CS3210: Isolation mechanisms

Taesoo Kim

Administrivia

- Lab1/2?
- (Oct 4) Quiz #1. Lab1-3, Ch 0-3, Appendix A/B
- (Oct 6) Time to brainstorm project ideas

Summary of last lectures

- Power-on → BIOS → bootloader → kernel (→ init, first user code)
- Abstractions: process and system call
- Example: shell
- OS designs: monolithic vs. micro kernels

What's operating system (again)?

- OS design focuses on:
  - **Abstracting** the hardware for convenience and portability
  - **Multiplexing** the hardware among multiple applications
  - **Isolating** applications that might contain bugs
  - Allowing **sharing** among applications

Today: isolation

- Isolation vs. protection?
- What's unit of isolation in OS?

The unit of isolation: "process"

- Prevent process X from wrecking or spying on process Y
  - (e.g., memory, cpu, FDs, resource exhaustion)
- Prevent a process from wrecking the operating system itself
  - (i.e. from preventing kernel from enforcing isolation)
- In the face of bugs or malice
  - (e.g. a bad process may try to trick the h/w or kernel)
- Q: can we isolate a process from kernel?

Isolation mechanisms in operating systems

1. User/kernel mode flag (aka ring)
2. Address spaces
3. Timeslicing (later)
4. System call interface

Hardware isolation in x86

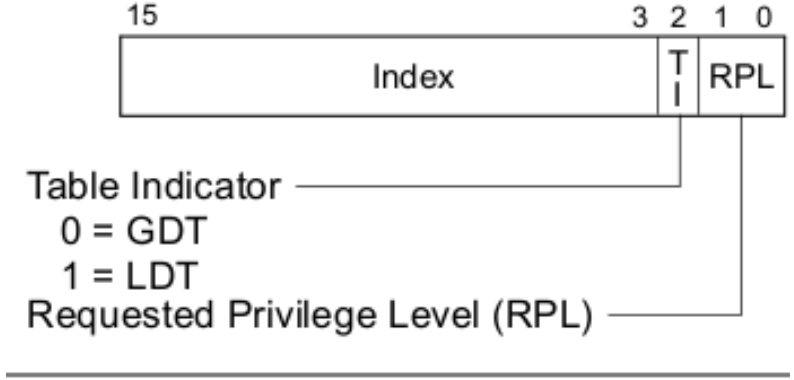


Figure 3-6. Segment Selector

- x86 support: kernel/user mode flag
- CPL (current privilege level): lower 2 bits of `%cs`
  - 0: kernel, privileged
  - 3: user, unprivileged

Hardware isolation in x86 (aka ring)

