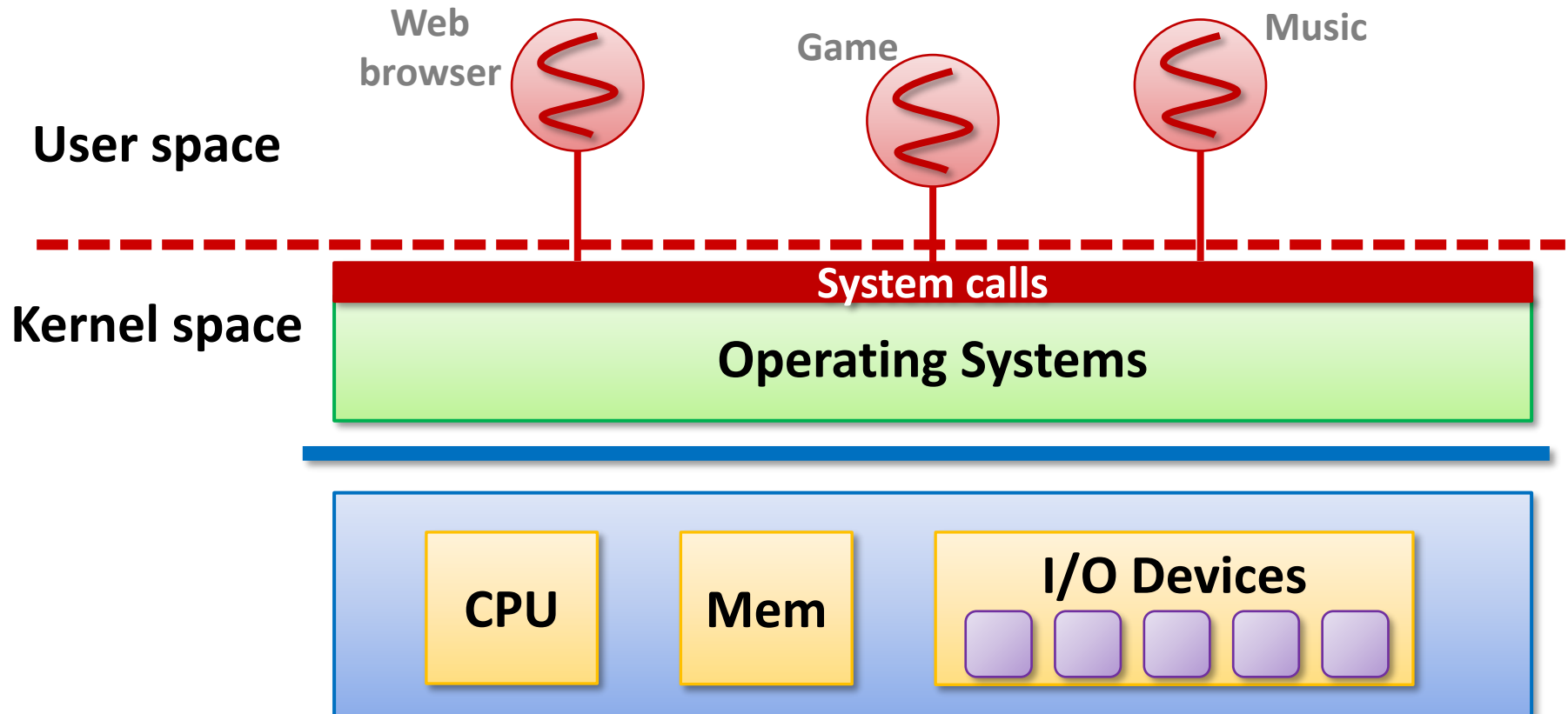


Project #1: System Call

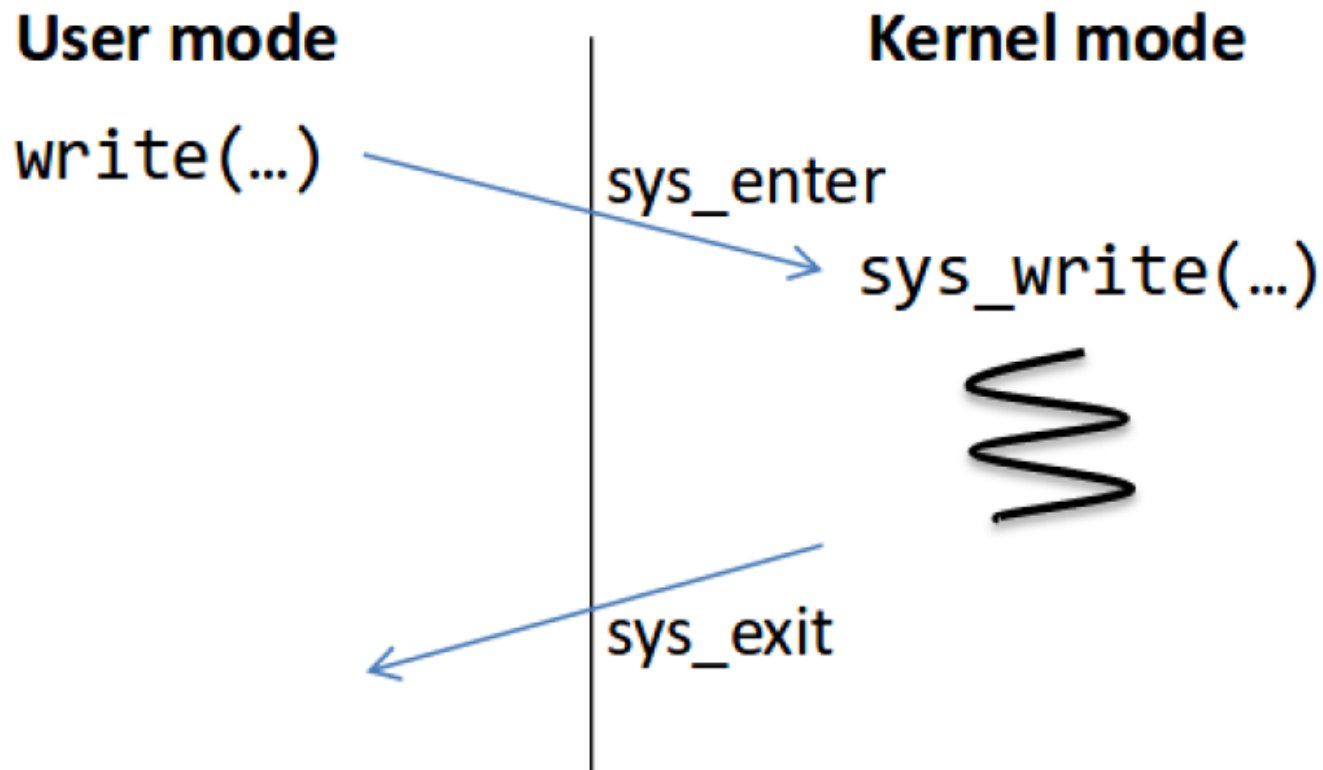
Instructor: Sungyong Ahn

Operating system



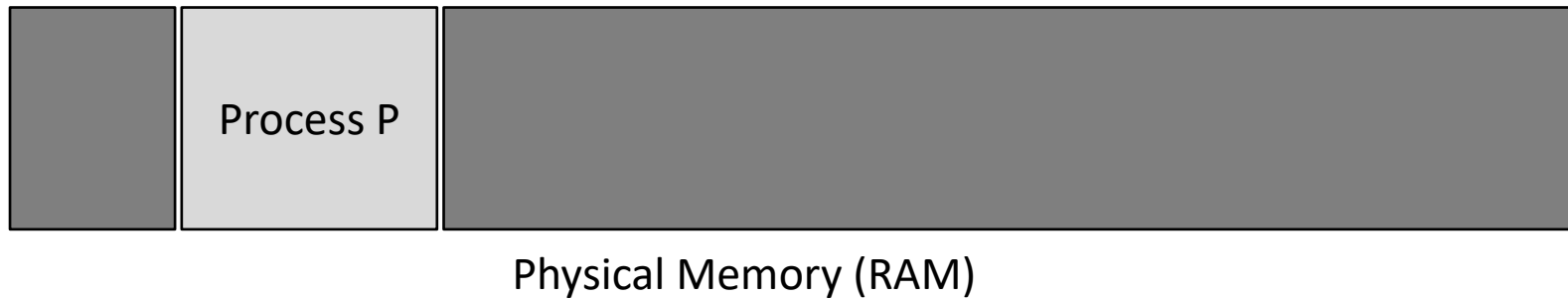
System Call

- An interface for accessing the kernel from user space



Trap Handling Process

■ Intel architecture



