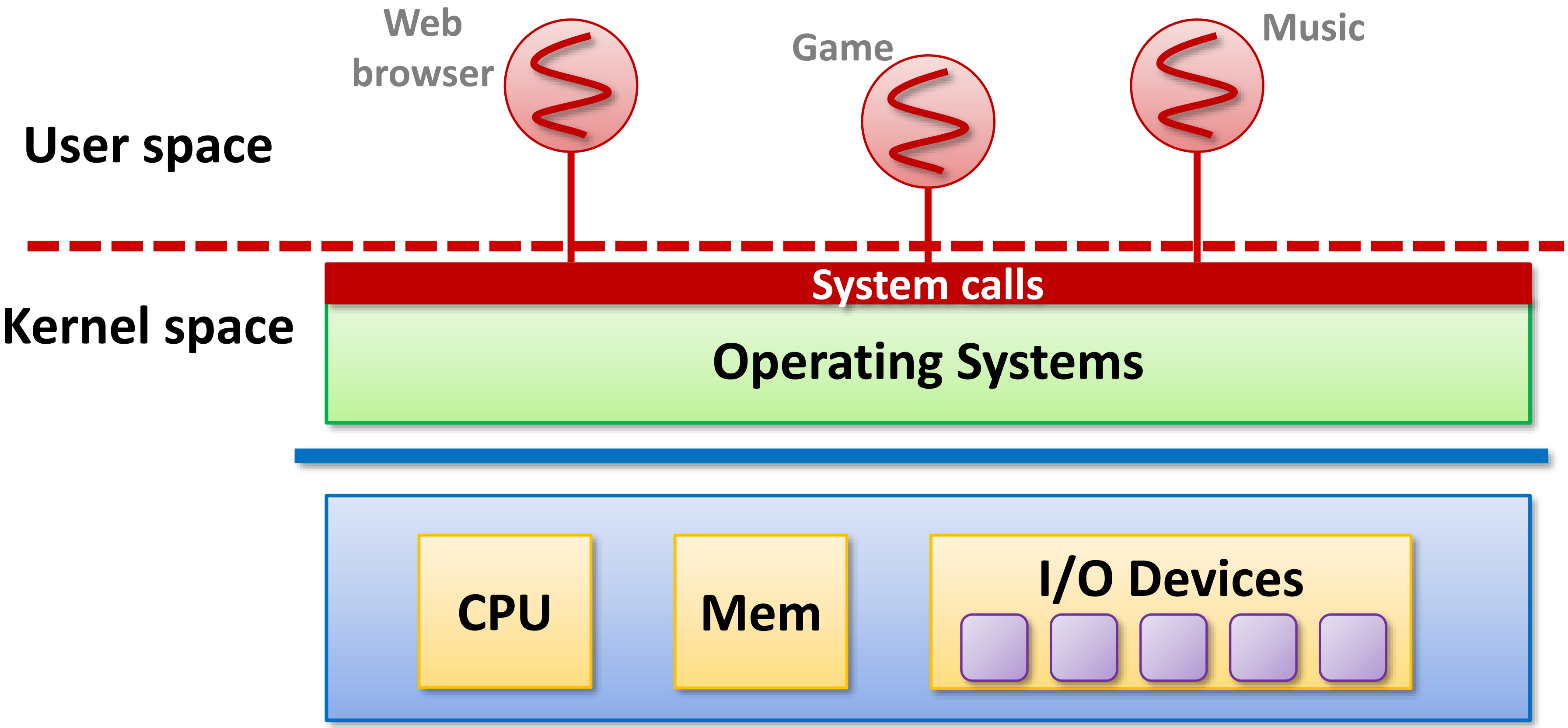
Project #1: System Call

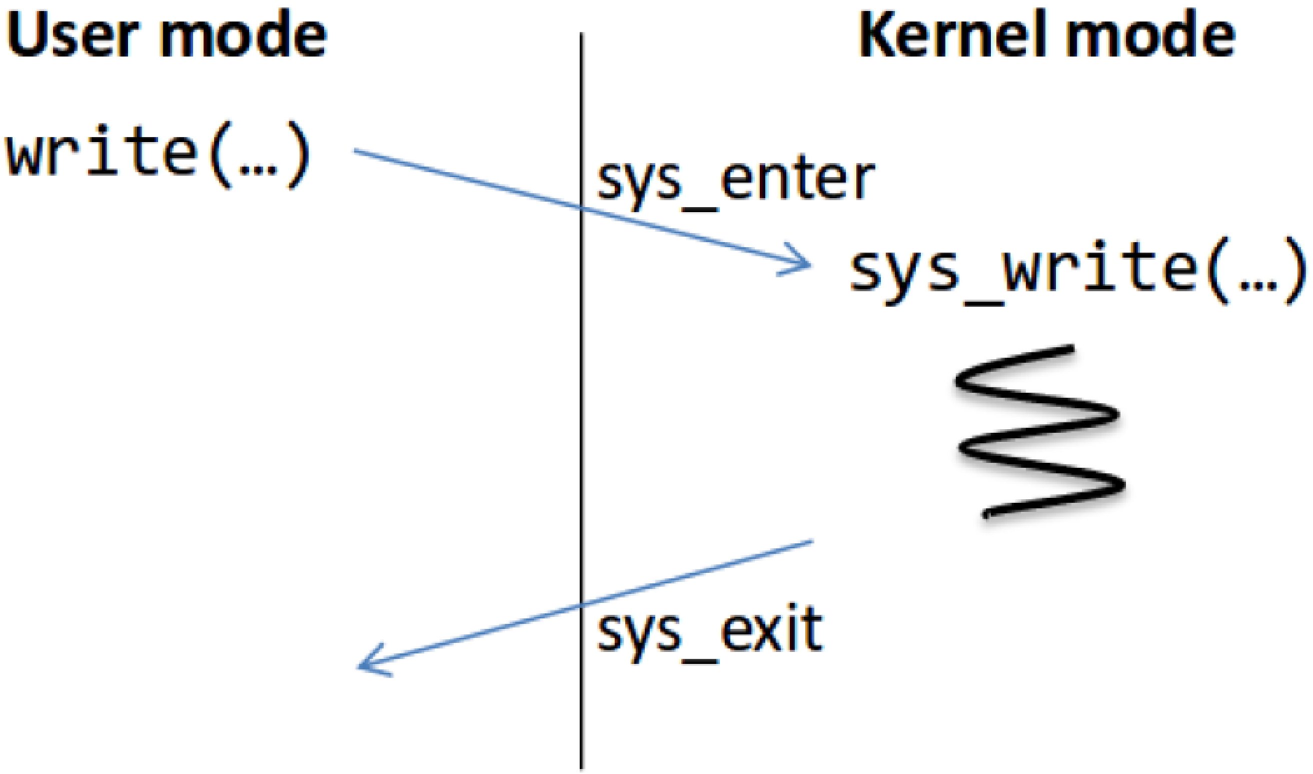
Instructor: Sungyong Ahn

Operating system



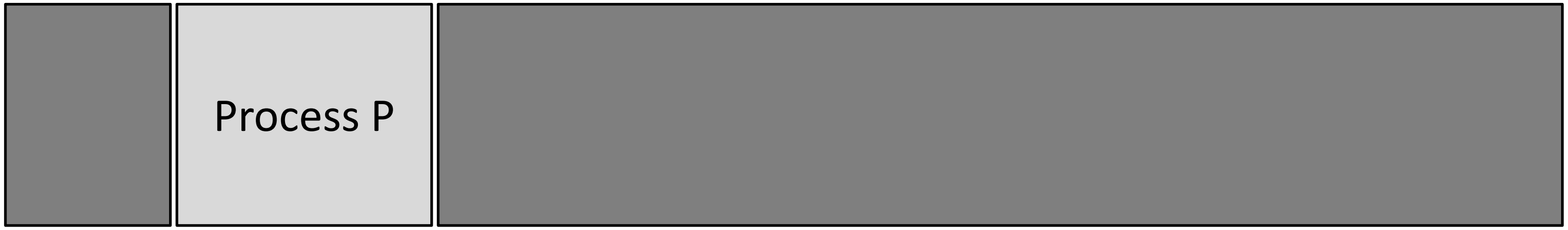
System Call

- An interface for accessing the kernel from user space



Trap Handling Process

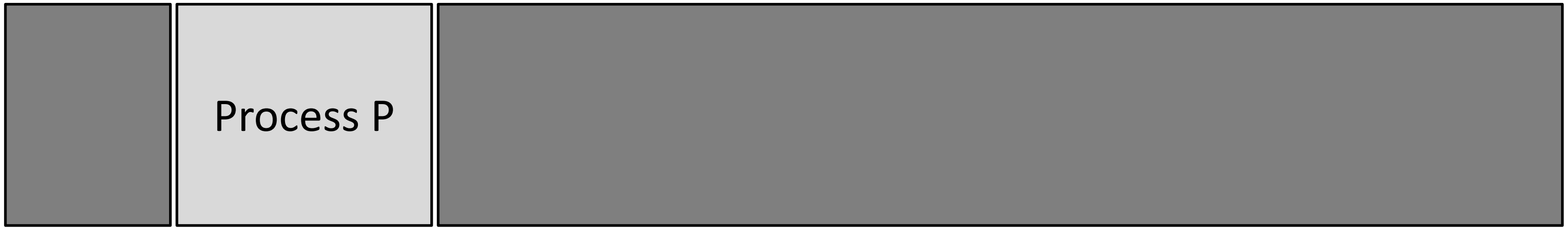
- **Intel architecture**



Physical Memory (RAM)

Trap Handling Process (Cont’d)

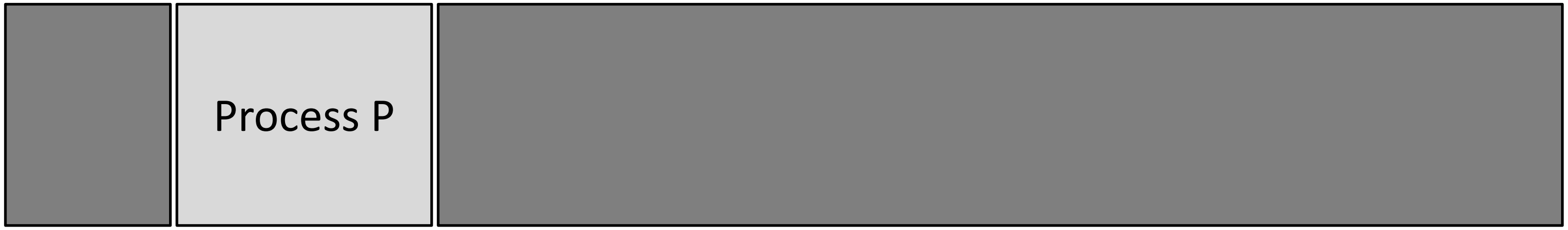
- **Process P can only see its own memory because of user mode (other areas, including kernel, are hidden)**



Physical Memory (RAM)

Trap Handling Process (Cont'd)

