Operating Systems

Lab. Class

Project #1

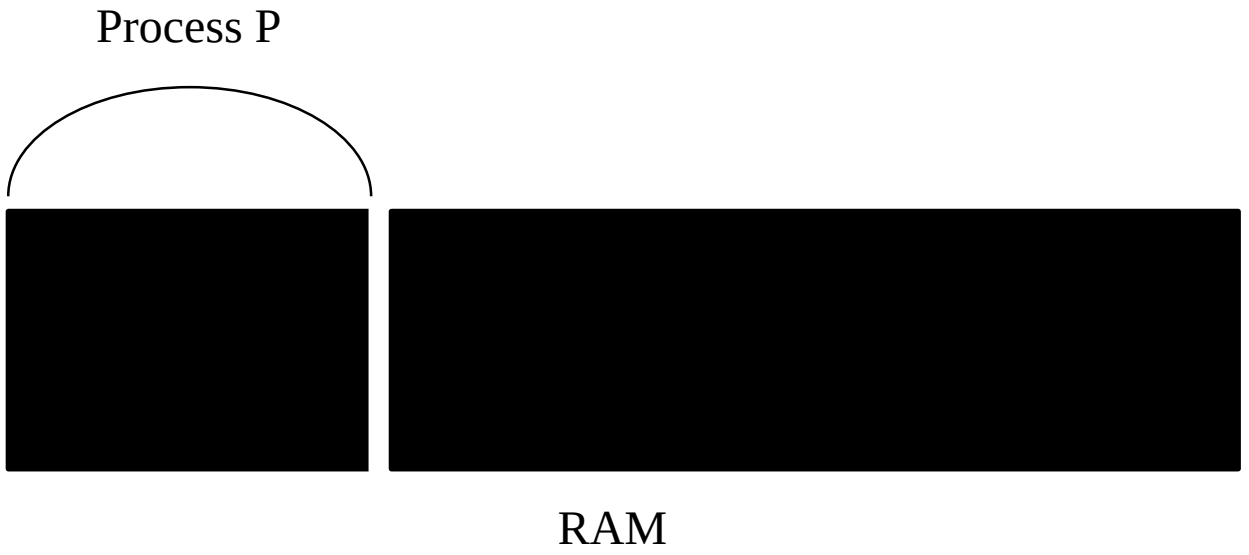
Project Plan

- 5 projects
  0. Install xv6
  1. System call
  2. Scheduling
  3. Virtual memory
  4. Concurrency
  5. File system

