操作系统原理

第二章：计算机系统结构

洪明坚

重庆大学软件学院

February 19, 2016

目录

Computer system structure

Computer system structure

- We need to have a general knowledge of the structure of a computer system before we can explore the details of the operating system.

Computer system structure

- We need to have a general knowledge of the structure of a computer system before we can explore the details of the operating system.
    - You should be familiar with these concepts in the course *Computer architecture*.

Computer system structure

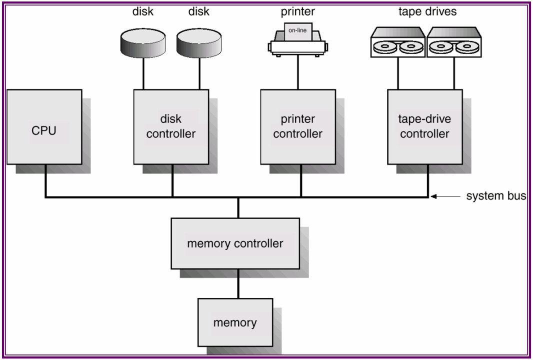
- We need to have a general knowledge of the structure of a computer system before we can explore the details of the operating system.
  - You should be familiar with these concepts in the course *Computer architecture*.
  - But, some of them will be explored in-depth with the operating system in mind.

Computer system structure

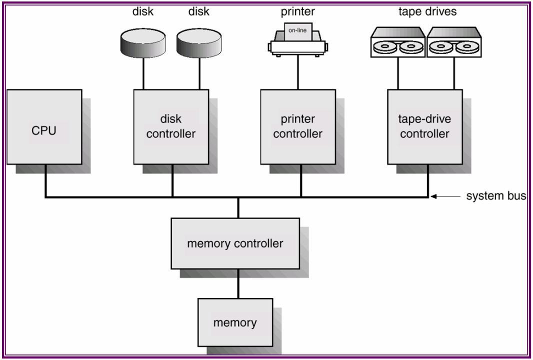
- We need to have a general knowledge of the structure of a computer system before we can explore the details of the operating system.
  - You should be familiar with these concepts in the course *Computer architecture*.
  - But, some of them will be explored in-depth with the operating system in mind.

Overall structure

Overall structure

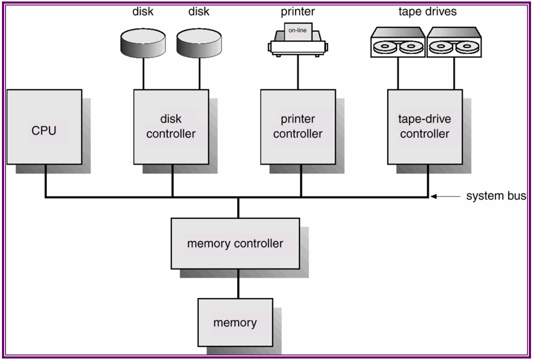


Overall structure



- The CPU and device controllers can execute concurrently, competing for memory cycles.

Overall structure



- The CPU and device controllers can execute concurrently, competing for memory cycles.
- To ensure orderly access to the memory, a memory controller is provided to synchronize access to the memory.

Bootstrap

Bootstrap

- When users just power on a computer, there is no already running operating system available.

Bootstrap

- When users just power on a computer, there is no already running operating system available.
    - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.

Bootstrap

- When users just power on a computer, there is no already running operating system available.
  - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.
  - Then the control is transfered to the entry of the operating system with a very basic environment.

Bootstrap

- When users just power on a computer, there is no already running operating system available.
  - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.
  - Then the control is transfered to the entry of the operating system with a very basic environment.
  - This procedure is named **bootstrap** (or simply **boot**) an operating system.

Bootstrap

- When users just power on a computer, there is no already running operating system available.
  - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.
  - Then the control is transfered to the entry of the operating system with a very basic environment.
  - This procedure is named **bootstrap** (or simply **boot**) an operating system.
  - The program which boots the operating system is called **boot-loader**.

Bootstrap

- When users just power on a computer, there is no already running operating system available.
  - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.
  - Then the control is transfered to the entry of the operating system with a very basic environment.
  - This procedure is named **bootstrap** (or simply **boot**) an operating system.
  - The program which boots the operating system is called **boot-loader**.
- Examples
  - **NTLDR** - boot-loader for Windows NT/XP (resides in C:\).

Bootstrap

- When users just power on a computer, there is no already running operating system available.
  - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.
  - Then the control is transfered to the entry of the operating system with a very basic environment.
  - This procedure is named **bootstrap** (or simply **boot**) an operating system.
  - The program which boots the operating system is called **boot-loader**.
- Examples
  - **NTLDR** - boot-loader for Windows NT/XP (resides in C:\).
  - **GRUB** - one of boot-loaders for the Unix/Linux.

