CS 134
Operating Systems

Feb 6, 2019

Isolation Mechanisms

Multiple processes

- Having multiple pieces of code running leads to:
  - Multiplexing
  - Isolation
  - Interaction/sharing/communication

Isolation: most constraining consideration

- Isolation determines much of the basic design

- Much of the reason why we need processes
  - Separate address space
  - Separately scheduled CPU

What is isolation

- Process is a unit of isolation
  - Process A can't (due to bugs or malice):
  - Spy on, modify, or wreck process B:
    - memory
    - CPU
    - resources
    - FDs
  - Wreck the OS:
    - Prevent the OS from enforcing isolation

What are the HW isolation mechanisms?

- User/Kernel mode
- Address spaces
- Timeslicing
- System call interface

User/Kernel mode

- Controls whether instruction can access privileged HW
- On x86, called CPL (Current Processor Level): bottom two bits of `%cs`
  - CPL==0: Kernel mode—privileged
  - CPL==3: User mode—unprivileged
- On x86, CPL protects everything relevant to isolation:
  - Writes to `%cs` (to protect CPL)
  - Every memory read/write
  - I/O port access
  - Register access (`eflags`, ...)

Hardware isolation in x86 (ring)

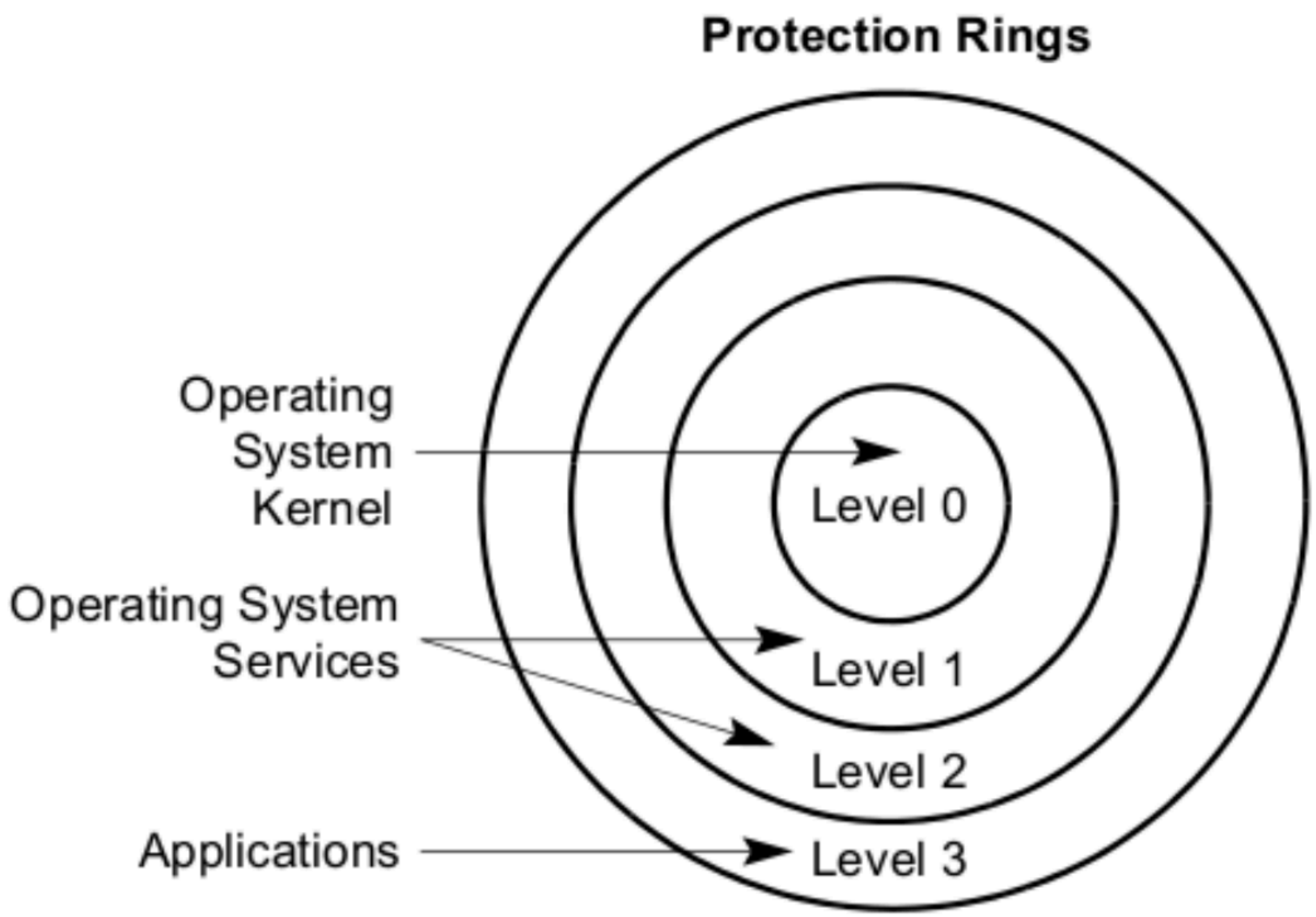


Figure 5-3. Protection Rings

How to do a system call: switching to a lower CPL

- How x86 actually does it
  - Combined instruction that:
    - sets CPL=0
    - calls into kernel code
      - But only into well-defined location(s)

