OSDI Lab4

Task scheduling

Outline

- System call trap
- Privilege level switch
- Stack switch
- Context switch

System call

- In x86 system, it use “int” instruction to invoke a system trap.

```
static inline int32_t
syscall(int num, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
        int32_t ret;

        // Generic system call: pass system call number in AX,
        // up to five parameters in DX, CX, BX, DI, SI.
        // Interrupt kernel with T_SYSCALL.
        //
        // The "volatile" tells the assembler not to optimize
        // this instruction away just because we don't use the
        // return value.
        //
        // The last clause tells the assembler that this can
        // potentially change the condition codes and arbitrary
        // memory locations.

        asm volatile("int %1\n"
                : "=a" (ret)
                : "i" (T_SYSCALL),
                  "a" (num),
                  "d" (a1),
                  "c" (a2),
                  "b" (a3),
                  "D" (a4),
                  "S" (a5)
                : "cc", "memory");

        //if(check && ret > 0)
        //      panic("syscall %d returned %d (> 0)", num, ret);

        return ret;
}
```

X86 Protection Ring

- Why need protection ring?
  - Some instruction do not use in user/application mode, such like reload interrupt table, setup page table,…,etc.
- Linux only use level 0 and level 3 as kernel and user mode.

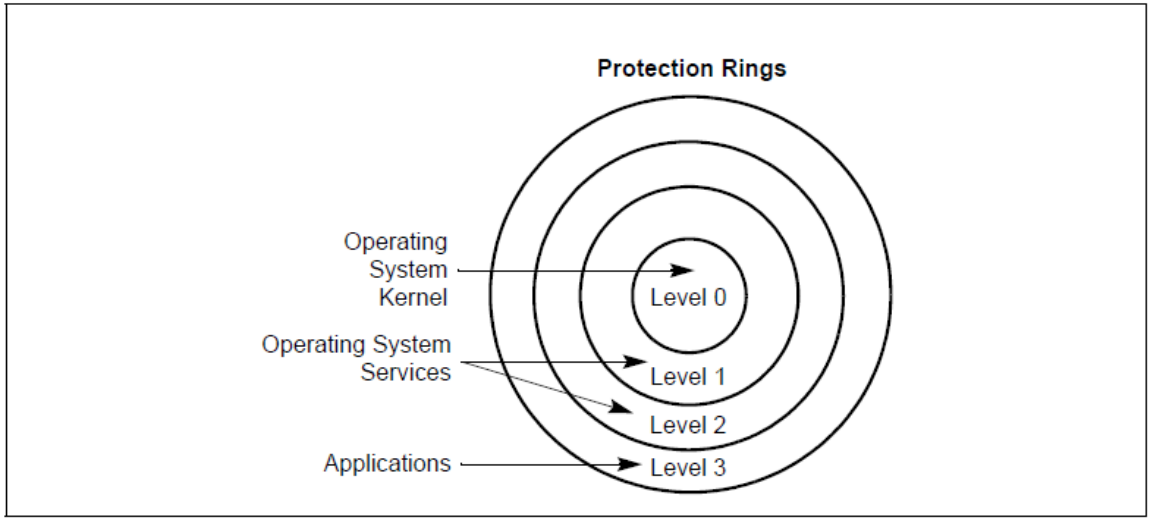
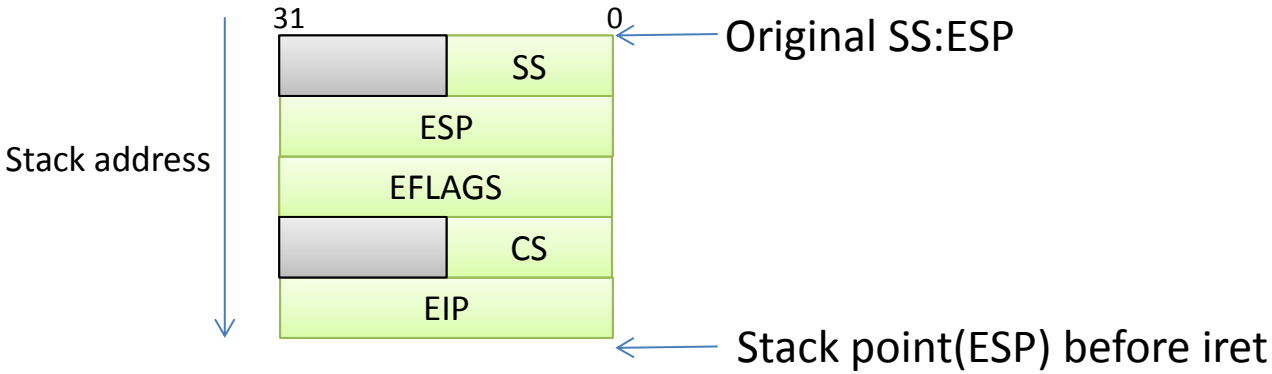