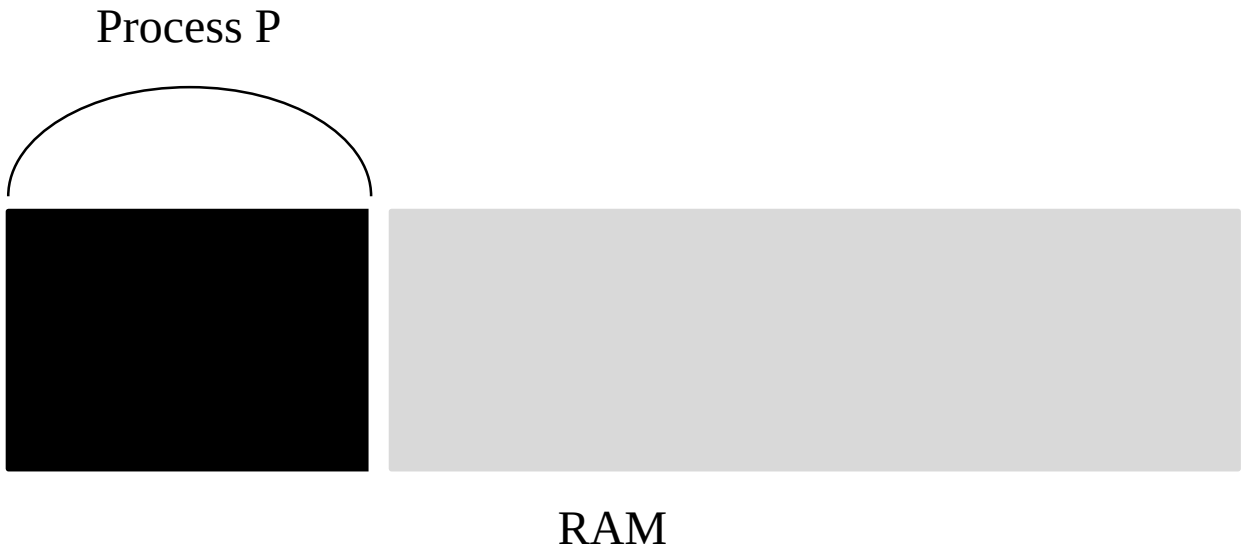
- Individual projects

Trap Handling Process

- Intel architecture

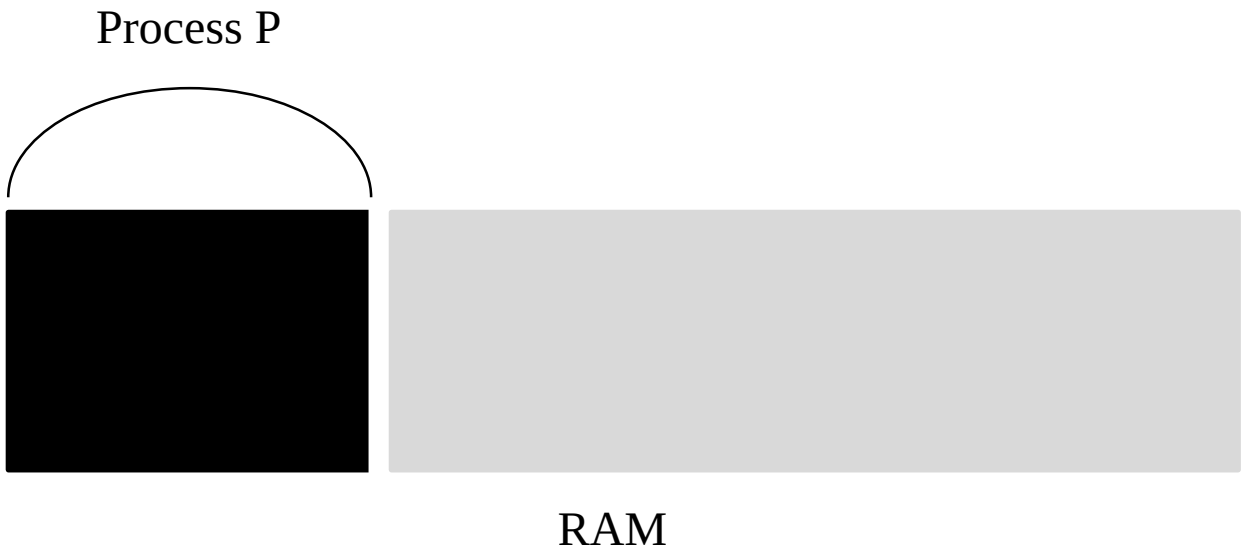


Trap Handling Process (Cont'd)



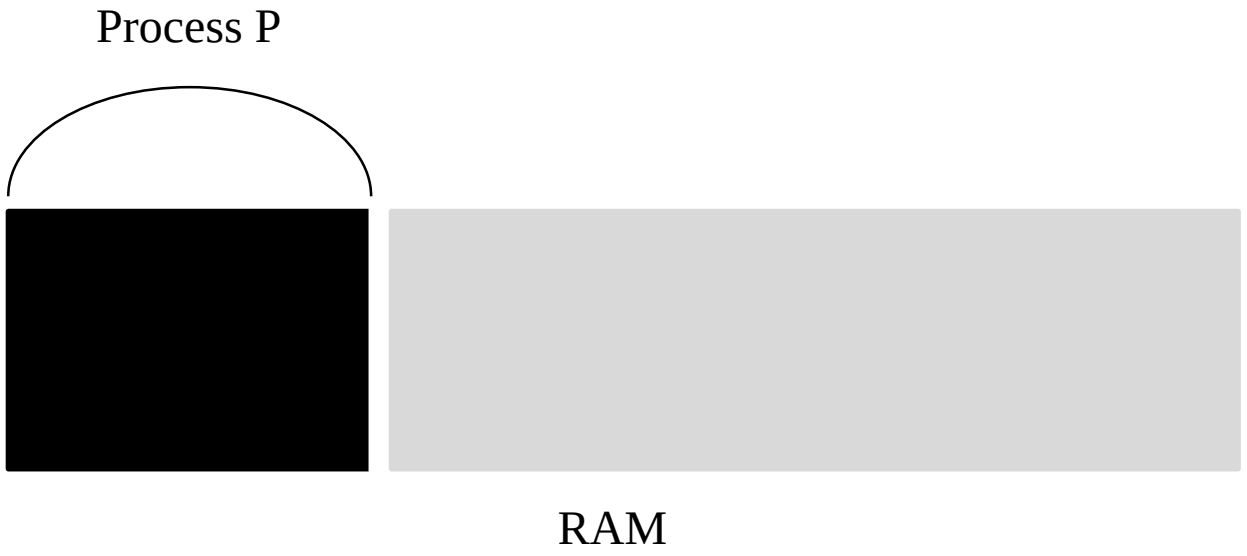
P can only see its own memory because of **user mode**
(other areas, including kernel, are hidden)

