Mechanisms for entering the system

Yolanda Becerra Fontal
Juan José Costa Prats

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech
2019-2020 QT

Content

- Introduction
- Mechanisms for entering the system
  - Initialization
  - Management
  - Example
- Procedure for entering the system
- Procedure to exit from system
- Exceptions
- Interrupts
- System calls
- Summary

Content

- **Introduction**
- Mechanisms for entering the system
  - Initialization
  - Management
  - Example
- Procedure for entering the system
- Procedure to exit from system
- Exceptions
- Interrupts
- System calls
- Summary

Introduction

- OS implements access to machine resources
  - Isolate users from low-level machine-dependent code
  - Group common code for all users: save disk space
  - Implement resource allocation policies
    - Arbitrate the usage of the machine resources in multi-user and multiprogrammed environments
  - Prevent machine and other users from user damage
    - Some instructions can not be executed by user codes: I/O instructions, halt,…

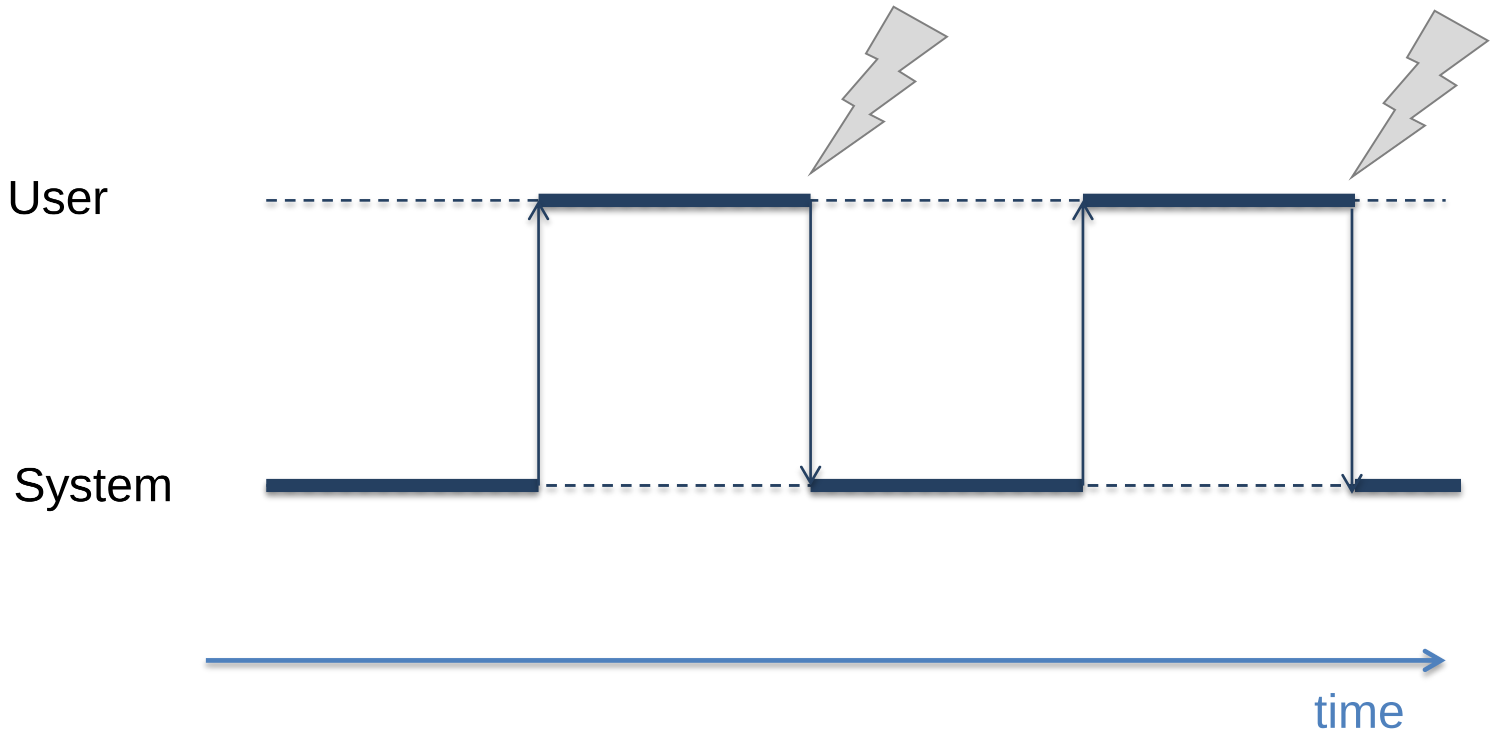
Privilege levels (I)

- Requirement:
  - Prevent users from direct access to resources
    - Ask the OS for services
- Privilege instructions
  - Instructions that only can execute the OS
  - HW support is needed
  - When a privilege instruction is executed, the hw checks if it is executing system code
    - If not → exception
- How to distinguish user code from system code?
  - Privilege levels
    - At least 2 different levels
    - System execution mode vs User execution mode
  - Intel defines 4 different privilege levels.

