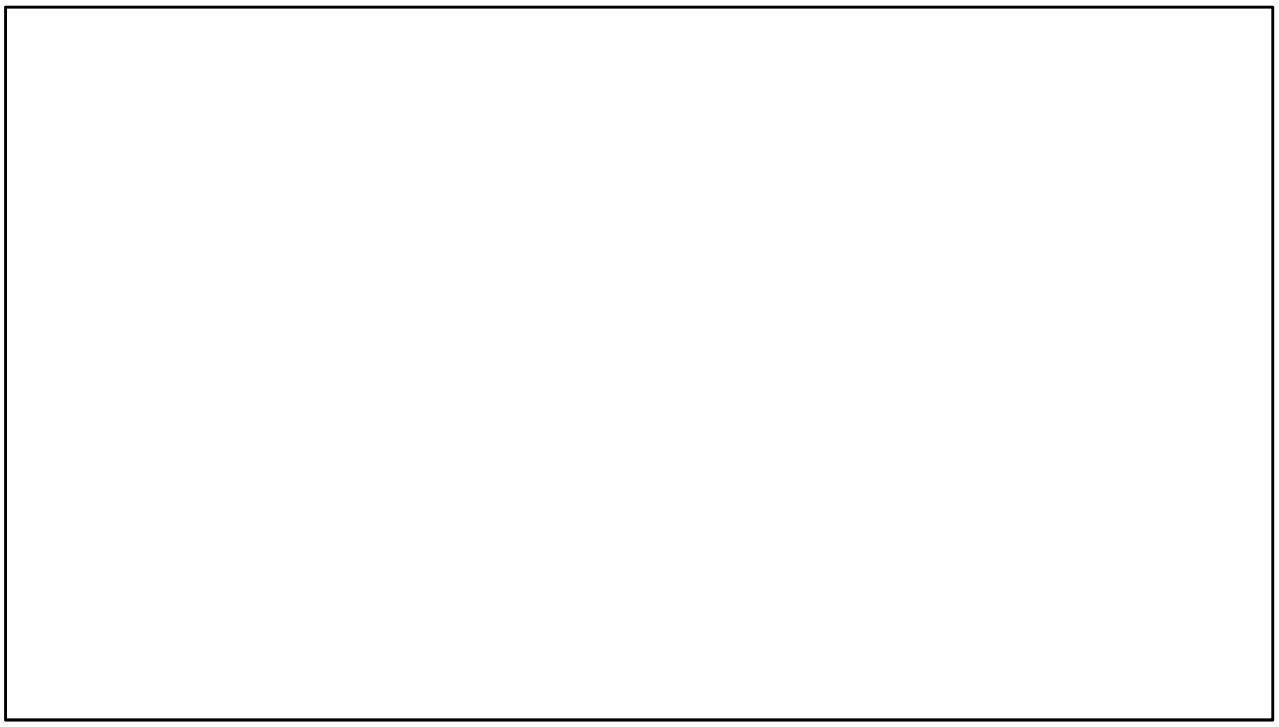


Optional assignment

- sleep
- wakeup
- wait_for_all
- thread_exit

Next class

- Homework 4 and 5 solutions
- If we have time, I will take questions from the topics that have been covered until now

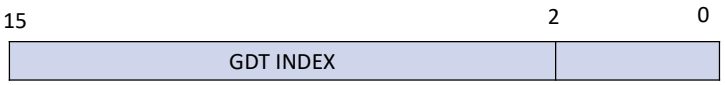
Global descriptor table (GDT)

- The OS creates a GDT in memory
- The GDTR register contains the base address of the GDT
- lgdt instruction is used to load the GDTR
  - lgdt instruction takes 6-byte memory operands
  - 4-byte base address (physical), and 2 bytes size
  - lgdt is a privilege instruction
- The GDT can have at most 2^{13} entries

Segment register

- A segment register contains a 16-bit value
  - The top 13 bits of segment registers contain the index in the GDT
  - The lower 2 bits of the cs segment register contains the current privilege level (CPL)
    - In user-mode, the last two bits of cs register is always 3
    - In kernel-mode, the last two bits of cs register is always 0
  - The hardware identifies the CPL using the lower 2 bits of the cs segment register
  - User applications can't directly modify the CPL
  - CPL is changed during entry to the kernel

Segment selector



Higher 13-bits contains the index in the GDT table.