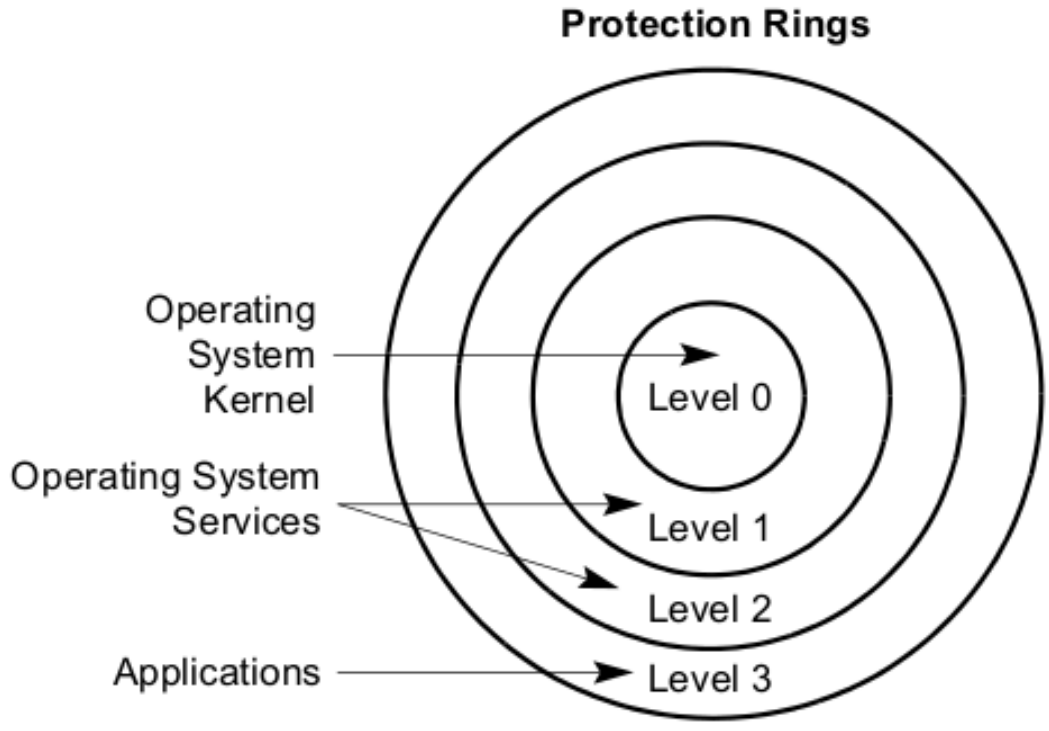


Figure 5-3. Protection Rings

What does "ring 0" protect?

- Protects everything relevant to isolation
  - writes to `%cs` (to defend CPL)
  - every memory read/write
  - I/O port accesses
  - control register accesses (`eflags`, `%cs4`,...)

How to switch b/w rings (ring 0 ↔ ring 3)?

- Controlled transfer: system call
  - `int` or `sysenter` instruction set CPL to 0
  - set CPL to 3 before going back to user space

Making system calls in xv6 (usys.S)

```
01    #include "syscall.h"
02    #include "traps.h"
03
04    #define SYSCALL(name)          \
05      .globl name;                 \
06      name:                        \
07        movl $SYS_ ## name, %eax; \
08        int $T_SYSCALL;            \
09        ret
10
11    SYSCALL(fork)
12    SYSCALL(exit)
13    ...
```

Returning back to userspace (trapasm.S)

- `syscall()` → `trapret()` → `iret`

```
01    .globl trapret
02    trapret:
03      popal
04      popl %gs
05      popl %fs
06      popl %es
07      popl %ds
08      addl $0x8, %esp  # trapno and errcode
09      iret
```

How to isolate process memory?

- Idea: "address space"
  - Give each process own memory space
  - Prevent it from accessing other memory (kernel or other processes)
- x86 provides "paging hardware" (next week)
  - MMU: VA → PA

Virtual address space in xv6

DEMO: memdump

```
$ ./memdump 0x0                    ; Q1?
$ ./memdump 0xffffffff000000       ; Q2?
$ ./memdump ????                   ; Q3?
```

How to isolate CPU?

- Prevent a process from hogging the CPU, e.g. buggy infinite loop
- Cooperative vs uncooperative scheduling
- xv6 relies on clock interrupt for context switching (next week)

How to represent a process in xv6 (proc.h)?

```
01  struct proc {
02    uint sz;                     // Size of process memory (byt
03    pde_t* pgdir;                // Page table
04    char *kstack;                // Bottom of kernel stack
05    enum procstate state;        // Process state
06    int pid;                     // Process ID
07    struct proc *parent;         // Parent process
08    struct trapframe *tf;        // Trap frame for current sysc
09    struct context *context;     // swtch() here to run process
10    void *chan;                  // If non-zero, sleeping on ch
11    int killed;                  // If non-zero, have been kill
12    struct file *ofile[NOFILE];  // Open files
13    struct inode *cwd;           // Current directory
14    char name[16];               // Process name (debugging)
15  };
```