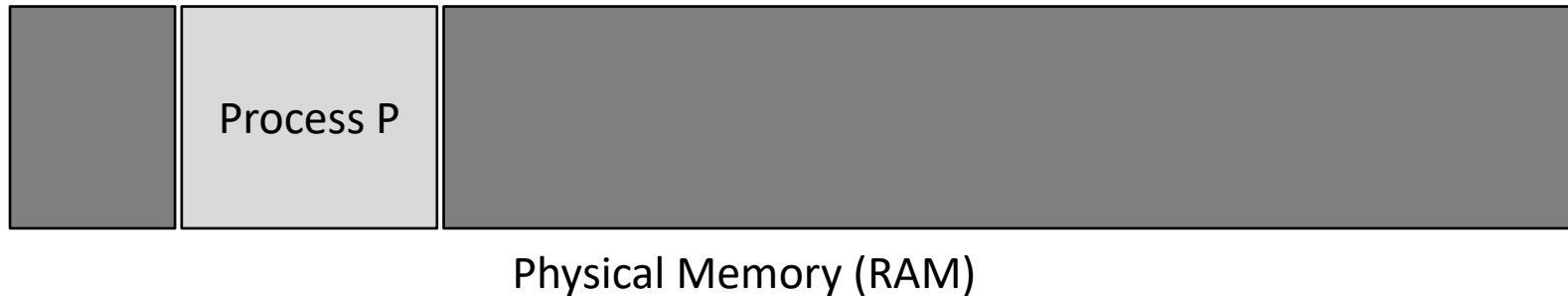
Trap Handling Process (Cont'd)

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



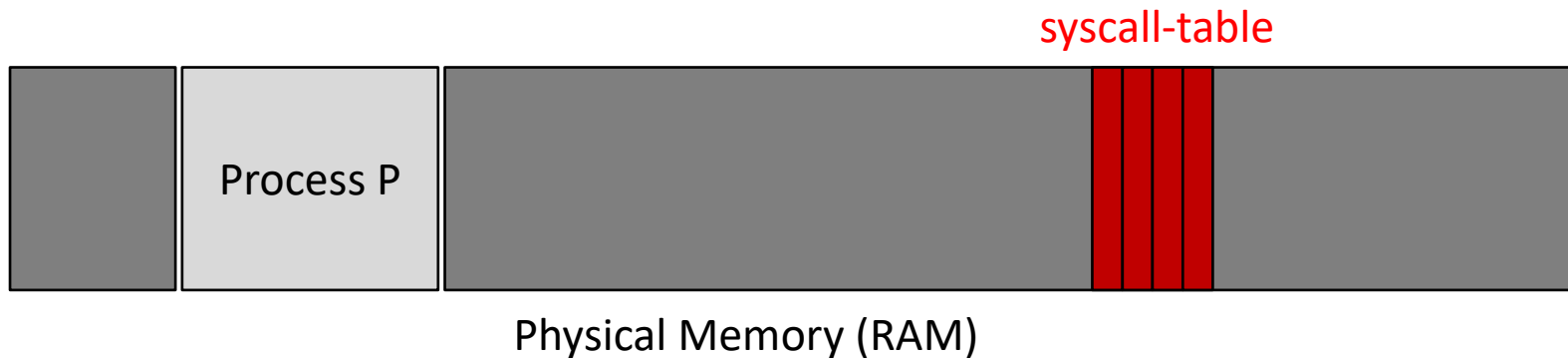
Trap Handling Process (Cont'd)

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



Trap Handling Process (Cont'd)

