Operating System

Lab 3 – xv6: System Calls

Hanul Kim

Dept. of Applied Artificial Intelligence

hukim@seoultech.ac.kr

Computer Vision Lab

서울과학기술대학교 컴퓨터비전 연구실

RISC-V Privilege Architecture

Privilege levels

- Privilege levels are used to provide protection between different component of software stack, and attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised

RISC-V privilege levels

- Machine (M)
- Supervisor (S)
- User (U)

RISC-V Privilege Architecture

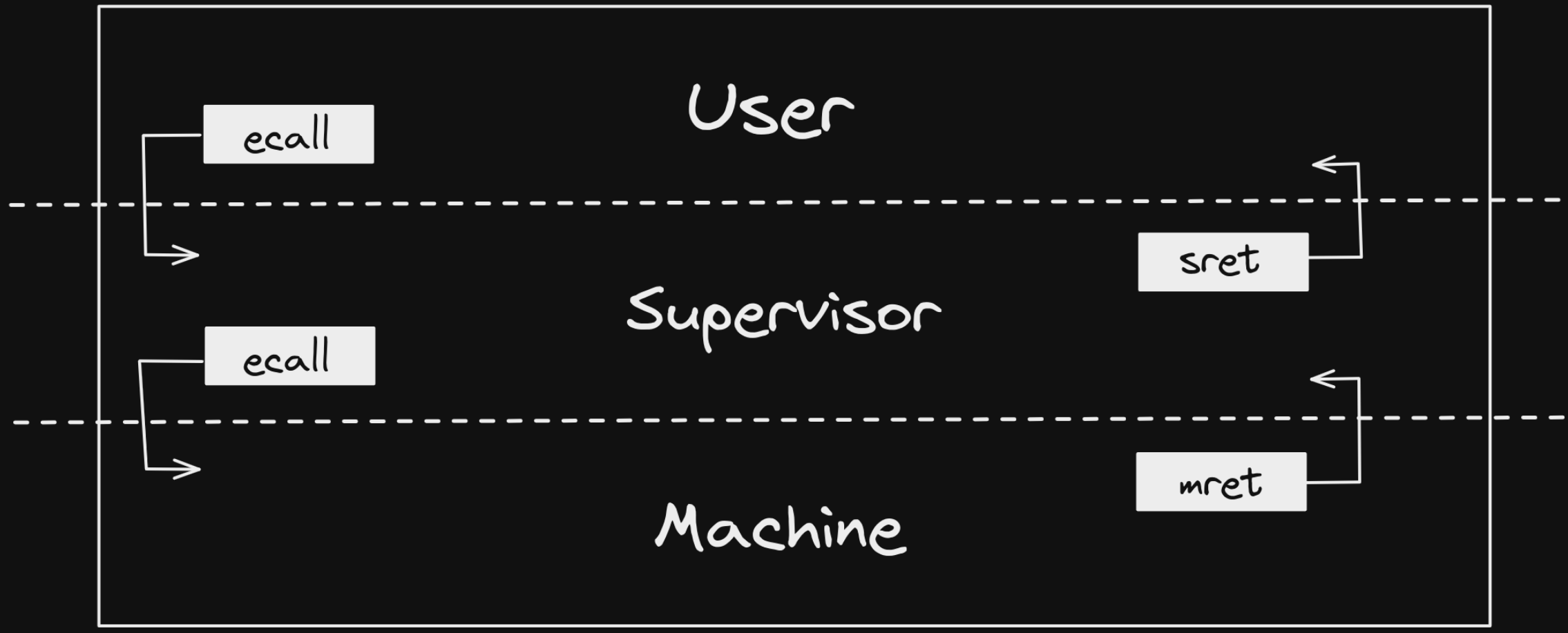
Control and Status Registers (CSR)

- Each privilege mode has some control and status registers CSR

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Configuration			
0x10A	SRW	senvcfg	Supervisor environment configuration register.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.
Debug/Trace Registers			
0x5A8	SRW	scontext	Supervisor-mode context register.

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
0xF15	MRO	mconfigptr	Pointer to configuration data structure.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
0x310	MRW	mstatush	Additional machine status register, RV32 only.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
0x34A	MRW	mtinst	Machine trap instruction (transformed).
0x34B	MRW	mtval2	Machine bad guest physical address.