Default segment register

- cs is the default segment register for instruction pointer (EIP)
- ss is default segment register of for stack pointer (ESP) and frame pointer (EBP)
- ds is default segment register for memory operands (except those which contain esp and ebp)
- es is default segment register in string instructions
- fs and gs are optional can be explicitly used in memory operands

schedule

```
struct thread *ready_list;
struct thread *cur_thread;

void schedule() {
  struct thread *prev = cur_thread;
  struct thread *next =  pop_front(ready_list);
  cur_thread = next;
  update_mmu_mappings(next);
  context_switch(prev, next);
}
```

The OS loads the GDT corresponding to the target process or updates the single GDT with the entries of the target process during scheduling.

update_mmu_mapping

- How to implement update_mmu_mapping, when processes want to use multiple entries in the GDT
  - updating all entries in the GDT may be expensive
- Create per-process GDT
  - load the GDT corresponding to next process in update_mmu_mapping
    - using lgdt

Example

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

movl $100, %ds:(%eax)

what is the PA, where
%eax = 100
%ds = (1 << 3) | 3;

The higher 13-bits in %ds contains 1. The PA is computed using adding base at index 1 (3000) to VA (100).

Example

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

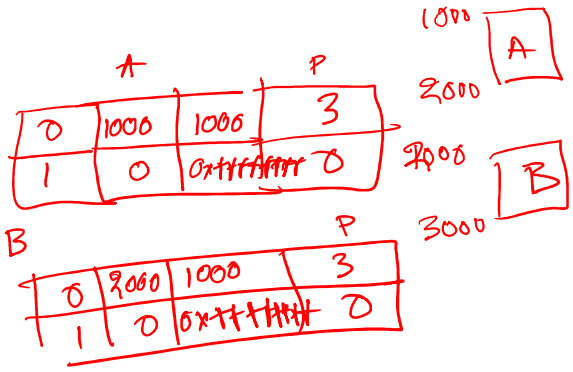
10: movl $100, %ds:(%eax)

Let us say that the VA of mov instruction is 10.
what is the PA of mov, where %cs = (2 << 3) | 3;

%cs is the default segment register for EIP. In this case, EIP of mov instruction is 10. Because, the higher 13 bits in %cs register contains 2, PA of EIP can be obtained by adding 8192 to 10.

Code isolation

- How does OS prevent an application from directly jumping to its code



Code isolation

- How does OS prevent an application from directly jumping to its code
  - OS code is stored outside the process address space

Example

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

1000 - 1256
3000 - 3128
8192 - 8192+4096

Which are the addresses where OS code is not present?

If the table shown on this slide is the current GDT, OS code can't be present as physical addresses in the range [1000-1256], [3000-3128], and [8192-8192+4096].

Interrupt_handler:

```
push %eax
push %edx
push %ecx
call schedule
pop %ecx
pop %edx
pop %eax
iret
```

CPU jumps to interrupt handler on interrupt. Can we use the same index in the cs register as the user program?

No, because OS code is outside the process address space (i.e., not mapped in the user-accessible GDT entries.)

On interrupt, the CPU needs to load the cs segment register with the value of the OS's cs register before jumping to the target interrupt handler.

Interrupt_handler:

```
push %eax
push %edx
push %ecx
call schedule
pop %ecx
pop %edx
pop %eax
iret
```

OS uses a different index in cs.
On interrupt, cs is set automatically to the OS's index in GDT.

How does the hardware know the cs value of OS?

Interrupt_handler:

```
push %eax
push %edx
push %ecx
call schedule
pop %ecx
pop %edx
pop %eax
iret
```

OS uses a different index in cs.
On interrupt, cs is set automatically to the OS's index in GDT.

How does the hardware know the cs value of OS?

IDT contains cs:EIP pairs.

Interrupt descriptor table (IDT)

- IDT contains cs:EIP pairs
- On interrupt 1, EIP is set to eip1 and cs is set to os_cs

Interrupt

- How does OS restore the cs of user program after returning from an interrupt?