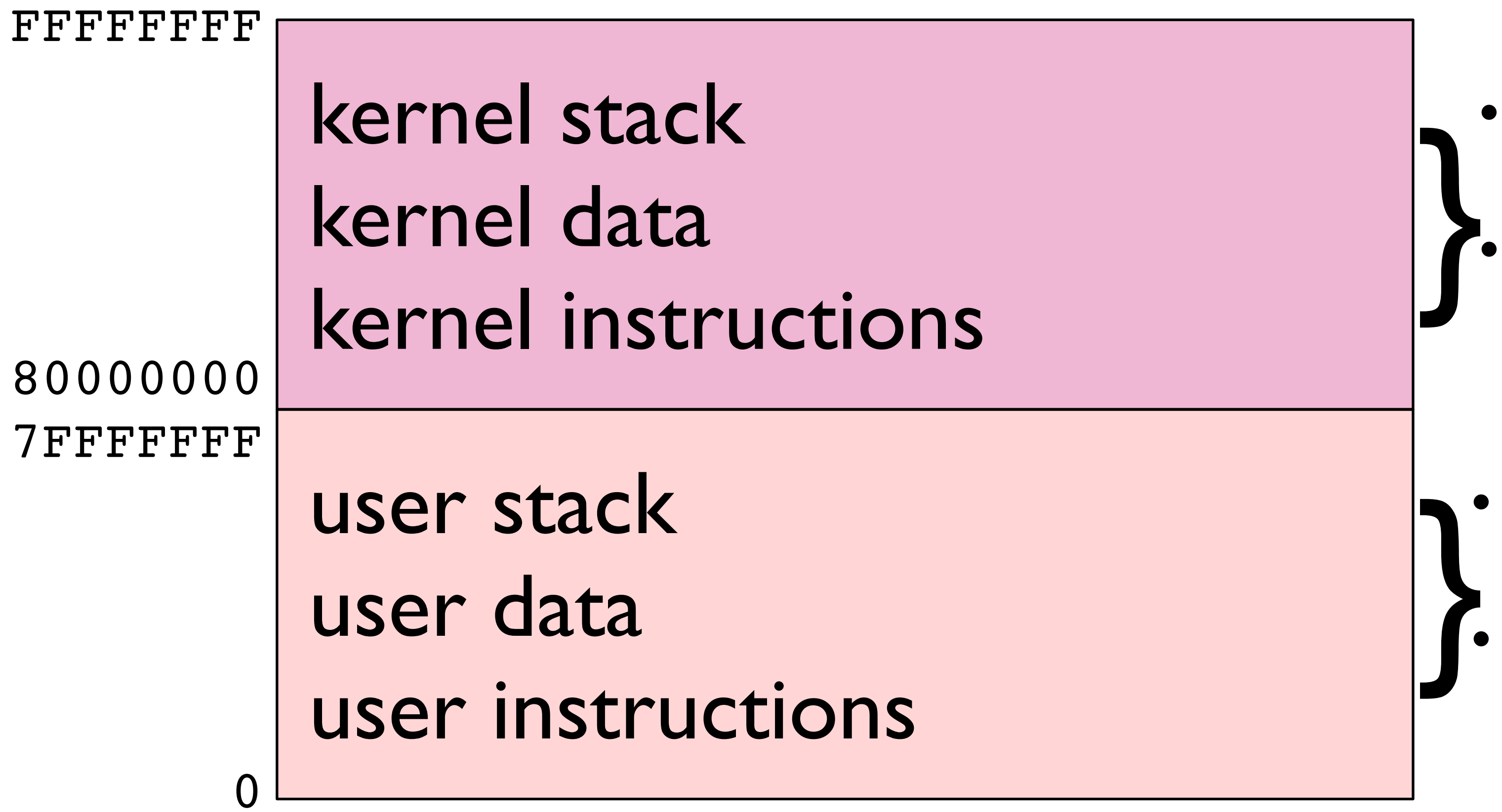
```
%eax = sys_call_number
int 64
```

  - Also, combined instruction that:
    - Restores CPL `iret`
    - Returns to user instructions

Well-defined notion of user/kernel mode

- If CPL == 0:
  - Executing via entry point into kernel
- If CPL == 3:
  - Executing user instructions

Simplified xv6 user/kernel virtual address space setup



- Page tables prevent access to upper area in user mode.
- Every process has same mappings for this range

- Separate address space for each process
- Every process has its own mappings for this range

System call starting point

- sh.c writing it's "$ " prompt

```
int
getcmd(char *buf, int nbuf)
{
  printf(2, "$ ");
  …
```

sh.c

```
int write(int, const void*, int);
…
void printf(int, const char*, ...);
```

user.h

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

…
SYSCALL(write)
```

usys.s

```
static void
putc(int fd, char c)
{
  write(fd, &c, 1);
}

void
printf(int fd, const char *fmt, ...)
{
  …
  putc(fd, c);
```

printf.c

```
#define SYS_write  16
…
```

syscall.h

```
00000cec <write>:
SYSCALL(write)
     cec:          mov    $0x10,%eax
     cf1:          int    $0x40
     cf3:          ret
```

sh.asm

System call: making the call

```
#define SYS_write  16
…
```

syscall.h

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

…
SYSCALL(write)
```

usys.s

```
00000cec <write>:
SYSCALL(write)
     cec:              mov    $0x10,%eax
     cf1:              int    $0x40
     cf3:              ret
```

sh.asm

When `int $0x40` is the next instruction:

- **`info reg`**
  ```
  eax 0x10
  esp 0x3f3c
  eip 0xcf1