Trap Handling Process (Cont'd)



P wants to call kill()

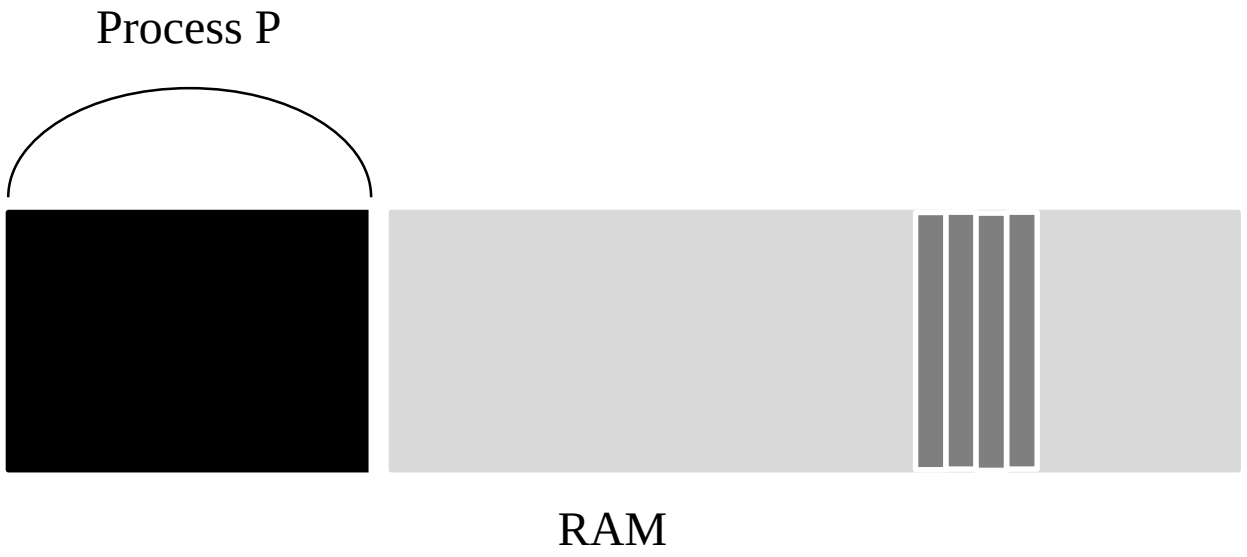
Trap Handling Process (Cont'd)




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

Trap Handling Process (Cont'd)

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

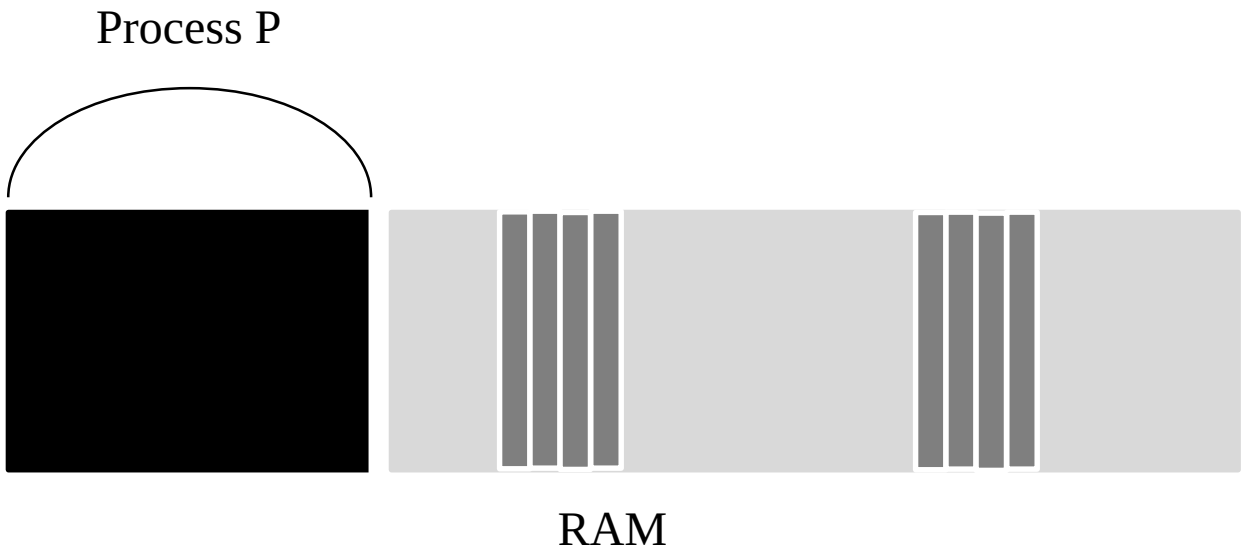



```
movl $6, %eax;    int $64
```

syscall-table index

Trap Handling Process (Cont'd)