Bootstrap

- When users just power on a computer, there is no already running operating system available.
  - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.
  - Then the control is transfered to the entry of the operating system with a very basic environment.
  - This procedure is named **bootstrap** (or simply **boot**) an operating system.
  - The program which boots the operating system is called **boot-loader**.
- Examples
  - **NTLDR** - boot-loader for Windows NT/XP (resides in C:\).
  - **GRUB** - one of boot-loaders for the Unix/Linux.
- Bear in mind that

Bootstrap

- When users just power on a computer, there is no already running operating system available.
  - We must load the operating system kernel from some persistent storages, such as disk and network server, to memory.
  - Then the control is transfered to the entry of the operating system with a very basic environment.
  - This procedure is named **bootstrap** (or simply **boot**) an operating system.
  - The program which boots the operating system is called **boot-loader**.
- Examples
  - **NTLDR** - boot-loader for Windows NT/XP (resides in C:\).
  - **GRUB** - one of boot-loaders for the Unix/Linux.
- Bear in mind that
  - **The boot-loader is NOT part of the operating system.**

Questions

- Any questions?

Interrupt

Interrupt

- Modern computers and operating systems are **interrupt driven**.

Interrupt

- Modern computers and operating systems are **interrupt driven**.
  - Peripheral devices use **interrupt** to signal the CPU that something has happened.

Interrupt

- Modern computers and operating systems are **interrupt driven**.
  - Peripheral devices use **interrupt** to signal the CPU that something has happened.
- When the CPU is interrupted, it must **serve the interrupt** by

Interrupt

- Modern computers and operating systems are **interrupt driven**.
    - Peripheral devices use **interrupt** to signal the CPU that something has happened.
- When the CPU is interrupted, it must **serve the interrupt** by
    1. *Hardware*: saves some of registers and branches to the **interrupt service routine (ISR)**;

Interrupt

- Modern computers and operating systems are **interrupt driven**.
  - Peripheral devices use **interrupt** to signal the CPU that something has happened.
- When the CPU is interrupted, it must **serve the interrupt** by
  1. *Hardware*: saves some of registers and branches to the **interrupt service routine (ISR)**;
  2. *Assembly language procedure in ISR*: saves rest of registers if necessary and sets up a convenient environment;

Interrupt

- Modern computers and operating systems are **interrupt driven**.
  - Peripheral devices use **interrupt** to signal the CPU that something has happened.
- When the CPU is interrupted, it must **serve the interrupt** by
  1. *Hardware*: saves some of registers and branches to the **interrupt service routine (ISR)**;
  2. *Assembly language procedure in ISR*: saves rest of registers if necessary and sets up a convenient environment;
  3. *C language procedure in ISR*: does serve the interrupt, typically reads and buffers input data from peripheral device;

Interrupt

- Modern computers and operating systems are **interrupt driven**.
  - Peripheral devices use **interrupt** to signal the CPU that something has happened.
- When the CPU is interrupted, it must **serve the interrupt** by
  1. *Hardware*: saves some of registers and branches to the **interrupt service routine (ISR)**;
  2. *Assembly language procedure in ISR*: saves rest of registers if necessary and sets up a convenient environment;
  3. *C language procedure in ISR*: does serve the interrupt, typically reads and buffers input data from peripheral device;
  4. *C language procedure in ISR*: returns to the *assembly language procedure in ISR*;

Interrupt

- Modern computers and operating systems are **interrupt driven**.
  - Peripheral devices use **interrupt** to signal the CPU that something has happened.
- When the CPU is interrupted, it must **serve the interrupt** by
  1. *Hardware*: saves some of registers and branches to the **interrupt service routine (ISR)**;
  2. *Assembly language procedure in ISR*: saves rest of registers if necessary and sets up a convenient environment;
  3. *C language procedure in ISR*: does serve the interrupt, typically reads and buffers input data from peripheral device;
  4. *C language procedure in ISR*: returns to the *assembly language procedure in ISR*;
  5. *Assembly language procedure in ISR*: restores saved registers and returns to the location being interrupted.

Interrupt Vector

Interrupt Vector

- Usually, a computer system has several peripheral devices.

Interrupt Vector

- Usually, a computer system has several peripheral devices.
- When an interrupt occurs, CPU must know which device triggered it.

Interrupt Vector

- Usually, a computer system has several peripheral devices.
- When an interrupt occurs, CPU must know which device triggered it.
  - Computer system assigns each device an *unique* interrupt request number (e.g., an 8-bit integer), or simply **IRQ**.