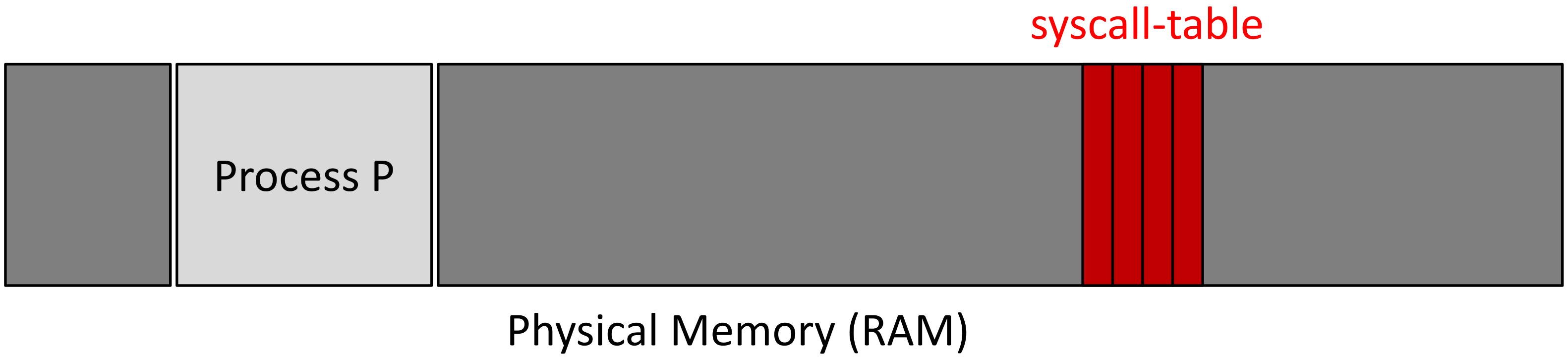
- Process P wants to call `kill()` system call



Physical Memory (RAM)

Trap Handling Process (Cont’d)

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

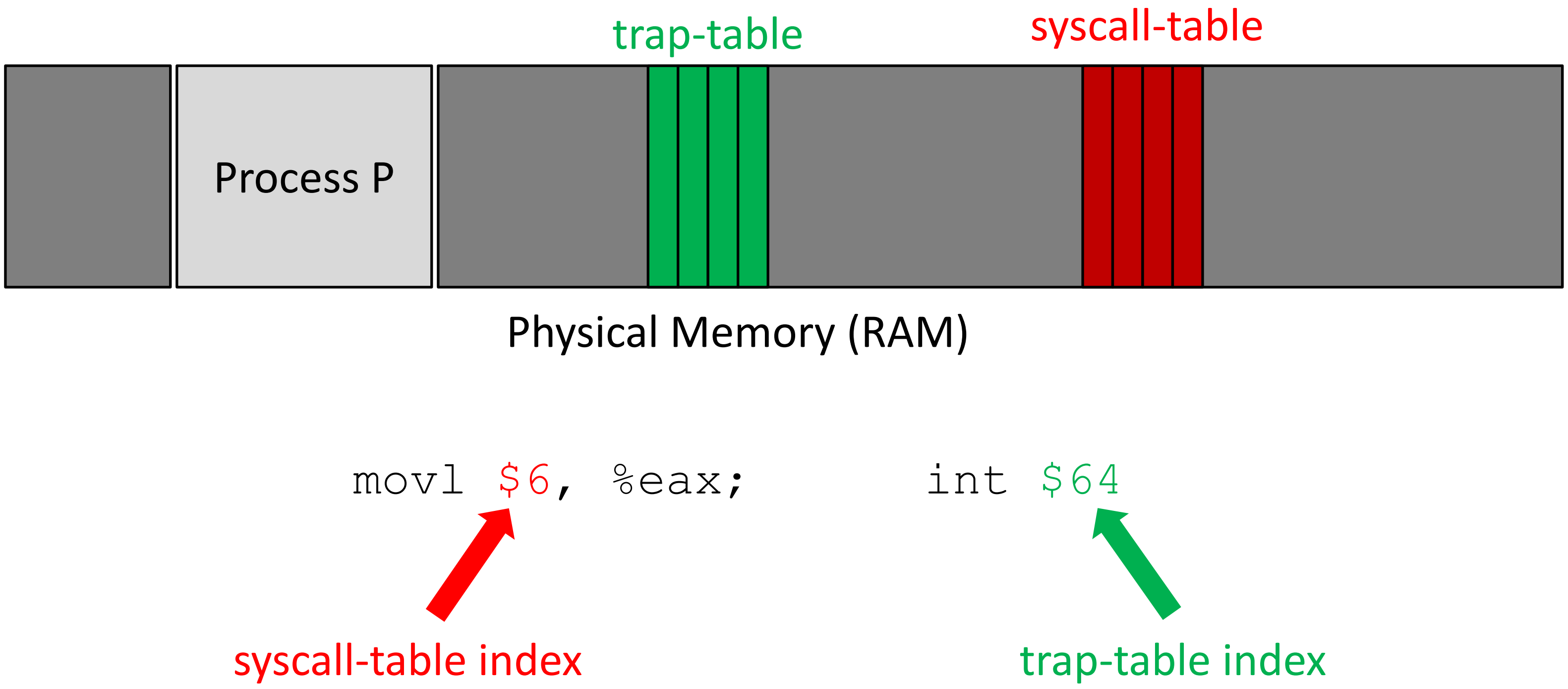


```
movl $6, %eax;      int $64
```

syscall-table index

Trap Handling Process (Cont'd)

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



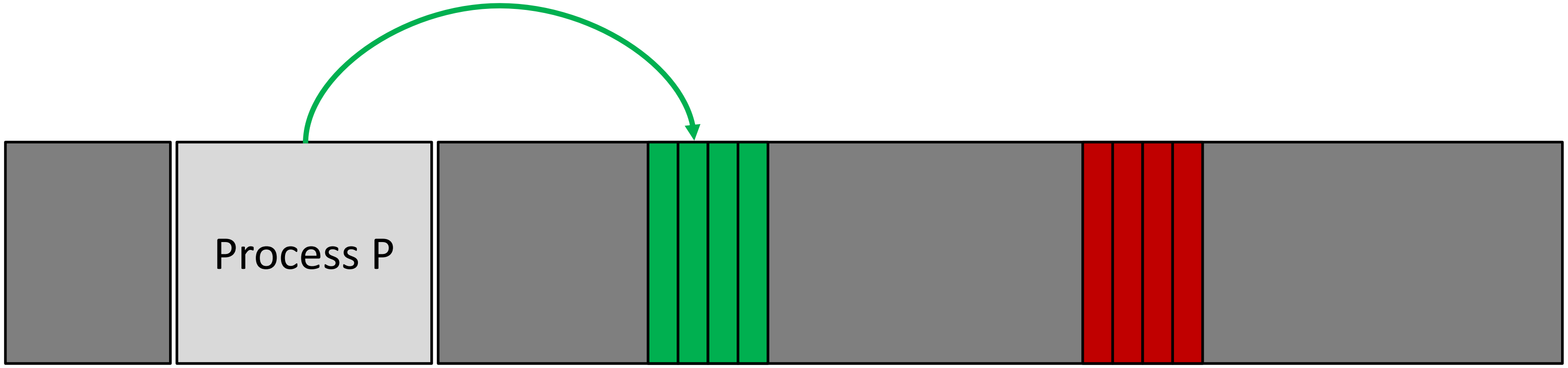
Physical Memory (RAM)

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

syscall-table index

trap-table index

Trap Handling Process (Cont’d)



Physical Memory (RAM)

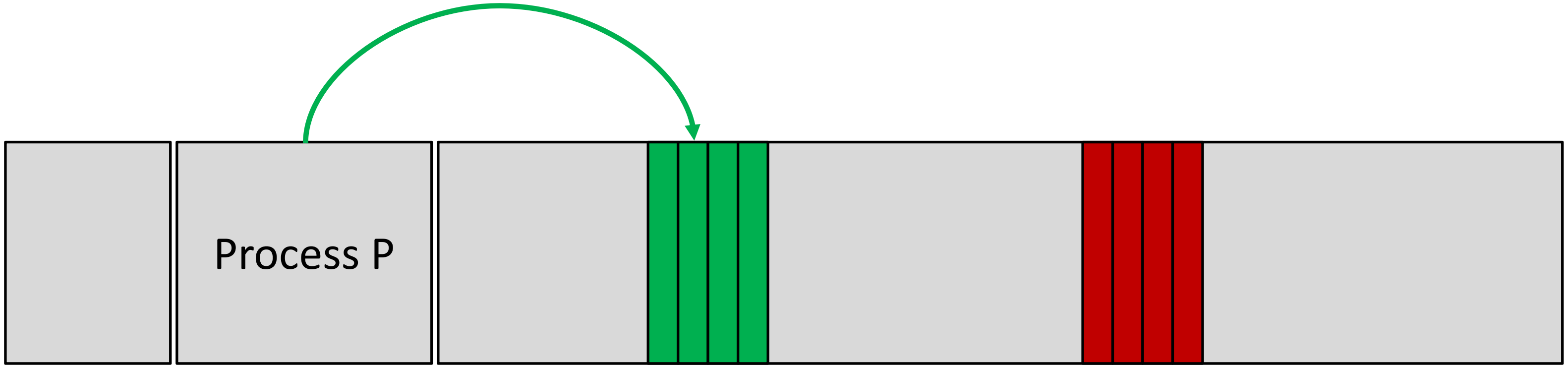
```
movl $6, %eax;        int $64
```

syscall-table index

trap-table index

Trap Handling Process (Cont’d)

Kernel mode: we can do anything!



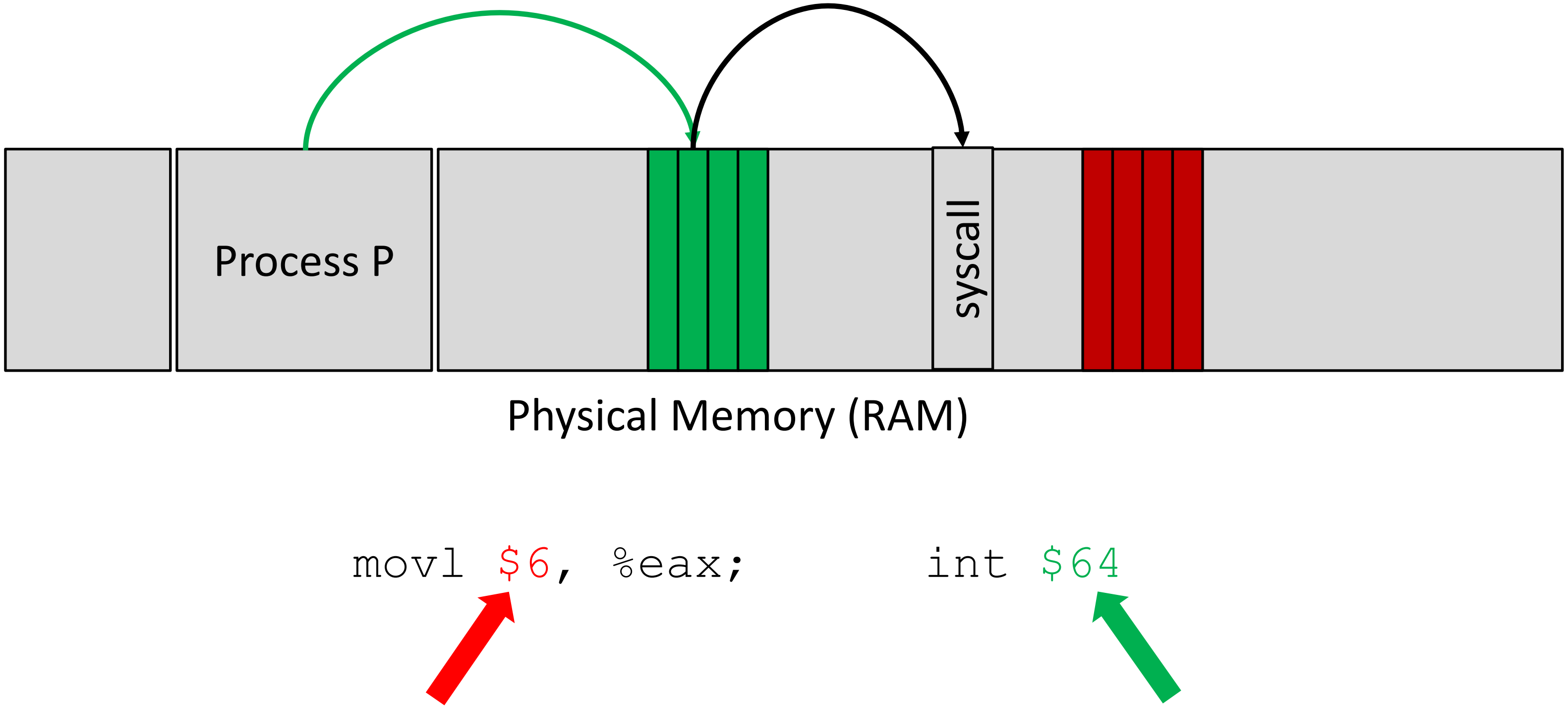
Physical Memory (RAM)