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

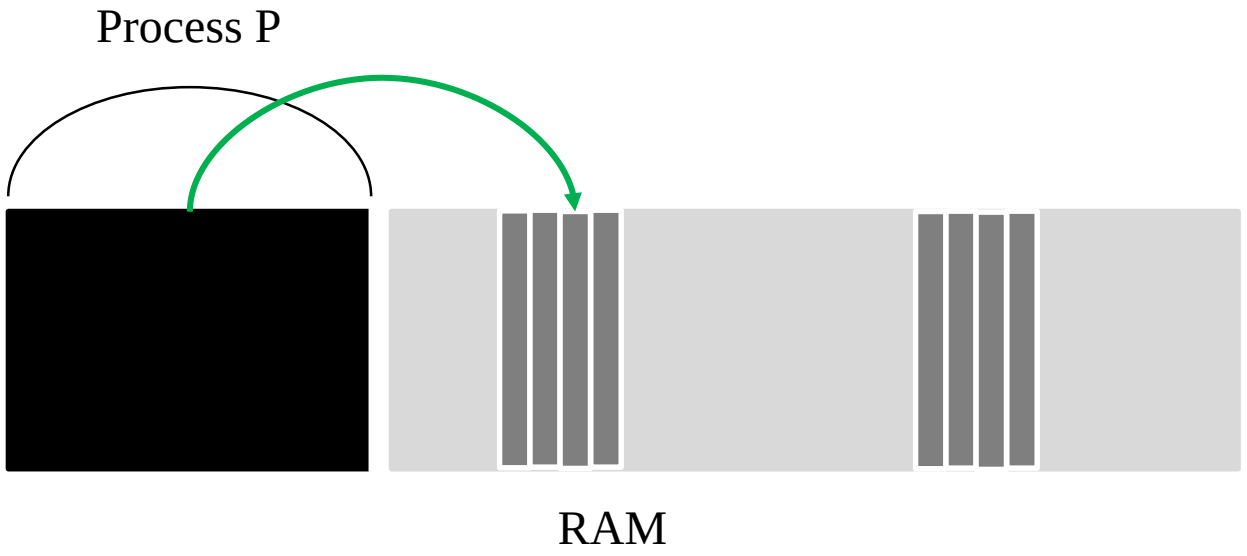



```
movl $6, %eax;     int $64
```

syscall-table index

trap-table index

Trap Handling Process (Cont'd)




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

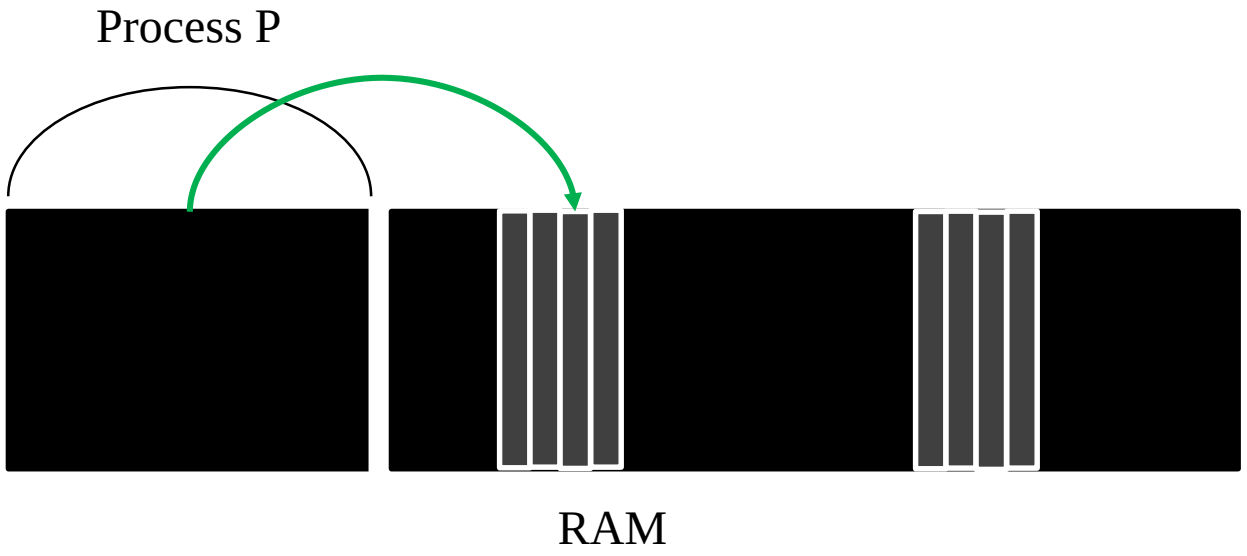
syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)

Kernel mode: we can do anything!