Interrupt Vector

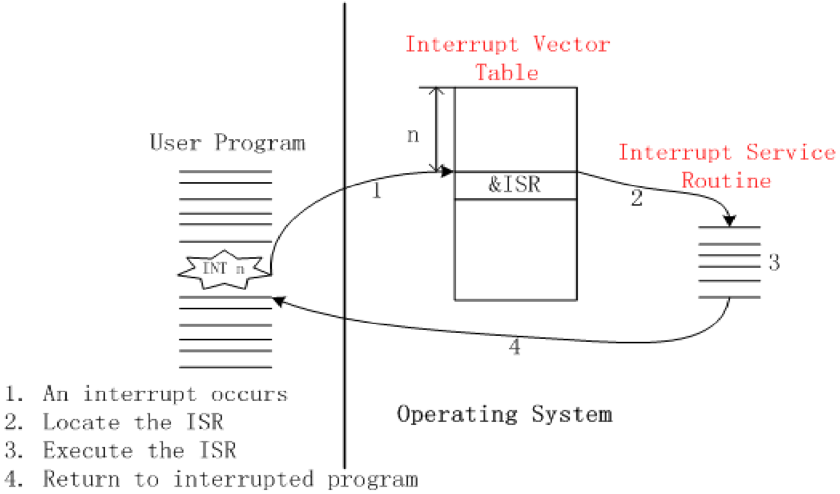
- Usually, a computer system has several peripheral devices.
- When an interrupt occurs, CPU must know which device triggered it.
    - Computer system assigns each device an *unique* interrupt request number (e.g., an 8-bit integer), or simply **IRQ**.
    - The addresses of all ISRs are collected into a table called **interrupt vector**.

Interrupt Vector

- Usually, a computer system has several peripheral devices.
- When an interrupt occurs, CPU must know which device triggered it.
  - Computer system assigns each device an *unique* interrupt request number (e.g., an 8-bit integer), or simply **IRQ**.
  - The addresses of all ISRs are collected into a table called **interrupt vector**.
  - When servicing an interrupt, CPU uses IRQ to index the interrupt vector to fetch the address of ISR and branches to it.

Put them all together

Put them all together



Exception

Exception

- Interrupt

Exception

- Interrupt
  - Triggered by peripheral devices;

Exception

- Interrupt
  - Triggered by peripheral devices;
  - Asynchronous.

Exception

- Interrupt
  - Triggered by peripheral devices;
  - Asynchronous.
- **Exception**

Exception

- Interrupt
  - Triggered by peripheral devices;
  - Asynchronous.
- **Exception**
  - Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero and invalid memory access.

Exception

- Interrupt
  - Triggered by peripheral devices;
  - Asynchronous.
- **Exception**
  - Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero and invalid memory access.
  - Synchronous.

Exception

- Interrupt
  - Triggered by peripheral devices;
  - Asynchronous.
- **Exception**
  - Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero and invalid memory access.
  - Synchronous.
- Other than the above, handling of interrupts and exceptions is identical.

Exception

- Interrupt
  - Triggered by peripheral devices;
  - Asynchronous.
- **Exception**
  - Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero and invalid memory access.
  - Synchronous.
- Other than the above, handling of interrupts and exceptions is identical.
  - Exception is also known as **software-generated interrupt** or **synchronous interrupt**.

Questions

- Any questions?

I/O structure

[1] Also known as blocking or non-overlapping I/O.
[2] Also known as non-blocking or overlapping I/O.

I/O structure

- When CPU is doing I/O with a peripheral device, two methods is available:

[1] Also known as blocking or non-overlapping I/O.

[2] Also known as non-blocking or overlapping I/O.

I/O structure

- When CPU is doing I/O with a peripheral device, two methods is available:
  - (a) synchronous [1]

[1] Also known as blocking or non-overlapping I/O.
[2] Also known as non-blocking or overlapping I/O.

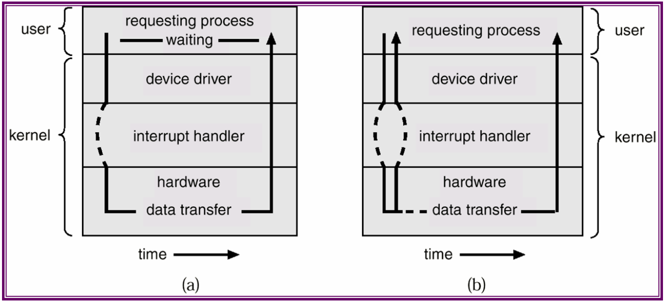
I/O structure

- When CPU is doing I/O with a peripheral device, two methods is available:
  - (a) synchronous [1] and (b) asynchronous [2] I/O.

[1] Also known as blocking or non-overlapping I/O.
[2] Also known as non-blocking or overlapping I/O.

I/O structure

- When CPU is doing I/O with a peripheral device, two methods is available:
  - (a) synchronous [1] and (b) asynchronous [2] I/O.



[1] Also known as blocking or non-overlapping I/O.

[2] Also known as non-blocking or overlapping I/O.

How CPU accesses peripheral devices?

How CPU accesses peripheral devices?

- CPU accesses devices through device controllers.

How CPU accesses peripheral devices?

- CPU accesses devices through device controllers.
    - Device controllers include registers to hold commands and the data being transferred.

How CPU accesses peripheral devices?

- CPU accesses devices through device controllers.
  - Device controllers include registers to hold commands and the data being transferred.
  - How CPU accesses these registers?

How CPU accesses peripheral devices?

- CPU accesses devices through device controllers.
  - Device controllers include registers to hold commands and the data being transferred.
  - How CPU accesses these registers?
- Two methods:

How CPU accesses peripheral devices?