Privilege levels (II)

- How to scale privileges?
  - Intel offers interrupts
    - Interrupt Driven Operating System
  - When an interrupt/exception happens
    - Hw changes the current privilege level and enables the execution of privilege instructions
  - When the interrupt/exception management ends
    - Hw changes the current privilege level to unable the execution of privilege instructions

Interrupt driven OS



Content

- Introduction
- **Mechanisms for entering the system**
  - Initialization
  - Management
  - Example
- Procedure for entering the system
- Procedure to exit from system
- Exceptions
- Interrupts
- System calls
- Summary

Mechanisms for entering the system

- Exceptions
  - **Synchronous**, produced by the **CPU** control unit after terminating the execution of an instruction
- Interrupts
  - **Asynchronous**, produced by **other hardware devices** at arbitrary times
- System calls
  - **Synchronous**: **assembly instruction** to cause it
    - Trap (in Pentium: INT, sysenter…)
  - Mechanism to request OS services
- All of them are managed through the **interrupts vector**
  - New arquitectures implement a fast system call mechanism that skip the interrupts vector: sysenter instruction

Interrupts Vector

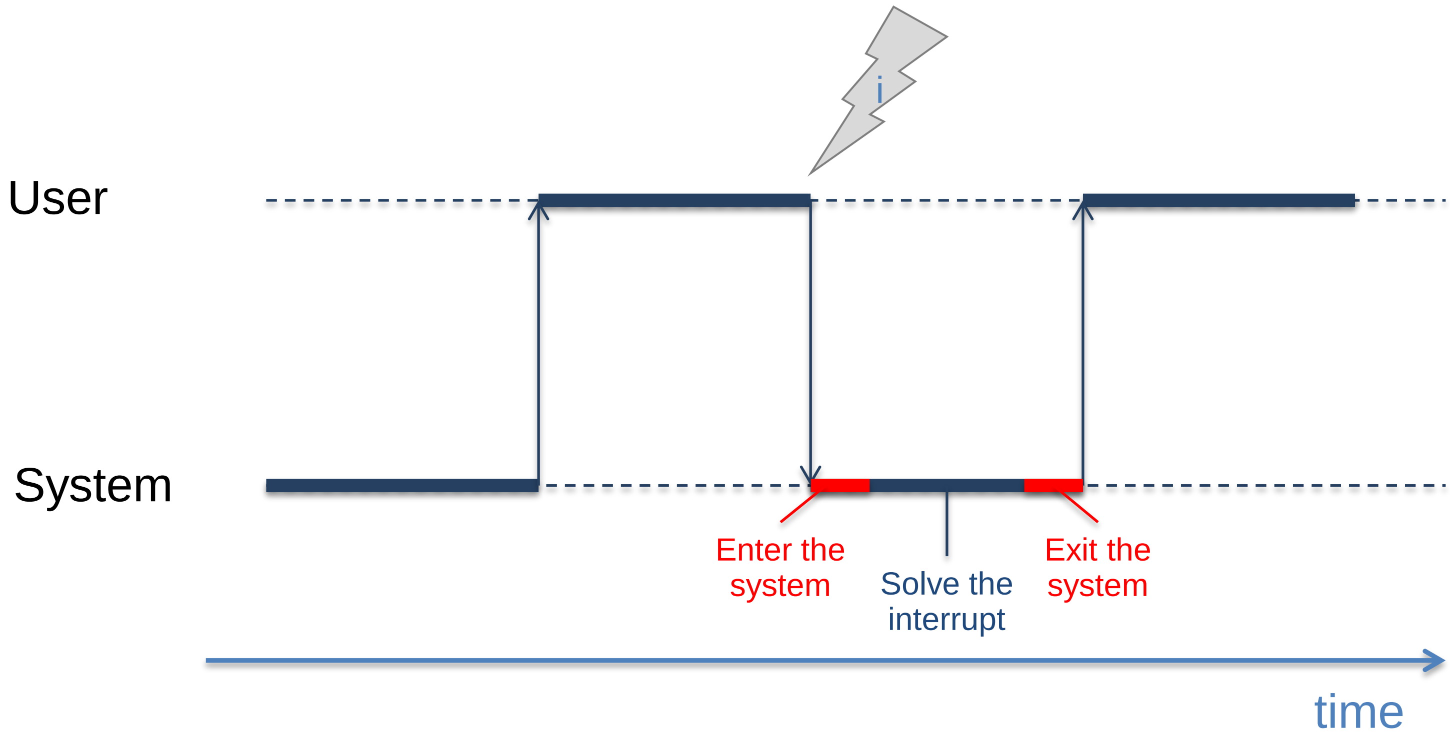
- Pentium
  - IDT: Interrupt Descriptor Table: 256 entries

- Three groups of entries, one for each kind of event:
  - 0 - 31: Exceptions
  - 32 - 47: Masked interrupts
  - 48 - 255: Software interrupts (Traps)

Initialization

- Each entry in the IDT, identifying an interrupt number, has:
  - A code address
    - Entry point to the routine's code to be executed
  - A privilege level
    - The minimum needed to execute the previous code