Interrupt

- How does OS restore the cs of user program after returning from an interrupt?
  - hardware pushes the cs on the stack before rewriting it with OS's cs value
  - iret restores the cs from the stack

What about stack?

- Can hardware push cs, eip, eflags on the user-program stack?

What about stack?

- Can hardware push cs, eip, eflags on the user-program stack?
  - No, because of the following reason
  - User program sets the %esp to a virtual address outside the limit of the %ss segment register and waits for an interrupt without accessing the stack
  - After an interrupt, the CPU tries to push the cs, eip, eflags on an invalid address inside the ring-0 – causing the system to crash

What about stack?

- How does application thread wait for an interrupt without accessing the stack?

base limit
1000 – 1000

```
movl $0xffff0000, %esp
movl $0x10000000, %eax
1:
  cmp $1, %eax
  jne 1b;
```

In this example, the application is setting the stack pointer to an invalid address and looping in an infinite loop. An interrupt at some point will force CPU to save the cs, eip, and eflags at an invalid stack location in ring-0, ultimately causing some bad things to happen.

What about stack?

- The OS can’t trust the user program stack pointer
- The OS allocates a kernel stack for each user thread
- The hardware automatically switches to the kernel stack on interrupt

What about stack?

- How does the CPU know, where is the kernel stack?

What about stack?

- How does the CPU know, where is the kernel stack?
  - A special structure TSS contains the address of kernel's stack and stack segment
    - OS allocates the kernel stack and modifies the TSS before scheduling a new process

schedule

```
struct thread *ready_list;
struct thread *cur_thread;

void schedule() {
  struct thread *prev = cur_thread;
  struct thread *next =  pop_front(ready_list);
  cur_thread = next;
  update_tss(next->kstack, next->kss);
  update_mmu_mappings(next);
  context_switch(prev, next);
}
```

OS kernel creates a kernel stack corresponding to each user stack. During scheduling, the OS rewrites the kernel stack and stack segment values in the tss structure with the target process' kernel stack pointer and stack segment.

Kernel stack

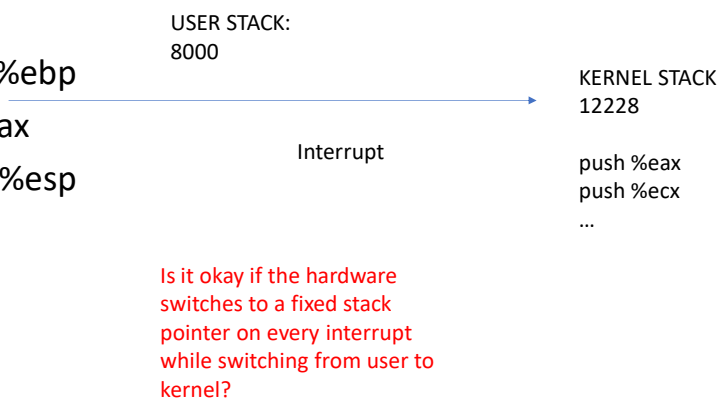
Corresponding to every user stack, the OS allocates a kernel stack.
The starting address of the kernel stack is stored in the tss structure.

User stack : 4096 - 8192
Kernel stack : 8192 – 12228

tss->stack = 12228

Kernel stack

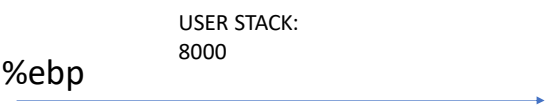
```
foo:
  push %ebp
  mov %esp, %ebp
  mov $1, %eax
  mov %ebp, %esp
  pop %ebp
  ret
```



Is it okay if the hardware switches to a fixed stack pointer on every interrupt while switching from user to kernel?