  cs  0x1b
  ```
- **`x/4x $esp`**
  ```
  0x00000d8c 0x00000002  0x00003f5c 0x00000001
  ```
- **`x/c 0x00003f5c`**
  ```
  0x3f5c: 36 '$'
  ```
- **`x/i 0x00000d8c`**
  ```
  0xd8c <putc+32>:     leave
  0xd8d <putc+33>:     ret
  ```

Kernel entry: INT instruction

After `int $0x40`:

- **`info reg`**

```
eax            0x10
esp            0x8dffefe8
eip            0x80105408
cs             0x8
```

- **`x/6x $esp`**

Saved err, eip, cs, eflags, esp, ss

```
0x8dffefe8:    0x00000000    0x00000cf3    0x0000001b    0x00000202
0x8dffeff8:    0x00003f3c    0x00000023
```

```
80105537 <vector64>:
.globl vector64
vector64:
  pushl $0
80105537:          push   $0x0
  pushl $64
80105539:          push   $0x40
  jmp alltraps
8010553b:          jmp    80104ec2 <alltraps>
```

What INT did:

- Switched to process's kernel stack
- Saved some regs on kernel stack
- Set CPL to 0
- Start executing at kernel-supplied "vector"

Kernel entry: INT instruction

```
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
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

trapasm.S

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

trap.c

Kernel entry: INT instruction

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
…,
[SYS_write]   sys_write,
…
}
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;
  }
}
```

syscall.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

sysfile.c

Summary

- Intricate design for User/Kernel transition
- Kernel must take adversarial view of user process
  - Doesn’t trust user stack
  - Checks arguments
- Page table confines what memory user program can read/write