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

syscall-table index

trap-table index

Trap Handling Process (Cont’d)



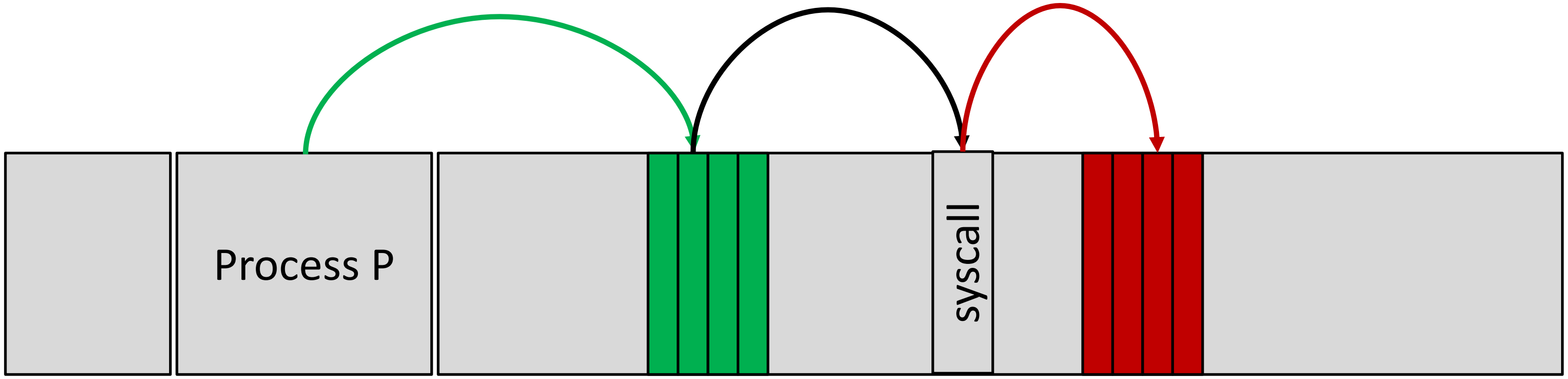
Physical Memory (RAM)

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

syscall-table index

trap-table index

Trap Handling Process (Cont'd)



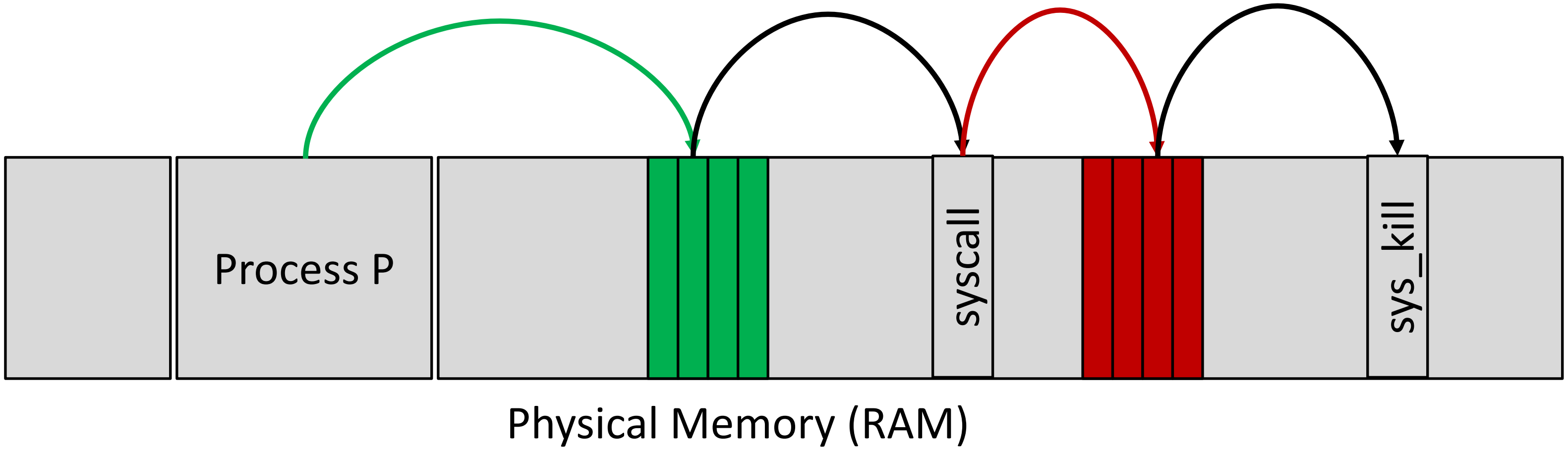
Physical Memory (RAM)

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

syscall-table index

trap-table index

Trap Handling Process (Cont'd)



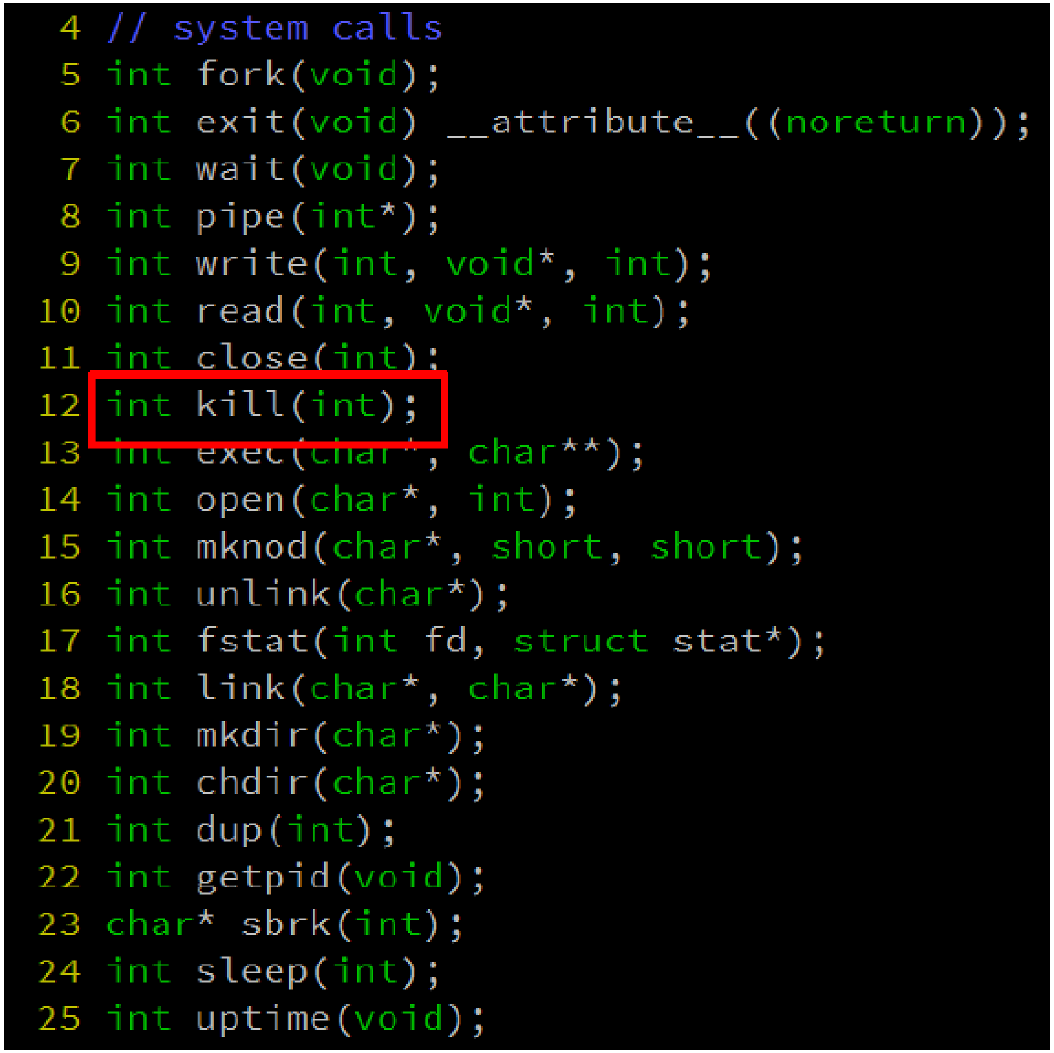
```
movl $6, %eax;        int $64
```

syscall-table index

trap-table index

User-level System Call API

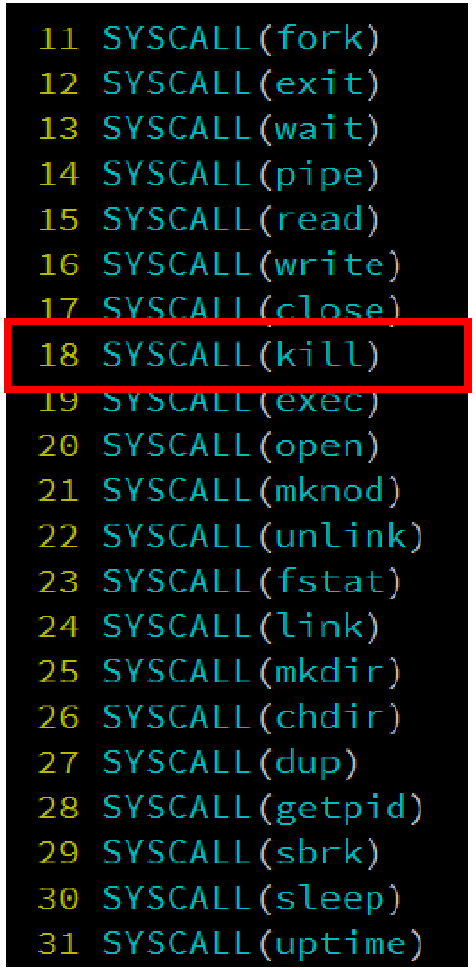
- `user.h`



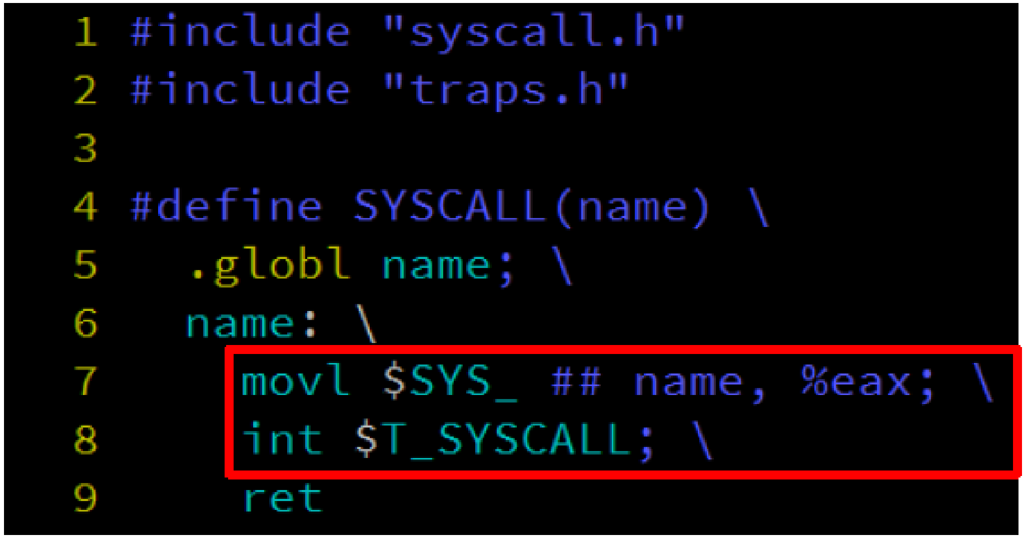
```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

User-level System Call API (Cont’d)

- `usys.S`