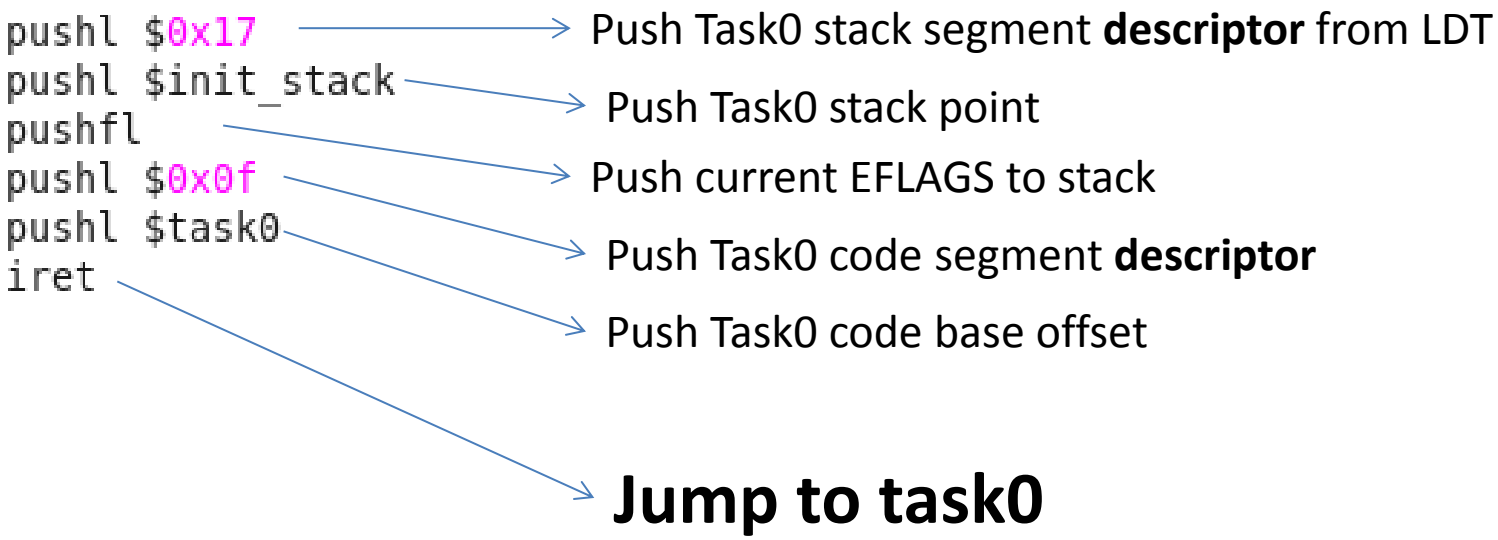
Figure 5-3. Protection Rings

Task Privilege Level Switch

- User -> Kernel
  - Just jump to a Gate Descriptor
    - Ex: int 0x80
- Kernel -> User mode
  - System only do it once at kernel startup
  - Simulate the interrupt return stack placement and use “iret”(interrupt return) instruction



Kernel -> User mode Example

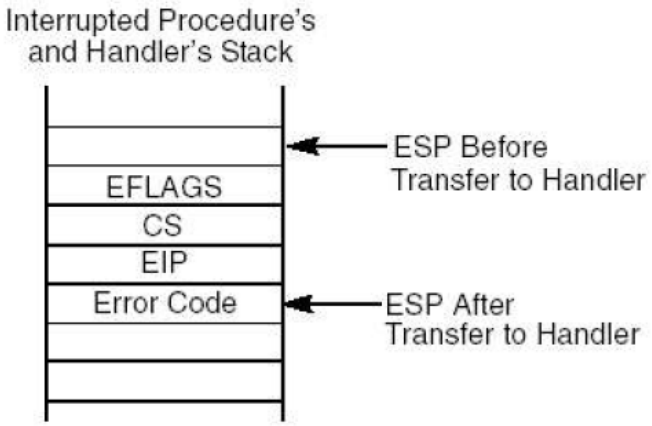


Exception Entry Mechanism

Kernel»Kernel

(New State)

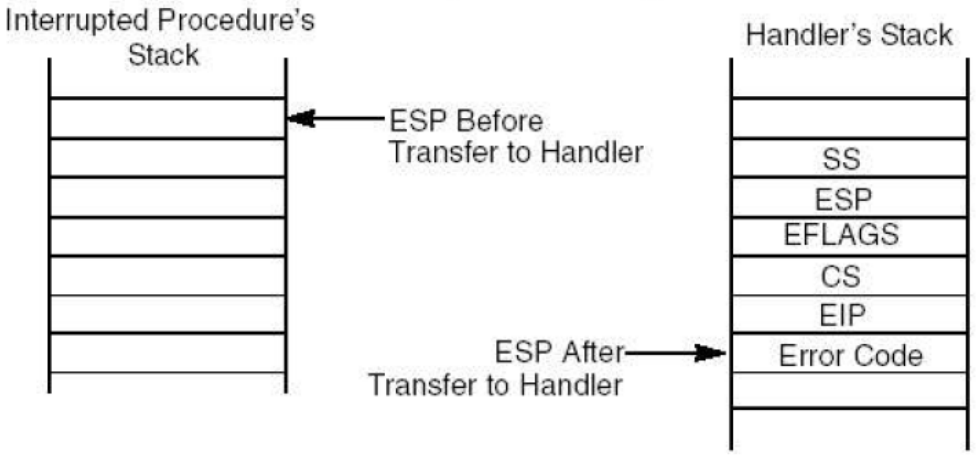
SS unchanged
ESP (new frame pushed)
CS:EIP (from IDT)



User»Kernel

(New State)

