Traps, interrupts, and drivers

Youjip Won

KAIST

Contents

- System calls, exceptions, and interrupts
- X86 protection
- Code: The first system call
- Code: Assembly trap handlers
- Code: C trap handler
- Code: System calls

System calls, exceptions, and interrupts

- System call
  - When a program asks for an operating system service.
- Exception
  - When a program performs an illegal action.
- Interrupt
  - When a device generates a signal to indicate that it needs attention from the operating system.
  - Kernel handles all interrupts because in most cases, only the kernel has the required privilege and state.

The first system call: `initcode.S`

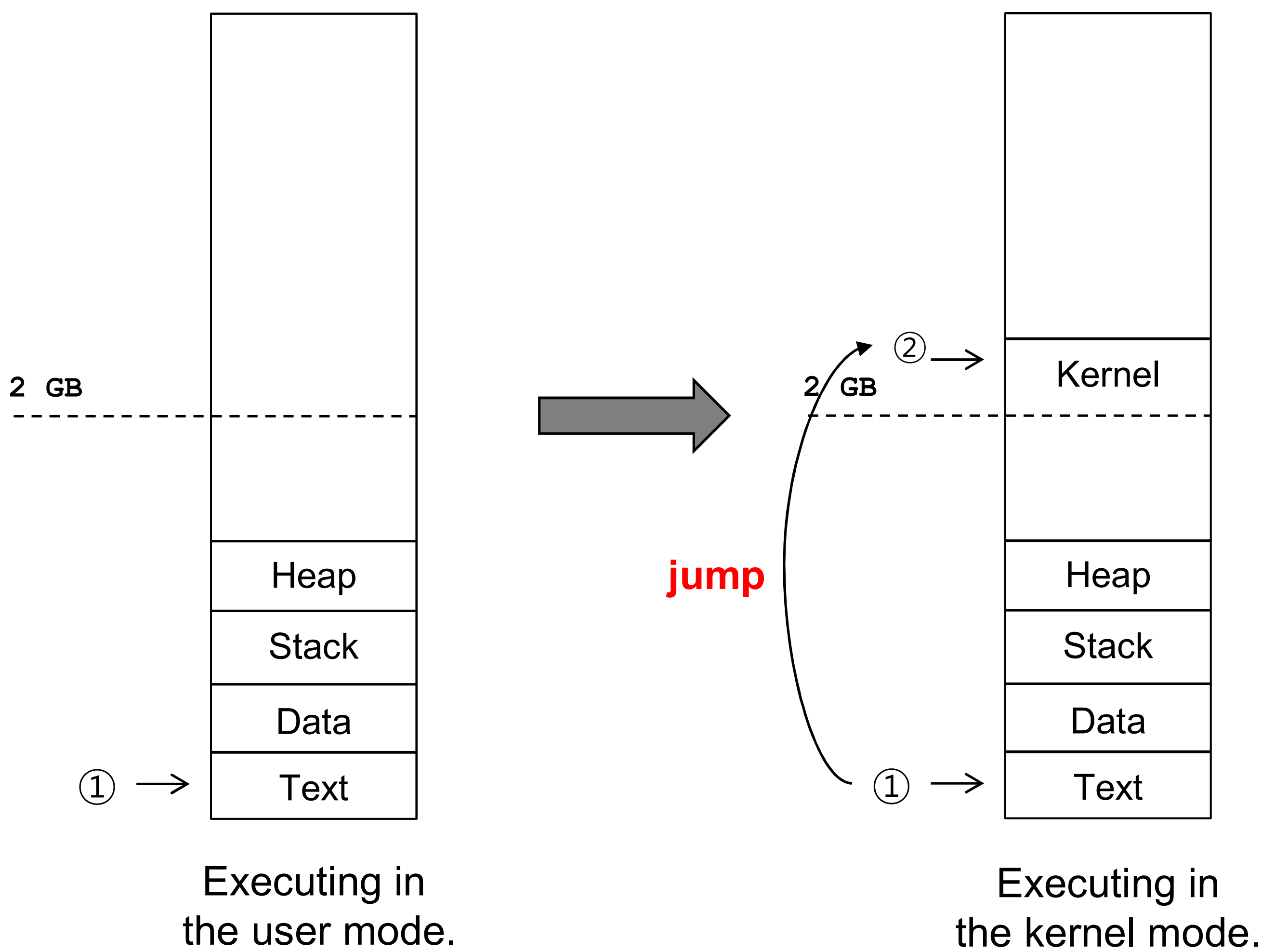
`exec ("init", ["init", NULL]);`

```
# Initial process execs /init.
…
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0 // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL
…
```

```
…
# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0

```

Switching from Kernel mode to user mode

Transfer the control from the user to the kernel.



When does it go into kernel?

- System call
  - Call `'int'` instruction.
  - `read(), fork()`
- Exception
  - Generated by the software

```
int i,j;
i = 1; j = 0;
i = i/j;
```

- Interrupt
  - Hardware interrupt signal, e.g. ntimer Interrupt.
  - Only the kernel has the privilege to handle it!

How does it go into kernel?

`int n`

- Assembly instruction
- It changes the mode from the user to the kernel if necessary.
- It executes interrupt handler.

- Interrupt Handler
  - A function that handles the interrupt.

① Find the address of interrupt handler and check the privilege level.

② Change the stack from the user stack to the kernel stack.

③ Push some value of the registers.

④ Load the address of interrupt handler to `%eip` register.

`int n`

① Find the address of interrupt handler and check the privilege level.

② Change the stack from the user stack to the kernel stack.

③ Push some value of the registers.

④ Load the address of interrupt handler to `%eip` register.