Kernel stack

```
foo:
  push %ebp
  mov %esp, %ebp
  mov $1, %eax
  mov %ebp, %esp
  pop %ebp
  ret
```

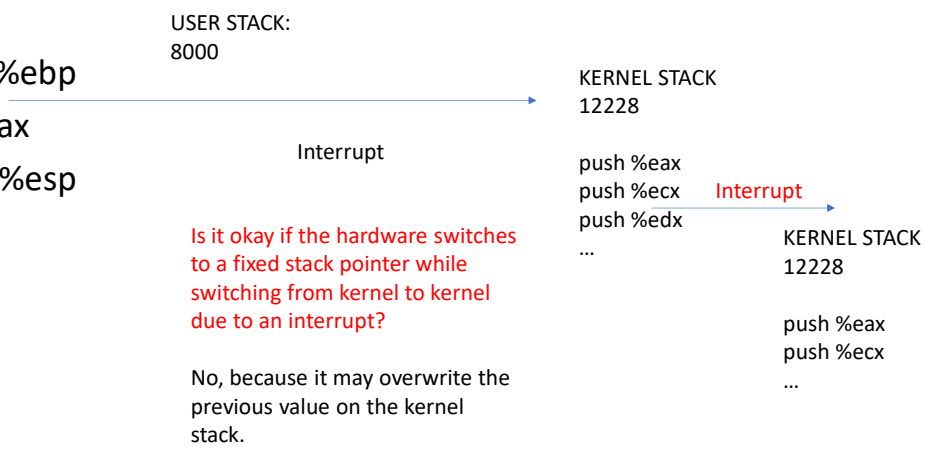


Is it okay if the hardware switches to a fixed stack pointer on every interrupt while switching from user to kernel?

Yes, on entry to the kernel, the stack is stateless. Similar to entry to main.

Kernel stack

```
foo:
  push %ebp
  mov %esp, %ebp
  mov $1, %eax
  mov %ebp, %esp
  pop %ebp
  ret
```



Is it okay if the hardware switches to a fixed stack pointer while switching from kernel to kernel due to an interrupt?

No, because it may overwrite the previous value on the kernel stack.

Kernel ss

- Why we need a different stack segment for the kernel?
  - Can we use the user's stack segment in the kernel?
  - Let us assume, for now, that the user application can't change the ss segment

Kernel ss

- Why we need a different stack segment for the kernel?
  - Can we use the user's stack segment in the kernel?
  - Let us assume, for now, that the user application can't change the ss segment
- If we execute, the kernel on user-stack the kernel local variables will be allocated on the user-stack that the user application can read after the interrupt handler returns
  - This may potentially leak the other processes data

The question is, can we map the kernel stack in the process address space, i.e., the process can read/write to the kernel stack. However, the hardware will automatically set the stack pointer with the address of the kernel stack on interrupt.

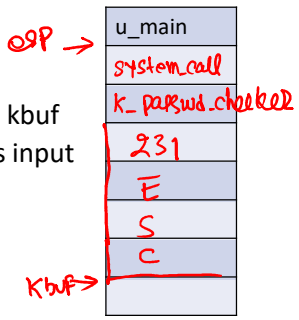
Kernel ss

```
void k_passwd_checker(char *passwd) {
   char kbuf[128];
   read_passwd(kbuf);                    // copy original passwd to kbuf
   if (!memcmp(kbuf, passwd, 128))       // cmp passwd with user's input
        allow_access();
   return error;
}

int u_main() {
   request_access("hello");
}
```

In this example, u_main is the user routine, and k_passwd_checker is the kernel routine. k_passwd_checker reads the original password in its local variable kbuf (allocated on kernel stack) and compares the password with the user's password. If they match, the OS provides access; otherwise, it returns an error status.

Kernel ss

```
void k_passwd_checker(char *passwd) {
   char kbuf[128];
   read_passwd(kbuf);                    // read original passwd in kbuf
   if (!memcmp(kbuf, passwd, 128))       // cmp passwd with user's input
        allow_access();
   return error;
}

int u_main() {
   request_access("hello");
   // read stack to get the value of kbuf, call access again with correct passwd
}
```



Notice that the kbuf is allocated on the kernel stack, and after returning from the system call, the real password has been copied to kbuf, which is still there on the kernel stack. After returning from the system call, the user program can inspect the kernel stack (because it is in the process' address space) to obtain the real password. It can again do the system call with the correct password to get access.