SS:ESP TSS ss0:esp0
CS:EIP (from IDT)
EFLAGS:
interrupt gates: clear IF



Reference: http://pdos.csail.mit.edu/6.828/

TSS Descriptor

- When the G flag is 0 in a TSS descriptor for a 32-bit TSS, the limit field must have a value equal to or greater than 67H

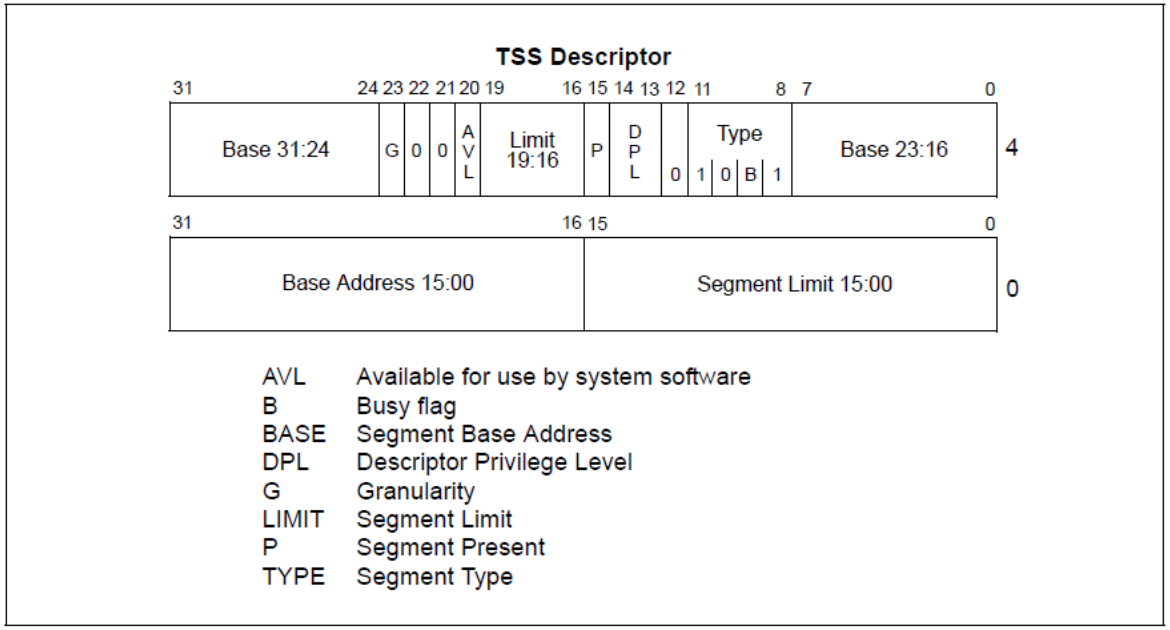


Figure 7-3. TSS Descriptor

type: 16bitTSS(0x1=available or 0x3=busy) or 32bitTSS(0x9=available or 0xB=busy)

Task-state segment (TSS)

- Specialized Segment for hardware supported multi-tasking
- **(we don't use this x86 feature)**
- In JOS's TSS
  - SS0:ESP0 kernel stack used by interrupt handlers.