Privilege Level (Mode)

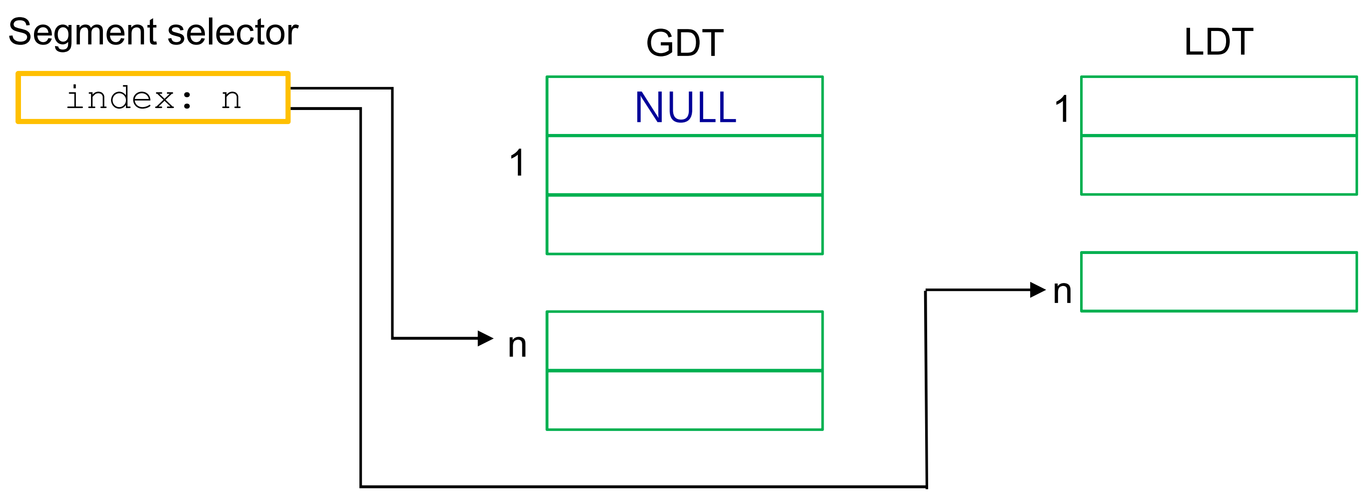
- Execution Privilege
- Four levels in x86.
  - It is called “ring”
  - 0 (kernel, the height), 1, 2, 3 (user, the lowest)

- **CPL:** current privilege level
  - RPL field (2 bit) of `%cs` register.

Recap: Segment Descriptor Table

- Kernel maintains the array of segment descriptors.
  - Per CPU: Global Descriptor Table
  - Per Process: Local Descriptor Table

- Segment Selector (`%cs, %es, %ds, %ss, %fs, %gs`)
  - It points an entry in a descriptor table.



Segment Selector

index	TI	RPL
15 … 3	2	1 … 0

`index:` index of the descriptor.

`TI:` table indicator, `0 = GDT, 1 = LDT`

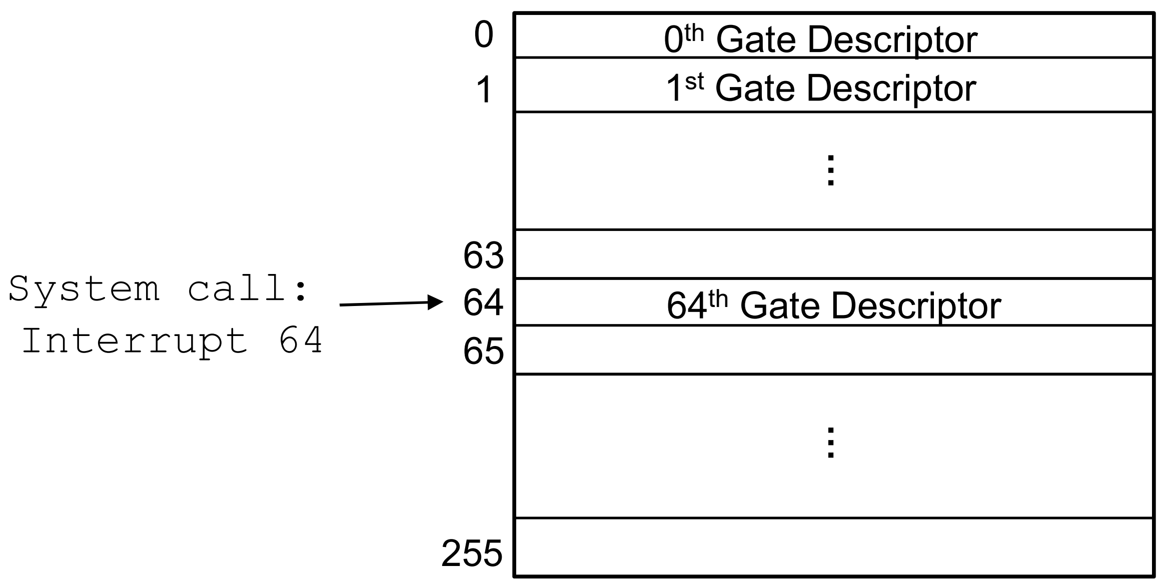
`RPL:` Request Privilege Level

Now we can find the current privilege level.

What next? Where is the privilege level of interrupt handler?

Interrupt Descriptor Table

- Kernel maintains the array of **gate descriptors**.
  - Each entry contains the address of interrupt handler.
  - When interrupt N is occurred, the interrupt handler at index N is executed.