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

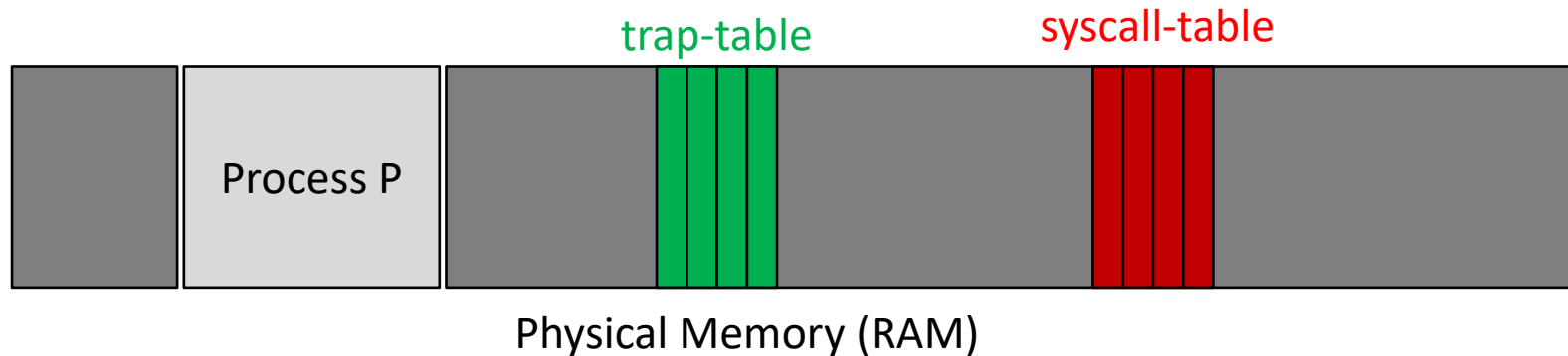


```
movl $6, %eax;      int $64
```

syscall-table index

Trap Handling Process (Cont'd)

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



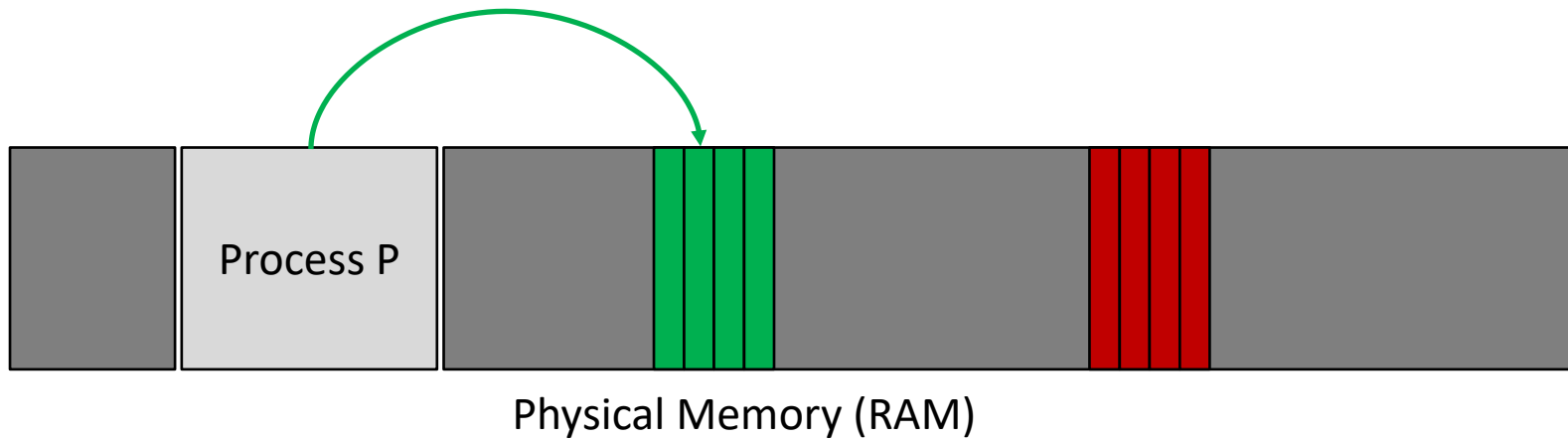
```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



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

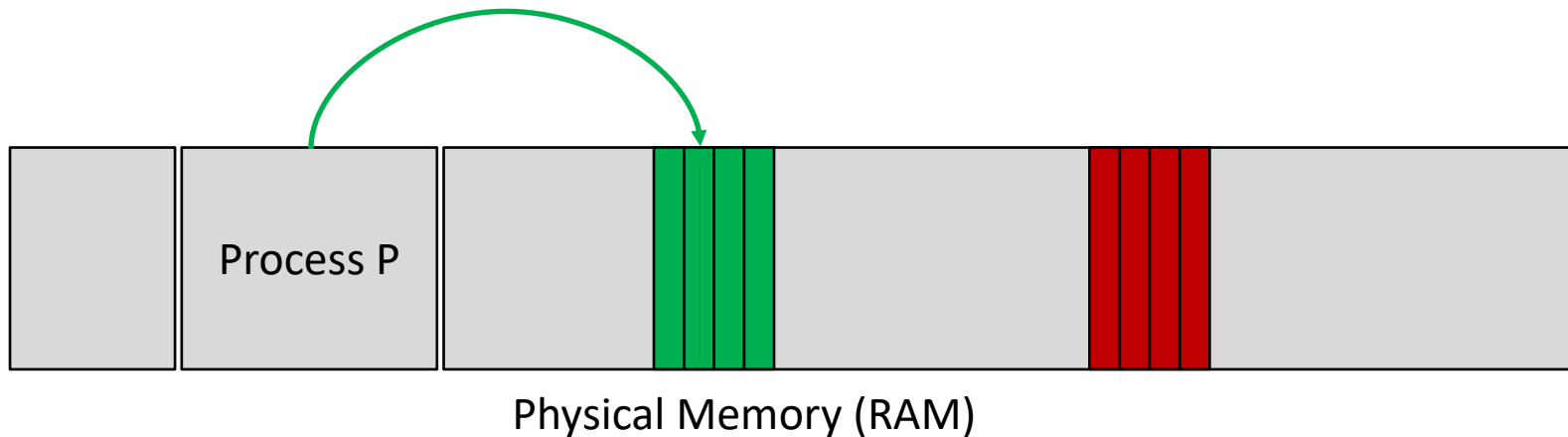
syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)

Kernel mode: we can do anything!



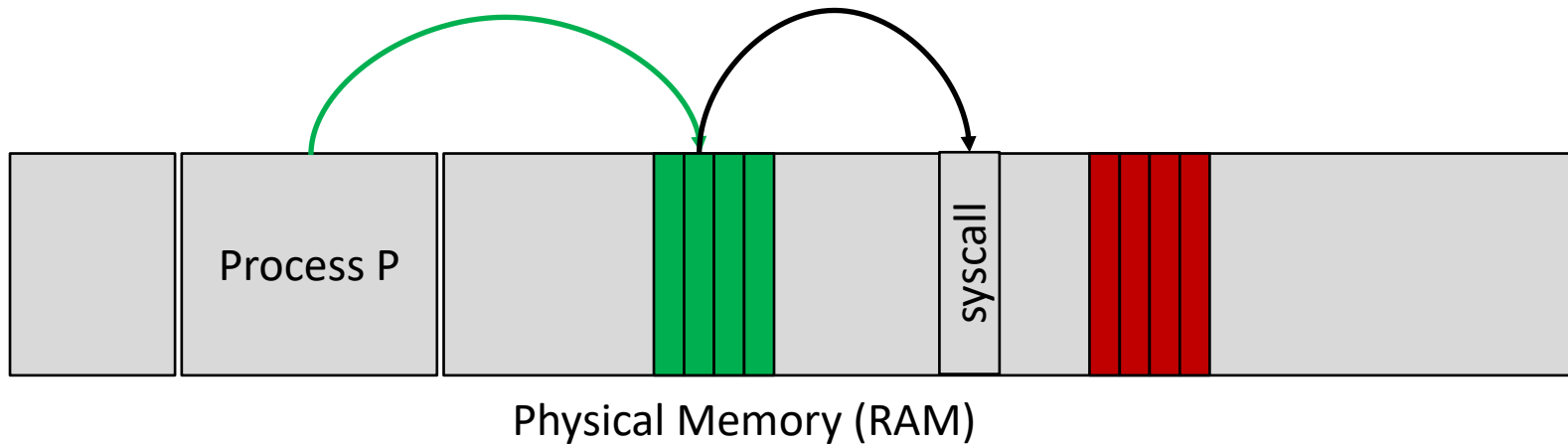
```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