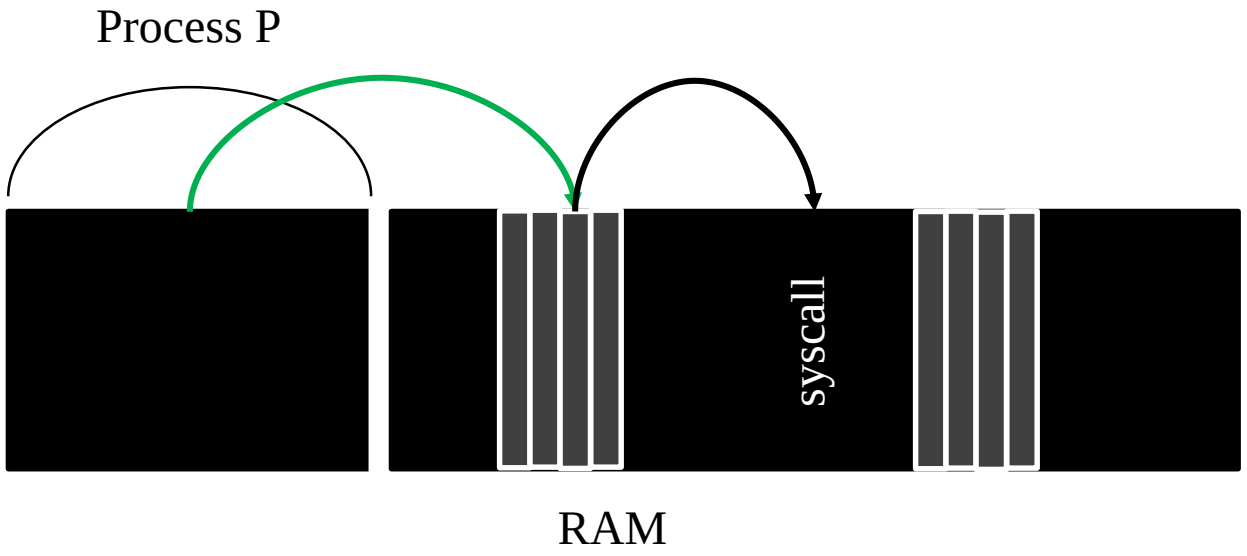

```
movl $6, %eax;
```

syscall-table index

```
int $64
```

trap-table index

Trap Handling Process (Cont'd)

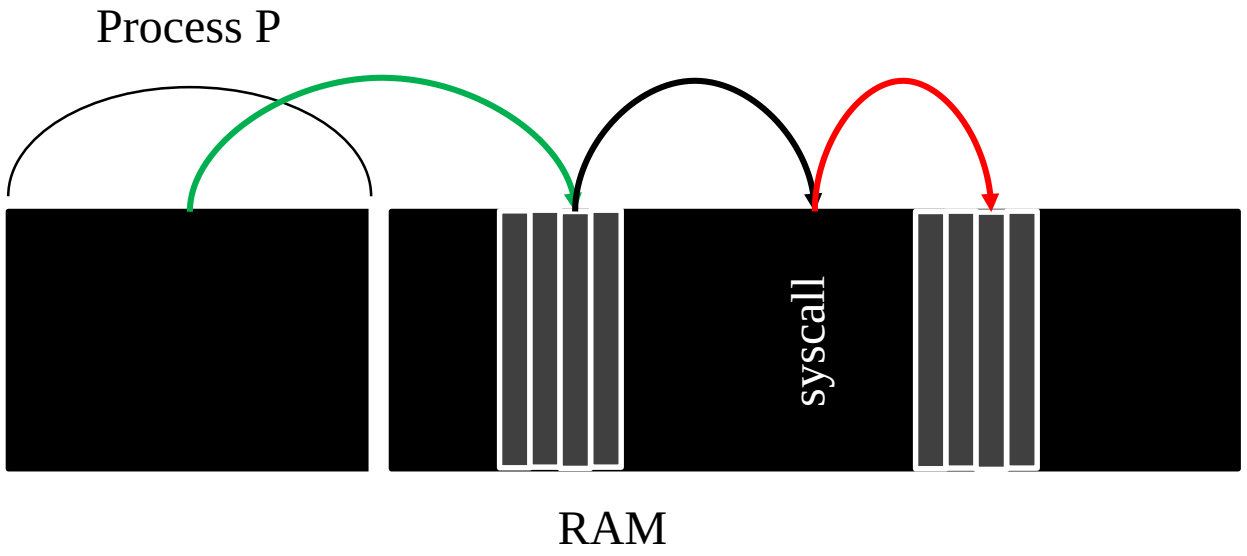



```
movl $6, %eax;     int $64
```

syscall-table index

trap-table index

Trap Handling Process (Cont'd)