- CPU accesses devices through device controllers.
  - Device controllers include registers to hold commands and the data being transferred.
  - How CPU accesses these registers?
- Two methods:
  - I/O port

How CPU accesses peripheral devices?

- CPU accesses devices through device controllers.
  - Device controllers include registers to hold commands and the data being transferred.
  - How CPU accesses these registers?
- Two methods:
  - I/O port
  - Memory-mapped I/O

I/O port (1/2)

I/O port (1/2)

- All registers within device controllers are collected.

I/O port (1/2)

- All registers within device controllers are collected.
  - An unique address (which is called **port**, an 8- or 16-bit integer) is assigned to each of them.

I/O port (1/2)

- All registers within device controllers are collected.
  - An unique address (which is called **port**, an 8- or 16-bit integer) is assigned to each of them.
- Special I/O instructions are designed to allow data transfers between these registers and memory.

I/O port (1/2)

- All registers within device controllers are collected.
  - An unique address (which is called **port**, an 8- or 16-bit integer) is assigned to each of them.
- Special I/O instructions are designed to allow data transfers between these registers and memory.
- Example: IBM-PC

I/O port (1/2)

- All registers within device controllers are collected.
  - An unique address (which is called **port**, an 8- or 16-bit integer) is assigned to each of them.
- Special I/O instructions are designed to allow data transfers between these registers and memory.
- Example: IBM-PC
  - 16-bit I/O ports are used to address the registers of device controllers.

I/O port (1/2)

- All registers within device controllers are collected.
  - An unique address (which is called **port**, an 8- or 16-bit integer) is assigned to each of them.
- Special I/O instructions are designed to allow data transfers between these registers and memory.
- Example: IBM-PC
  - 16-bit I/O ports are used to address the registers of device controllers.
  - Two special I/O instructions: **IN** and **OUT** are included in the INTEL x86 CPU.

I/O port (1/2)

- All registers within device controllers are collected.
  - An unique address (which is called **port**, an 8- or 16-bit integer) is assigned to each of them.
- Special I/O instructions are designed to allow data transfers between these registers and memory.
- Example: IBM-PC
  - 16-bit I/O ports are used to address the registers of device controllers.
  - Two special I/O instructions: **IN** and **OUT** are included in the INTEL x86 CPU.
    - IN *reg*, *port* - Read a byte/word from *port* to CPU register *reg*.

I/O port (1/2)

- All registers within device controllers are collected.
    - An unique address (which is called **port**, an 8- or 16-bit integer) is assigned to each of them.
- Special I/O instructions are designed to allow data transfers between these registers and memory.
- Example: IBM-PC
    - 16-bit I/O ports are used to address the registers of device controllers.
    - Two special I/O instructions: **IN** and **OUT** are included in the INTEL x86 CPU.
        - IN *reg*, *port* - Read a byte/word from *port* to CPU register *reg*.
        - OUT *port*, *reg* - Write the content of CPU register *reg* to *port*.

I/O port (2/2)

I/O port (2/2)

- Part of PC I/O port address map

I/O port (2/2)

- Part of PC I/O port address map

Range (hex)	**Function**
000-01F	1st DMA controller
020-03F	1st Programmable Interrupt Controller (PIC)
040-05F	Programmable Interval Timer (System timer)
060-06F	Keyboard
220-233	Sound card
3D0-3DF	Color Graphics Adapter

I/O port (2/2)

- Part of PC I/O port address map

Range (hex)	Function
000-01F	1st DMA controller
020-03F	1st Programmable Interrupt Controller (PIC)
040-05F	Programmable Interval Timer (System timer)
060-06F	Keyboard
220-233	Sound card
3D0-3DF	Color Graphics Adapter

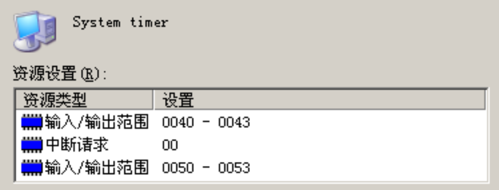
- Example: system timer

I/O port (2/2)

- Part of PC I/O port address map

Range (hex)	Function
000-01F	1st DMA controller
020-03F	1st Programmable Interrupt Controller (PIC)
040-05F	Programmable Interval Timer (System timer)
060-06F	Keyboard
220-233	Sound card
3D0-3DF	Color Graphics Adapter

- Example: system timer



Memory-mapped I/O (1/3)

Memory-mapped I/O (1/3)

- In the method of I/O port,

Memory-mapped I/O (1/3)

- In the method of I/O port,
  - we can view I/O ports as another separate address space, independent of memory address space.

Memory-mapped I/O (1/3)

- In the method of I/O port,
  - we can view I/O ports as another separate address space, independent of memory address space.
- Registers within device controller is just a piece of storage.

Memory-mapped I/O (1/3)

- In the method of I/O port,
  - we can view I/O ports as another separate address space, independent of memory address space.
- Registers within device controller is just a piece of storage.
  - Why not access these registers using the same method as memory?