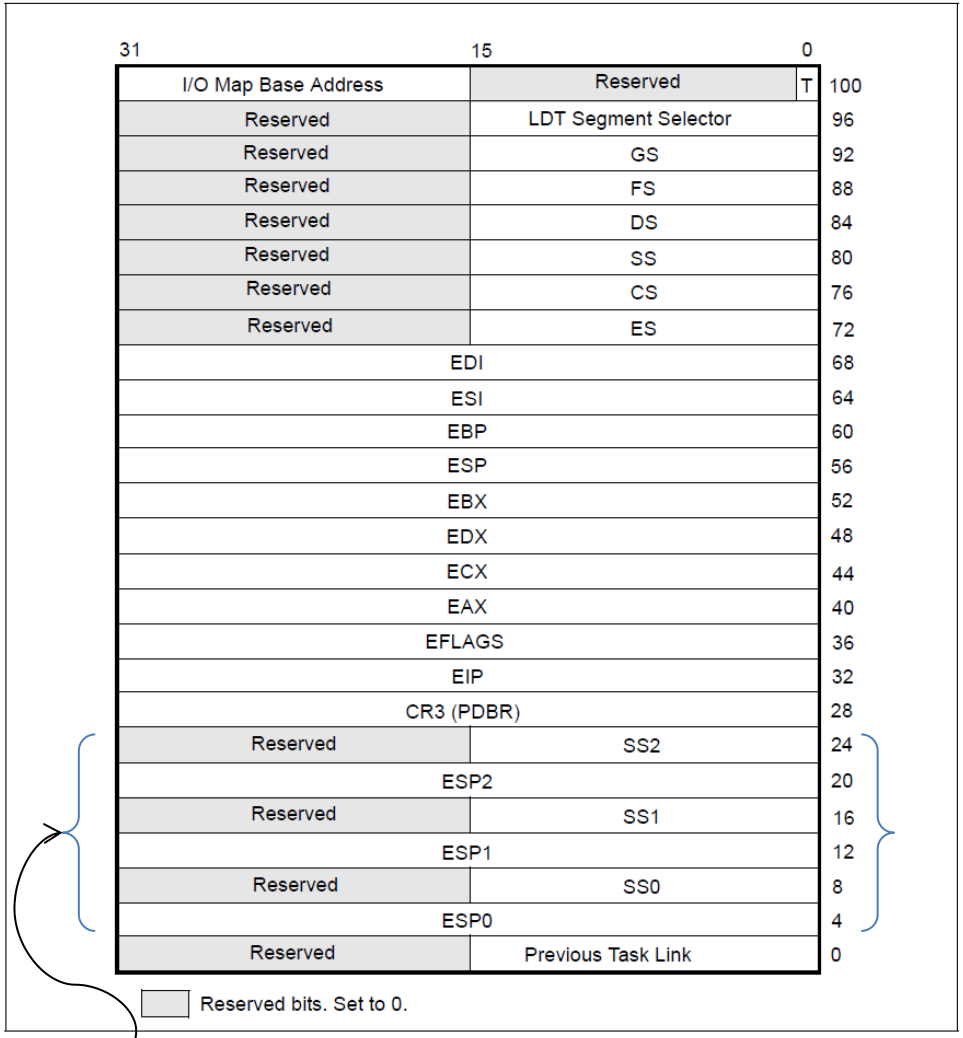


Figure 7-2. 32-Bit Task-State Segment (TSS)

Privilege level-0, -1, and -2 stack pointer fields

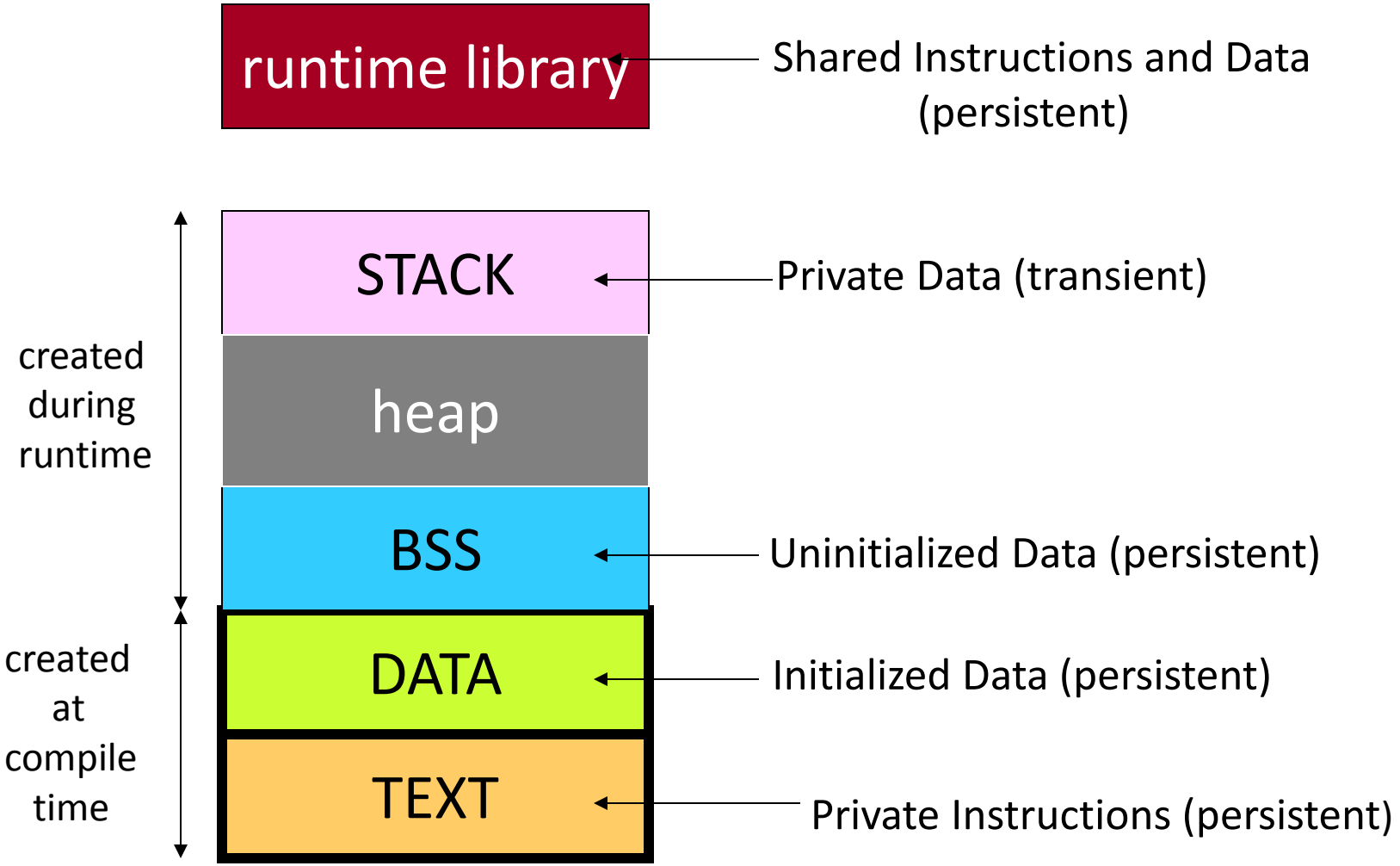
Reference: http://pdos.csail.mit.edu/6.828/

Program Model

- Programs consist of data and instructions
- Data consists of constants and variables, which may be ‘persistent’ or ‘transient’
- Instructions may be ‘private’ or ‘shared’
- These observations lead to a conceptual model for the management of programs, and to special processor capabilities that assist in supporting that conceptual model