Kernel ss

- If the kernel uses the user's ss, the user program can overwrite the return address on the stack to crash the system
  - e.g., the user application can overwrite schedule1 with an invalid address to crash the system
  - How?

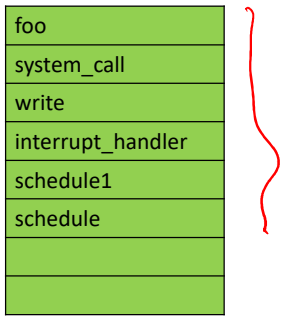
Kernel ss

- If the kernel uses the user's ss, the user program can overwrite the return address on the stack to crash the system
  - e.g., the user application can overwrite schedule1 with an invalid address to crash the system
  - How?

Even though the current thread is in the OS, the other threads of the current application can access the stack.

Kernel ss

```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```

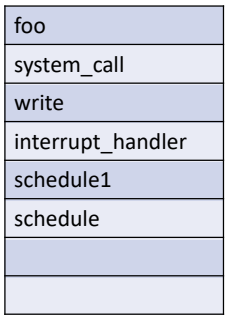


foo was running
foo did write system call
interrupt was triggered
schedule1 was invoked
schedule was invoked
schedule picked bar
right now, context_switch is executing

Let us assume that the user application doesn’t corrupt the stack pointer, and the interrupt handler executes the OS on the user’s stack. Let’s say foo was the current thread, and it is being preempted due to a timer interrupt. The call stack at the start of the context_switch routine is shown on this slide.

Kernel ss

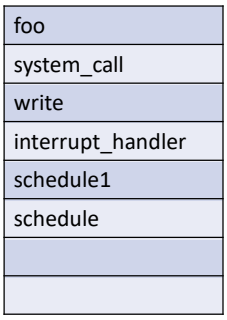
```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```



context_switch switches to bar’s stack

Kernel ss

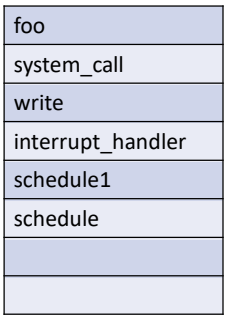
```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```



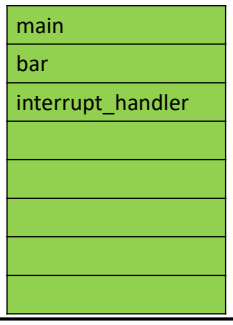
bar returns to schedule

Kernel ss

```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```

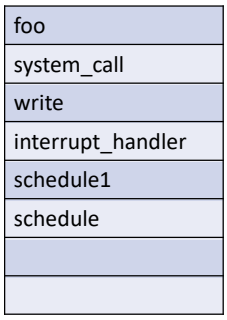


bar returns to schedule1



Kernel ss

```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```

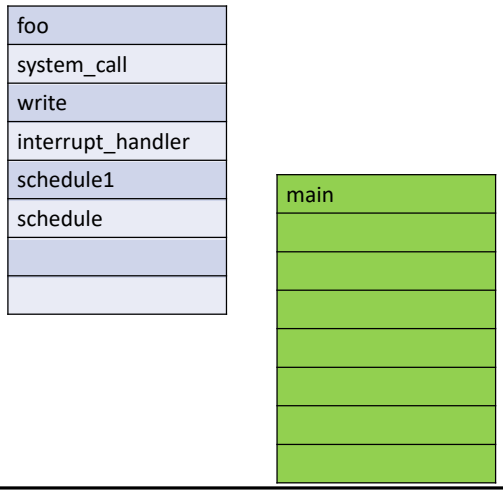


bar returns to interrupt_handler

Kernel ss

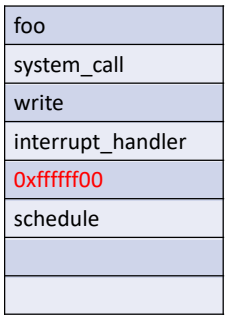
```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```

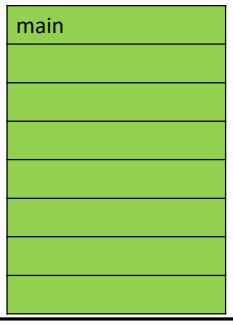
bar is running.