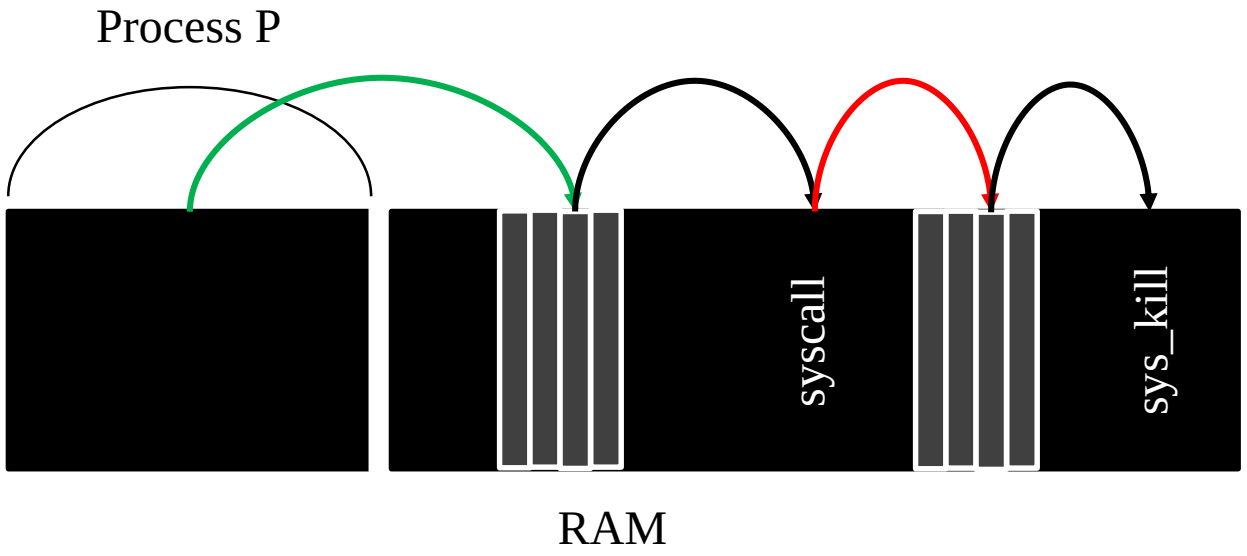



```
movl $6, %eax;     int $64
```

syscall-table index

trap-table index

Trap Handling Process (Cont'd)




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

syscall-table index

```
int $64
```

trap-table index

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   exit();
17 }
```

Trap Handling Process on Xv6 (Cont'd)

- 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 (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
```

Trap Handling Process on Xv6 (Cont'd)

- 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 Handling Process on Xv6 (Cont'd)

- 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
```

Trap Handling Process on Xv6 (Cont'd)

- trap.c

```
11 // Interrupt descriptor table (shared by all CPUs).
12 struct gatedesc idt[256];
13 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
14 struct spinlock tickslock;
15 uint ticks;
16
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 }
```

Trap Handling Process on Xv6 (Cont'd)

- vectors.S

```
 1 # generated by vectors.pl - do not edit
 2 # handlers
 3 .globl alltraps
 4 .globl vector0
 5 vector0:
 6   pushl $0
 7   pushl $0
 8   jmp alltraps
 9 .globl vector1
10 vector1:
11   pushl $0
12   pushl $1
13   jmp alltraps
14 .globl vector2
```

```
1278 # vector table
1279 .data
1280 .globl vectors
1281 vectors:
1282   .long vector0
1283   .long vector1
1284   .long vector2
1285   .long vector3
1286   .long vector4
1287   .long vector5
1288   .long vector6
1289   .long vector7
1290   .long vector8
1291   .long vector9
1292   .long vector10
```

Trap Handling Process on Xv6 (Cont'd)

- trapasm.S