```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

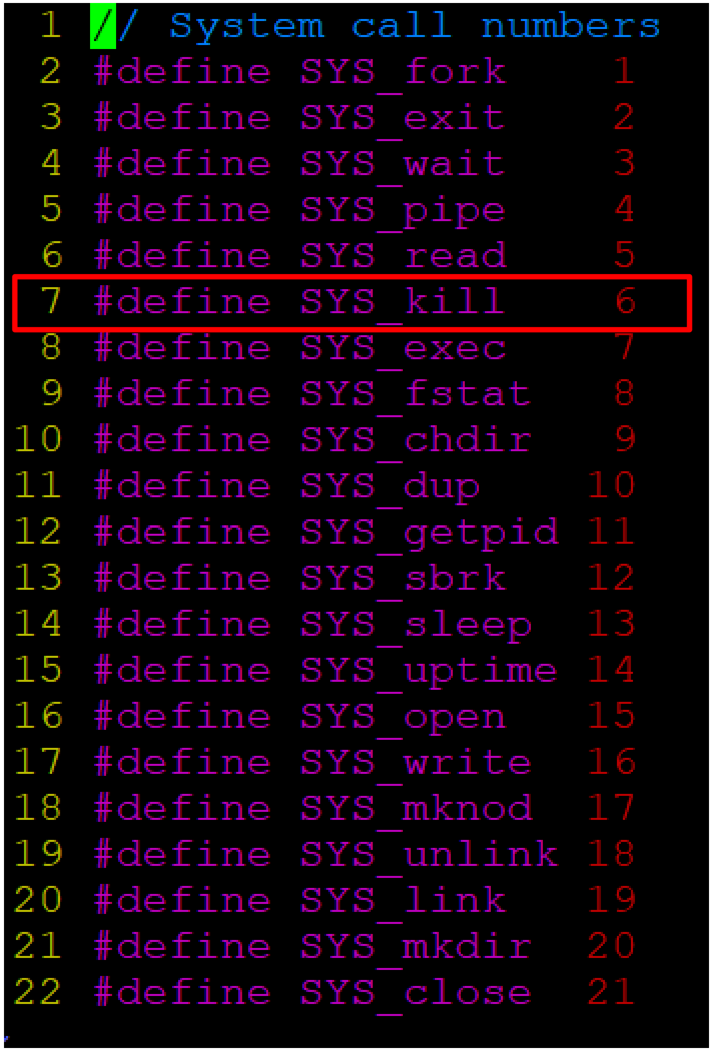


```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

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

System Call Number

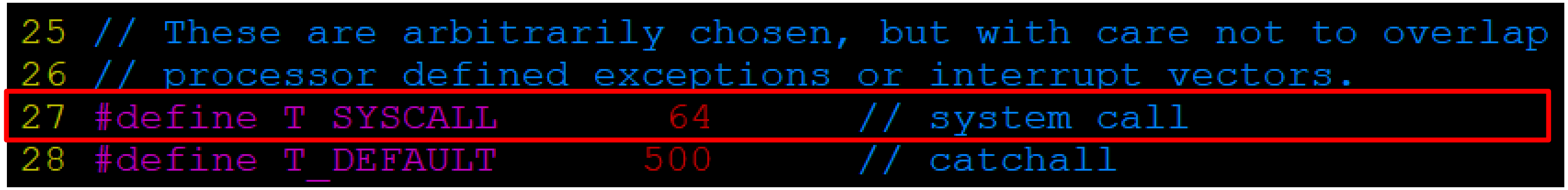
- syscall.h



```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

Trap Number

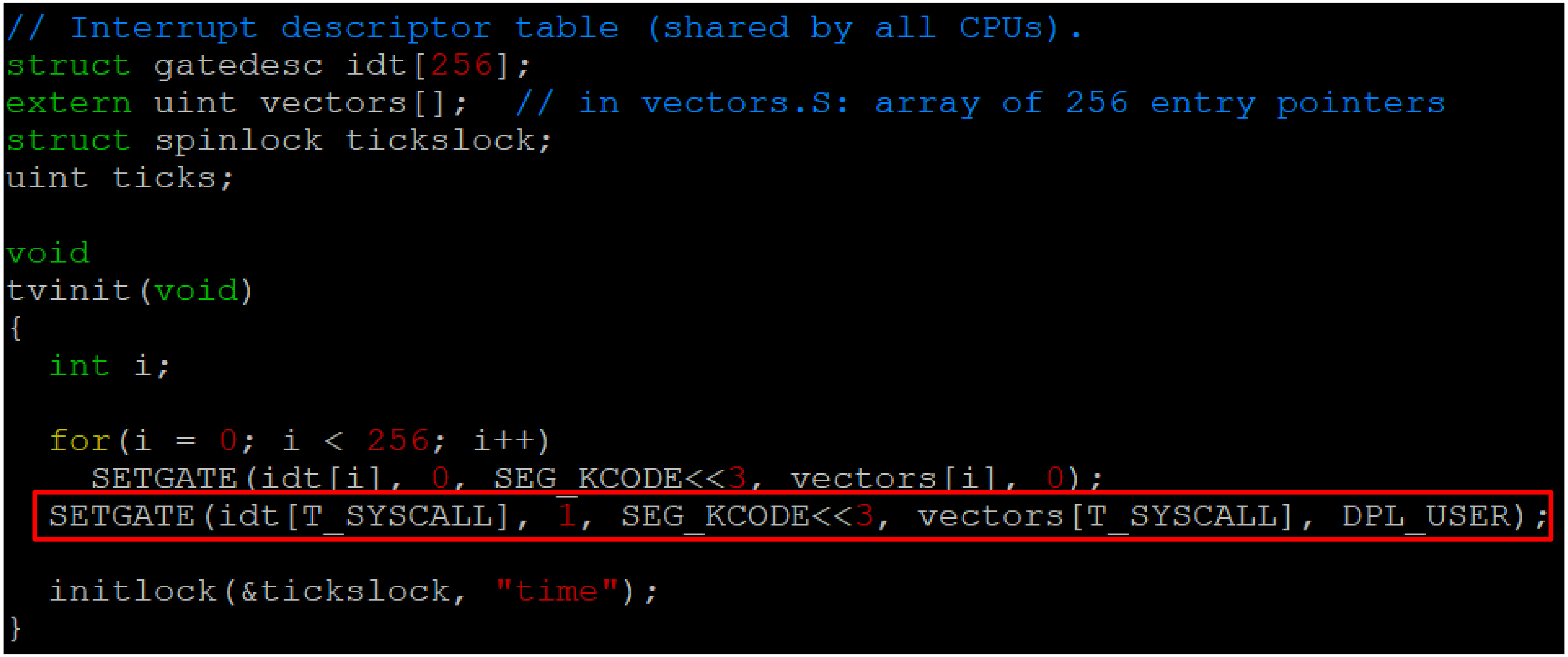
- traps.h



```
// 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
```