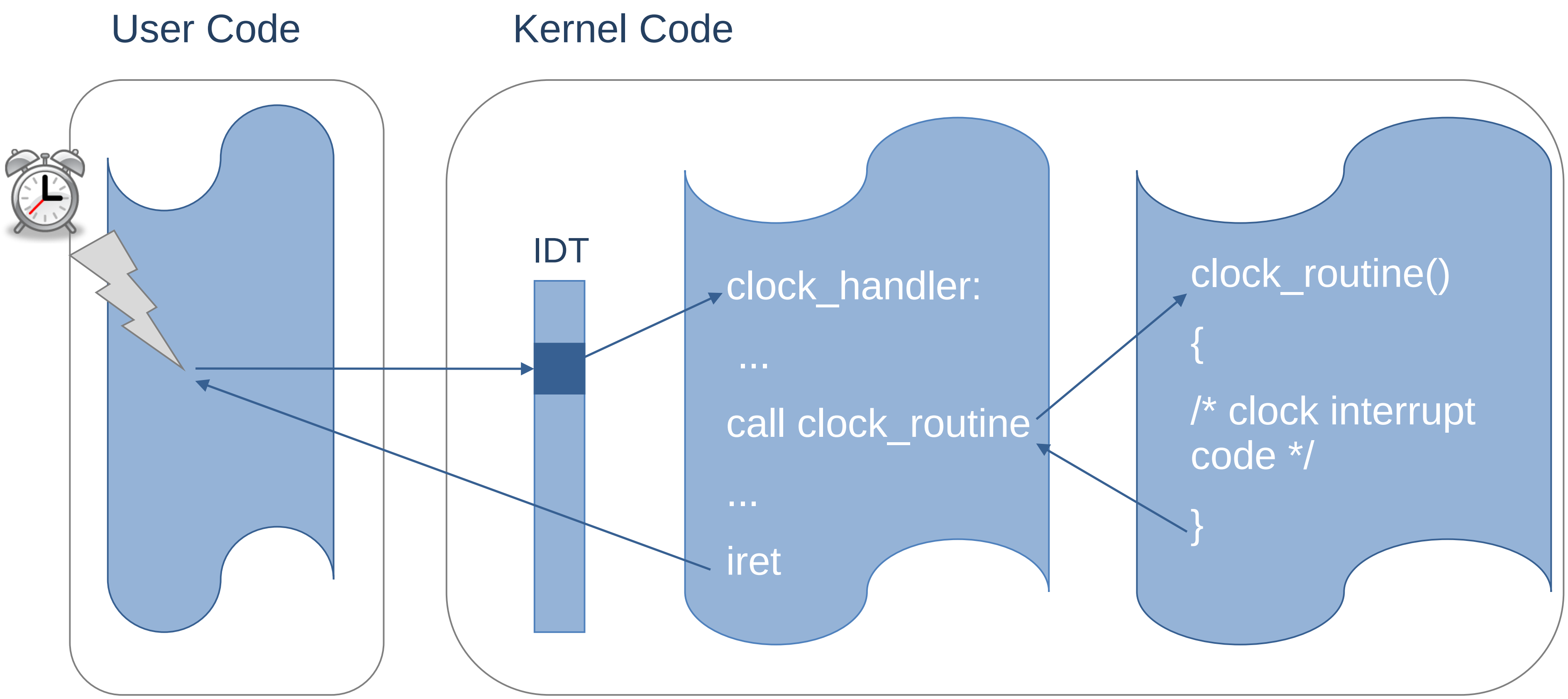
Management code



Management Code

- It could be done in a single routine
  - Divided in two parts: hw context mgmt + solve int.
- Hw context mgmt
  - Entry point handler
  - Basic hardware context management
  - Assembly code
  - Call to a Interrupt Service Routine
- Solve interrupt
  - Interrupt Service Routine
  - High level code (C for example)
  - Specific algorithm for each interrupt

Example: clock interrupt behavior



Content

- Introduction
- Mechanisms for entering the system
  - Initialization
  - Management
  - Example
- **Procedure for entering the system**
- Procedure to exit from system
- Exceptions
- Interrupts
- System calls
- Summary