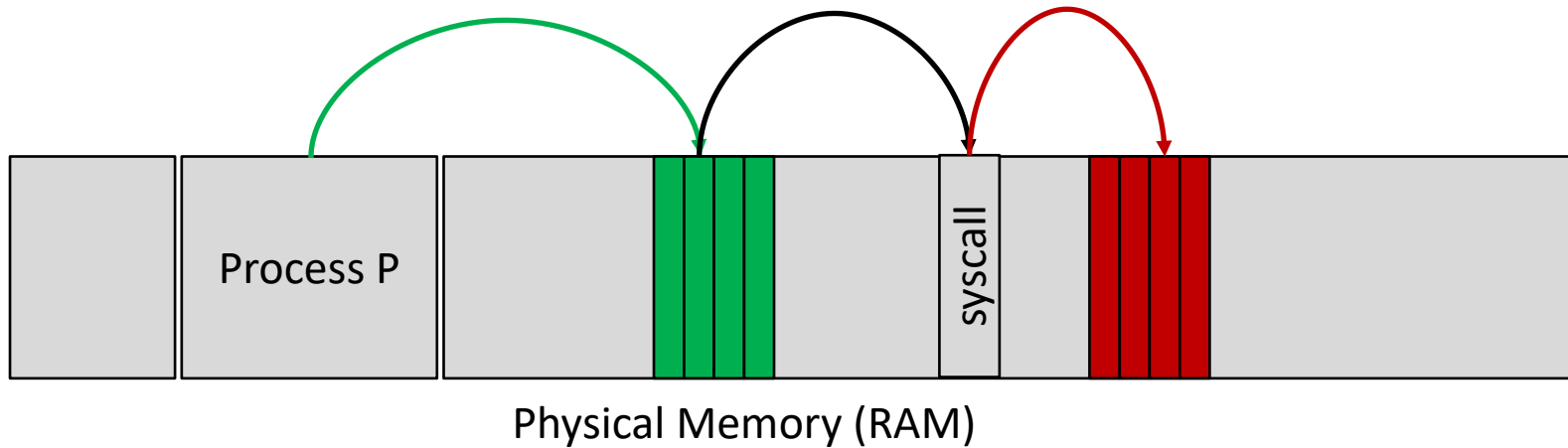
```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



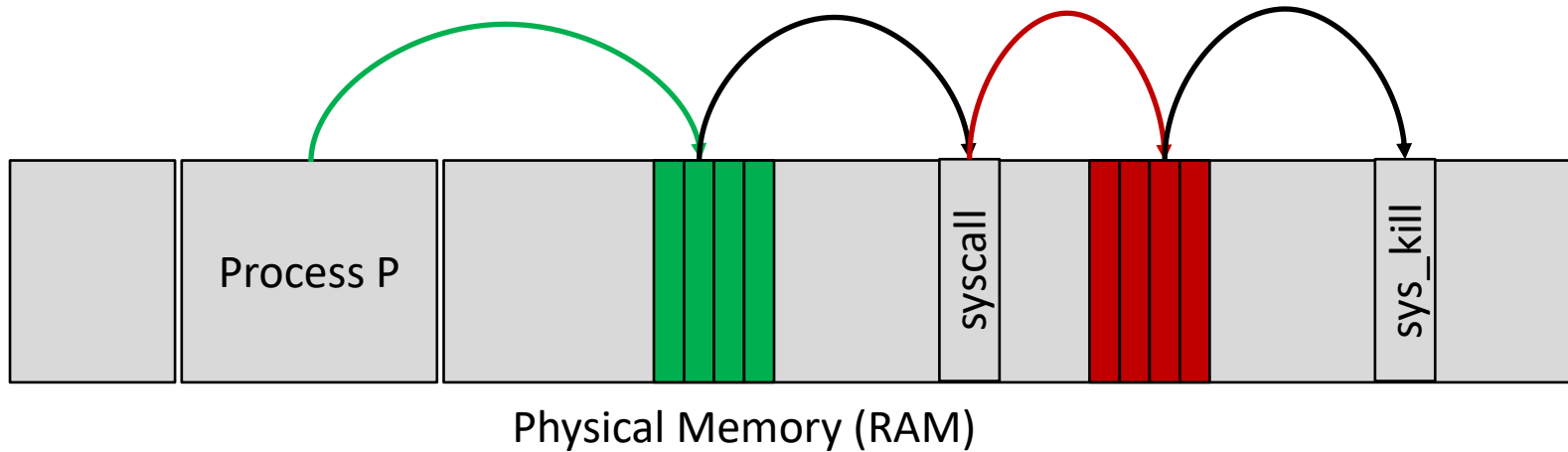
```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)