Kernel ss

```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```

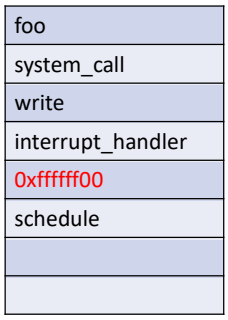




bar is running
bar overwrites the stack of foo in a way that, return address of schedule is now 0xffffff00

Kernel ss

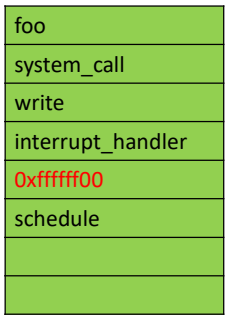
```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```



bar was executing
interrupt was triggered
scheduler picked foo
context_switch is called
right now, context_switch is executing

Kernel ss

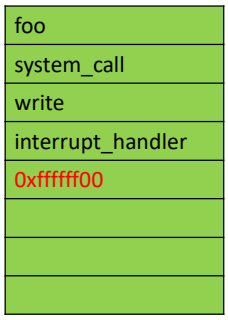
```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```



context_switch switches to foo’s stack

Kernel ss

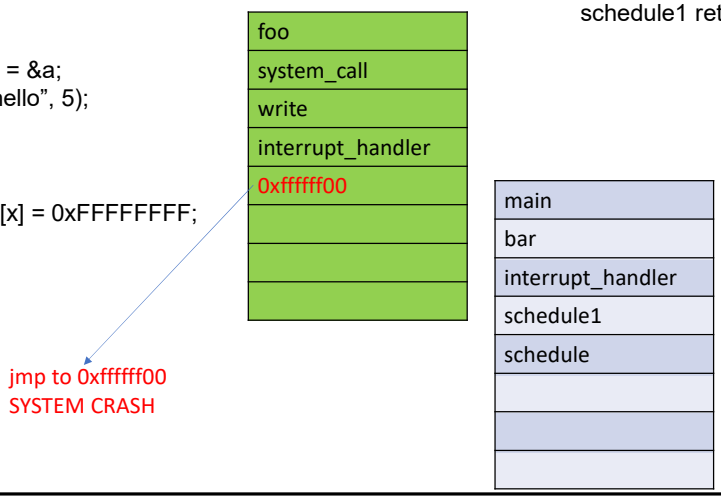
```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```



context_switch returns to schedule

Kernel ss

```
unsigned *foo_stack;

int foo() {
   int a;
   foo_stack = &a;
   write(1, “hello”, 5);
}

int bar() {
   foo_stack[x] = 0xFFFFFFFF;
}
```



Even though the current thread can’t change the return address inside the kernel, the other threads of the same process which are executing in the user-mode can change the return addresses of the other threads executing inside the kernel. For this reason, we must have to keep the kernel stack outside the process’ address space.

Interrupt

- How does OS restore the ss and esp of the user program after returning from an interrupt?

Interrupt

- How does OS restore the ss and esp of the user program after returning from an interrupt?
  - ss and esp of user program is pushed automatically on the kernel stack by the hardware on switching from user to kernel
  - iret restores the ss and esp from the stack when returning to the user mode

ss and esp

- Does the CPU need to switch stack when interrupted inside the kernel

```
interrupt_handler:
push %eax
push %ecx
push %edx
…
```

ss and esp

- Does the CPU need to switch stack when interrupted inside the kernel
  - why we did it for the user-program?
  - why kernel can't do the same thing?

interrupt_handler:

interrupt

push %eax

push %ecx

push %edx

…

ss and esp

- Does the CPU need to switch stack when interrupted inside the kernel
  - why we did it for the user-program?
  - why kernel can't do the same thing?