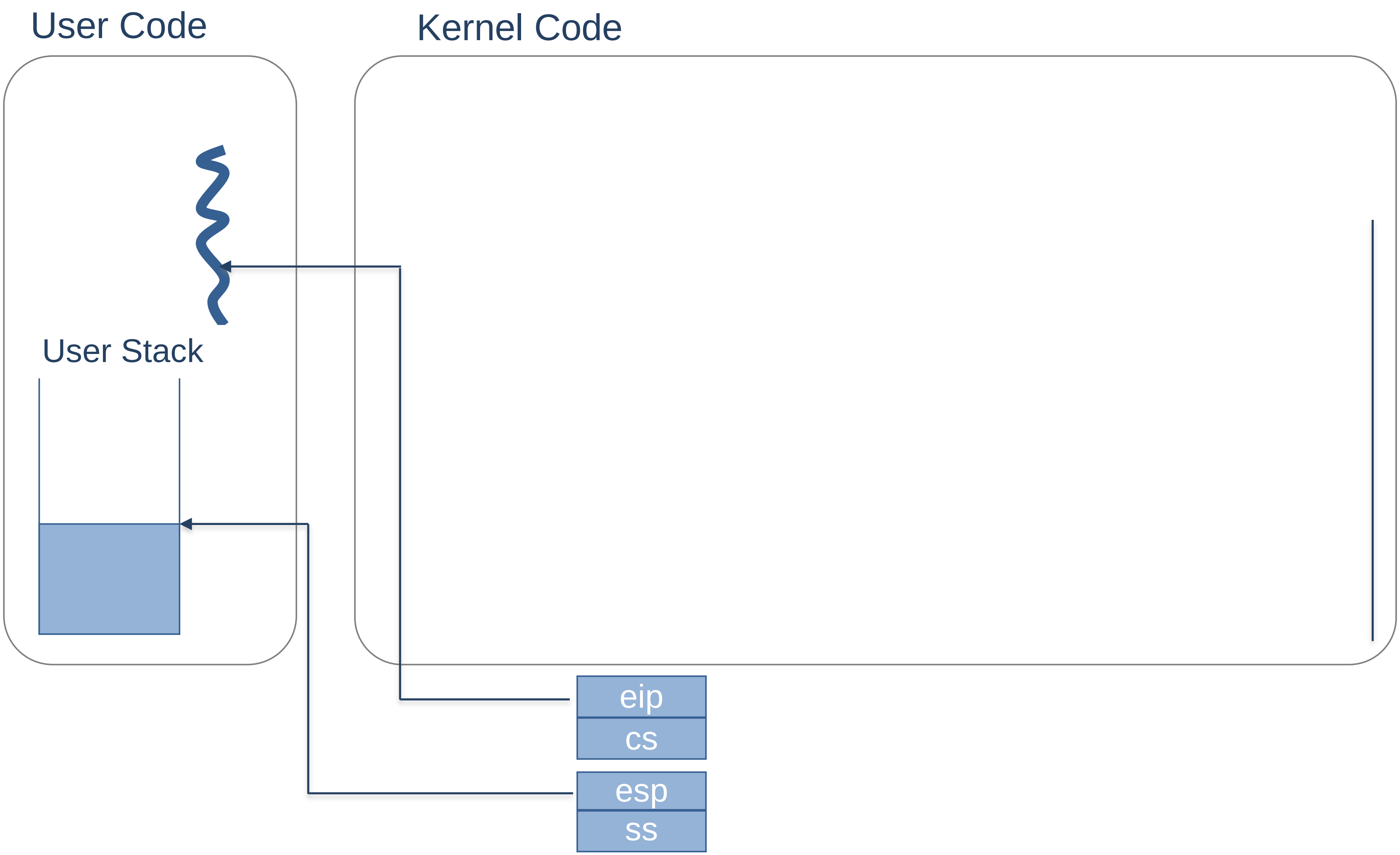
Procedure for entering the system

- Switch to protected execution mode
  - User Mode → Kernel Mode
- Save hardware context: CPU registers
  - ss, esp, psw, cs i eip
  - General purpose registers
- Execute service routine

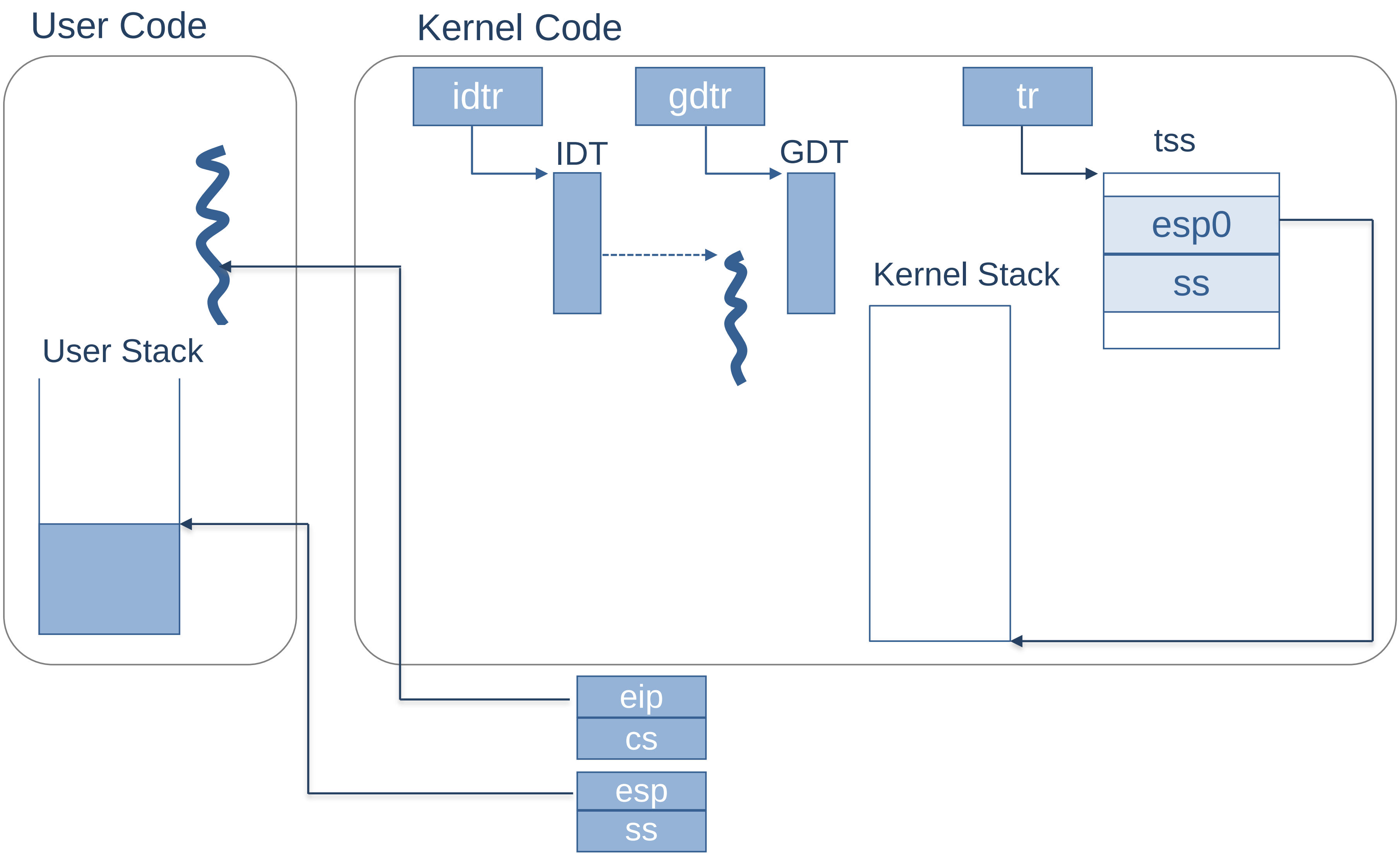
HW (switch to protected mode; save ss, esp, psw, cs i eip)

handler (save general purpose registers; execute service routine)