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

syscall-table index

```
int $64
```

trap-table index

User-level System Call API

■ 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);
```

User-level System Call API (Cont'd)

■ `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
```

System Call Number

■ syscall.h

```
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 Number

■ traps.h

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

Interrupt Descriptor Table initialization

■ trap.c

```
// 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");
}
```

Interrupt Vector Table

■ 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 Handler: alltraps

■ 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
```

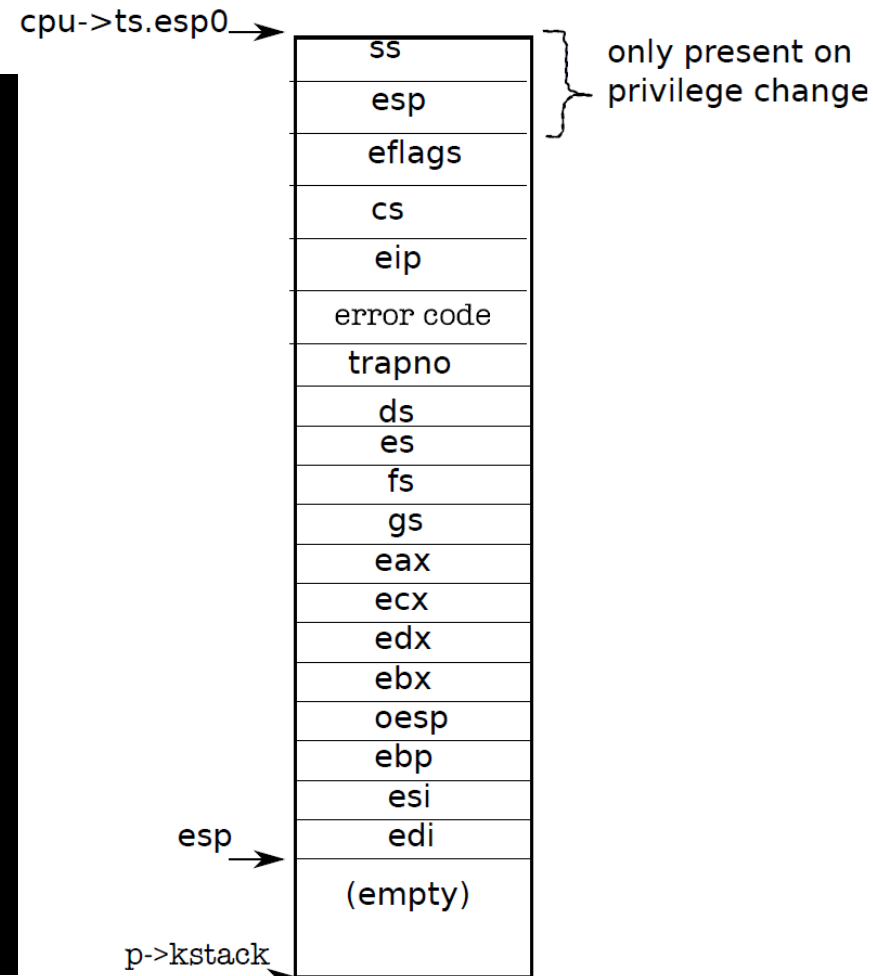


Figure 3-2. The trapframe on the kernel stack

Trap Handler: trap

■ 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;      // useful only in kernels
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

System Call Handler: syscall

■ 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
```

System Call Handler: sys_kill

■ 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 }
```

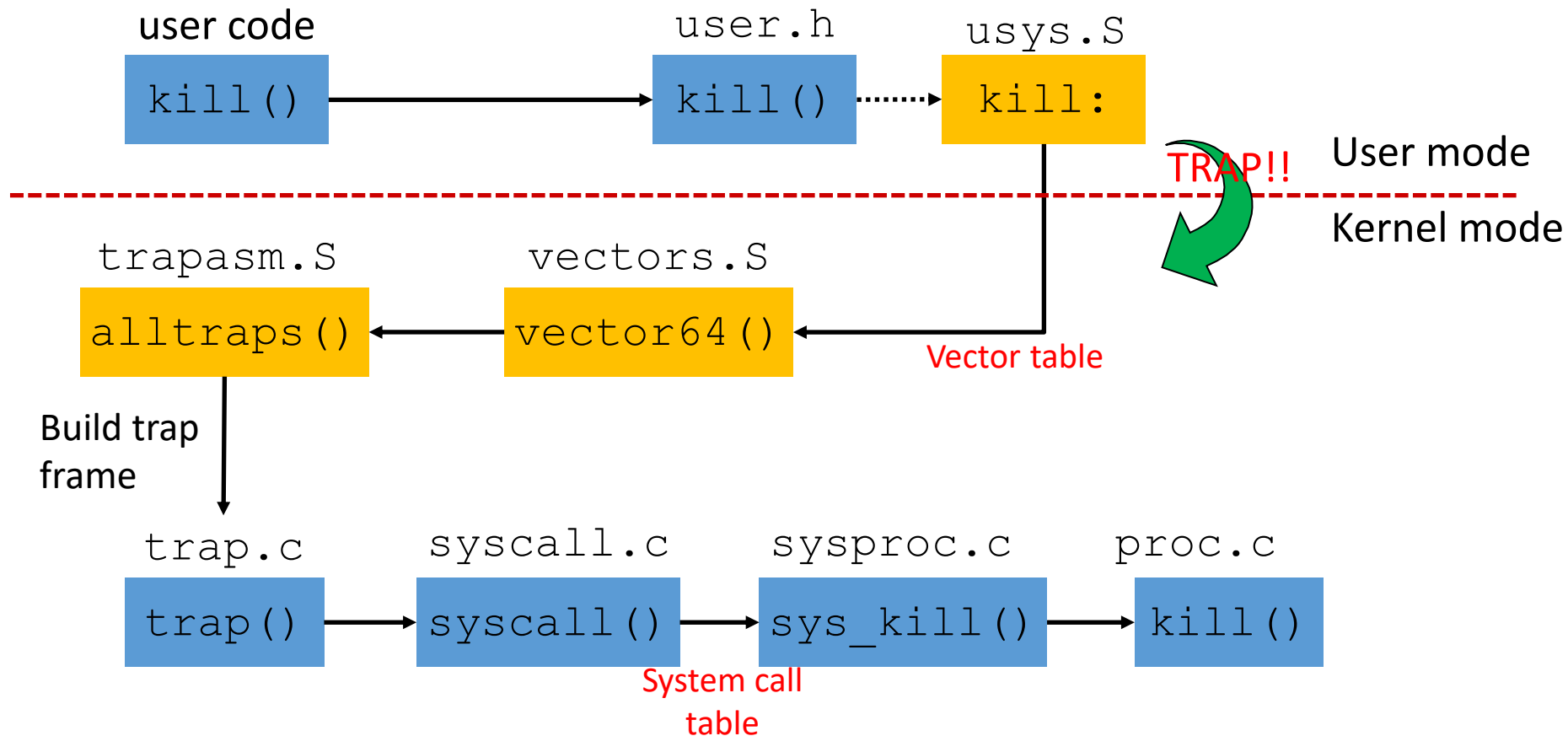
■ proc.c

```
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

■ `kill.c`

