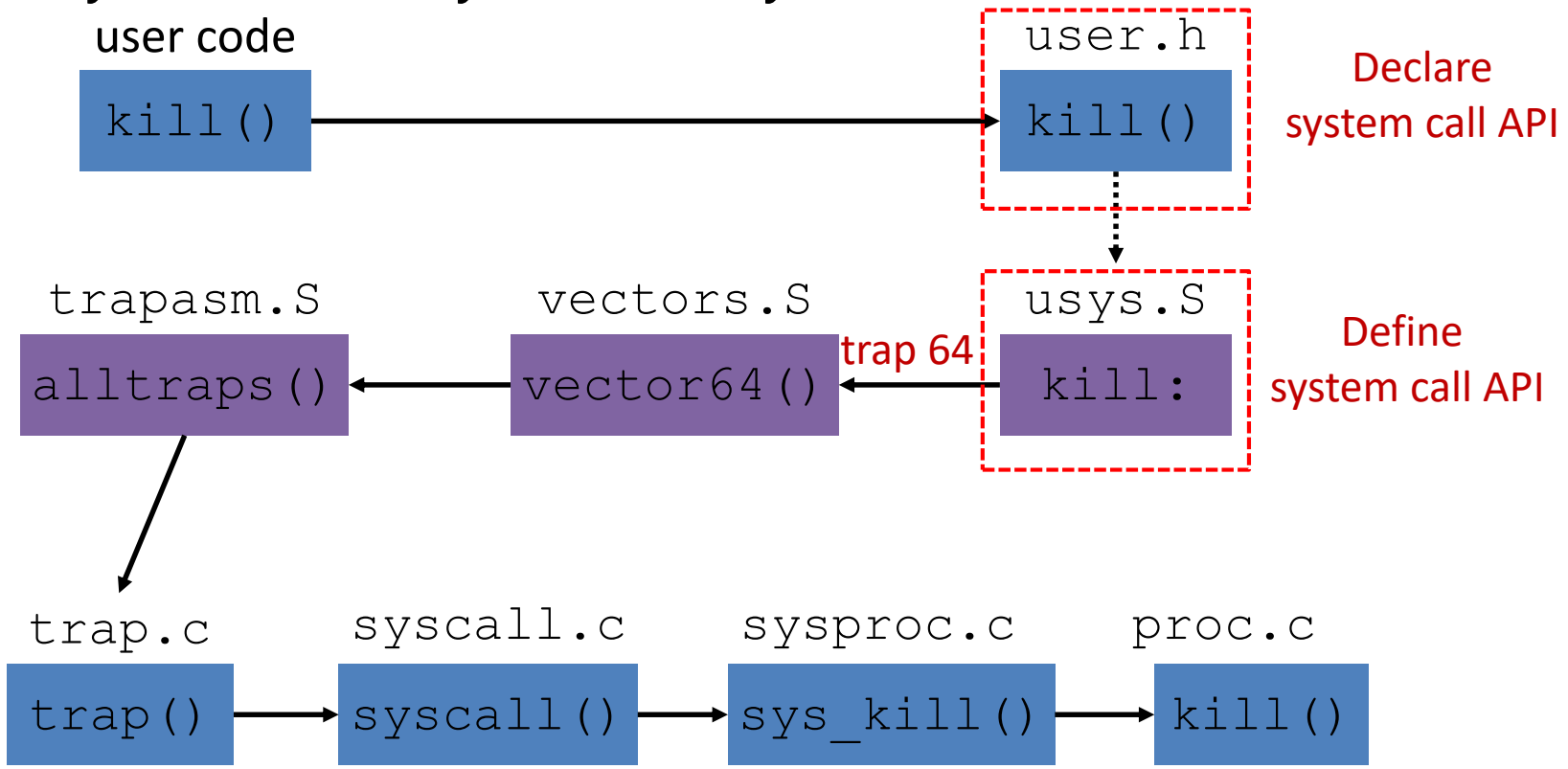
How to Make New System Call on Xv6

- Declare a library routine for your new system call



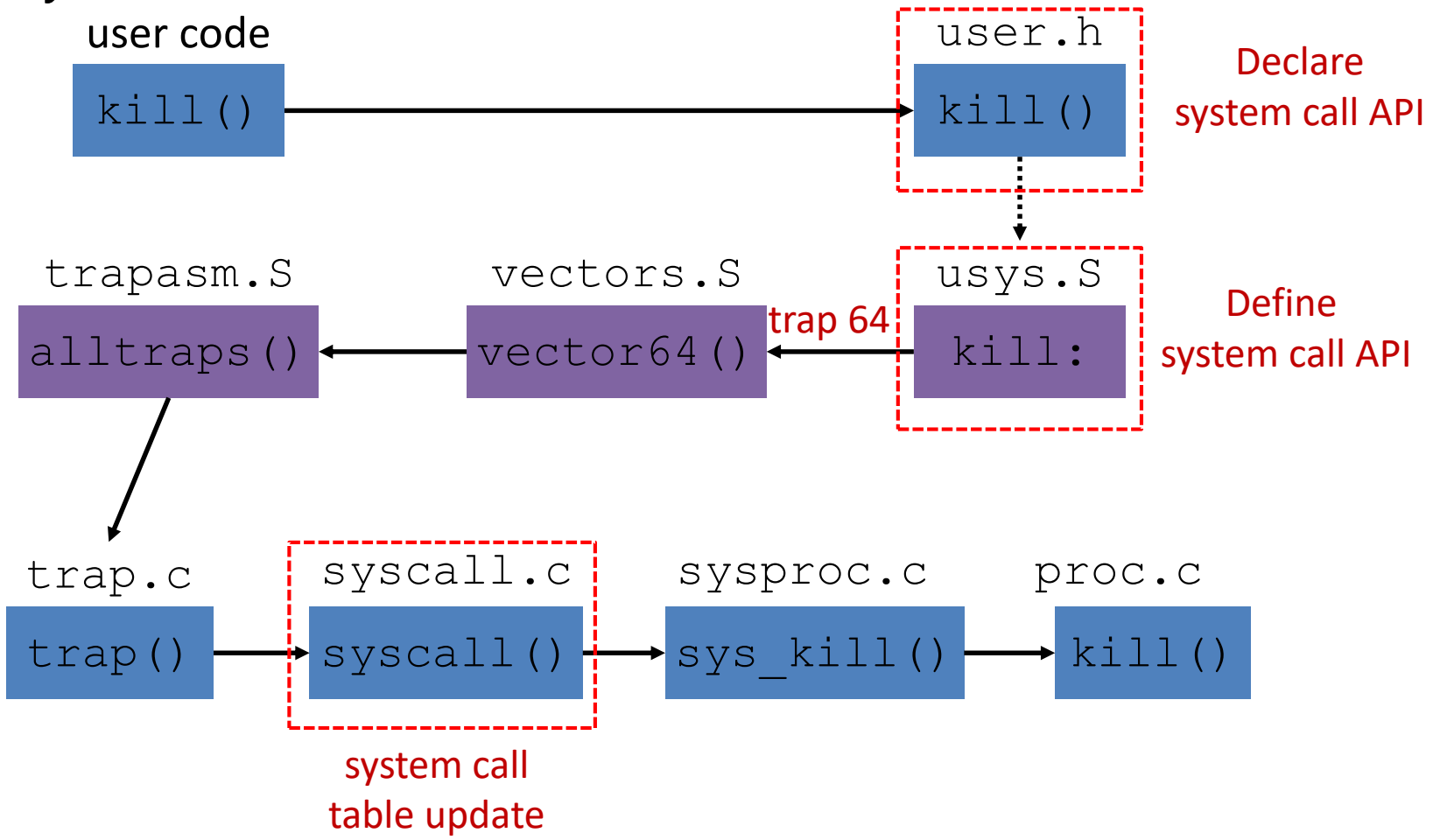
How to Make New System Call on Xv6

- Define a library routine for your new system call

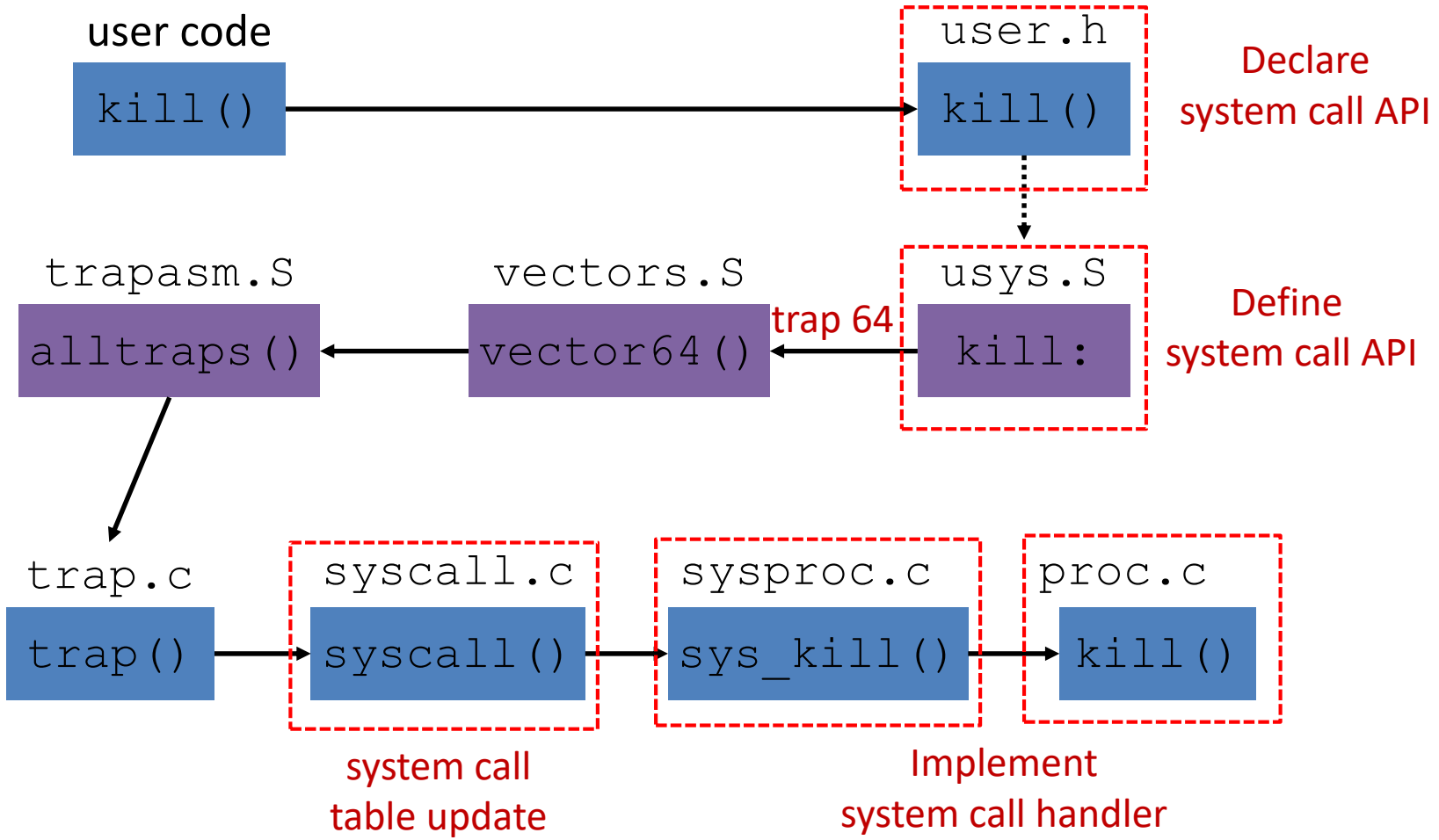


How to Make New System Call on Xv6

- Update the system call table



How to Make New System Call on Xv6



Hand-in Procedures

- Download template
  - https://github.com/KilhoLee/xv6-ssu.git (pull or clone)
  - `cp templates/xv6_ssu_syscall.tar.gz [YOUR_DIRECTORY]`
  - `cd [YOUR_DIRECTORY]`
  - `tar xvzf xv6_ssu_syscall.tar.gz`
  - ** DO CODING **

- Compress your code with your ID (ex> 20201234)
  - `$tar cvzf xv6_ssu_syscall_20201234.tar.gz xv6_ssu_syscall`
  - Must do `$make clean` before compressing, if not, you will get penalty.
  - Double-check whether the source codes are correct!!

- Submit your `tar.gz` file through class.ssu.ac.kr