Reference: www.cs.usfca.edu/~cruse/cs630f06/lesson08.ppt

Conceptual Program-Model



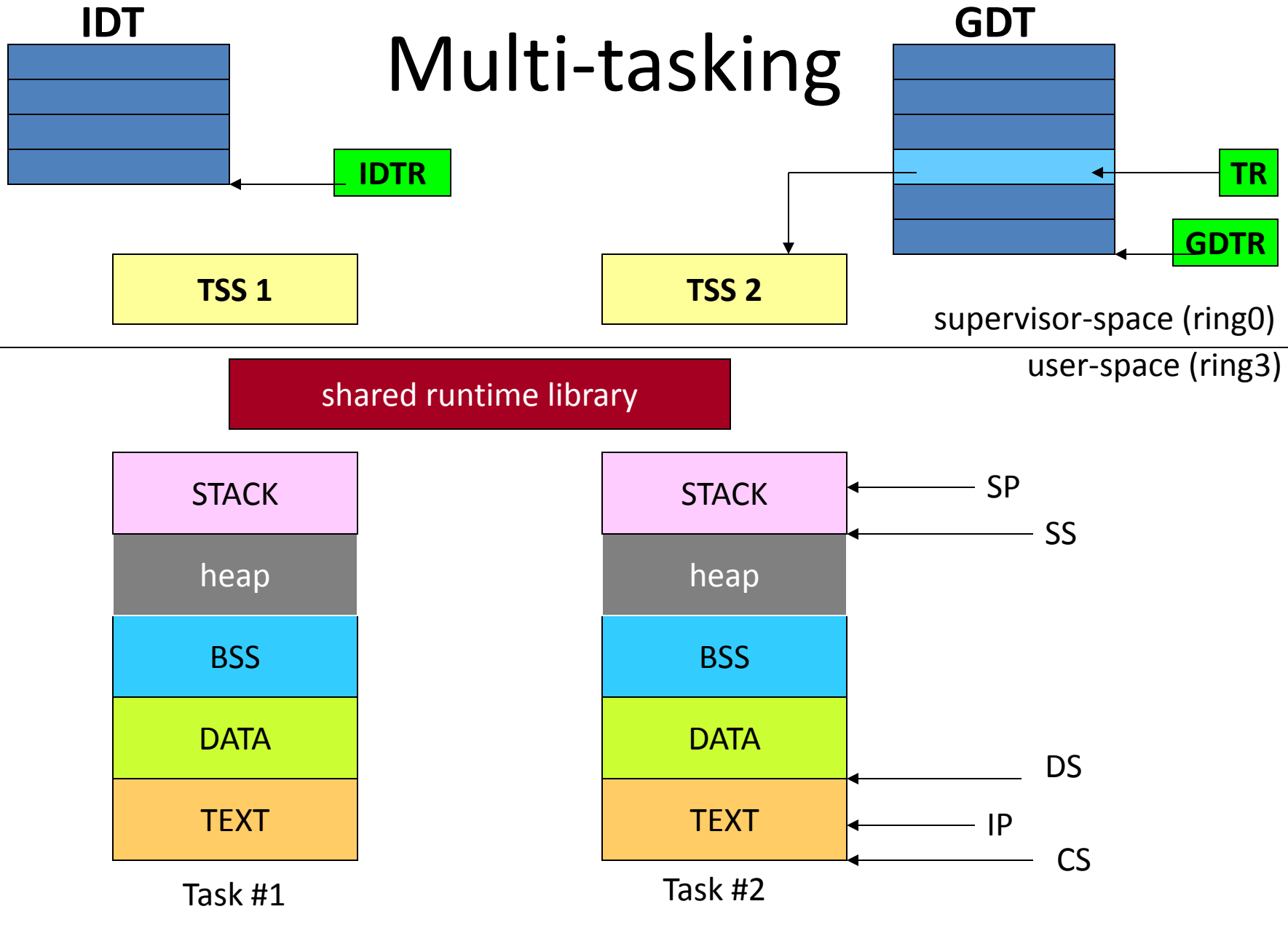
Reference: www.cs.usfca.edu/~cruse/cs630f06/lesson08.ppt

Task Isolation

- The CPU is designed to assist the system software in isolating the private portions of one program from those of another while they both are residing in physical memory, while allowing them also to share certain instructions and data in a controlled way
- This ‘sharing’ includes access to the CPU, whereby the tasks take turns at executing

Reference: www.cs.usfca.edu/~cruse/cs630f06/lesson08.ppt

Multi-tasking



Reference: www.cs.usfca.edu/~cruse/cs630f06/lesson08.ppt

Context-Switching

- The CPU can perform a ‘context-switch’ to save the current values of all its registers (in the memory-area referenced by the TR register), and to load new values into all its registers (from the memory-area specified a new Task-State Segment selector)
- There are four ways to trigger this ‘switch’ operation on x86 processors