Memory-mapped I/O (1/3)

- In the method of I/O port,
  - we can view I/O ports as another separate address space, independent of memory address space.
- Registers within device controller is just a piece of storage.
  - Why not access these registers using the same method as memory?
  - In this case, a (unique) **memory address** is assigned to every register,**NOT a port address**.

Memory-mapped I/O (1/3)

- In the method of I/O port,
  - we can view I/O ports as another separate address space, independent of memory address space.
- Registers within device controller is just a piece of storage.
  - Why not access these registers using the same method as memory?
  - In this case, a (unique) **memory address** is assigned to every register,**NOT a port address**.
- That's the memory-mapped I/O.

Memory-mapped I/O (1/3)

- In the method of I/O port,
  - we can view I/O ports as another separate address space, independent of memory address space.
- Registers within device controller is just a piece of storage.
  - Why not access these registers using the same method as memory?
  - In this case, a (unique) **memory address** is assigned to every register,**NOT a port address**.
- That's the memory-mapped I/O.
  - Memory-mapped I/O uses the same bus to address both memory and I/O devices

Memory-mapped I/O (1/3)

- In the method of I/O port,
  - we can view I/O ports as another separate address space, independent of memory address space.
- Registers within device controller is just a piece of storage.
  - Why not access these registers using the same method as memory?
  - In this case, a (unique) **memory address** is assigned to every register,**NOT a port address**.
- That's the memory-mapped I/O.
  - Memory-mapped I/O uses the same bus to address both memory and I/O devices
  - In order to accommodate the I/O devices, areas of CPU addressable space must be reserved for I/O rather than memory.

Memory-mapped I/O (2/3)

Memory-mapped I/O (2/3)

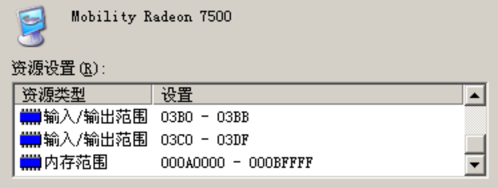
- Example: video controller of IBM-PC

Memory-mapped I/O (2/3)

- Example: video controller of IBM-PC
  - Each location on the screen is mapped to a memory location.

Memory-mapped I/O (2/3)

- Example: video controller of IBM-PC
  - Each location on the screen is mapped to a memory location.



Memory-mapped I/O (3/3)

Memory-mapped I/O (3/3)

- Advantages

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O.

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O.
- Disadvantages

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O.
- Disadvantages
  - Most computers nowadays have some form of caching memory words. But,

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O.
- Disadvantages
  - Most computers nowadays have some form of caching memory words. But, caching a device controller register would be disastrous.

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O.
- Disadvantages
  - Most computers nowadays have some form of caching memory words. But, caching a device controller register would be disastrous.
- Modern computer systems use both of them,

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O.
- Disadvantages
  - Most computers nowadays have some form of caching memory words. But, caching a device controller register would be disastrous.
- Modern computer systems use both of them,
  - with memory-mapped I/O for data buffers and separate I/O ports for the command registers,

Memory-mapped I/O (3/3)

- Advantages
  - Every instruction that can reference memory can also reference device controller registers.
    - The device drivers can be written entirely in C.
  - No special protection mechanism is needed to keep user processes from performing I/O.
- Disadvantages
  - Most computers nowadays have some form of caching memory words. But, caching a device controller register would be disastrous.
- Modern computer systems use both of them,
  - with memory-mapped I/O for data buffers and separate I/O ports for the command registers,
  - as in the previous example of *Mobility Radeon 7500*.

Questions

- Any questions?

Questions

- Any questions?

Hardware protection

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation**

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation** - Prevent user programs taking over part of the OS and using this to overwrite other programs or even modify the OS itself.

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation** - Prevent user programs taking over part of the OS and using this to overwrite other programs or even modify the OS itself.
  - **Privileged instructions**

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation** - Prevent user programs taking over part of the OS and using this to overwrite other programs or even modify the OS itself.
  - **Privileged instructions** - Prevent user programs disrupting the normal operation of the system by issuing illegal I/O instructions.

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation** - Prevent user programs taking over part of the OS and using this to overwrite other programs or even modify the OS itself.
  - **Privileged instructions** - Prevent user programs disrupting the normal operation of the system by issuing illegal I/O instructions.
  - **Memory protection**

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation** - Prevent user programs taking over part of the OS and using this to overwrite other programs or even modify the OS itself.
  - **Privileged instructions** - Prevent user programs disrupting the normal operation of the system by issuing illegal I/O instructions.
  - **Memory protection** - Prevent a user program directly accessing the memory of another user program or even operating system.