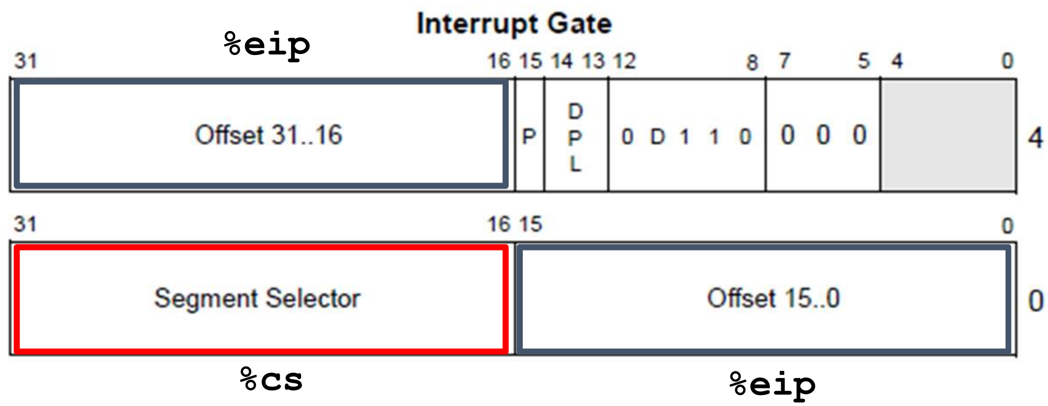


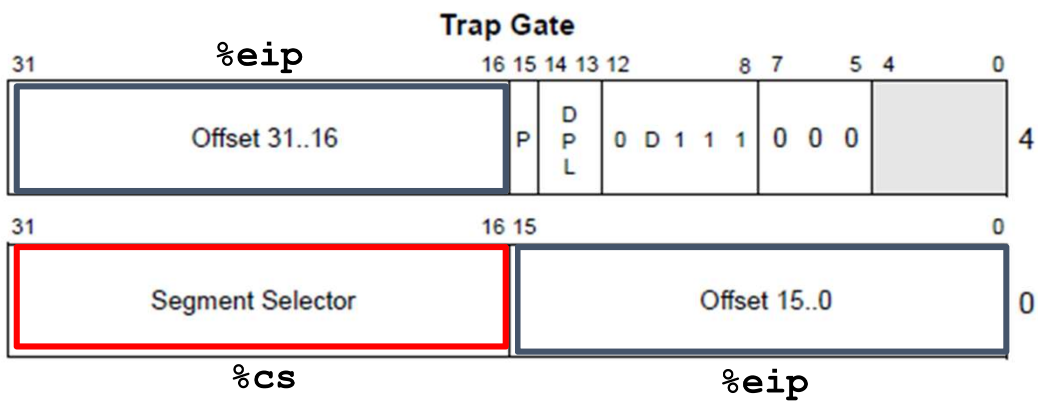
Interrupt Descriptor Table

Gate Descriptor

- Special data structure in x86.
- Offset 31..16 & 15..0: Value of `%eip` register
- Segment Selector: Value of `%cs` register

Address of interrupt handler

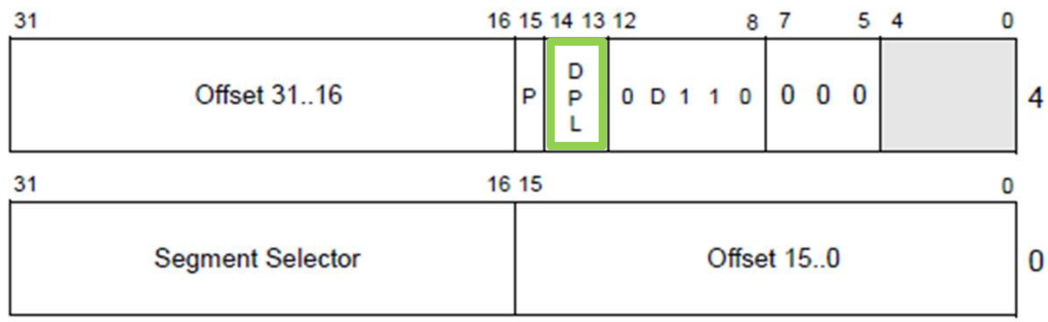




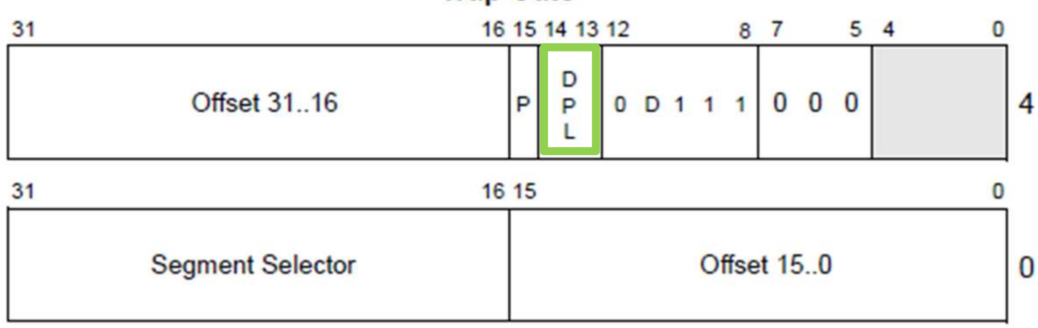
Gate Descriptor

- Special data structure in x86.
- **DPL : Privilege level of interrupt handler.**

Interrupt Gate



Trap Gate



struct gate descriptor: mmu.h

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;    // low 16 bits of offset in segment
  uint cs : 16;          // code segment selector
  uint args : 5;         // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;         // reserved(should be zero I guess)
  uint type : 4;         // type(STS_{IG32,TG32})
  uint s : 1;            // must be 0 (system)
  uint dpl : 2;          // descriptor(meaning new) privilege level
  uint p : 1;            // Present
  uint off_31_16 : 16;   // high bits of offset in segment
};
```

Initializing interrupt descriptor table: `main.c`

```
int
main(void)
{
                                     …
  tvinit();          // gate descriptors
                                     …
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
  userinit();        // first user process
  mpmain();          // finish this processor's setup
}
```

- The interrupt descriptor table is initialized when the system is booting up.
  - The main function for kernel calls the function `tvinit()`.
  - The `tvinit()` initializes the interrupt descriptor table.

Initializing interrupt descriptor table: `trap.c`

- Initializing 256 entries
- All entries: disable interrupt, can be called only inside the kernel.
- System call: interruptible, can be called from the user (DPL_USER == 3).