interrupt_handler:

push %eax ← **interrupt**

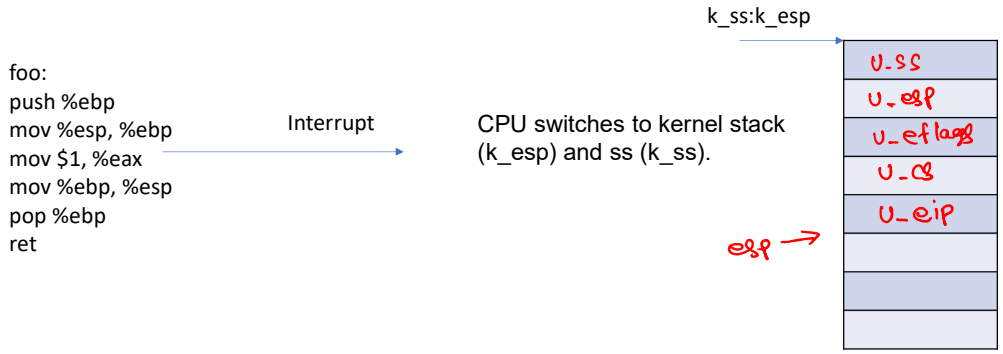
push %ecx

push %edx

…

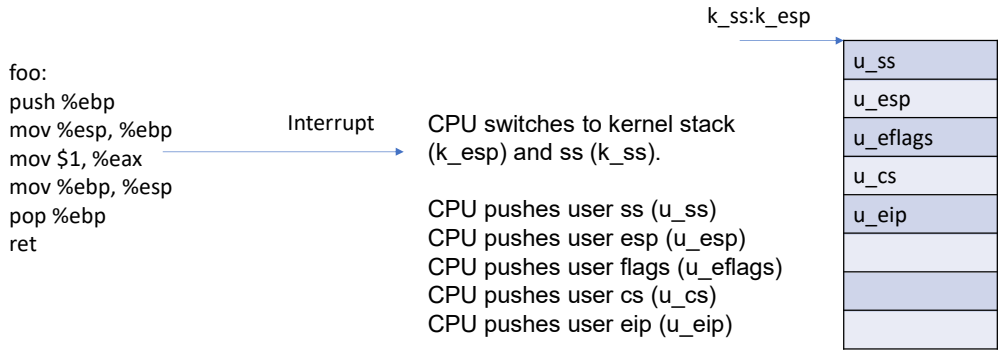
If we switch the stack, the old value will be overwritten; no need to switch stack because kernel's %esp and %ss are trusted.

Interrupts in user-mode

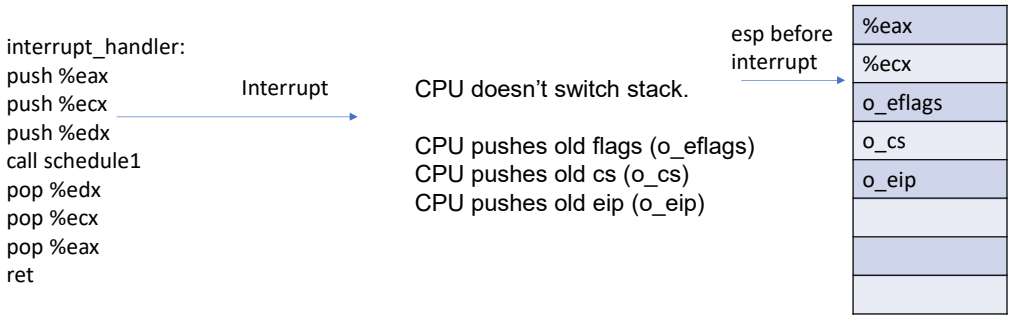


On switching from user to kernel, the hardware modifies %ss, %esp, %eflags, %cs, and %eip. The hardware pushes the old values of these registers on the kernel stack (after switching from user stack to kernel stack).