```
 3   # vectors.S sends all traps here.
 4 .globl alltraps
 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 and per-cpu segments.
14   movw $(SEG_KDATA<<3), %ax
15   movw %ax, %ds
16   movw %ax, %es
17   movw $(SEG_KCPU<<3), %ax
18   movw %ax, %fs
19   movw %ax, %gs
20
21   # Call trap(tf), where tf=%esp
22   pushl %esp
23   call trap
24   addl $4, %esp
```

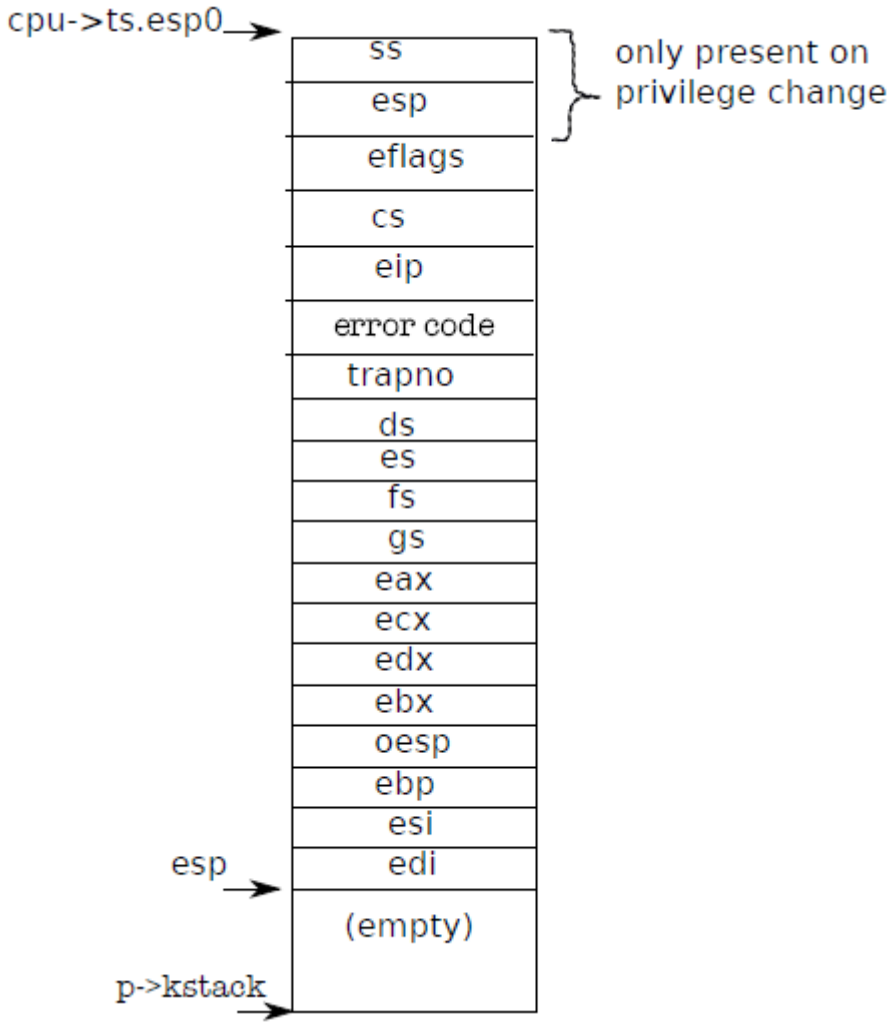


Figure 3-2. The trapframe on the kernel stack

Trap Handling Process on Xv6 (Cont'd)

- trap.c

```
36 void
37 trap(struct trapframe *tf)
38 {
39   if(tf->trapno == T_SYSCALL){
40     if(proc->killed)
41       exit();
42     proc->tf = tf;
43     syscall();
44     if(proc->killed)
45       exit();
46     return;
47   }
```

Trap Handling Process on Xv6 (Cont'd)

- syscall.c

```
102 static int (*syscalls[])(void) = {
103 [SYS_fork]    sys_fork,
104 [SYS_exit]    sys_exit,
105 [SYS_wait]    sys_wait,
106 [SYS_pipe]    sys_pipe,
107 [SYS_read]    sys_read,
108 [SYS_kill]    sys_kill
```

```
 84 extern int sys_exit(void);
 85 extern int sys_fork(void);
 86 extern int sys_fstat(void);
 87 extern int sys_getpid(void);
 88 extern int sys_kill(void);
 89 extern int sys_link(void);
```

```
126 void
127 syscall(void)
128 {
129   int num;
130
131   num = proc->tf->eax;
132   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
133     proc->tf->eax = syscalls[num]();
134   } else {
135     cprintf("%d %s: unknown sys call %d\n",
136             proc->pid, proc->name, num);
137     proc->tf->eax = -1;
138   }
139 }
```

Trap Handling Process on Xv6 (Cont'd)

- 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

```
411 int
412 kill(int pid)
413 {
414   struct proc *p;
415
416   acquire(&ptable.lock);
417   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
418     if(p->pid == pid){
419       p->killed = 1;
```

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 (Cont'd)

- 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
$
```