Interrupt Descriptor Table initialization

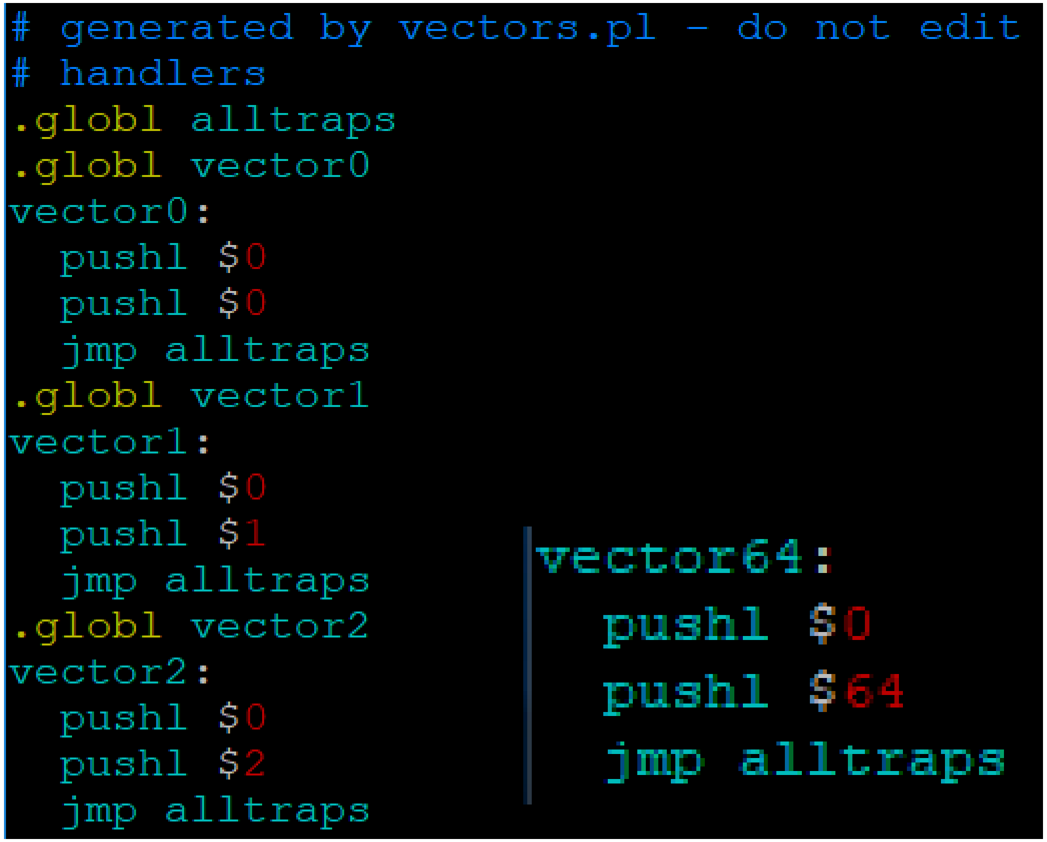
- trap.c



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

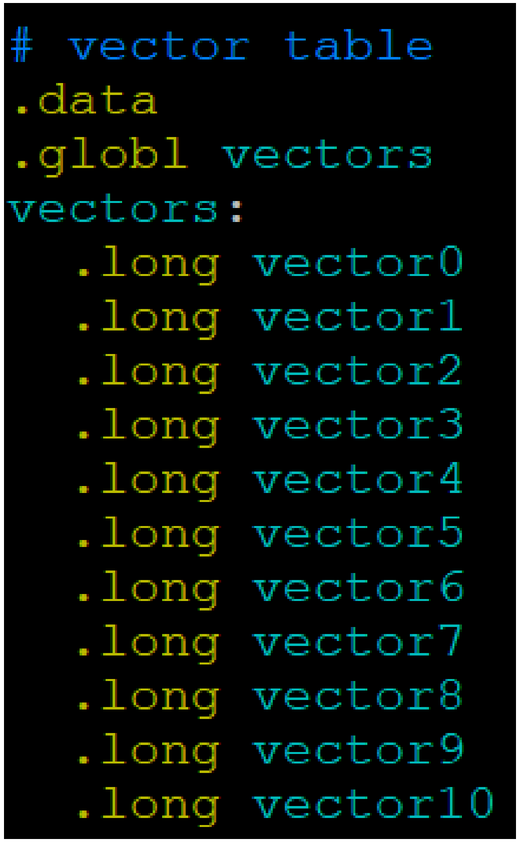
Interrupt Vector Table

- `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
```

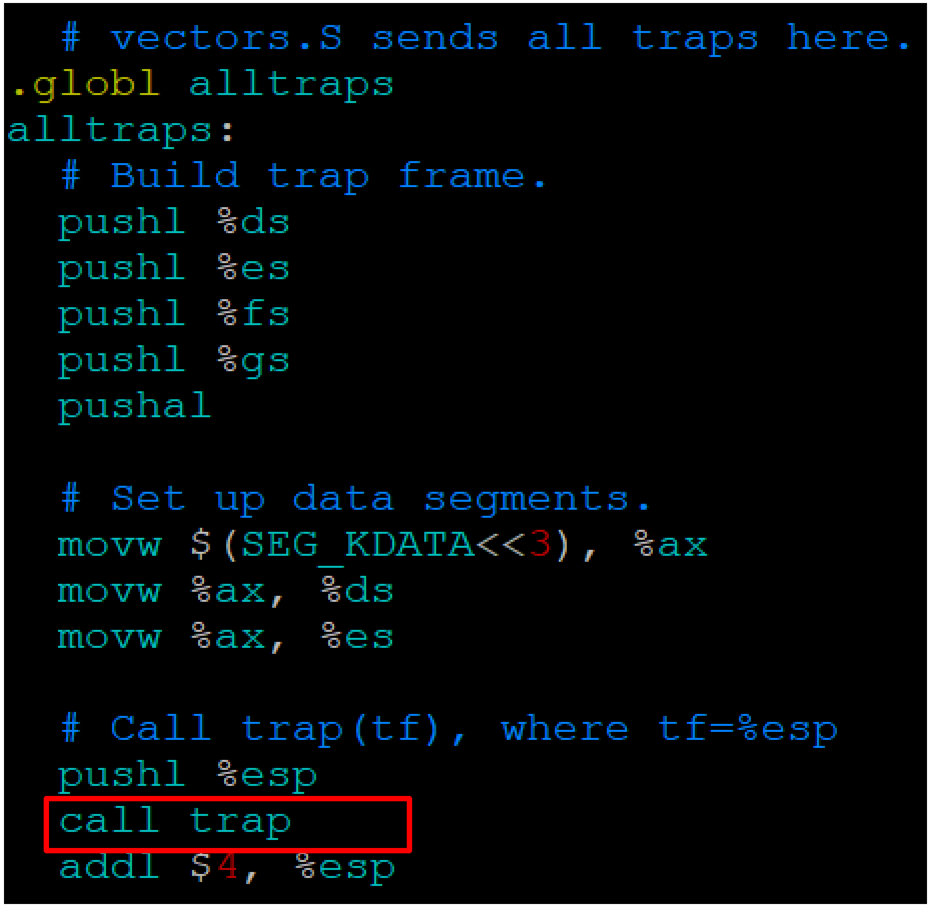
```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```



```
# 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
```

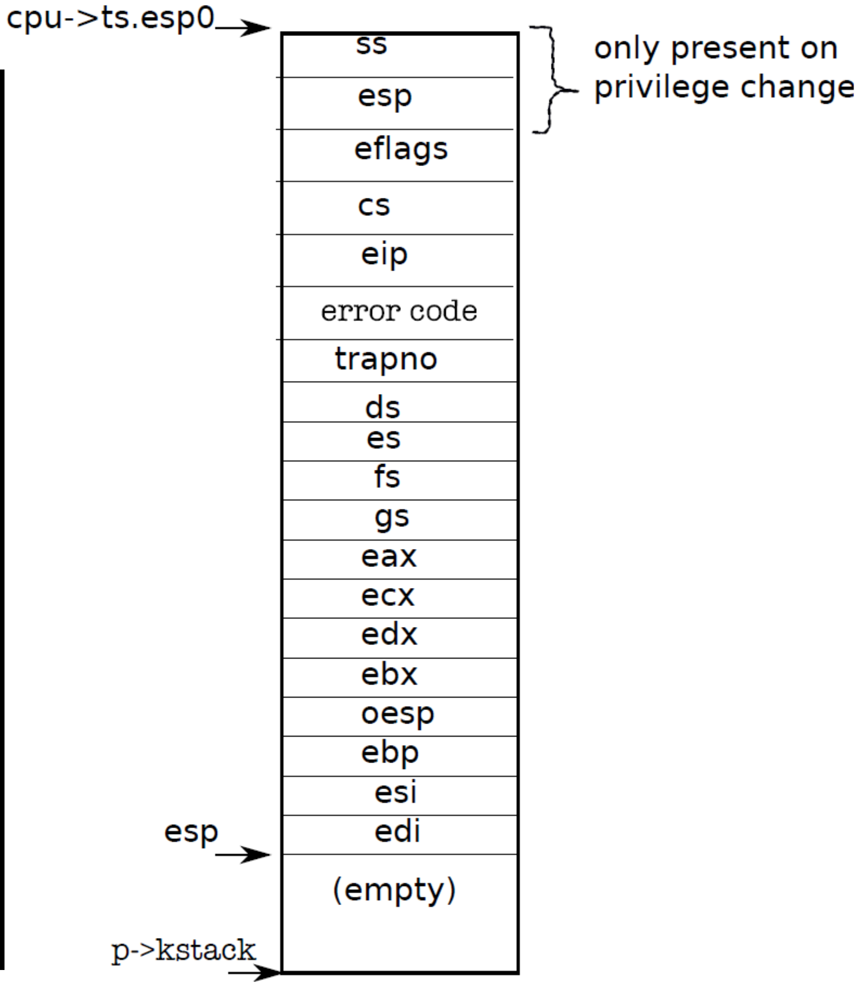
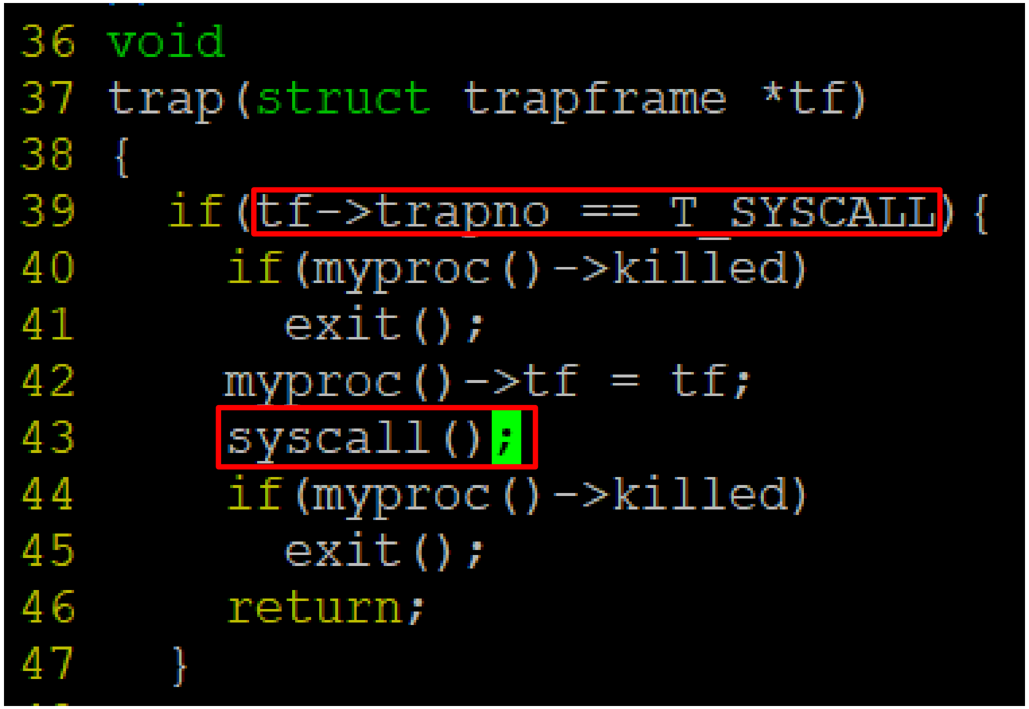


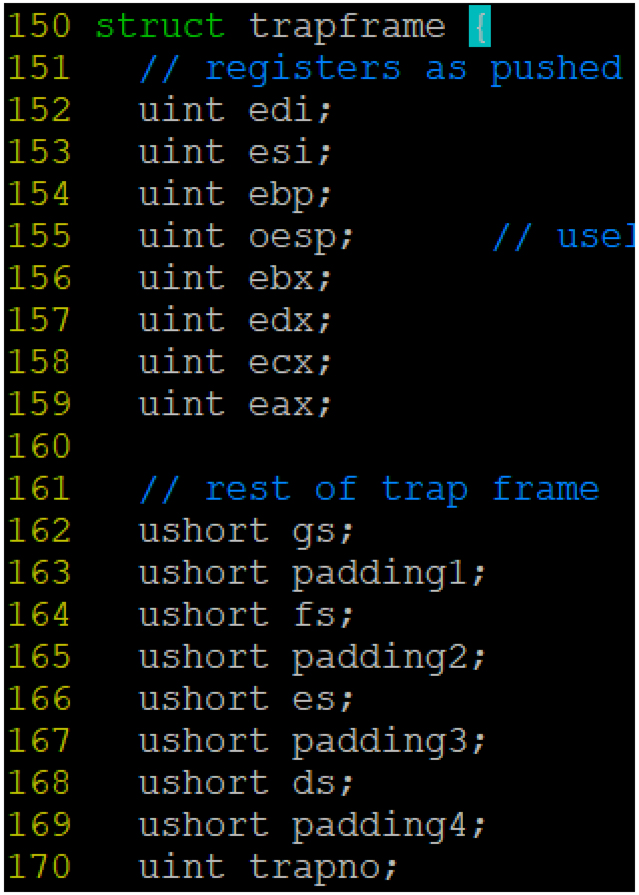
Figure 3-2. The trapframe on the kernel stack