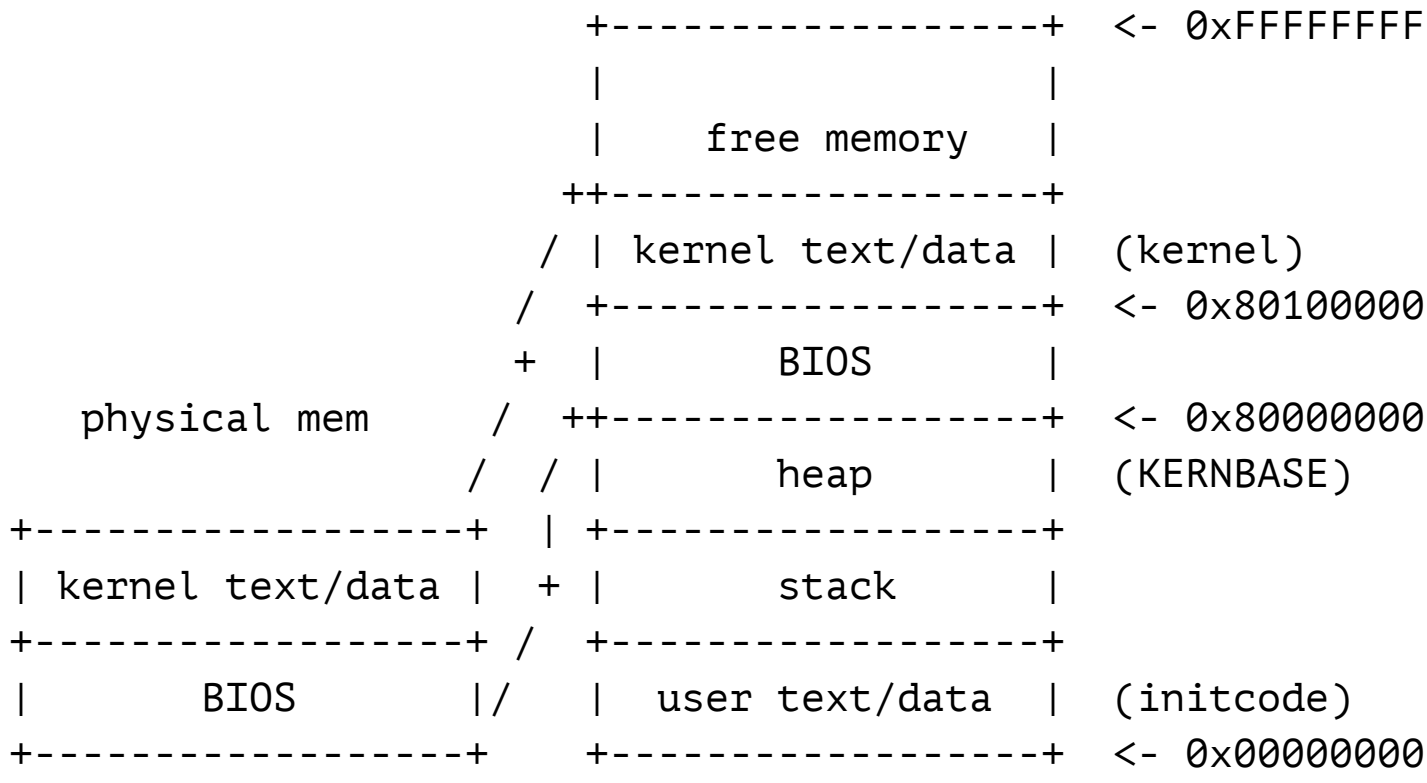
Code: first kernel code (entry.S)

- entry point of `kernel`
- enable paging
- setup stack
- handover to `main` in `main.c`