RISC-V Privilege Levels



xv6 Booting

xv6 boot

1. RISC-V computer runs a boot loader stored in read-only memory
2. The boot loader load the kernel into memory at 0x800000 (no virtual memory yet)
3. kernel/entry.S is executed in machine mode
4. kernel/entry.S sets up a stack so that xv6 can run a C program. Then, kernel/start.c is executed

```
1         # qemu -kernel loads the kernel at 0x80000000
2         # and causes each hart (i.e. CPU) to jump there.
3         # kernel.ld causes the following code to
4         # be placed at 0x80000000.
5   .section .text
6   .global _entry
7   _entry:
8         # set up a stack for C.
9         # stack0 is declared in start.c,
10        # with a 4096-byte stack per CPU.
11        # sp = stack0 + (hartid * 4096)
12        la sp, stack0
13        li a0, 1024*4
14        csrr a1, mhartid
15        addi a1, a1, 1
16        mul a0, a0, a1
17        add sp, sp, a0
18        # jump to start() in start.c
19        call start
20  spin:
21        j spin
```

xv6 Booting

5. kernel/start.c sets up the machine level CSRs. One of important thing is to set a timer for timer interrupt.
6. At the end of kernel/start.c, the privilege level is reduced into the supervised mode by calling mret and jump to kernel/main.c

```
19  // entry.S jumps here in machine mode on stack0.
20  void
21  start()
22  {
23    // set M Previous Privilege mode to Supervisor, for mret.
24    unsigned long x = r_mstatus();
25    x &= ~MSTATUS_MPP_MASK;
26    x |= MSTATUS_MPP_S;
27    w_mstatus(x);
28
29    // set M Exception Program Counter to main, for mret.
30    // requires gcc -mcmodel=medany
31    w_mepc((uint64)main);
32
33    // disable paging for now.
34    w_satp(0);
35
36    // delegate all interrupts and exceptions to supervisor mode.
37    w_medeleg(0xffff);
38    w_mideleg(0xffff);
39    w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);
40
41    // configure Physical Memory Protection to give supervisor mode
42    // access to all of physical memory.
43    w_pmpaddr0(0x3fffffffffffffull);
44    w_pmpcfg0(0xf);
45
46    // ask for clock interrupts.
47    timerinit();
48
49    // keep each CPU's hartid in its tp register, for cpuid().
50    int id = r_mhartid();
51    w_tp(id);
52
53    // switch to supervisor mode and jump to main().
54    asm volatile("mret");
55  }
```

xv6 Booting

7. kernel/main.c prepares a lot of thing that you have learned in the OS class such as
   - Page-based virtual memory systems.
   - Trap handlers for system calls
   - ...

```
10  void
11  main()
12  {
13    if(cpuid() == 0){
14      consoleinit();
15      printfinit();
16      printf("\n");
17      printf("xv6 kernel is booting\n");
18      printf("\n");
19      kinit();         // physical page allocator
20      kvminit();       // create kernel page table
21      kvminithart();   // turn on paging
22      procinit();      // process table
23      trapinit();      // trap vectors
24      trapinithart();  // install kernel trap vector
25      plicinit();      // set up interrupt controller
26      plicinithart();  // ask PLIC for device interrupts
27      binit();         // buffer cache
28      iinit();         // inode table
29      fileinit();      // file table
30      virtio_disk_init(); // emulated hard disk
31      userinit();      // first user process
32      __sync_synchronize();
33      started = 1;
34    } else {
35      while(started == 0)
36        ;
37      __sync_synchronize();
38      printf("hart %d starting\n", cpuid());
39      kvminithart();    // turn on paging
40      trapinithart();   // install kernel trap vector
41      plicinithart();   // ask PLIC for device interrupts
42    }
43
44    scheduler();
45  }
```