```
struct gatedesc idt[256];

                                          …

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");
}
```

```
SETGATE(Gate descriptor,
        istrap?, %cs value,
        %eip value, DPL);
```

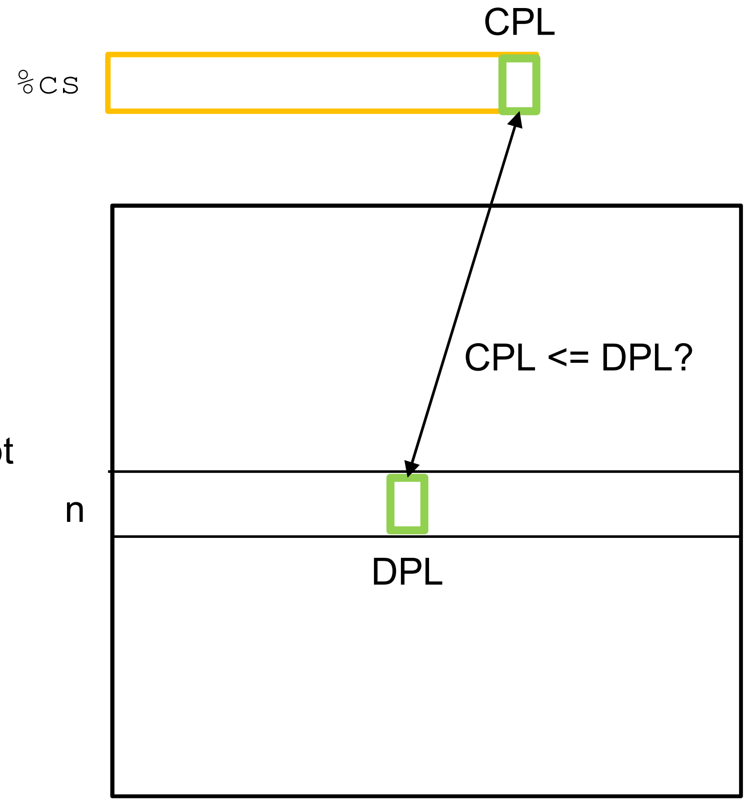
why?

Checking the privilege level

- **CPL:** RPL field of %cs register.
- **DPL:** DPL field of gate descriptor.

- `CPL <= DPL:`
  - If CPL is lower (higher), Do next.
  - The execution privilege of current context is higher than the execution privilege of interrupt handler.

- `CPL >  DPL:`
  - Otherwise, Kernel panic!!!
  - Raise Interrupt 13 (T_GPFLT).



Interrupt descriptor table

Checking the privilege level (Cont.)

- The CPL of user mode context is 3.
- Only 64th Gate descriptor's DPL is set to be 3 (== DPL_USER). Others are 0.
- User can call only `'int 64'`.
- Hardware interrupt is regardless of whether the execution is in kernel or user context.

```
struct gatedesc idt[256];
...
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");
}
```

```
SETGATE(Gate descriptor,
        istrap?, %cs value,
        %eip value, DPL);
```

why?

`int n`

① Find the address of interrupt handler and check the privilege level.

② Change the stack from the user stack to the kernel stack.

③ Push some value of the registers.

④ Load the address of interrupt handler to `%eip` register.

Changing the stack

- `CPL <= DPL:`
  - The process can start this point only when the above condition is satisfied.
  - `CPL == DPL:`
    - The privilege level is not updated. The stack does not have to be changed.
  - `CPL <  DPL:`
    - The privilege level is updated, stack should be changed from the user stack to the kernel stack.

How does the processor find the location of the kernel stack?

Task State Segment

- Data structure in x86 CPU that holds information on a task.
- Per-CPU data structure

```
struct cpu {
  uchar apicid;                  // Local APIC ID
  struct context *scheduler;     // swtch() here to enter scheduler
  struct taskstate ts;           // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];     // x86 global descriptor table
  volatile uint started;         // Has the CPU started?
  int ncli;                      // Depth of pushcli nesting.
  int intena;                    // Were interrupts enabled before pushcli?
  struct proc *proc;             // The process running on this cpu or null
};
```

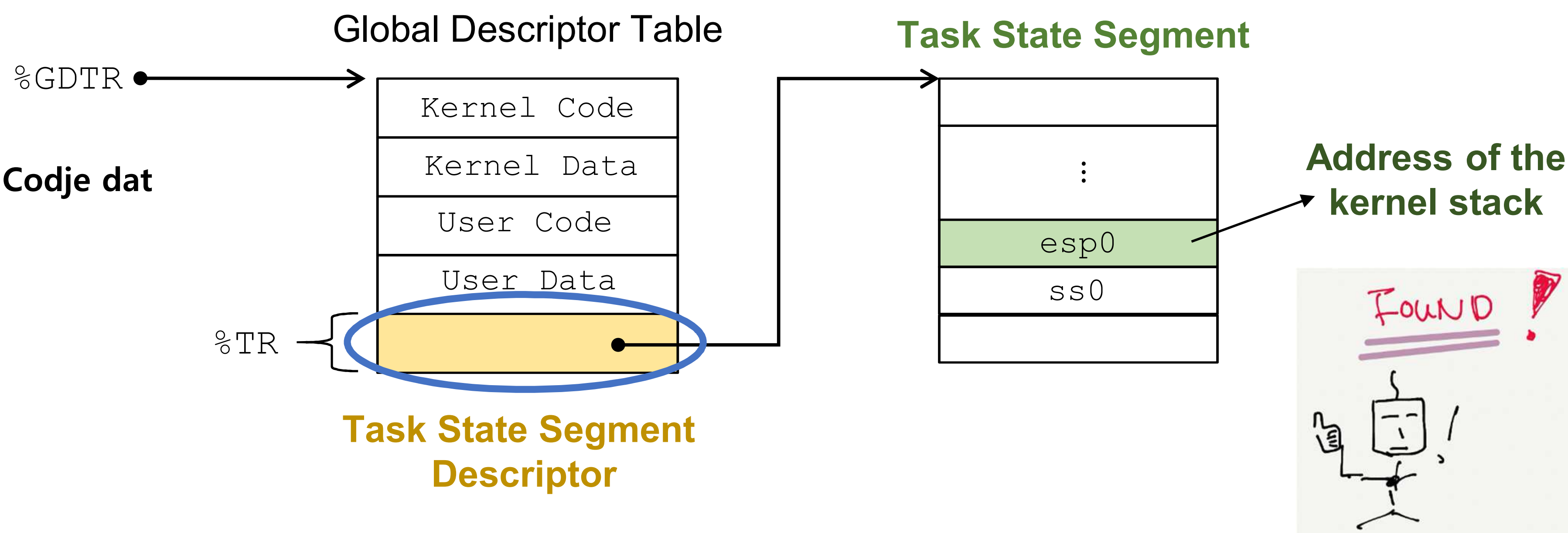
Task State Segment (Cont.)

- Task State Segment has `%esp` & `%ss` register value of kernel mode process.
- `esp0`: Line 109
  - Address of the kernel stack
- `ss0`: Line 110
  - Segment Selector for stack.
  - Processor refers to it when `push/pop` instruction is executed.