Reference: www.cs.usfca.edu/~cruse/cs630f06/lesson08.ppt

How to cause a task-switch

- Use an ‘ljmp’ instruction (long jump):

  ljmp $task_selector, $0

- Use an ‘lcall’ instruction (long call):

  lcall $task_selector, $0

- Use an ‘int-n’ instruction (with a task-gate):

  int $0x80

- Use an ‘iret’ instruction (with NT=1):

  iret

Reference: www.cs.usfca.edu/~cruse/cs630f06/lesson08.ppt

Task Linking

- Use for nested task switch, when “Icall” instruction will set NT flag=1

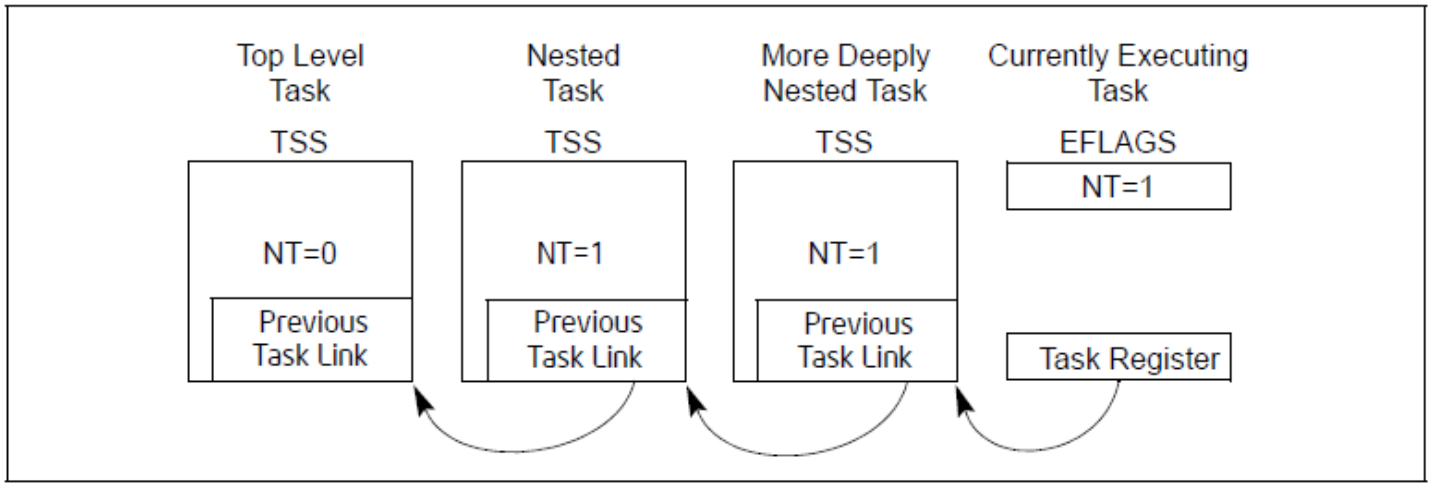
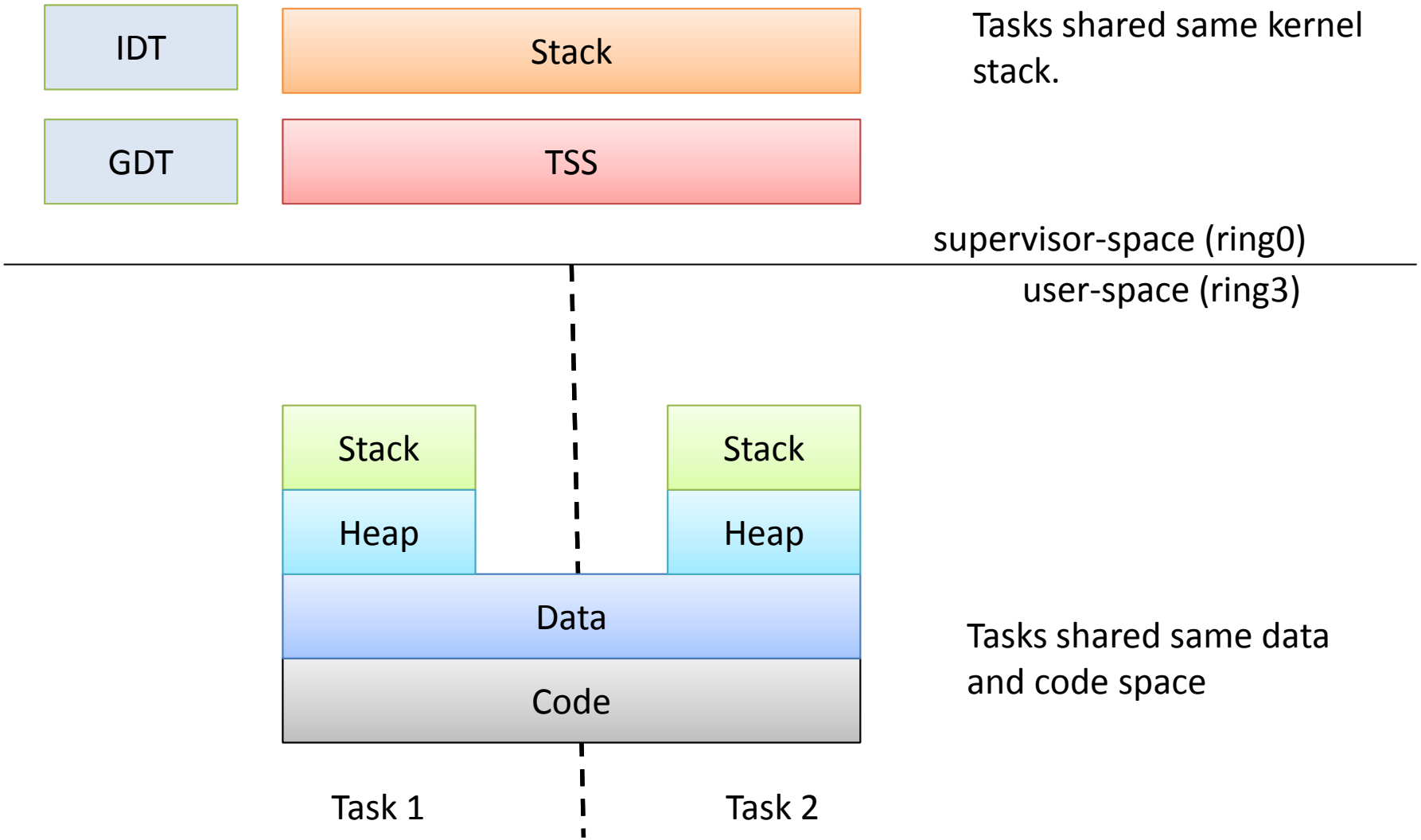