xv6 Booting

8. kernel/main.c creates the first user process called init

```
218  // a user program that calls exec("/init")
219  // assembled from ../user/initcode.S
220  // od -t xC ../user/initcode
221  uchar initcode[] = {
222    0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,
223    0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,
224    0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,
225    0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,
226    0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,
227    0x74, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,
228    0x00, 0x00, 0x00, 0x00
229  };
```

```
1   # Initial process that execs /init.
2   # This code runs in user space.
3
4   #include "syscall.h"
5
6   # exec(init, argv)
7   .globl start
8   start:
9           la a0, init
10          la a1, argv
11          li a7, SYS_exec
12          ecall
```

```
231  // Set up first user process.
232  void
233  userinit(void)
234  {
235    struct proc *p;
236
237    p = allocproc();
238    initproc = p;
239
240    // allocate one user page and copy initcode's instructions
241    // and data into it.
242    uvmfirst(p->pagetable, initcode, sizeof(initcode));
243    p->sz = PGSIZE;
244
245    // prepare for the very first "return" from kernel to user.
246    p->trapframe->epc = 0;      // user program counter
247    p->trapframe->sp = PGSIZE;  // user stack pointer
248
249    safestrcpy(p->name, "initcode", sizeof(p->name));
250    p->cwd = namei("/");
251
252    p->state = RUNNABLE;
253
254    release(&p->lock);
255  }
```

xv6 Booting

9. The init process creates the shell process; sh


```
14  int
15  main(void)
16  {
17    int pid, wpid;
18
19    if(open("console", O_RDWR) < 0){
20      mknod("console", CONSOLE, 0);
21      open("console", O_RDWR);
22    }
23    dup(0);  // stdout
24    dup(0);  // stderr
25
26    for(;;){
27      printf("init: starting sh\n");
28      pid = fork();
29      if(pid < 0){
30        printf("init: fork failed\n");
31        exit(1);
32      }
33      if(pid == 0){
34        exec("sh", argv);
35        printf("init: exec sh failed\n");
36        exit(1);
37      }
38
39      for(;;){
40        // this call to wait() returns if the shell exits,
41        // or if a parentless process exits.
42        wpid = wait((int *) 0);
43        if(wpid == pid){
44          // the shell exited; restart it.
45          break;
46        } else if(wpid < 0){
47          printf("init: wait returned an error\n");
48          exit(1);
49        } else {
50          // it was a parentless process; do nothing.
51        }
52      }
53    }
54  }
```

xv6: System calls

Exceptions

- Events other than branches that changes the normal flow of instruction execution

Terminology (RISC-V Convention)

- Interrupt: External asynchronous event
- Exception: An unusual condition occurring at run time associated with an instruction
- Trap: The transfer of control to a trap handler caused by either an exception or an interrupt

xv6: System calls

Supervisor CSRs for trap handling

- stvec: The kerenl write the address of its trap handler here. When the trap instruction (ecall) is called, the RISC-V CPU jumps to the address in stvec to handle a trap
  - For traps from userspace, stvec stores the address of uservec (kernel/trampoline.S)
  - For traps from kernelspace, stvec store the address of kernelvec (kernel/kernelvec.S)
- sepc: When a trap occur, RISC-V save the current PC address here
- scause: RISC-V puts a number here to describe the reason for the trap
  - system calls, exceptions, interrupts

xv6: System calls

Flow of exec system call in initcode.S

1. trap instruction (ecall)

- Do many things …

1. If the trap is a device interrupt, and the `sstatus` SIE bit is clear, don't do any of the following.
2. Disable interrupts by clearing the SIE bit in `sstatus`.
3. Copy the `pc` to `sepc`.
4. Save the current mode (user or supervisor) in the SPP bit in `sstatus`.
5. Set `scause` to reflect the trap's cause.

- Set the mode to supervisor
- Copy stvec to the pc (jump to uservec)

```
1   # Initial process that execs /init.
2   # This code runs in user space.
3
4   #include "syscall.h"
5
6   # exec(init, argv)
7   .globl start
8   start:
9           la a0, init
10          la a1, argv
11          li a7, SYS_exec
12          ecall
```

xv6: System calls

2. uservec (kernel/trampoline.S)