```
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    exit();
17 }
```

Test with User Program

■ Makefile

```

159 UPROGS=\
160     _cat\
161     _echo\
162     _forktest\
163     _grep\
164     _init\
165     _kill\
166     _ln\
167     _ls\
168     _mkdir\
169     _rm\
170     _sh\
171     _stressfs\
172     _usertests\
173     _wc\
174     _zombie\

```

■ 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 own .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

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

Project #1. Make System Calls

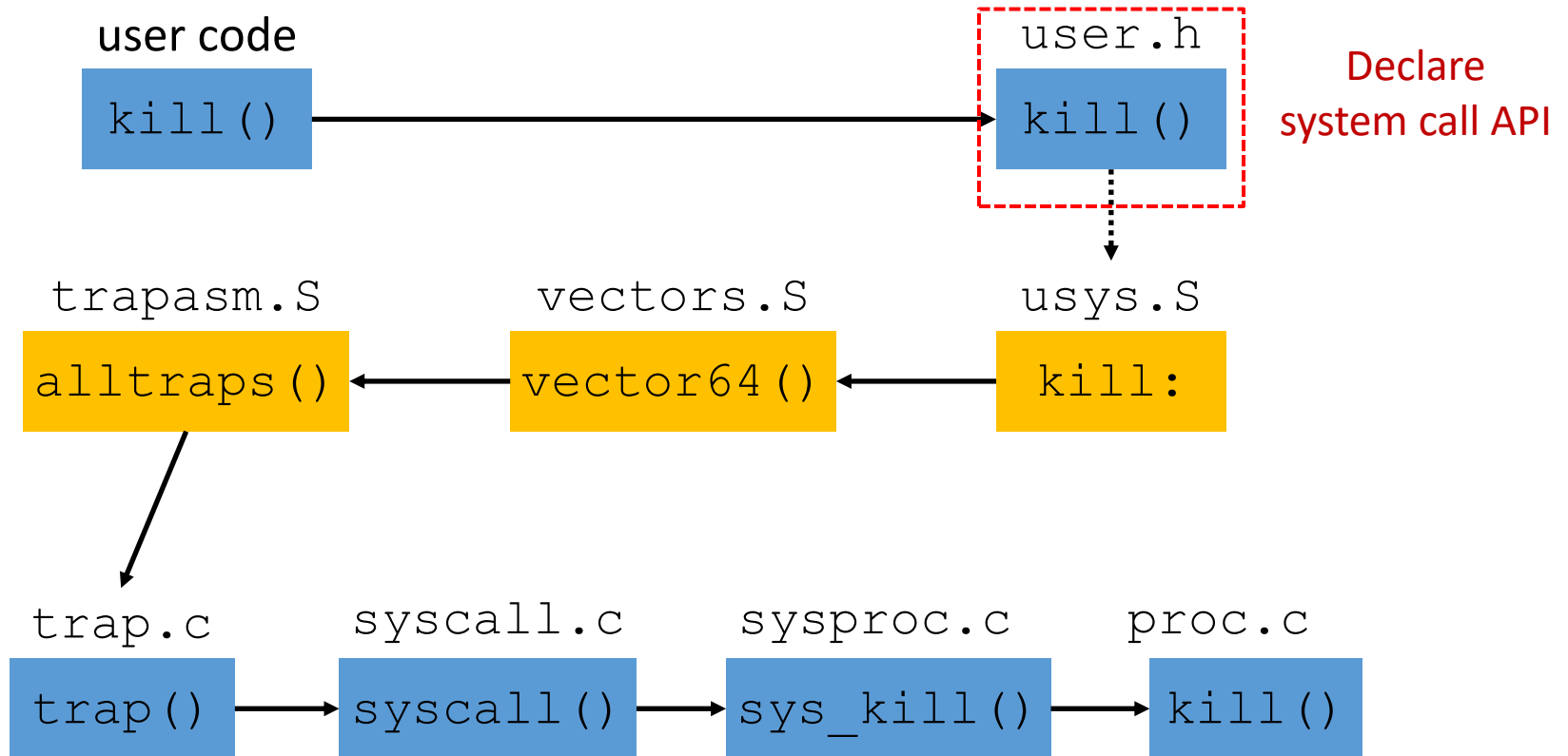