```
106 // Task state segment format
107 struct taskstate {
108   uint link;          // Old ts selector
109   uint esp0;          // Stack pointers and segment selectors
110   ushort ss0;         //   after an increase in privilege level
111   ushort padding1;
                                   …
145 };
```

Finding the Task State Segment

- GDTR register & TS register.
  - %GDTR: the start address of global descriptor table.
  - %TR: the index of **task state segment descriptor in the global descriptor table.**
  - This descriptor points the **Task State Segment**.



Setting the Task State Segment Descriptor

- `switchuvm()` switches the stack from kernel to user process `p`.
  - SEG_TSS is 5.
  - It sets the entry at index 5 in GDT to Task state segment (Line 167 ~ 168).
  - It sets the value of TS register to 5 (Line 175).

```
157 void switchuvm(struct proc *p) {
                                              …
166   pushcli();
167   mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
168                                 sizeof(mycpu()->ts)-1, 0);
169   mycpu()->gdt[SEG_TSS].s = 0;
170   mycpu()->ts.ss0 = SEG_KDATA << 3;
171   mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
                                              …
175   ltr(SEG_TSS << 3);
176   lcr3(V2P(p->pgdir));  // switch to process's address space
177   popcli();
178 }
```

Setting the Task State Segment

- `switchuvm()` switches the stack from kernel to user.
  - It stores the value of current values of `ss0` and `esp0`.
  - `p->kstack` stores the start address of kernel stack page.

```
void switchuvm(struct proc *p) {
                                    …
  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
                                    …
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

Changing the stack

- `CPL <= DPL:`
  - The process can start this point only when the above condition is satisfied.
  - `CPL == DPL:`
    - The privilege level is not updated, stack change is not required.
  - `CPL <  DPL:`
    - Save the current value of %ss & %esp, temporarily.
    - Fetch the new value to %ss & %esp from Task State Segment.
    - Push original value of %ss & %esp to new stack.

`int n`

① Find the address of interrupt handler and check the privilege level.

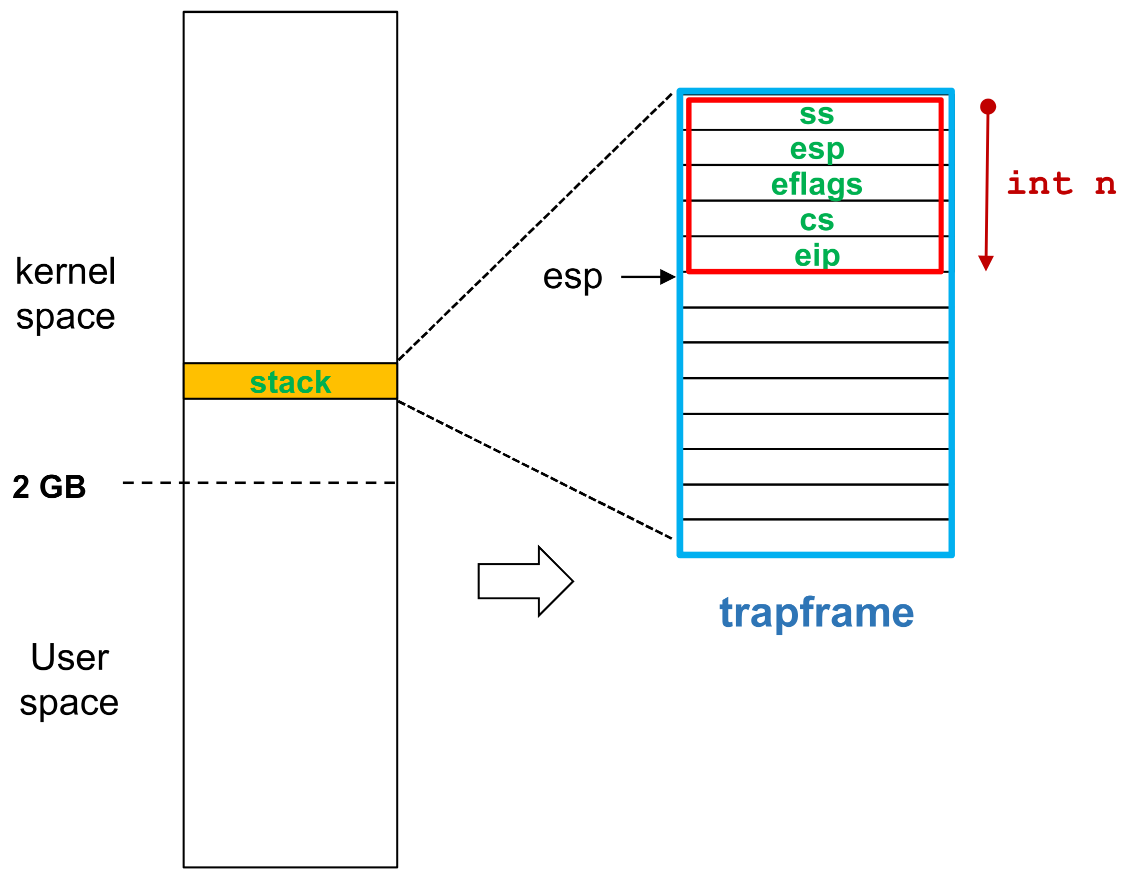
② Change the stack from the user stack to the kernel stack.