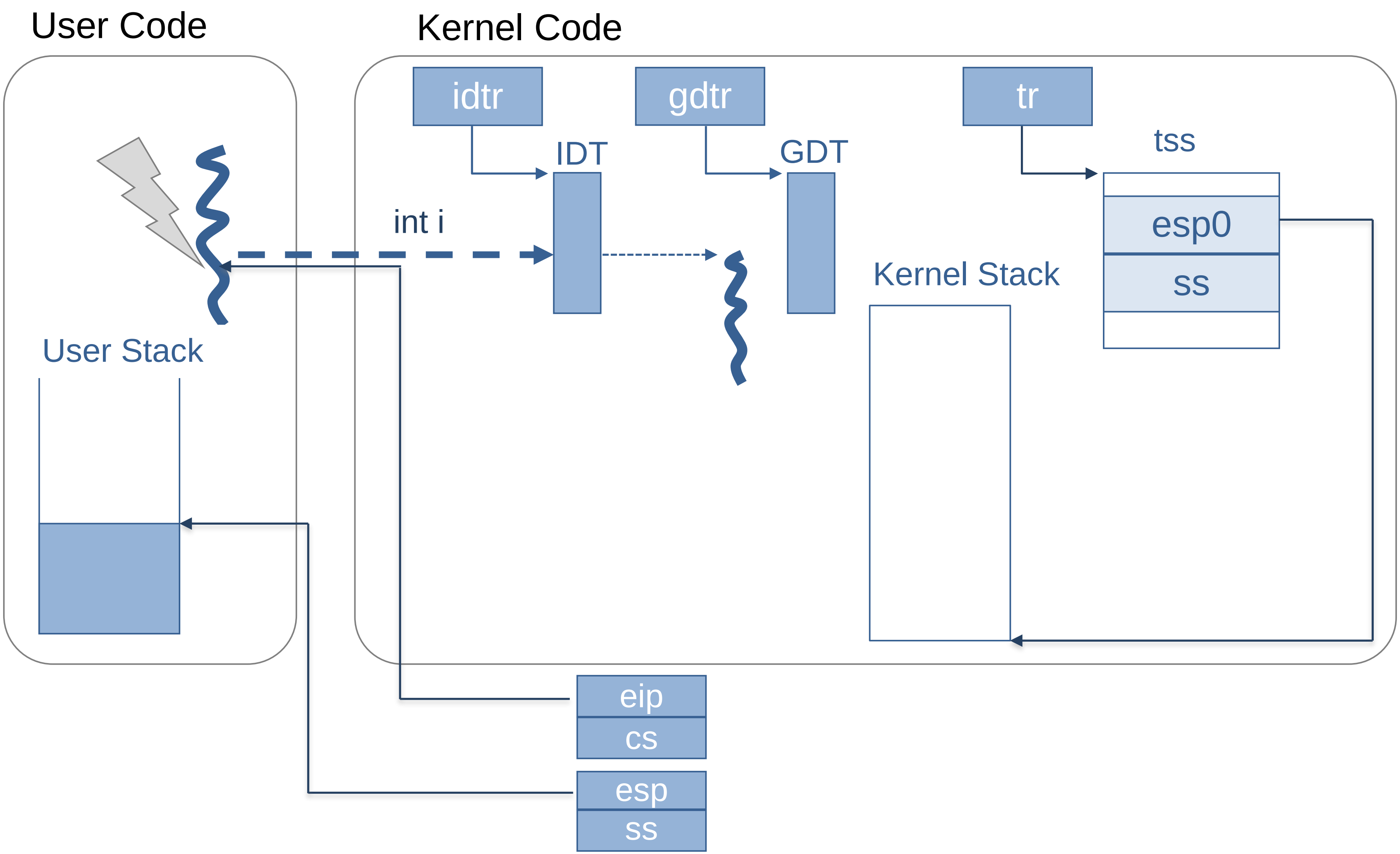
Procedure for entering the system



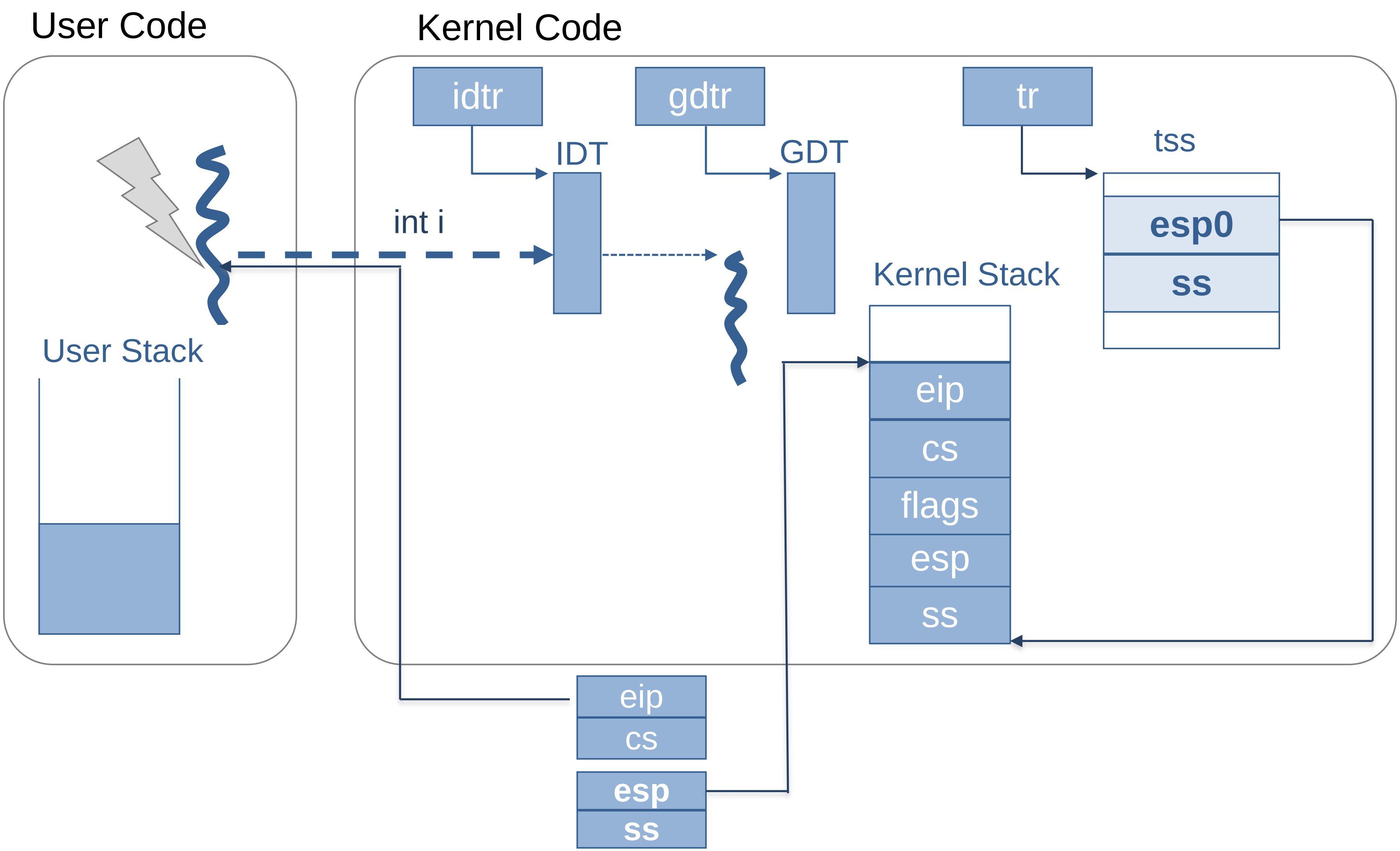
Procedure for entering the system



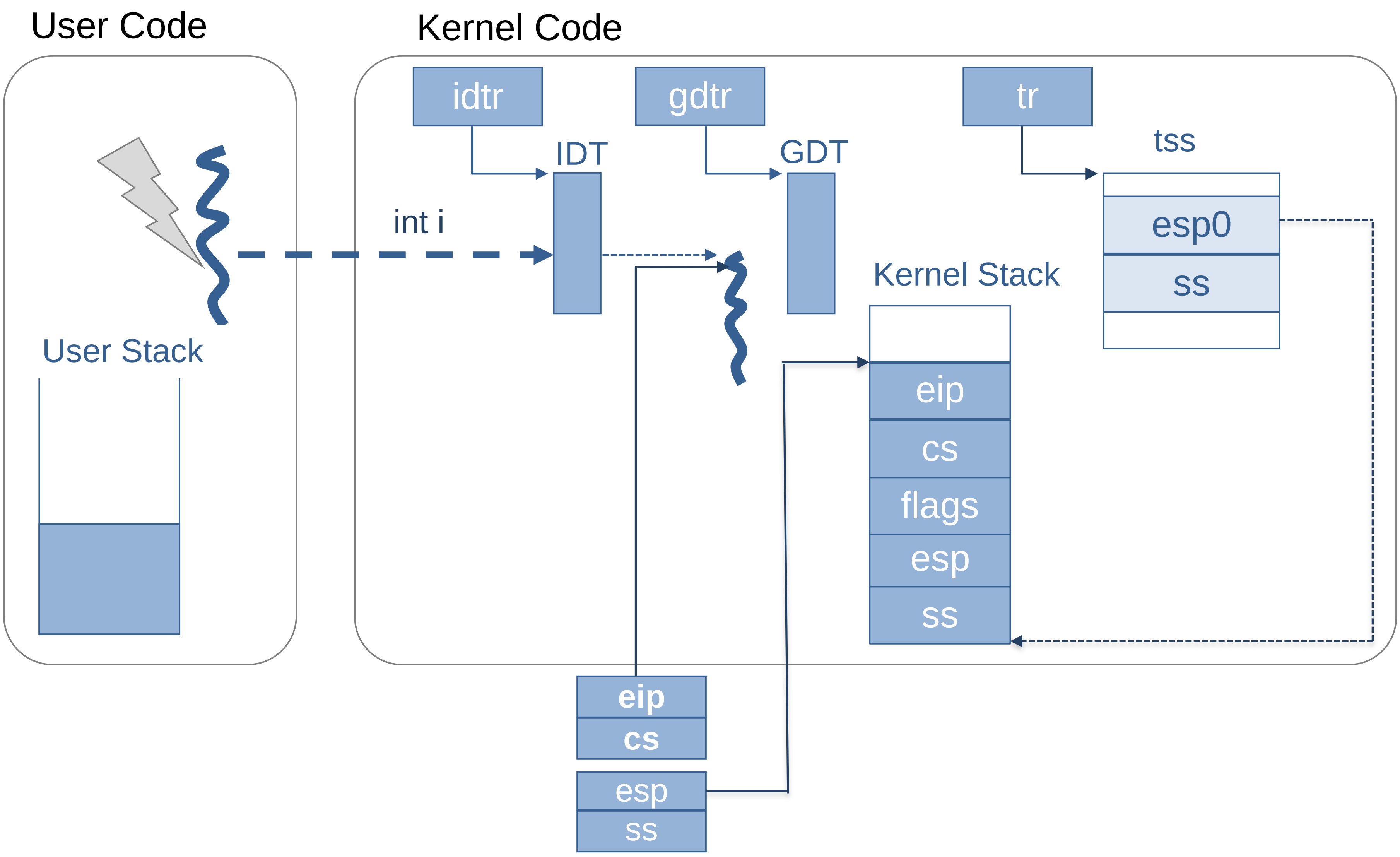
Procedure for entering the system



Procedure for entering the system



Procedure for entering the system



Content

- Introduction
- Mechanisms for entering the system
  - Initialization
  - Management
  - Example
- Procedure for entering the system
- **Procedure to exit from system**
- Exceptions
- Interrupts
- System calls
- Summary