Trap Handler: `trap`

- `trap.c`



```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

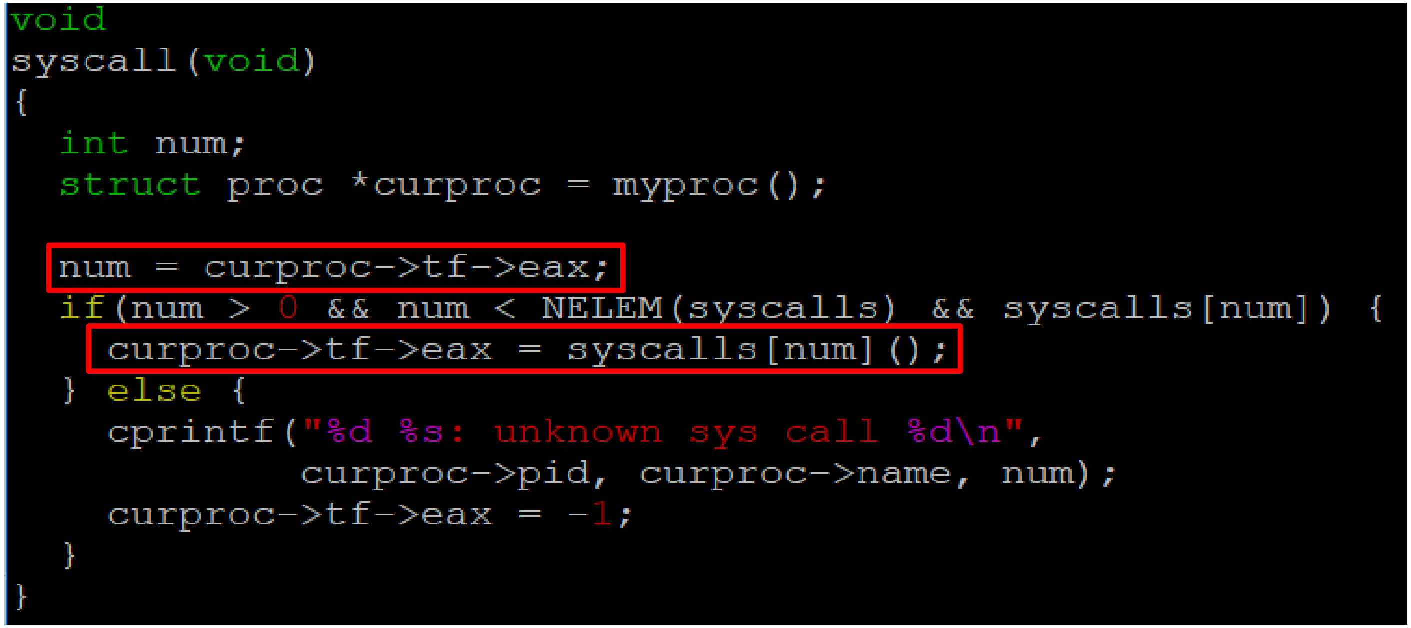


```
struct trapframe {
  // registers as pushed
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // usel
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;
```

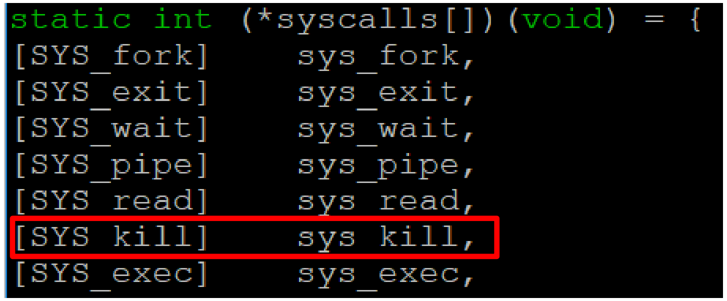
x86.h

System Call Handler: `syscall`

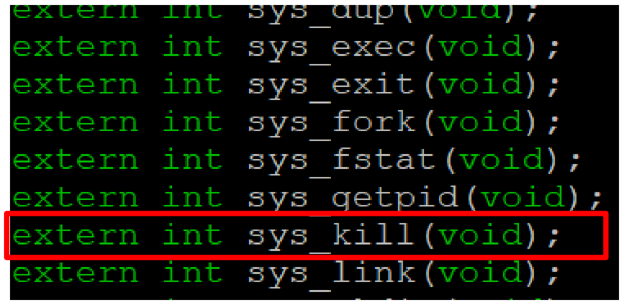
- **`syscall.c`**



```
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,
```

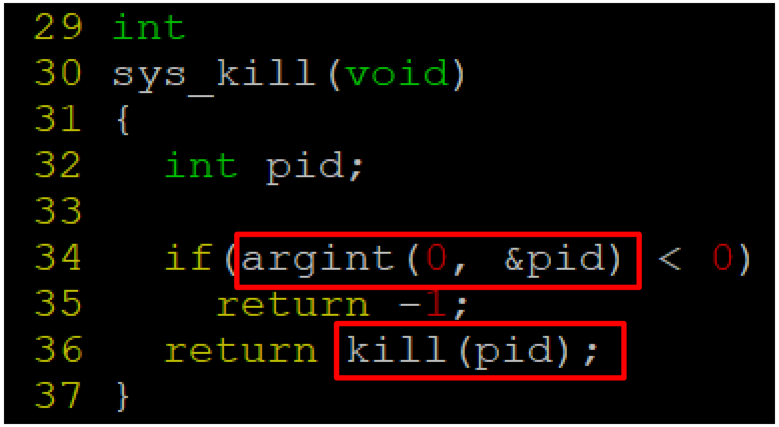


```
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
```

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

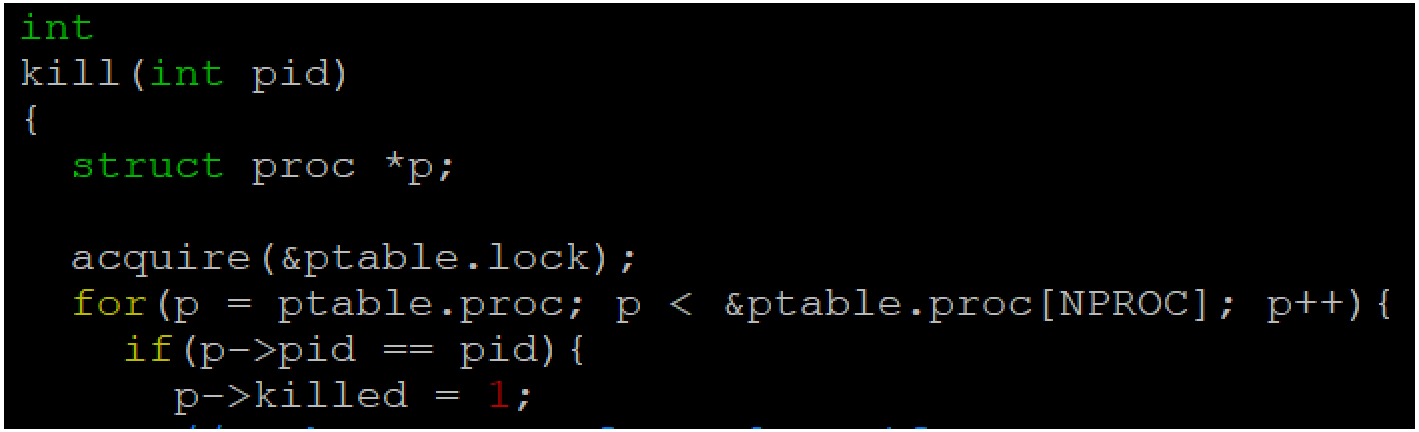
System Call Handler: sys_kill

- sysproc.c



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

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

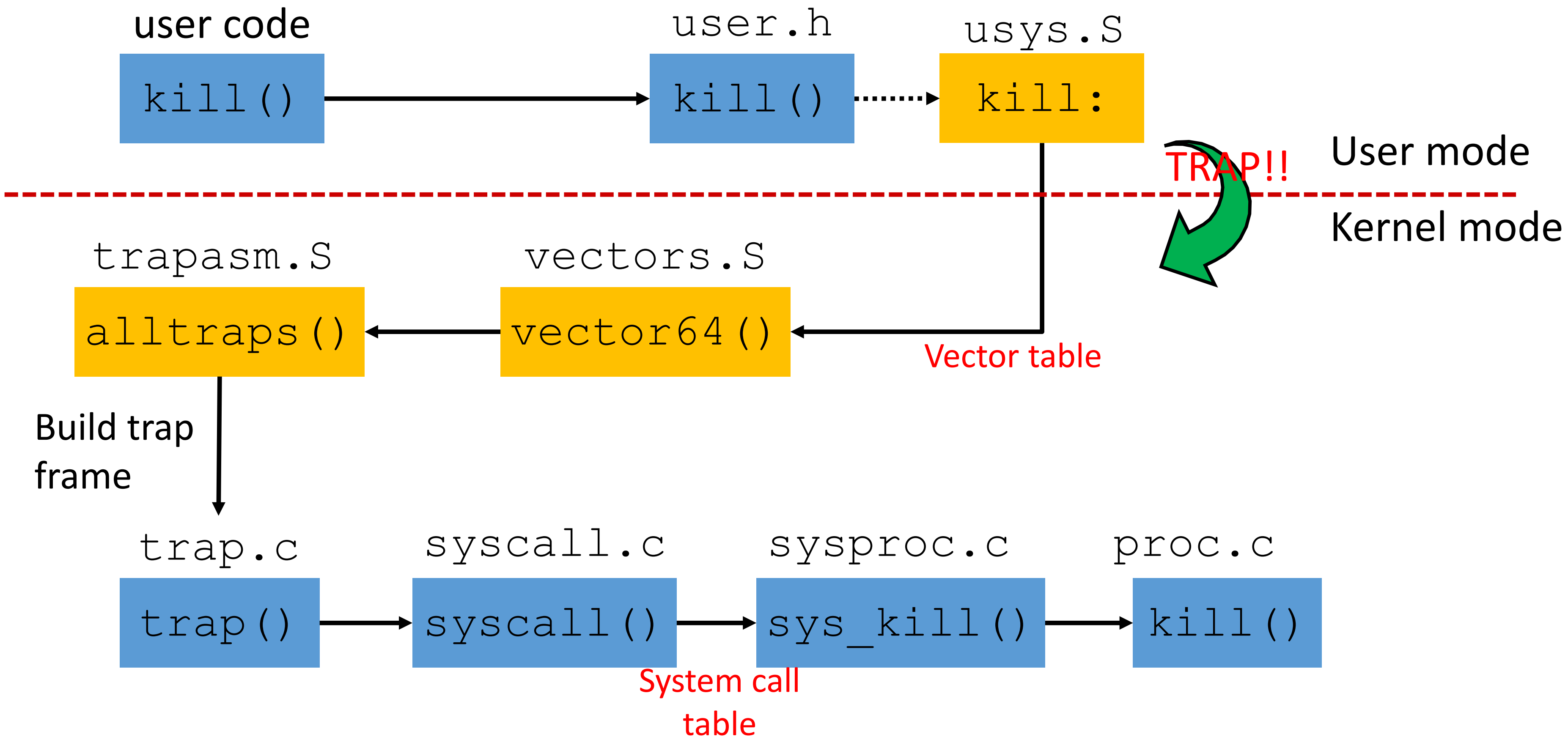
- proc.c