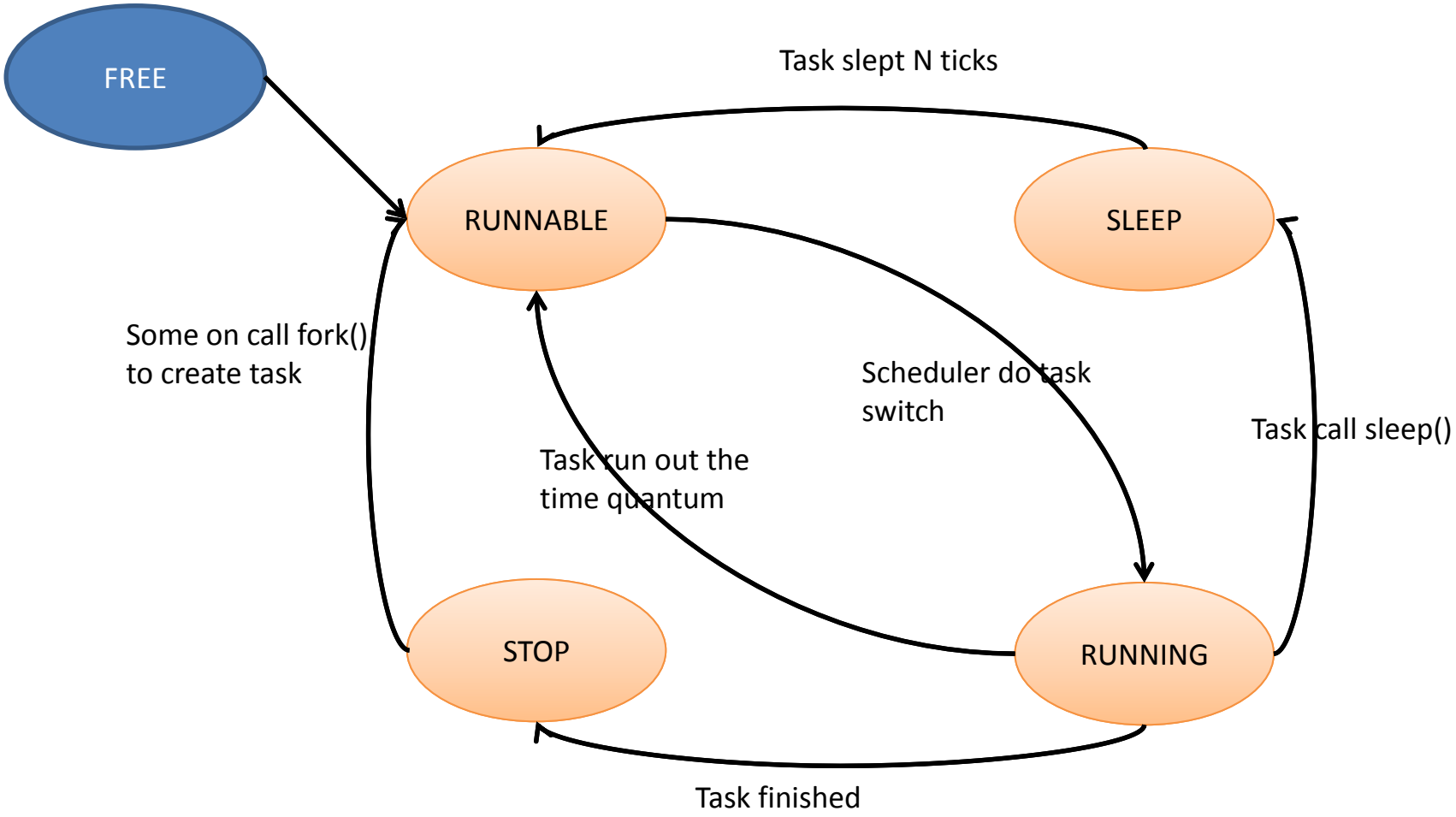
Figure 7-8. Nested Tasks

Task space model in lab4



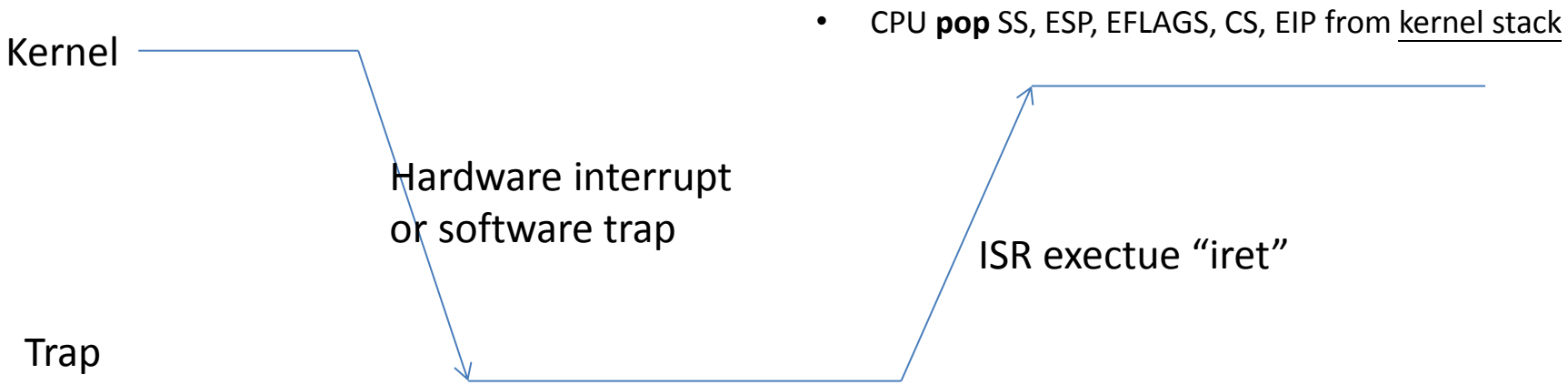
Task state diagram in lab4

Define in kernel/task.h



Interrupt/Exception behavior

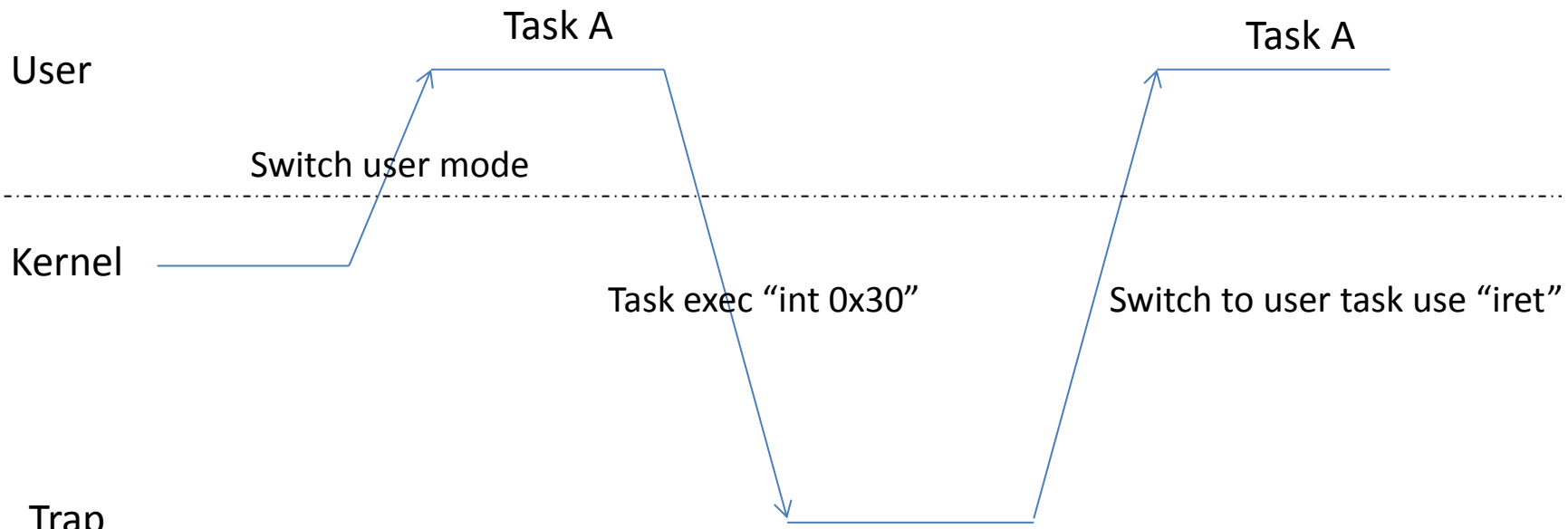
- External interrupt generated from Programmable Interrupt Controller (PIC or APIC)
- Software interrupt could generated from user trap (e.g. int xx) or exceptions (e.g. stack fault, divide zero…)



- CPU **pop** SS, ESP, EFLAGS, CS, EIP from kernel stack

- CPU found the trap gate in IDT
- CPU check the Privilege level (DPL)
- CPU **push** EIP, CS, EFLAGS, ESP, SS to kernel stack
- CPU load the code segment for IDT
- CPU jump trap entry (use trap gate)

System call



- Task pass the system call’s parameter in EAX,EBX… registers (e.g. call number, buffer address…)
- CPU do the trap process (check IDT, DPL,…)
- Do system call service

- **ISR save the task A’s trap stack and registers to task A’s Trapframe**
- ISR pass the system call value in EAX register
- Switch to user task A

Context switch

- Similar to system call but it generated by timer interrupt or sleep() system call.