Procedure to exit the system

- Restore HW context
  - General purpose registers
  - ss, esp, flags, cs, eip
- Switch execution mode
  - Kernel mode → User mode

handler (General purpose registers)

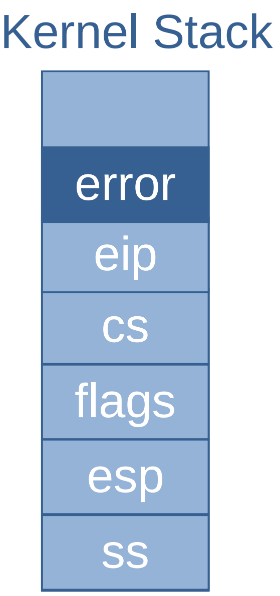
HW (iret instruction) (ss, esp, flags, cs, eip; Kernel mode → User mode)

Content

- Introduction
- Mechanisms for entering the system
  - Initialization
  - Management
  - Example
- Procedure for entering the system
- Procedure to exit from system
- **Exceptions**
- Interrupts
- System calls
- Summary

Exceptions: Stack layout

- There are some exceptions that push a parameter of 4 bytes (a hardware error code) to the kernel stack after entering the system:



Exception: IDT

# IDT	Exception	Error Code
0	Divide Error	
1	Debug Exception	
2	NMI Interrupt	
3	Breakpoint	
4	Overflow	
5	BOUND Range Exceeded	
6	Invalid Opcode (Undefined Opcode)	
7	Device Not Available (No Math Coprocessor)	
8	Double Fault	✓
9	Coprocessor Segment Overrun (reserved)	
10	Invalid TSS	✓
11	Segment Not Present	✓
12	Stack-Segment Fault	✓
13	General Protection	✓
14	Page Fault	✓
15	*(Intel reserved. Do not use.)*	
16	x87 FPU Floating-Point Error (Math Fault)	
17	Alignment Check	✓
18	Machine Check	
19	SIMD Floating-Point Exception	
20	Virtualization Exception	
21-31	(Intel reserved. Do not use.)	

Exception´s handler

- Save hardware context
- Call exception service routine
- Restore hardware context
- Remove error code (if present) from kernel stack
- Return to user (iret)

Content

Interrupt´s handler

- Similar to exception, but:
  - No hardware error code in kernel stack
  - It is necessary to notify the interrupt controller when the interrupt management finishes
    - Meaning that a new interrupt can be processed
    - End Of Interrupt (EOI)

Content

- Introduction
- Mechanisms for entering the system
  - Initialization
  - Management
  - Example
- Procedure for entering the system
- Procedure to exit from system
- Exceptions
- Interrupts
- **System calls**
- Summary

Handling system calls

- Why cannot be invoked like a regular user function?
- Which is the mechanism to identify the system call?
- How to pass parameters to the kernel?
- How to get results from the kernel?

System calls: invocation and identification

- Assembly instruction that causes a software generated interrupt
  - **int** assembly instruction (*int idt_entry*)
  - Alternative: **sysenter** assembly instruction: fast system call mechanism
- An entry point per syscalls?
  - Limitation for the potential number of syscalls
- A single entry point is used for all system calls
  - int
    - 0x80 for Linux
    - 0x2e for Windows
  - sysenter
    - system call handler @ is kept on a control register: SYSENTER_EIP_MSR
- And an extra parameter (EAX) to identify the requested service
- A table is used to translate the user service request to a kernel function to execute

System calls: parameters and results

- Parameter passing: Stack is NOT shared
  - Linux: syscall handler expects parameters in the registers
    - (first parameter) ebx, ecx, edx, esi, edi, ebp
    - Copy parameters from user stack
  - Windows: Use a register to pass a pointer to parameters
    - EBX
- Returning results:
  - EAX register: contains error code