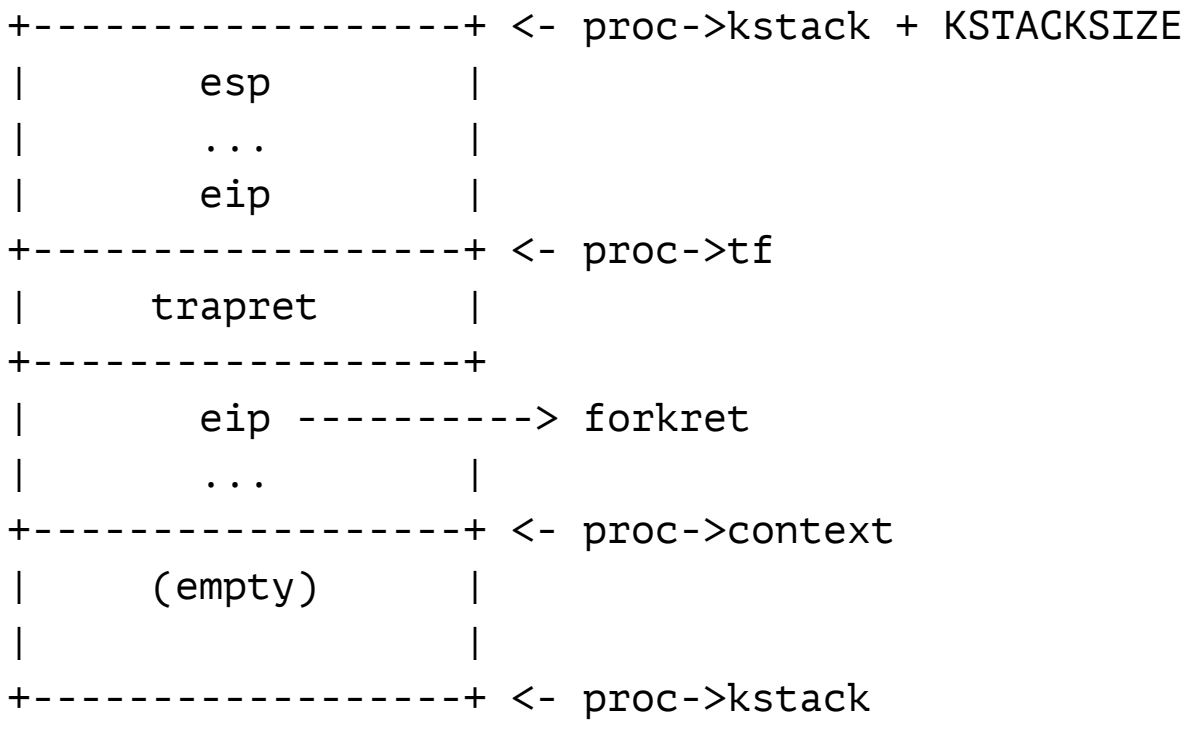
Code: the first process (proc.c)

- allocate a proc with `allocproc()`
- setup vm: `setupkvm()` and `inituvm()`
- setup tf to launch `initcode.S`

The first address space in xv6



Code: a new kernel stack (proc.c)



Code: running the first process

- `mpmain()`
- `scheduler()`
- runs `initcode.S`

Code: the first system call (initcode.S)

- handover to "/init" (Q: why not just invoke "/init"?)

```
01  .globl start
02  start:
03    pushl $argv  // argv[] = {init, 0}
04    pushl $init  // init[] = "/init\0"
05    pushl $0     // where caller pc would be
06    movl $SYS_exec, %eax
07    int $T_SYSCALL
```

Code: the /init process (init.c)

```
01  int main(void) {
02    open("console", O_RDWR) // Q1?
03    dup(0);                 // Q2?
04    dup(0);                 // Q3?
05    for(;;) {
06      if (!fork())          // Q4?
07        exec("sh", argv);   // Q5?
08      wait();
09    }
10  }
```

Exercise: system calls in xv6

- Next Lecture: Bring your Thumb Drive!

```
$ git clone git://tc.gtisc.gatech.edu/cs3210-pub
```

or

```
$ cd cs3210-pub
$ git pull
```

References

- Intel Manual
- UW CSE 451
- OSPP
- MIT 6.828
- Wikipedia
- The Internet