```
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;
```

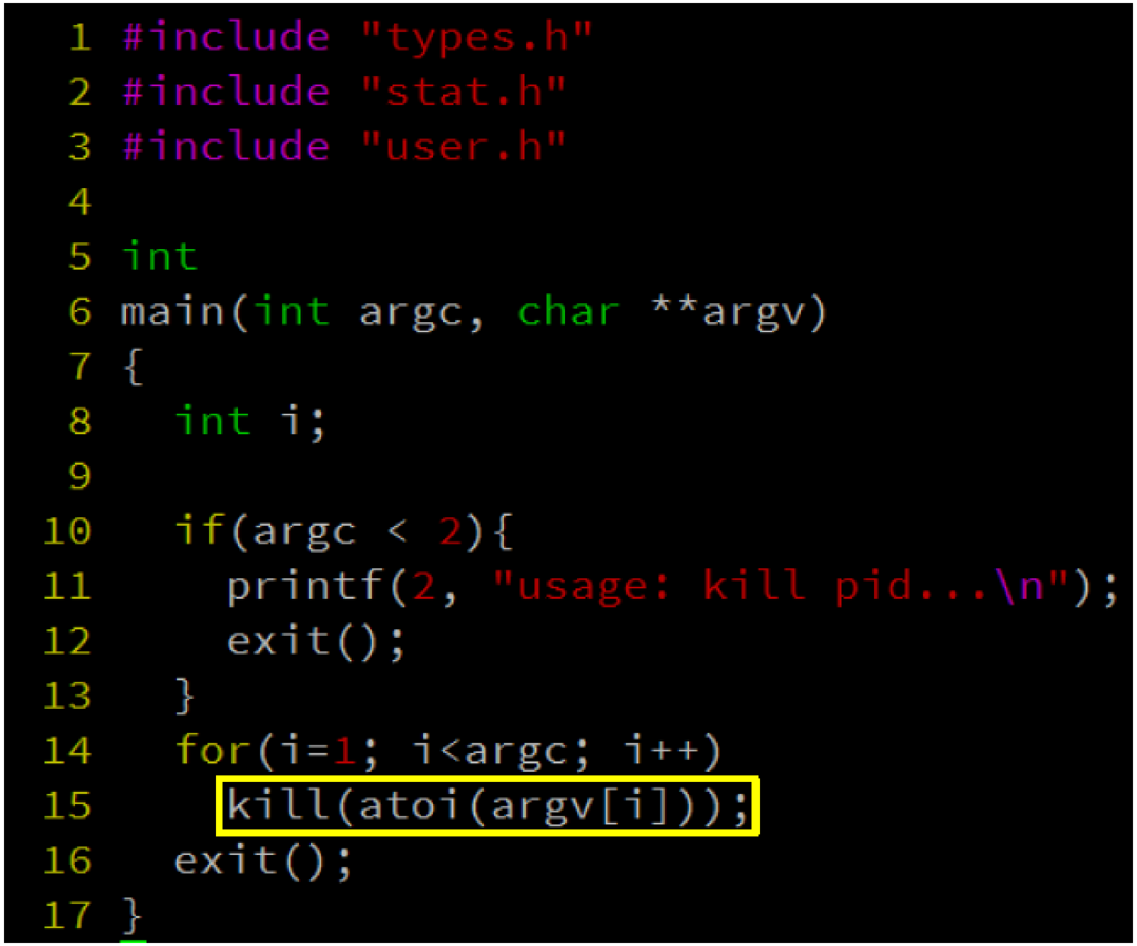
Trap Handling Process on Xv6

- `kill` system call



Test with User Program

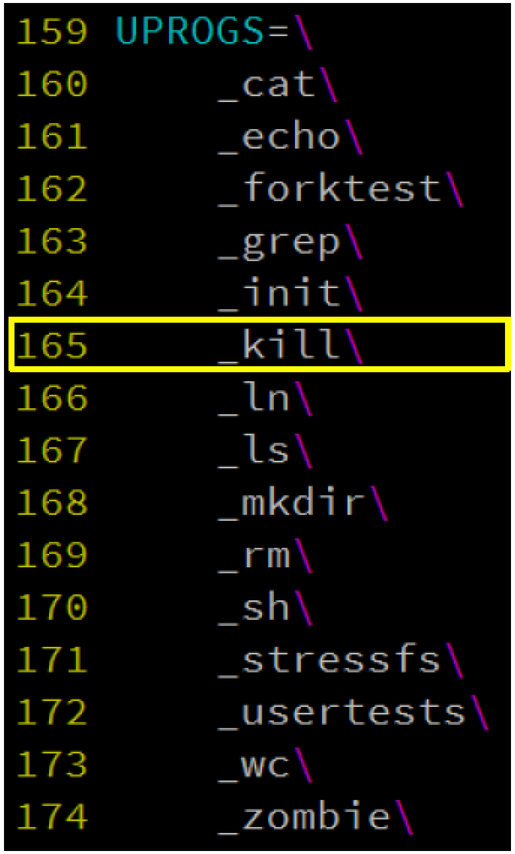
- Example: `kill` system call
- `kill.c`