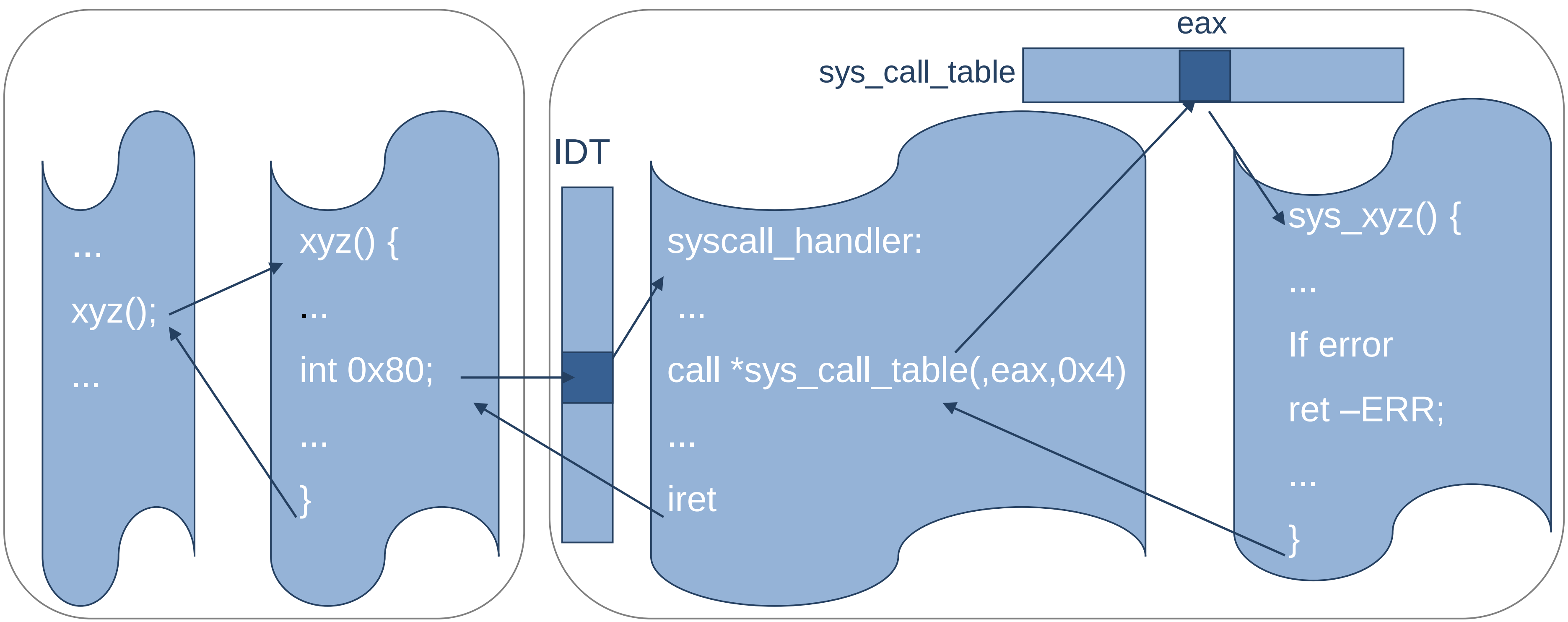
System call wrappers

- System must provide the users with an easy and portable way to use them
  - New layer: wrappers
    - wrap all the gory details in a simple function call
- Wrapper responsibilities
  - Invoke the system call handler
    - Responsible for parameter passing
    - Identify the system call requested
    - Generate the trap
  - Return the result to the user code
    - Use errno variable to codify type of error and returns -1 to users

System call mechanism overview

User Code

Kernel Code



System call invocation in application program

Wrapper for system call

system call handler

system call service routine

Fast System calls: sysenter/sysexit

- Avoid interrupt mechanism
- Avoid privilege check → Always user to sys
- 3 control registers initialized at boot time
  - SYSENTER_CS_MSR: contains kernel cs selector
  - SYSENTER_EIP_MSR: contains kernel entry point
  - SYSENTER_ESP_MSR: points to the TSS base @
    - NOT USED AS STACK!
    - used to load ESP with the TSS´s field esp0
    - avoid modifications in the task_switch code

modifications to wrapper

- vsyscall_page
  - Shared page: linked with system library
  - elf code:
    - defines kernel_vsyscall function
      - if sysenter is not available: int 0x80 + ret
      - else

```
pushl %ecx
pushl %edx
pushl %ebp
movl %esp, %ebp
sysenter
....
```

    - defines SYSENTER_RETURN

```
popl %ebp
popl %edx
popl %ecx
ret
```

sysenter

- change to system mode
- loads cs ← SYSENTER_CS_MSR
- loads eip ← SYSENTER_EIP_MSR
- loads esp ← SYSENTER_ESP_MSR
- loads ss ← CS + 8
  - Stack segment must be defined at this position
    - (not a problem)

kernel entry point

- Trick: Change to real stack
  - At entry point ESP contains TSS base address
  - Load ESP ← TSS.esp0
- Configure kernel stack like the interrupt mechanism

```
pushl USER_DS
pushl %ebp
pushfl
pushl USER_CS
pushl $SYSENTER_RETURN
....
```

- And the rest as before (SAVE_ALL, check eax…)

exit

- after RESTORE_ALL
  - EDX ← EIP user (it is in the stack)
  - ECX ← ESP user (it is in the stack)
  - sysexit
    - change mode
    - change stack
    - returns to user code (vsyscall_page: SYSENTER_RETURN)

System call handler

- Save hardware context and prepare parameters for the service routine
  - Linux: stores registers with system call parameters at the top of the kernel stack
  - Windows: copy parameters from the address stored in ebx to the top of the kernel stack
- Execute system call service routine
  - Error checking: system calls identifiers
  - Using system_call_table
- Update kernel context with the system call result
- Restore hardware context
- Return to user

System calls service routines

- Check parameters
  - User code is NOT reliable
    - System MUST validate ALL data provided by users

- Access the process address space (if needed)
- Specific system call code algorithm

Content

- Introduction
- Mechanisms for entering the system
  - Initialization
  - Management
  - Example
- Procedure for entering the system
- Procedure to exit from system
- Exceptions
- Interrupts
- System calls
- **Summary**

Interrupt Handling Summary

- Save user context
- Restore system context
- Retrieve user parameters [if needed]
- Identify service [if needed]
- Execute service
- Return result [if needed]
- Restore user context

References

- [1] Understanding Linux Kernel 3rd ed. Chapter 4 Interrupts and Exceptions.
- [2] Understanding Linux Kernel 3rd ed. Chapter 9 System Calls.
- [3] Intel® 64 and IA-32 architectures software developer's manual volume 3: System programming guide. Chapter 6.
- [4] Intel® 64 and IA-32 architectures software developer's manual volume volume 2: Instruction set reference. sysenter, sysexit