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation** - Prevent user programs taking over part of the OS and using this to overwrite other programs or even modify the OS itself.
  - **Privileged instructions** - Prevent user programs disrupting the normal operation of the system by issuing illegal I/O instructions.
  - **Memory protection** - Prevent a user program directly accessing the memory of another user program or even operating system.
  - **CPU protection**

Hardware protection

- To ensure proper operation, we must protect the operating system and all other program and their data from any malfunctioning program.
- Hardware protection can be broken down different ways
  - **Dual-mode operation** - Prevent user programs taking over part of the OS and using this to overwrite other programs or even modify the OS itself.
  - **Privileged instructions** - Prevent user programs disrupting the normal operation of the system by issuing illegal I/O instructions.
  - **Memory protection** - Prevent a user program directly accessing the memory of another user program or even operating system.
  - **CPU protection** - Prevent a user program from getting stuck in an infinite loop and never returning control to the operating system.

Dual-mode operation

Dual-mode operation

- We need at least two separate modes of operation:

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode**

Dual-mode operation

- We need at least two separate modes of operation:
    - **User mode** - Execution on behalf of user programs;

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode**

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.
    - Also known as **supervisor, system, privileged or kernel mode**.

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.
    - Also known as **supervisor, system, privileged or kernel mode**.
- **Mode bit** is added to computer hardware to indicate the current mode: monitor (0) or user (1).

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.
    - Also known as **supervisor, system, privileged or kernel mode**.
- **Mode bit** is added to computer hardware to indicate the current mode: monitor (0) or user (1).
  - It's set to *monitor* at system boot time.

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.
    - Also known as **supervisor, system, privileged or kernel mode**.
- **Mode bit** is added to computer hardware to indicate the current mode: monitor (0) or user (1).
  - It's set to *monitor* at system boot time. The operating system is then loaded, and starts user programs in user mode.

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.
    - Also known as **supervisor, system, privileged or kernel mode**.
- **Mode bit** is added to computer hardware to indicate the current mode: monitor (0) or user (1).
  - It's set to *monitor* at system boot time. The operating system is then loaded, and starts user programs in user mode.
- When an interrupt or exception occurs hardware switches to monitor mode.

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.
    - Also known as **supervisor, system, privileged or kernel mode**.
- **Mode bit** is added to computer hardware to indicate the current mode: monitor (0) or user (1).
  - It's set to *monitor* at system boot time. The operating system is then loaded, and starts user programs in user mode.
- When an interrupt or exception occurs hardware switches to monitor mode.
  - Whenever the operating system gains control of the computer, it's in monitor mode.

Dual-mode operation

- We need at least two separate modes of operation:
  - **User mode** - Execution on behalf of user programs;
  - **Monitor mode** - Execution on behalf of operating system.
    - Also known as **supervisor, system, privileged or kernel mode**.
- **Mode bit** is added to computer hardware to indicate the current mode: monitor (0) or user (1).
  - It's set to *monitor* at system boot time. The operating system is then loaded, and starts user programs in user mode.
- When an interrupt or exception occurs hardware switches to monitor mode.
  - Whenever the operating system gains control of the computer, it's in monitor mode.
  - And the system always switches to user mode before passing control to a user program.

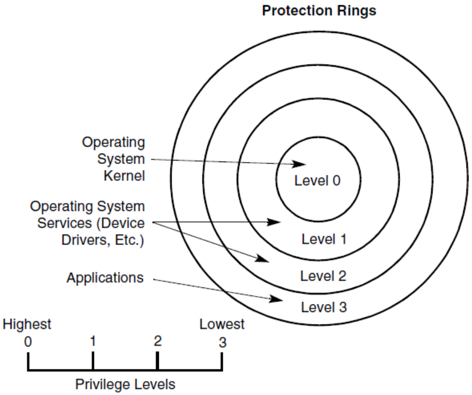
Example

Example

- INTEL IA-32 supports 4 modes to operate, named **protection rings**.

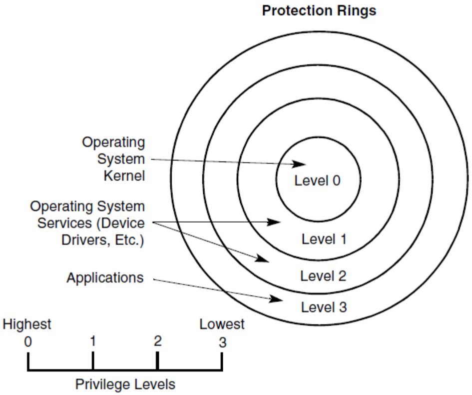
Example

- INTEL IA-32 supports 4 modes to operate, named **protection rings**.



Example

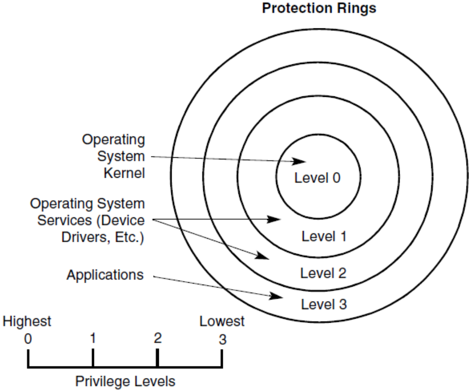
- INTEL IA-32 supports 4 modes to operate, named **protection rings**.



- But, most operating systems running on IA-32 only use 2 of 4.