Interrupts in user-mode



Interrupts in kernel-mode

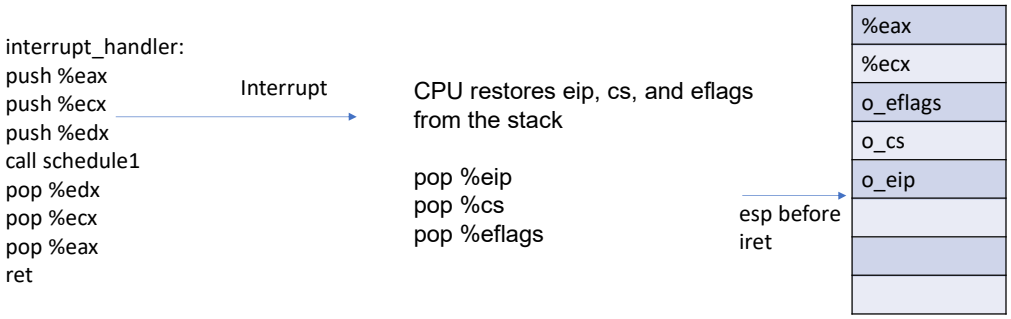


On interrupt in the kernel, the hardware doesn't switch stacks. So, the hardware pushes only three values: %eflags, %cs, and %eip on the kernel stack before overwriting them.

Current privilege level

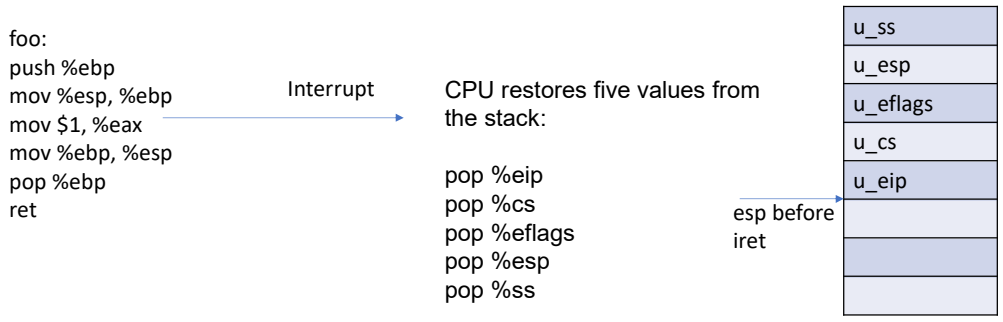
- How does CPU know the current privilege level (CPL)?
  - The last two bits of the cs register contain the CPL

iret to kernel-mode



On iret to kernel-mode, the hardware pops three values.

iret to user-mode



On iret to user-mode, the hardware pops five values.

iret

- How does CPU know how many values to pop (five or three)?

iret

- How does CPU know how many values to pop (five or three)?
  - After popping cs register if the last two bits in cs register is 3, it means that iret is returning to the user mode
    - In this case, CPU pops five values; otherwise, CPU pops three values

First process

- How does the first process return to the user-mode?

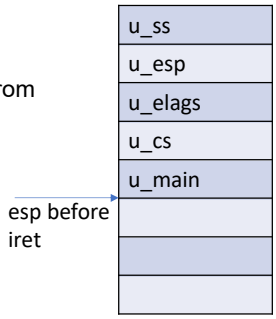
First process

OS manually setup the kernel stack before iret

CPU restores five values from the stack:

pop %eip
pop %cs
pop %eflags
pop %esp
pop %ss

After, iret the CPU jumps to main of the first process.



For the first process, the OS allocates a kernel stack, manually puts five values corresponding to the new thread on the kernel stack, sets the %esp to the newly created kernel stack, and executes the iret instruction.

Segmentation

- What happens if a user-application accesses a virtual address that is outside the limit of the segment?

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

movl $100, %ds:(%eax)

what is the PA, where
%eax = 1000
%ds = (1 << 3) | 3;

Segmentation

- What happens if a user-application accesses a virtual address that is outside the limit of the segment?
  - What is the problem with crashing the system?

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

```
movl $100, %ds:(%eax)
```

what is the PA, where
%eax = 1000
%ds = (1 << 3) | 3;

If we bring down the whole system when an application accesses an invalid address, then any process can bring the whole system down.

Segmentation

- What happens if a user-application accesses a virtual address that is outside the limit of the segment
  - The hardware generates an exception called general protection fault

idx	base	limit	DPL
0	1000	256	3
1	3000	128	3
2	8192	4096	3
3	0	0xffffffff	0

movl $100, %ds:(%eax)

what is the PA, where
%eax = 1000
%ds = (1 << 3) | 3;

We will discuss exceptions in the next class.