```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();
}
```

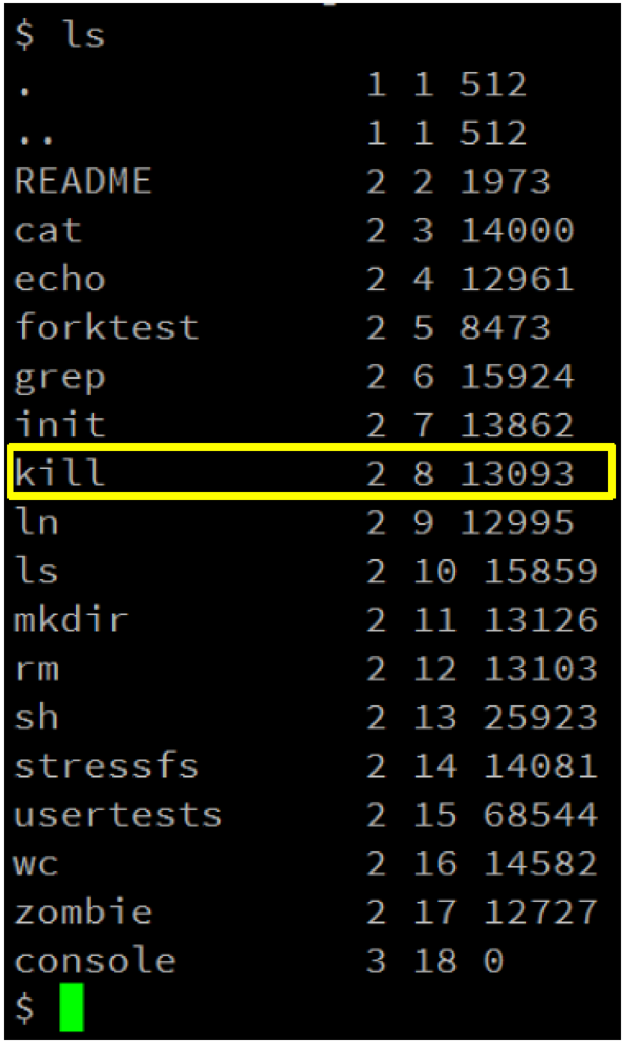
Test with User Program

- **Makefile**



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

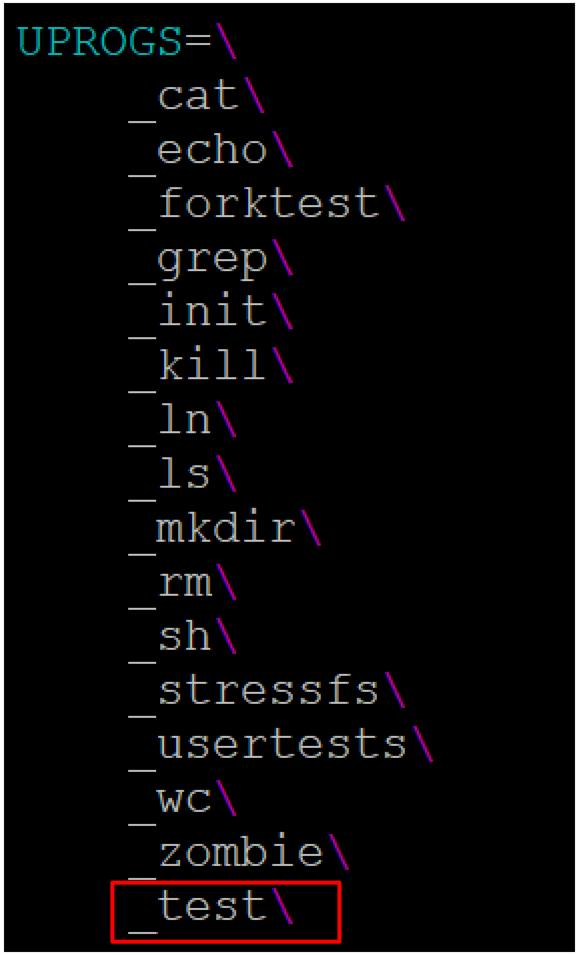
- **xv6**



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

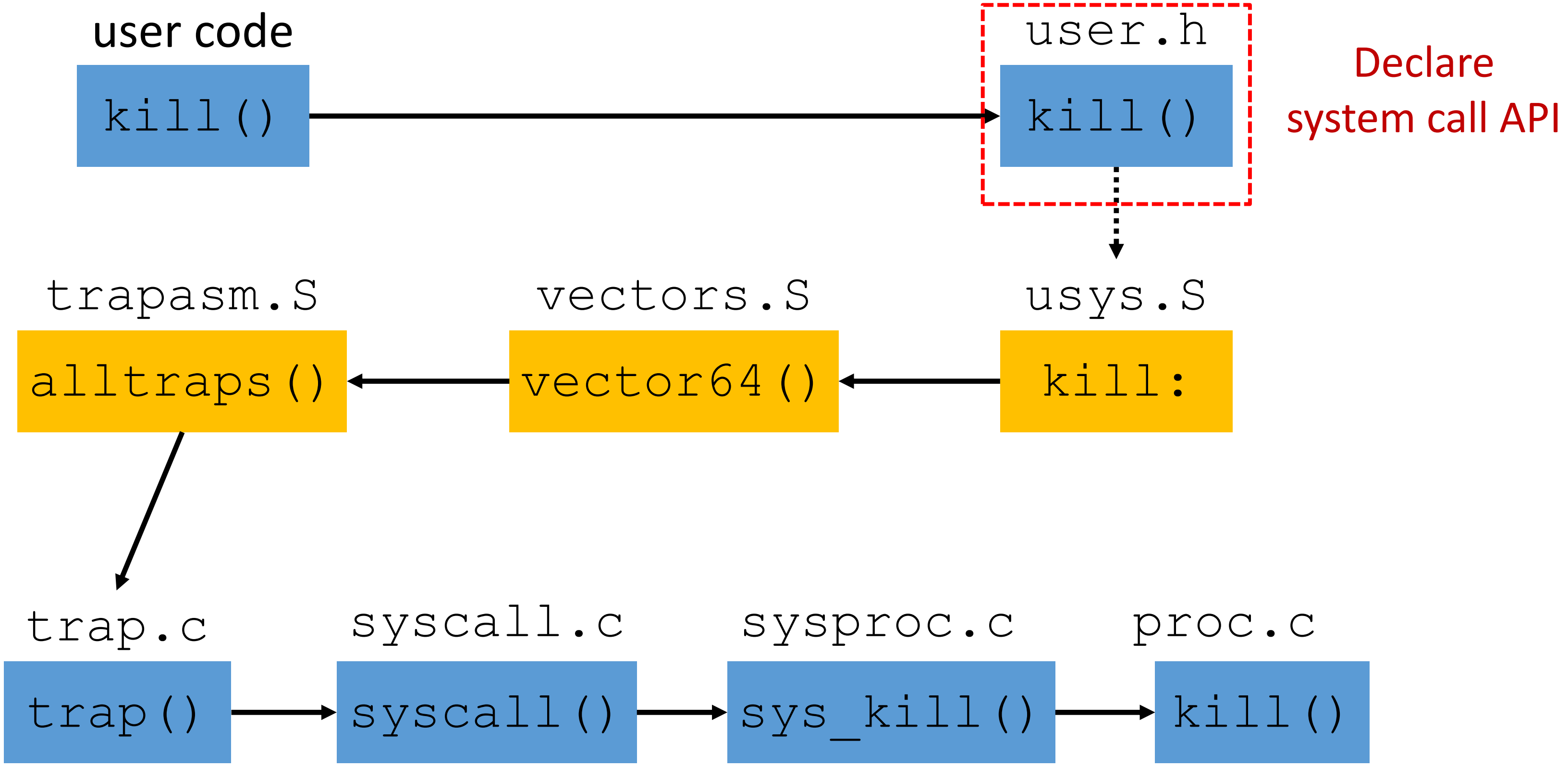
Project #1. Make System Calls

- **`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**

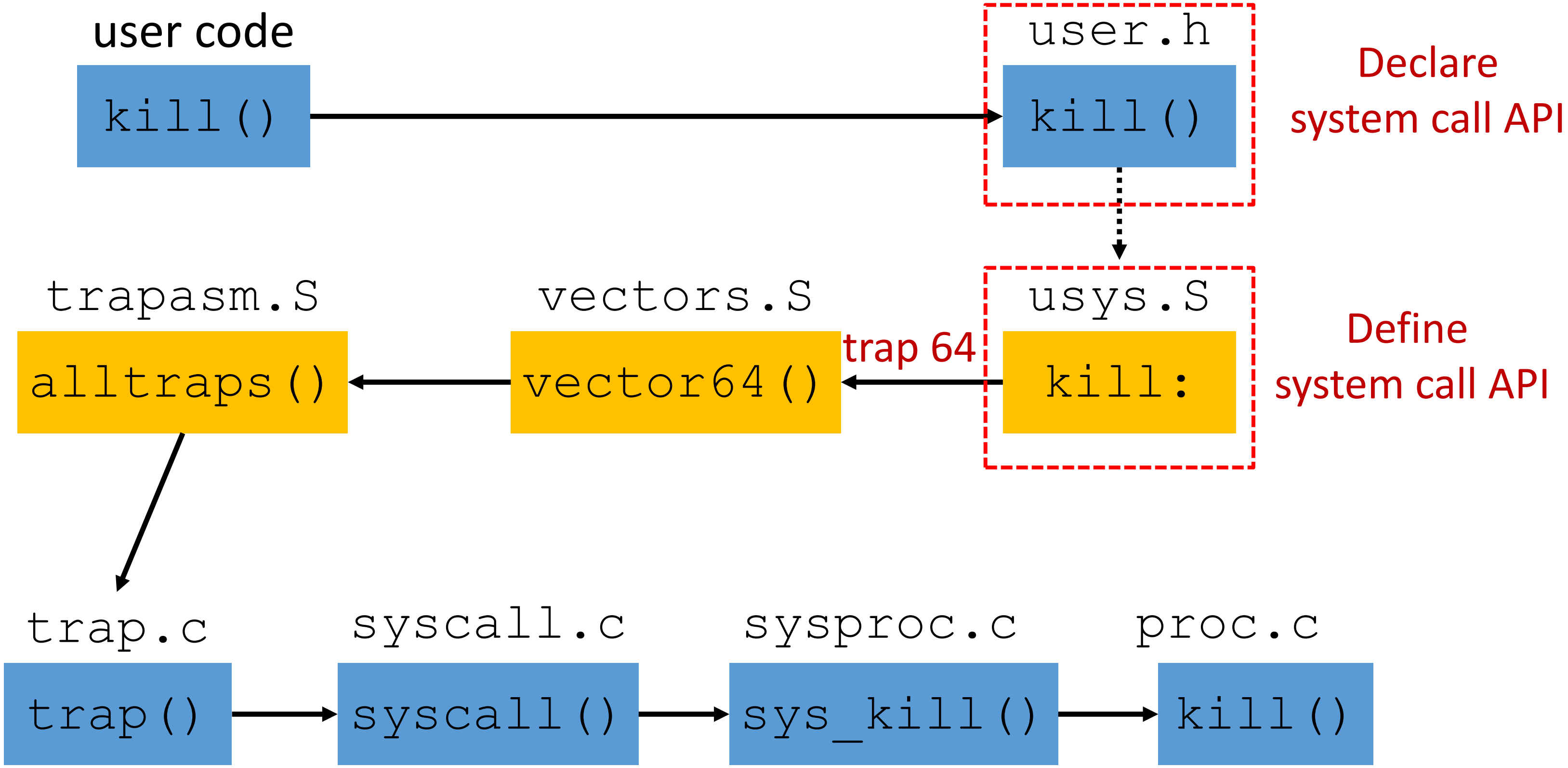
How to Make New System Call on Xv6

- Declare a library routine for your new system call



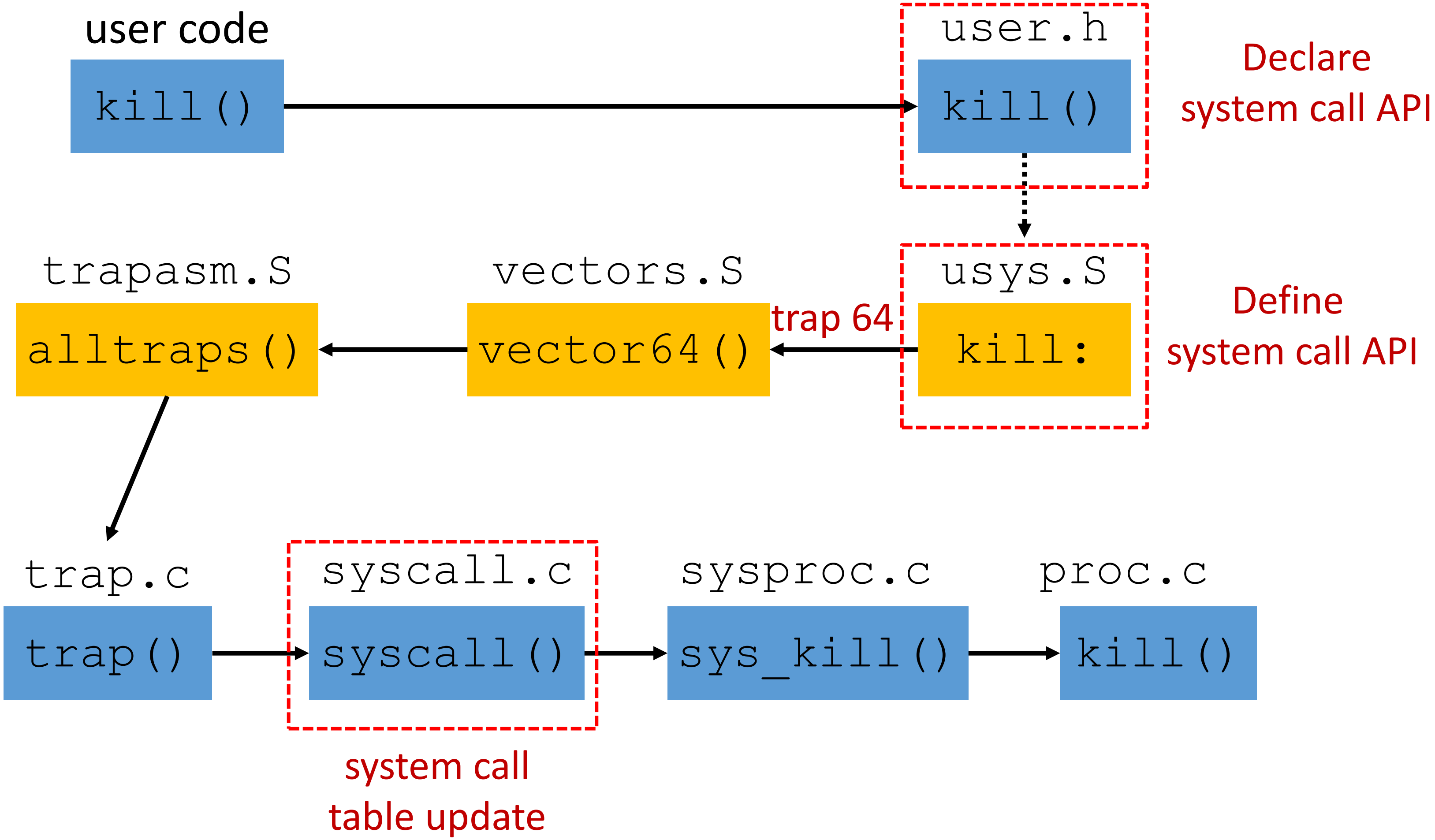
How to Make New System Call on Xv6

- Define a library routine for your new system call



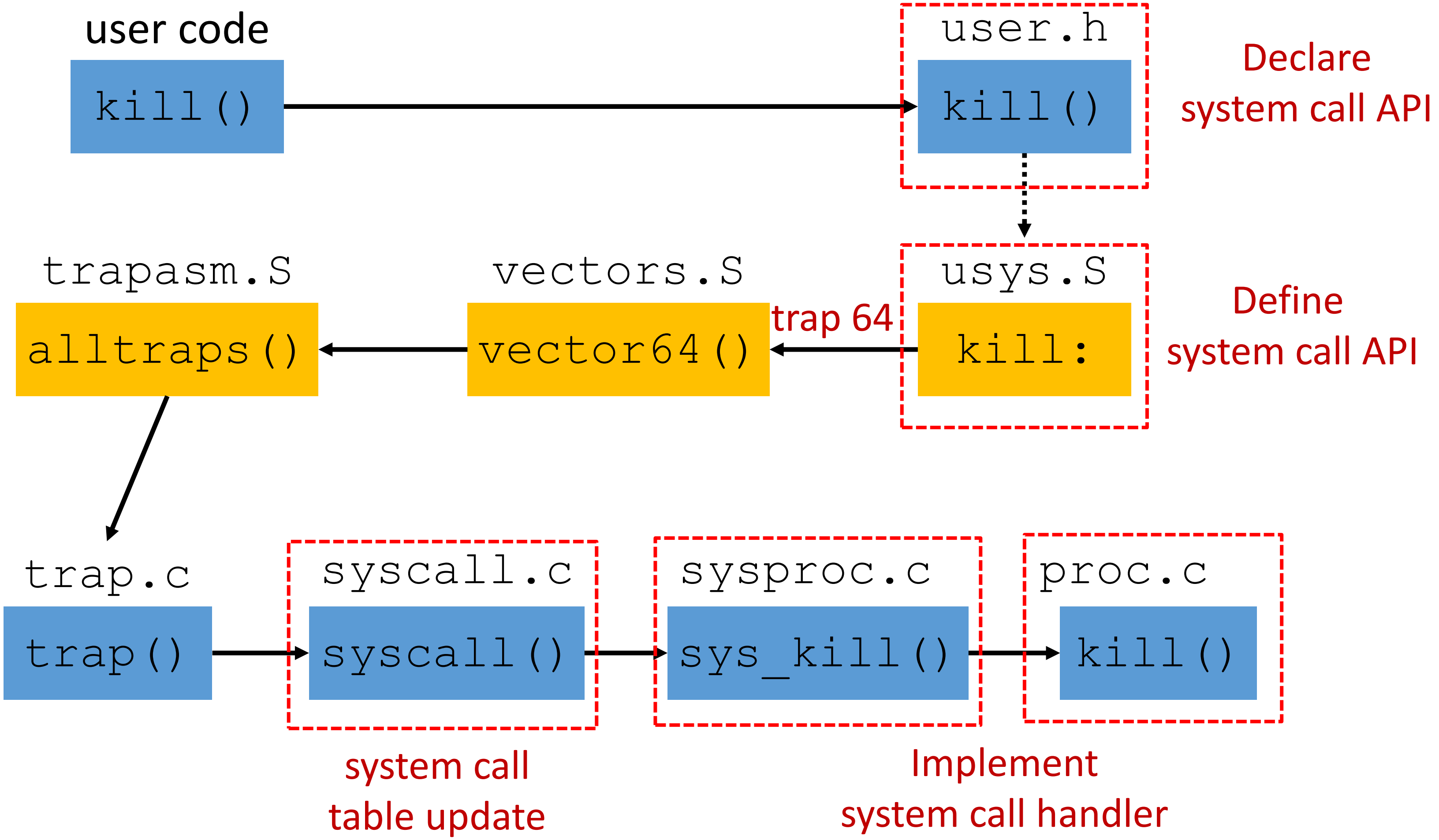
How to Make New System Call on Xv6

- 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: 4/6, 23:59**
  - Late submission penalty: -25% penalty of total mark per day

- **PLEASE DO NOT COPY !!**
  - YOU WILL GET **F** IF YOU COPIED