- Save the user registers to kernel stack
- Do many things …
- Jump to usertrap (kernel/trap.c)

```
21  uservec:
22          #
23          # trap.c sets stvec to point here, so
24          # traps from user space start here,
25          # in supervisor mode, but with a
26          # user page table.
27          #
28
29          # save user a0 in sscratch so
30          # a0 can be used to get at TRAPFRAME.
31          csrw sscratch, a0
32
33          # each process has a separate p->trapframe memory area,
34          # but it's mapped to the same virtual address
35          # (TRAPFRAME) in every process's user page table.
36          li a0, TRAPFRAME
37
38          # save the user registers in TRAPFRAME
39          sd ra, 40(a0)
40          sd sp, 48(a0)
41          sd gp, 56(a0)
42          sd tp, 64(a0)
43          sd t0, 72(a0)
44          sd t1, 80(a0)
45          sd t2, 88(a0)
46          sd s0, 96(a0)
47          sd s1, 104(a0)
48          sd a1, 120(a0)
49          sd a2, 128(a0)
50          sd a3, 136(a0)
51          sd a4, 144(a0)
52          sd a5, 152(a0)
53          sd a6, 160(a0)
54          sd a7, 168(a0)
55          sd s2, 176(a0)
56          sd s3, 184(a0)
57          sd s4, 192(a0)
```

xv6: System calls

3. usertrap (kernel/trap.c)

- If the cause of the trap is a system call, then call syscall (kernel/syscall.c)

```
36  void
37  usertrap(void)
38  {
39    int which_dev = 0;
40
41    if((r_sstatus() & SSTATUS_SPP) != 0)
42      panic("usertrap: not from user mode");
43
44    // send interrupts and exceptions to kerneltrap(),
45    // since we're now in the kernel.
46    w_stvec((uint64)kernelvec);
47
48    struct proc *p = myproc();
49
50    // save user program counter.
51    p->trapframe->epc = r_sepc();
52
53    if(r_scause() == 8){
54      // system call
55
56      if(killed(p))
57        exit(-1);
58
59      // sepc points to the ecall instruction,
60      // but we want to return to the next instruction.
61      p->trapframe->epc += 4;
62
63      // an interrupt will change sepc, scause, and sstatus,
64      // so enable only now that we're done with those registers.
65      intr_on();
66
67      syscall();
68    } else if((which_dev = devintr()) != 0){
69      // ok
70    } else {
71      printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid)
72      printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
73      setkilled(p);
74    }
75
```

xv6: System calls

4. syscall (kernel/syscall.c)

- Run exec syscall handler

```
131  void
132  syscall(void)
133  {
134    int num;
135    struct proc *p = myproc();
136
137    num = p->trapframe->a7;
138    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139      // Use num to lookup the system call function for num, call it,
140      // and store its return value in p->trapframe->a0
141      p->trapframe->a0 = syscalls[num]();
142    } else {
143      printf("%d %s: unknown sys call %d\n",
144              p->pid, p->name, num);
145      p->trapframe->a0 = -1;
146    }
147  }
```

```
434  uint64
435  sys_exec(void)
436  {
437    char path[MAXPATH], *argv[MAXARG];
438    int i;
439    uint64 uargv, uarg;
440
441    argaddr(1, &uargv);
442    if(argstr(0, path, MAXPATH) < 0) {
443      return -1;
444    }
445    memset(argv, 0, sizeof(argv));
446    for(i=0;; i++){
447      if(i >= NELEM(argv)){
448        goto bad;
449      }
450      if(fetchaddr(uargv+sizeof(uint64)*i, (uint64*)&uarg) < 0){
451        goto bad;
452      }
453      if(uarg == 0){
454        argv[i] = 0;
455        break;
456      }
457      argv[i] = kalloc();
458      if(argv[i] == 0)
459        goto bad;
460      if(fetchstr(uarg, argv[i], PGSIZE) < 0)
461        goto bad;
462    }
463
464    int ret = exec(path, argv);
465
466    for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
467      kfree(argv[i]);
468
469    return ret;
470
471   bad:
472    for(i = 0; i < NELEM(argv) && argv[i] != 0; i++)
473      kfree(argv[i]);
474    return -1;
475  }
```

xv6: System calls

5. usertrap (kernel/trap.c)