Process Structure in proc.h

```
49 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
50
51 // Per-process state
52 struct proc {
53   uint sz;                     // Size of process memory (bytes)
54   pde_t* pgdir;                // Page table
55   char *kstack;                // Bottom of kernel stack for this process
56   enum procstate state;        // Process state
57   int pid;                     // Process ID
58   struct proc *parent;         // Parent process
59   struct trapframe *tf;        // Trap frame for current syscall
60   struct context *context;     // swtch() here to run process
61   void *chan;                  // If non-zero, sleeping on chan
62   int killed;                  // If non-zero, have been killed
63   struct file *ofile[NOFILE];  // Open files
64   struct inode *cwd;           // Current directory
65   char name[16];               // Process name (debugging)
66 };
```

Process Dump in proc.c

```
450 //PAGEBREAK: 36
451 // Print a process listing to console.  For debugging.
452 // Runs when user types ^P on console.
453 // No lock to avoid wedging a stuck machine further.
454 void
455 procdump(void)
456 {
457   static char *states[] = {
458   [UNUSED]    "unused",
459   [EMBRYO]    "embryo",
460   [SLEEPING]  "sleep ",
461   [RUNNABLE]  "runble",
462   [RUNNING]   "run   ",
463   [ZOMBIE]    "zombie"
464   };
465   int i;
466   struct proc *p;
467   char *state;
468   uint pc[10];
469
470   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
471     if(p->state == UNUSED)
472       continue;
473     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
474       state = states[p->state];
475     else
476       state = "???";
477     cprintf("%d %s %s", p->pid, state, p->name);
478     if(p->state == SLEEPING){
479       getcallerpcs((uint*)p->context->ebp+2, pc);
480       for(i=0; i<10 && pc[i] != 0; i++)
481         cprintf(" %p", pc[i]);
482     }
483     cprintf("\n");
484   }
485 }
```

Process Dump in proc.c

```
450 //PAGEBREAK: 36
451 // Print a process listing to console.  For debugging.
452 // Runs when user types ^P on console.
453 // No lock to avoid wedging a stuck machine further.
454 void
455 procdump(void)
456 {
457   static char *states[] = {
458   [UNUSED]    "unused",
```

```
kisik@kisik-desktop:~/vm/os/xv6-skku$ make qemu-nox
dd if=/dev/zero of=xv6.img count=10000 status=none
dd if=bootblock of=xv6.img conv=notrunc status=none
dd if=kernel of=xv6.img seek=1 conv=notrunc status=none
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,
media=disk,format=raw -smp 1 -m 512

(process:11089): GLib-WARNING **: /build/glib2.0-prJhLS/glib2.0-2.48.2/./glib/gmem.c:483: custom memory a
llocation vtable not supported
$ 1 sleep  init 80103e0f 80103ea9 80104859 80105899 8010568b
2 sleep  sh 80103dd3 801002c2 80100f6c 80104b52 80104859 80105899 8010568b
```

```
473     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
474       state = states[p->state];
475     else
476       state = "???";
477     cprintf("%d %s %s", p->pid, state, p->name);
478     if(p->state == SLEEPING){
479       getcallerpcs((uint*)p->context->ebp+2, pc);
480       for(i=0; i<10 && pc[i] != 0; i++)
481         cprintf(" %p", pc[i]);
482     }
483     cprintf("\n");
484   }
485 }
```

Project Assignment #1 - Make a System Call

- Goal: make three system calls (getnice, setnice, ps)

- Name
  - getnice, setnice - get and set the nice value
  - ps - print out process(s)'s information

- 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. The range of valid nice value is [0, 40].

- 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.

Project Assignment #1 - Make a System Call (Cont'd)

- 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. The range of valid nice value is [0, 40].
  - The ps function prints out process(s)'s information, which includes **pid**, **nice**, **status**, and **name** 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.

- 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.

Project Template Code

- `git clone https://github.com/jinsoox/xv6-skku.git -b pa1`
- Modifications
  - halt system call
    - Halt xv6 program
  - make tarball
    - Compress your source codes into one .tar.gz file for submission
    - You should enter your ID & project no. on Makefile
  - CPUS=1

```
239 # SKKU operating system
240 PROJECTNUM=1
241 STUDENTID=2011311671 # enter your ID
242
243 # DO NOT EDIT
244 tarball:
245         make clean
246         tar cvzf ../xv6-project-$(PROJECTNUM)-$(STUDENTID).tar.gz .
```

Project Submission Procedure

- http://sys.skku.edu
  - Submit a tarball file made from "**make tarball**"

- Due date
  - 2017-03-26 23:59

- Since 2nd submission, -5% penalty of the project score
- Every one day delay, -25% penalty of the project score
  - You can use up to 5 *slip* days