③ Push the registers at the kernel stack.

④ Load the address of interrupt handler to `%eip` register.

Pushing some registers' value

- Push `%eflags`, `%cs`, and `%eip` to kernel stack.



struct trapframe

- First, the values of five registers are pushed by hardware (`%ss, %esp, %eflags, %cs, and %eip`).
- The remaining values are pushed later…

```
struct trapframe {
                    …

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings,
     such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

Stack grows.

Push and popped by hardware

`int n`

`int n`

① Find the address of interrupt handler and check the privilege level.

② Change the stack from the user stack to the kernel stack.

③ Push some value of the registers.

④ Load the address of interrupt handler to `%eip`.

Load the address of interrupt handler.

- Load `%eip` and `%cs` from the gate descriptor.
- 64th Gate Descriptor points to address of `vector64`.

0	0th Gate Descriptor
1	1st Gate Descriptor
	⋮
63	
64	64th Gate Descriptor
65	
	⋮
255	

Interrupt Descriptor Table

System Call

`%eip` → `vector64`

`%cs` → `SEG_KCODE <<3`

```
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
```

at `tvinit()`.

`int n`

① Find the address of interrupt handler and check the privilege level.

- Fetch n^{th} gate descriptor from IDT.
- Examine if current privilege (in %cs) is lower than or equal to DPL of gate descriptor.

② Change the stack from the user stack to the kernel stack.

- If current privilege is lower than DPL,
  - load %ss & %esp from task state segment.
  - push original value of %ss and %esp.

③ Push some value of the registers.

- Push `%eflags, %cs`, and `%eip` to the stack.

④ Load the address of interrupt handler.

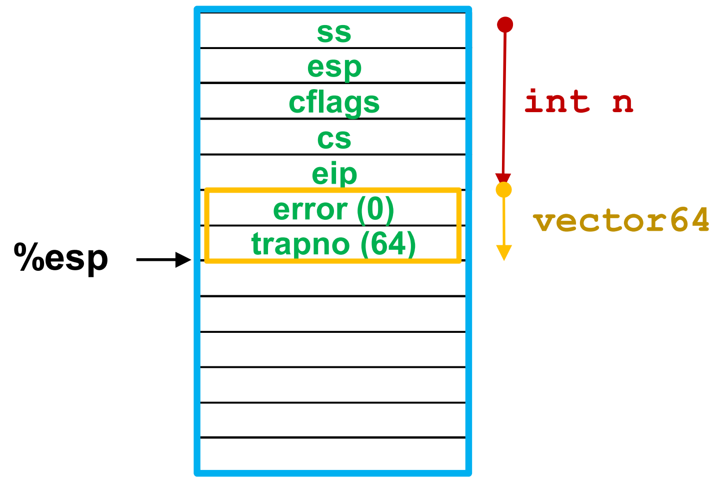
- `%cs`, `%eip` are loaded the n^{th} gate descriptor in IDT.
- Current privilege level is increased due to update of `%cs`.
- Begin to execute interrupt handler (`vector64` for system call).

Trap handler

- **Pushes error number and trap number to the trap frame.**
- **Example: vector 64 (system call)**
  - **It pushes 0.** This is value of error of the trapframe (Line 319).
  - **It pushes 64.** This is trap number of the trapframe (Line 320).
  - Then, call the function `alltraps`.

New value of `%eip` = address of **`vector64`**

```
# vectors.S
317 .globl vector64
318 vector64:
319   pushl $0
320   pushl $64
321   jmp alltraps
```



trapframe

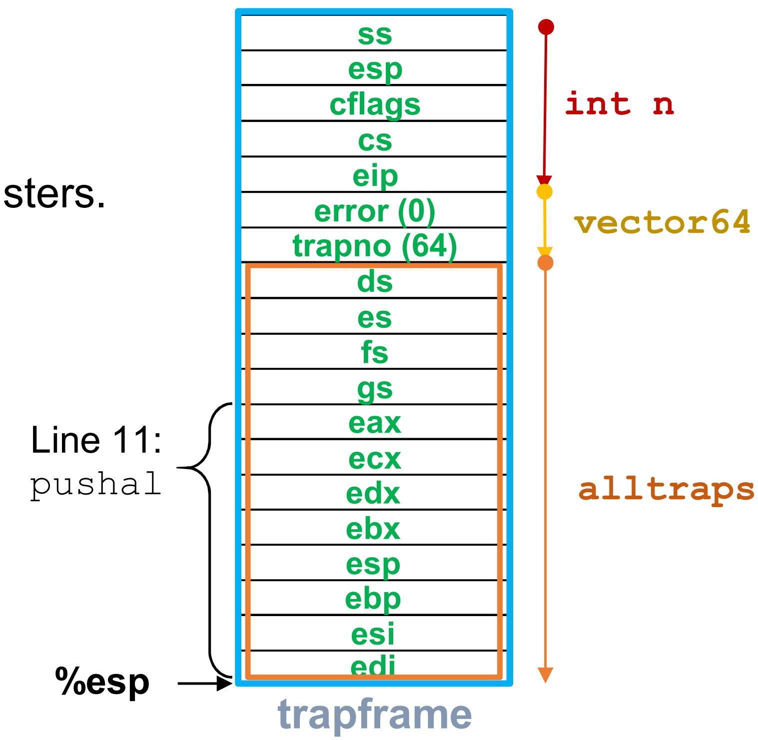
Trap handler: `alltraps`

- Pushes the rest of the registers at the trap frame.
- Jump to trap handler.