■ `int getreadcount(void)`

- Returns the value of a counter which is incremented every time any process calls the `read()` system call.
- You have to define a variable for the values of a read counter
 - (e.g. `readcount`)

■ Make a user program `readcount` using `getreadcount()` system call for testing

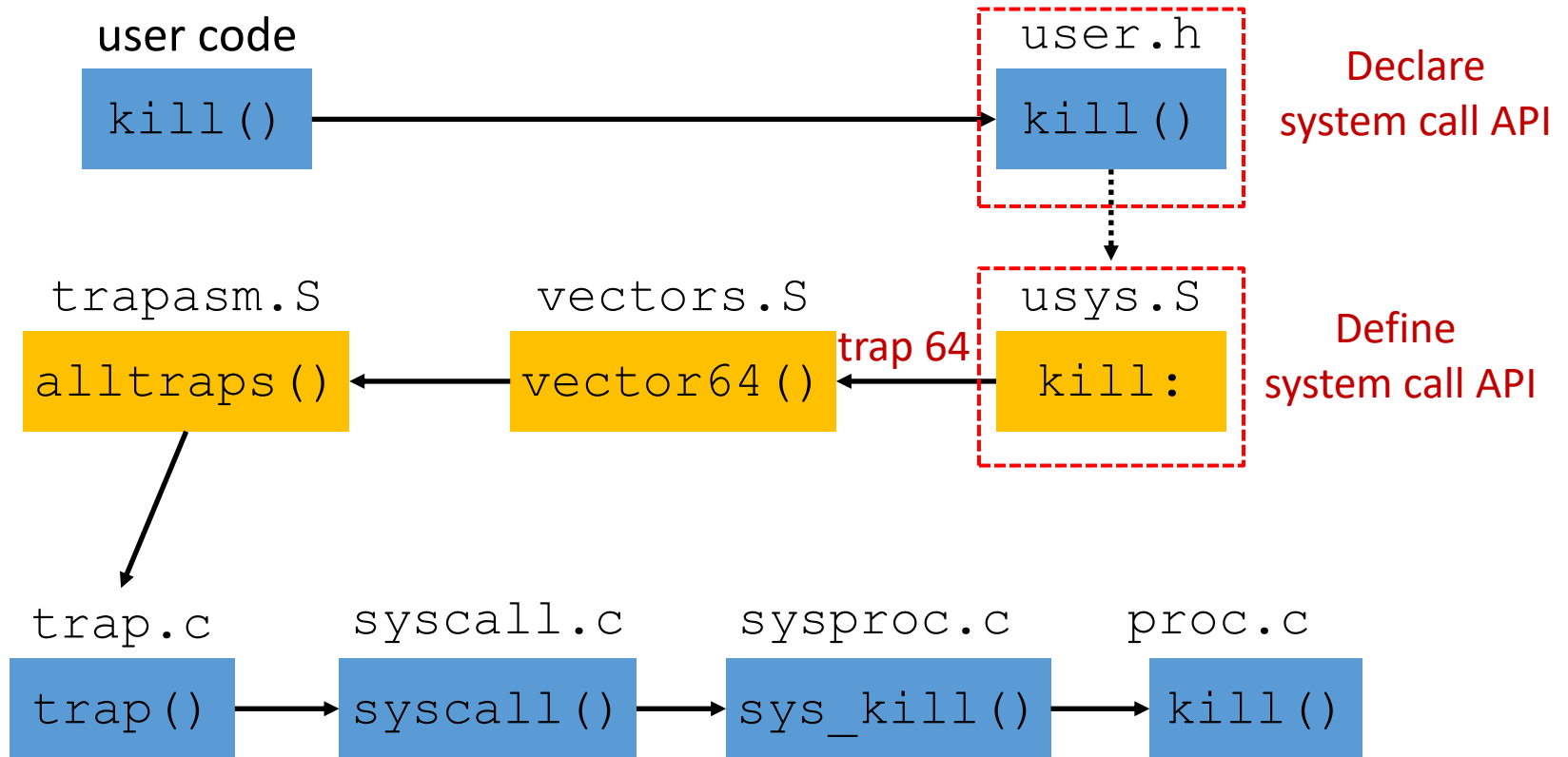
How to Make New System Call on Xv6

■ Declare a library routine for your new system call



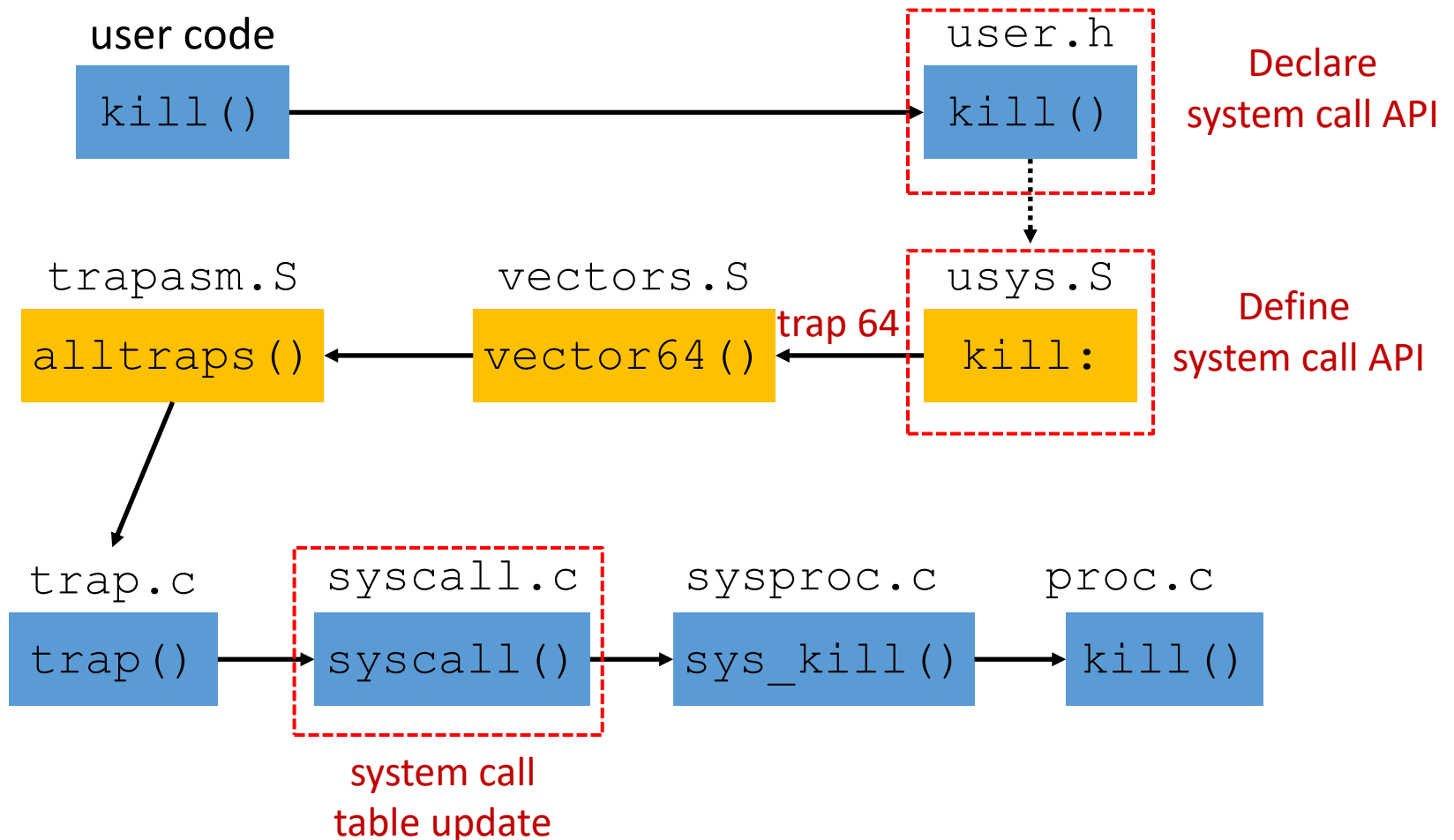
How to Make New System Call on Xv6

■ Define a library routine for your new system call



How to Make New System Call on Xv6

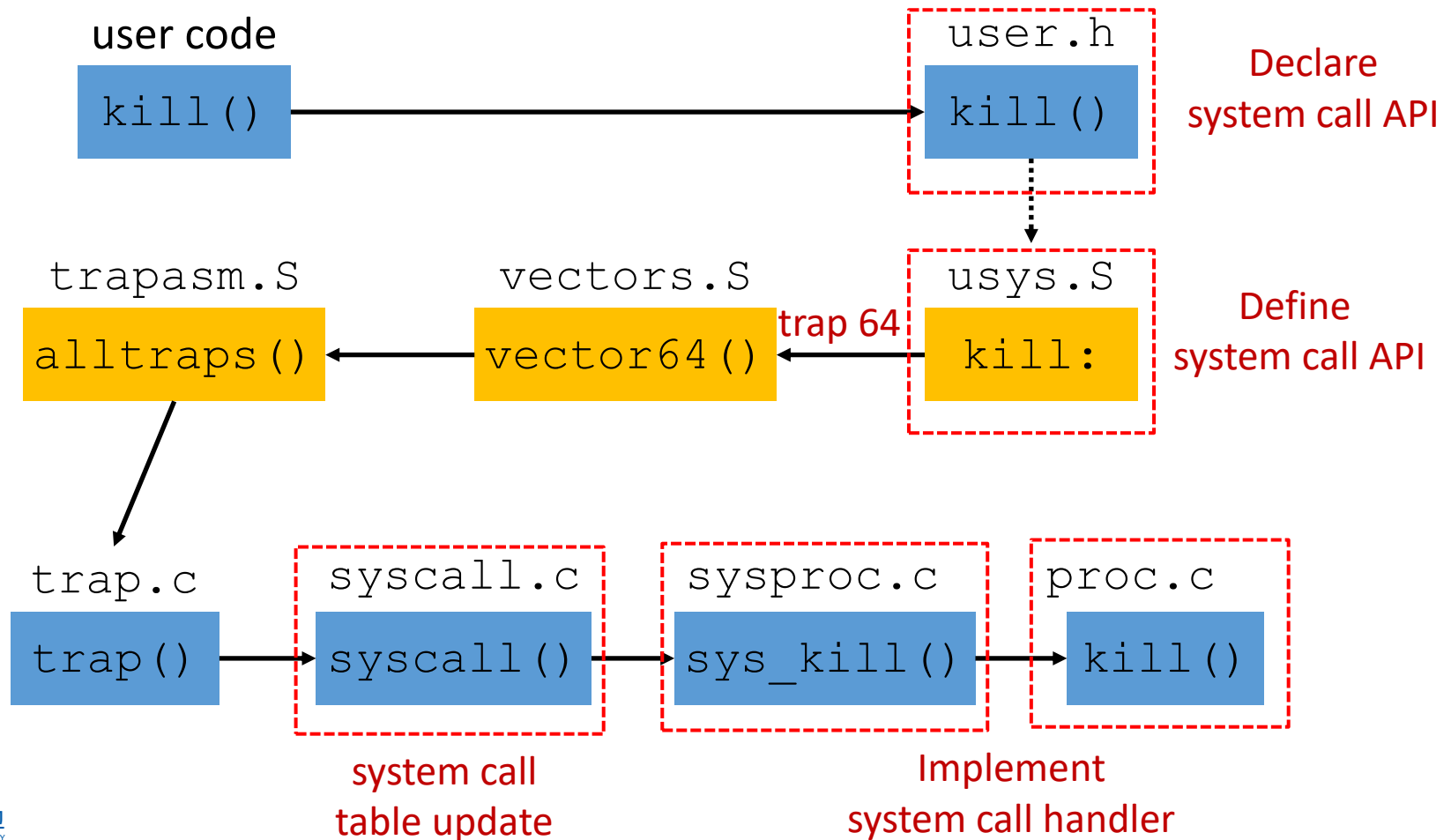
■ Update the system call table



How to Make New System Call on Xv6

■ Implement system call handler

- You have to implement your new system call handler in suitable source files, not in `sysproc.c` or `proc.c`



Submission

■ Compress your xv6 folder as `StudentID-1.tar.gz`

- `$make clean`
- `$tar -czvf StudentID-1.tar.gz ./xv6-public`
- Please command `$make clean` before compressing

■ Submit your `tar.gz` file through **PLATO**

■ Due date: **4/15, 23:59**

- Late submission penalty: **-25% penalty of total mark per day**

■ **PLEASE DO NOT COPY !!**

- **YOU WILL GET F IF YOU COPIED**