Example

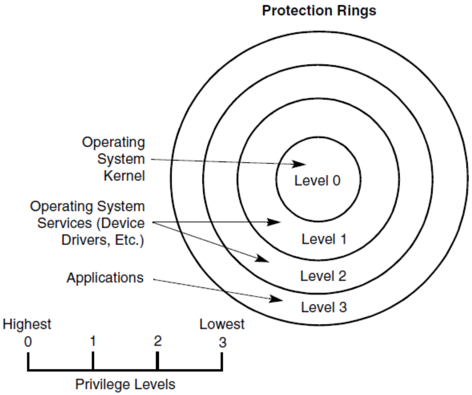
- INTEL IA-32 supports 4 modes to operate, named **protection rings**.



- But, most operating systems running on IA-32 only use 2 of 4.
  - Ring 0 as monitor mode;

Example

- INTEL IA-32 supports 4 modes to operate, named **protection rings**.



- But, most operating systems running on IA-32 only use 2 of 4.
  - Ring 0 as monitor mode; ring 3 as user mode.

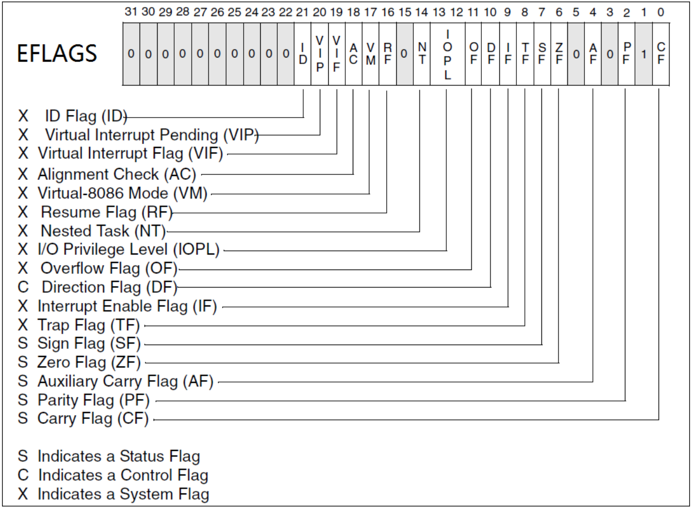
Example (cont'd)

Example (cont'd)

- **Mode bit** of INTEL IA-32

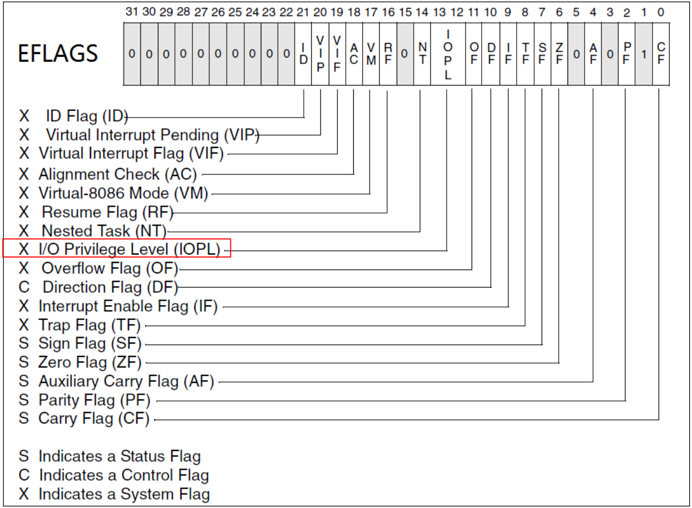
Example (cont'd)

- **Mode bit** of INTEL IA-32



Example (cont'd)

- **Mode bit** of INTEL IA-32



I/O protection

I/O protection

- All I/O instructions are privileged instructions.

I/O protection

- All I/O instructions are privileged instructions.
  - The hardware allows privileged instructions to be executed only in monitor mode.

I/O protection

- All I/O instructions are privileged instructions.
  - The hardware allows privileged instructions to be executed only in monitor mode.
  - If these instructions are to be executed in user mode, the hardware does not execute the instruction, but rather treats it as illegal and generates an exception.

I/O protection

- All I/O instructions are privileged instructions.
  - The hardware allows privileged instructions to be executed only in monitor mode.
  - If these instructions are to be executed in user mode, the hardware does not execute the instruction, but rather treats it as illegal and generates an exception.
  - For example, **IN** and **OUT** are 2 privileged instructions in INTEL IA-32.

I/O protection

- All I/O instructions are privileged instructions.
  - The hardware allows privileged instructions to be executed only in monitor mode.
  - If these instructions are to be executed in user mode, the hardware does not execute the instruction, but rather treats it as illegal and generates an exception.
  - For example, **IN** and **OUT** are 2 privileged instructions in INTEL IA-32.
- Must ensure that a user program could never gain control of the computer in monitor mode.