```
 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 segments.
14   movw $(SEG_KDATA<<3), %ax
15   movw %ax, %ds
16   movw %ax, %es
17
18   # Call trap(tf), where tf=%esp
19   pushl %esp
20   call trap
21   addl $4, %esp
```

Push the value of registers.

call the function trap.



Trap handler: `alltraps` (Cont.)

- Change the value of segment selector to access kernel's address space.

```
.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
```

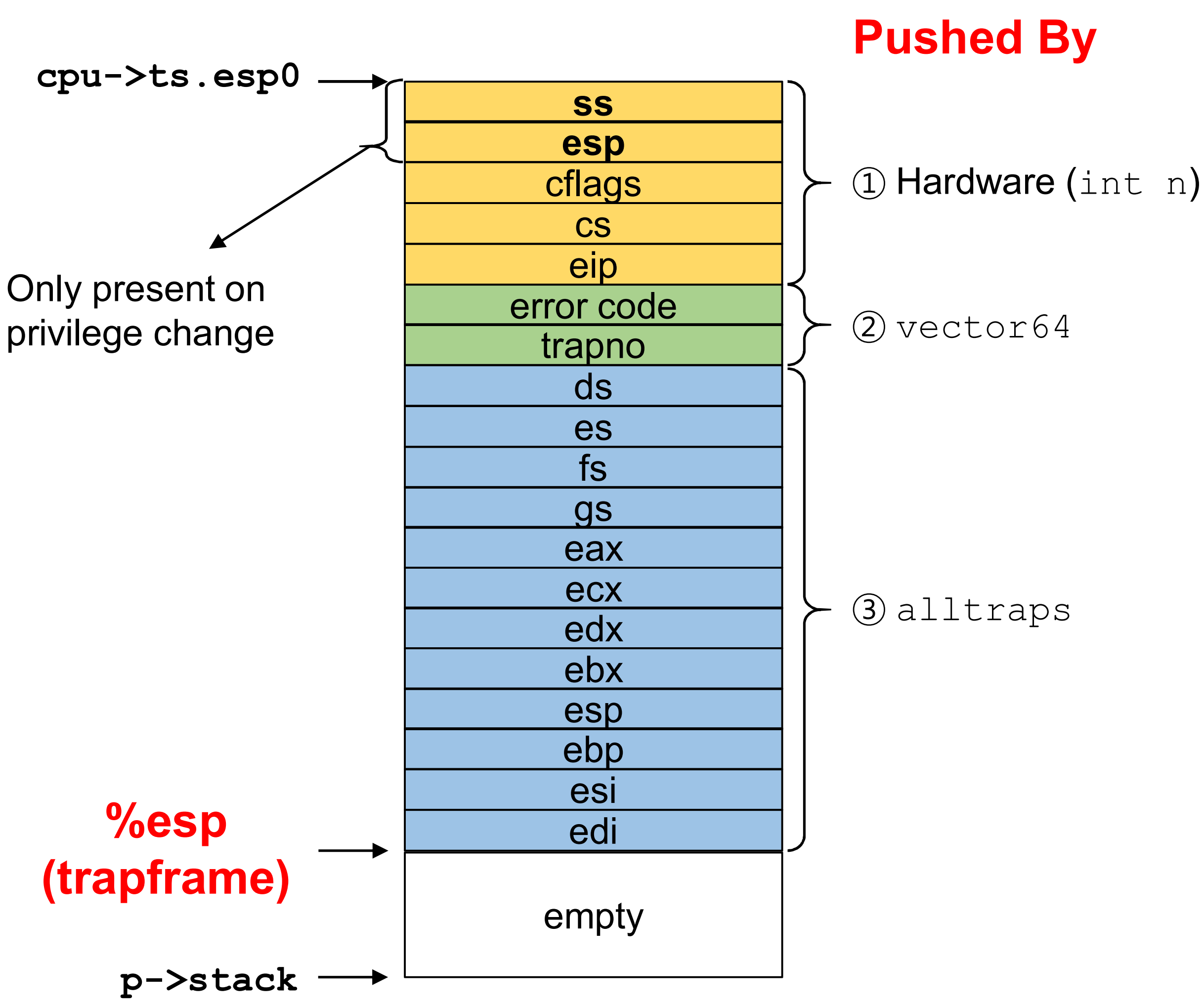
Change the value of segment selector

`%ds & %es`

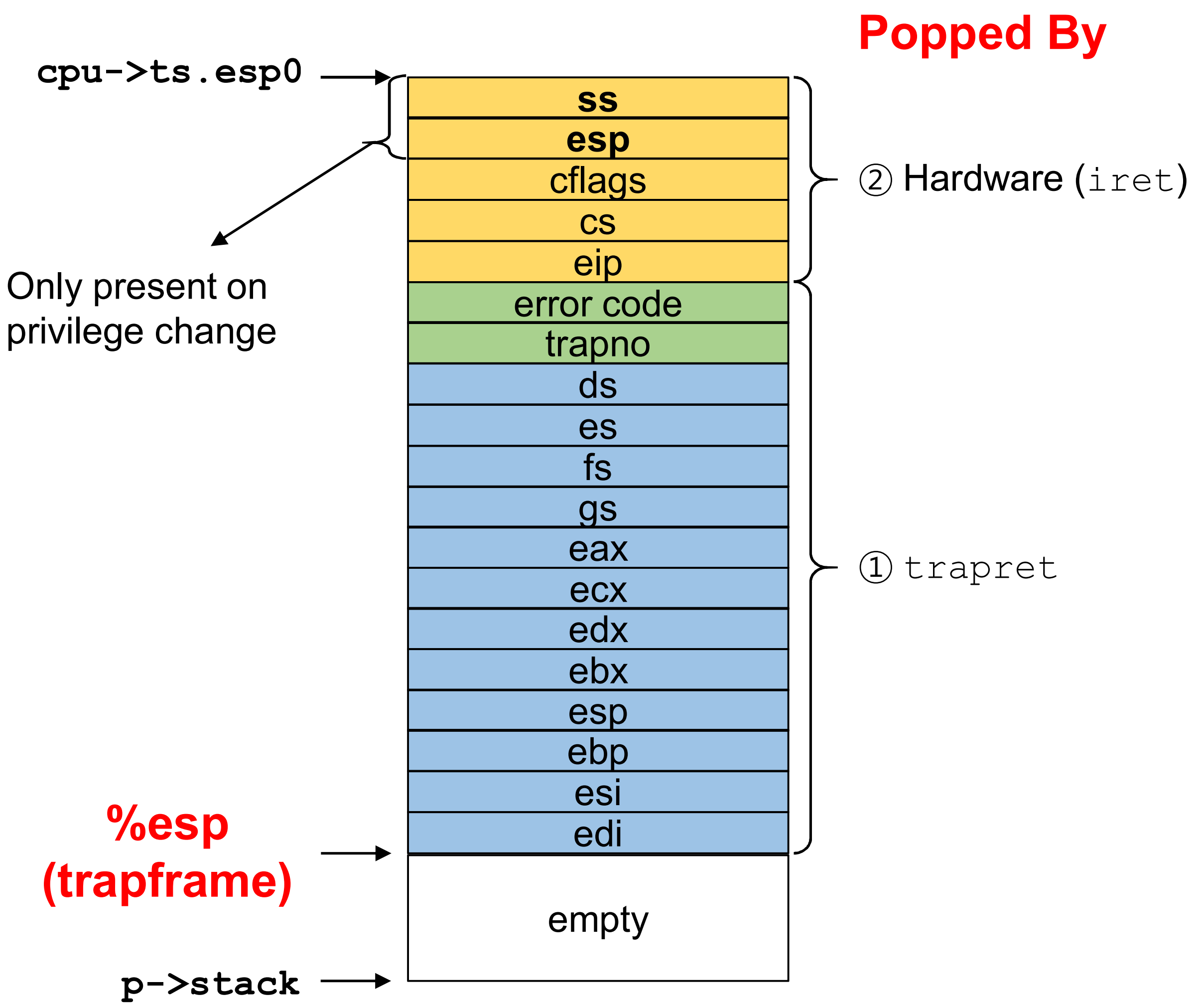
(SEG_UDATA<<3 | DPL_USER)

→ SEG_KDATA<<3

Kernel Stack Contents when entering the kernel



Kernel Stack Contents when getting out of the kernel



trap()

- `trameframe` is the parameter to `trap()` function.
- `%esp` points to the start (lowest) address of trapframe.
- pushes `%esp` at the stack before calling `trap()`.