- Call usertrapret (kernel/trampoline.S)

```
53    if(r_scause() == 8){
54      // system call
55
56      if(killed(p))
57        exit(-1);
58
59      // sepc points to the ecall instruction,
60      // but we want to return to the next instruction.
61      p->trapframe->epc += 4;
62
63      // an interrupt will change sepc, scause, and sstatus,
64      // so enable only now that we're done with those registers.
65      intr_on();
66
67      syscall();
68    } else if((which_dev = devintr()) != 0){
69      // ok
70    } else {
71      printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid)
72      printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
73      setkilled(p);
74    }
75
76    if(killed(p))
77      exit(-1);
78
79    // give up the CPU if this is a timer interrupt.
80    if(which_dev == 2)
81      yield();
82
83    usertrapret();
84  }
```

xv6: System calls

6. usertrapret (kernel/trampoline.S)

- Load the user registers from the kernel stack
- Do many things …
- Return-from-trap (sret)

```
100  .globl userret
101  userret:
102          # userret(pagetable)
103          # called by usertrapret() in trap.c to
104          # switch from kernel to user.
105          # a0: user page table, for satp.
106
107          # switch to the user page table.
108          sfence.vma zero, zero
109          csrw satp, a0
110          sfence.vma zero, zero
                                  ...
149          # return to user mode and user pc.
150          # usertrapret() set up sstatus and sepc.
151          sret
```

Adding xv6: System calls

We don’t discuss the implementation.

But there are some interesting points

- When the user program ps.c requests a system call, where is the ecall?
- The answer is usys.S, which usys.pl creates at the build time (make qemu). Note that user/user.h contains the prototypes of functions in usys.S, not in the functions in kernel/*

```
1  struct stat;
2
3  // system calls
4  int fork(void);
5  int exit(int) __attribute__((noreturn));
6  int wait(int*);
7  int pipe(int*);
8  int write(int, const void*, int);
9  int read(int, void*, int);
10 int close(int);
11 int kill(int);
12 int exec(const char*, char**);
13 int open(const char*, int);
14 int mknod(const char*, short, short);
15 int unlink(const char*);
16 int fstat(int fd, struct stat*);
17 int link(const char*, const char*);
18 int mkdir(const char*);
19 int chdir(const char*);
20 int dup(int);
21 int getpid(void);
22 char* sbrk(int);
23 int sleep(int);
24 int uptime(void);
```

```
# generated by usys.pl - do not edit
#include "kernel/syscall.h"
.global fork
fork:
 li a7, SYS_fork
 ecall
 ret
.global exit
exit:
 li a7, SYS_exit
 ecall
 ret
.global wait
wait:
 li a7, SYS_wait
 ecall
 ret
.global pipe
pipe:
 li a7, SYS_pipe
 ecall
 ret
.global read
read:
 li a7, SYS_read
 ecall
 ret
```

Adding xv6: System calls

- The OS keeps the processor list
- The PCB contains many information

```
11  struct proc proc[NPROC];
                                        ...
82  enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
83
84  // Per-process state
85  struct proc {
86    struct spinlock lock;
87
88    // p->lock must be held when using these:
89    enum procstate state;        // Process state
90    void *chan;                  // If non-zero, sleeping on chan
91    int killed;                  // If non-zero, have been killed
92    int xstate;                  // Exit status to be returned to parent's wait
93    int pid;                     // Process ID
94
95    // wait_lock must be held when using this:
96    struct proc *parent;         // Parent process
97
98    // these are private to the process, so p->lock need not be held.
99    uint64 kstack;               // Virtual address of kernel stack
100   uint64 sz;                   // Size of process memory (bytes)
101   pagetable_t pagetable;       // User page table
102   struct trapframe *trapframe; // data page for trampoline.S
103   struct context context;      // swtch() here to run process
104   struct file *ofile[NOFILE];  // Open files
105   struct inode *cwd;           // Current directory
106   char name[16];               // Process name (debugging)
107 };
```

```
int
ps(void)
{
  struct proc *p;
  printf("name \t pid \t state \n");
  for(p = proc; p < &proc[NPROC]; p++) {
    if(p->state == SLEEPING)
      printf("%s \t %d \t SLEEPING \n", p->name, p->pid);
    else if(p->state == RUNNABLE)
      printf("%s \t %d \t RUNNABLE \n", p->name, p->pid);
    else if(p->state == RUNNING)
      printf("%s \t %d \t RUNNING \n", p->name, p->pid);
  }
  return 0;
}
```

Thank You