```
  # Call trap(tf), where tf=%esp
  pushl %esp // Pointer to trapframe.
  call trap  // jump to 'real' handler.
  addl $4, %esp

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

trap ()

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

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
                …
```

If trap number is `T_SYSCALL`, `trap()` calls the `syscall()`.

```
  case T_IRQ0 + IRQ_IDE:
                …
  case T_IRQ0 + IRQ_IDE+1:
                …
  case T_IRQ0 + IRQ_KBD:
                …
  case T_IRQ0 + IRQ_COM1:
                …
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
                …
  default:
                …
  }
```

Code: The first system call `exec()`

- `initcode.S`
  - Process pushes the arguments for an `exec` call on the process's stack.
  - Put the system call number in `%eax`
  - The system call numbers match the entries in the syscalls array, a table of function pointers.

```
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0   // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

```

Fake return address

```
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,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};
```

syscall.c

syscall()

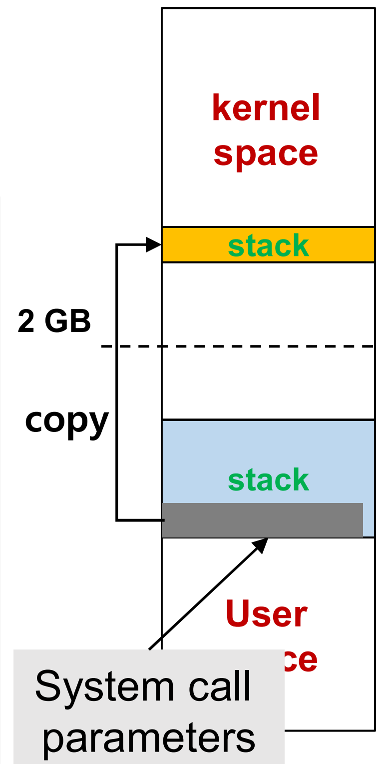
- checks the system call number at %eax.
- Then executes the proper function.

```
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;
  }
}
```

The parameter to system call

- Parameters to the system call are stored in the caller's stack.
- If the caller is user, the parameters are initially in the user stack.
- It copies the parameter to system call to its local variable.
  - `argint()`: copy the integer value to local variable.
  - `argptr()`: copy the pointer value to local variable.
  - `argstr()`: copy the string to local variable.

```
int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;

  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }
                                    …
```



Accessing caller’s stack

- System call accesses the caller’s stack by using trapframe.
- The trapframe has caller’s stack address at the `esp` variable.

```
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}

                                  …

int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}
```

Summary

- `int` instruction
  - The current privilege level should be lower than or equal to DPL of gate descriptor of IDT.
  - The %esp & %ss are loaded from task state segment pointed by segment descriptor in GDT.
  - The %eip & %cs are loaded from gate descriptor of IDT.

- `alltraps & trap()`
  - `alltraps` pushes the value of registers and calls `trap()`.
  - `trap()` calls the function with respect to trap number.
  - If trap number is 64 (system call), it calls `syscall()`.

- `syscall()`
  - `syscall()` calls the proper function with respect to the value of `%eax`.