Memory protection

Memory protection

- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:

Memory protection

- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register**

Memory protection

- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
    - **Base register** - holds the smallest legal physical memory address;

Memory protection

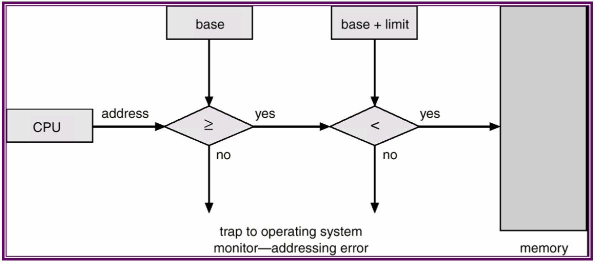
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** - holds the smallest legal physical memory address;
  - **Limit register**

Memory protection

- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** - holds the smallest legal physical memory address;
  - **Limit register** - contains the size of the range.

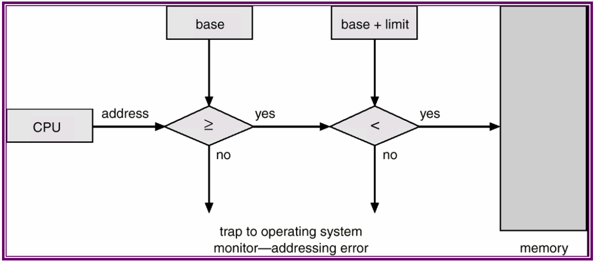
Memory protection

- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** - holds the smallest legal physical memory address;
  - **Limit register** - contains the size of the range.



Memory protection

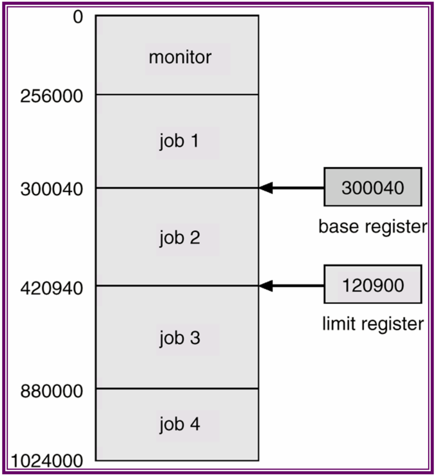
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** - holds the smallest legal physical memory address;
  - **Limit register** - contains the size of the range.



- We will return to this topic when entering *memory management*.

Example

Example



0
monitor
256000
job 1
300040
300040
base register
job 2
420940
120900
limit register
job 3
880000
job 4
1024000

CPU protection

CPU protection

- The operating system can enforce policies only if it gets a chance to run.

CPU protection

- The operating system can enforce policies only if it gets a chance to run.
    - If a malfunctioning program entered an infinite loop and never returns control to the operating system, then CPU was out of the control from the operating system.

CPU protection

- The operating system can enforce policies only if it gets a chance to run.
  - If a malfunctioning program entered an infinite loop and never returns control to the operating system, then CPU was out of the control from the operating system.
- **Timer**

CPU protection

- The operating system can enforce policies only if it gets a chance to run.
  - If a malfunctioning program entered an infinite loop and never returns control to the operating system, then CPU was out of the control from the operating system.
- **Timer** - interrupts CPU after a specified period to ensure operating system maintains control.

CPU protection

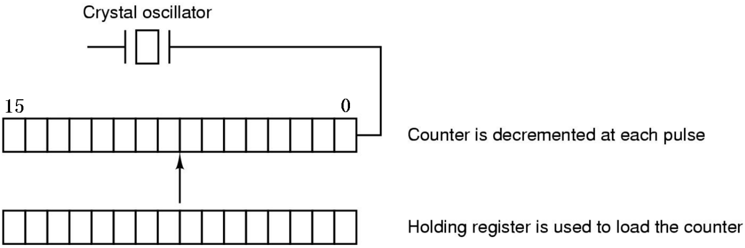
- The operating system can enforce policies only if it gets a chance to run.
  - If a malfunctioning program entered an infinite loop and never returns control to the operating system, then CPU was out of the control from the operating system.
- **Timer** - interrupts CPU after a specified period to ensure operating system maintains control.
  - Remember that when interrupt occurs, the operating system will get the control via ISR.

CPU protection

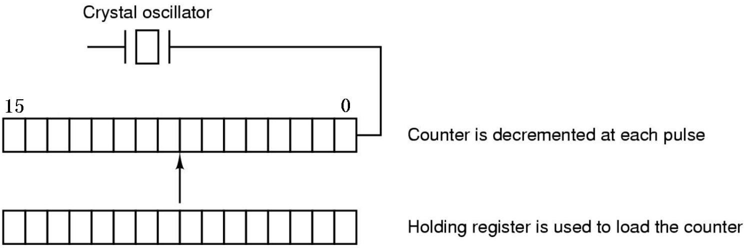
- The operating system can enforce policies only if it gets a chance to run.
  - If a malfunctioning program entered an infinite loop and never returns control to the operating system, then CPU was out of the control from the operating system.
- **Timer** - interrupts CPU after a specified period to ensure operating system maintains control.
  - Remember that when interrupt occurs, the operating system will get the control via ISR.

Timer

Timer

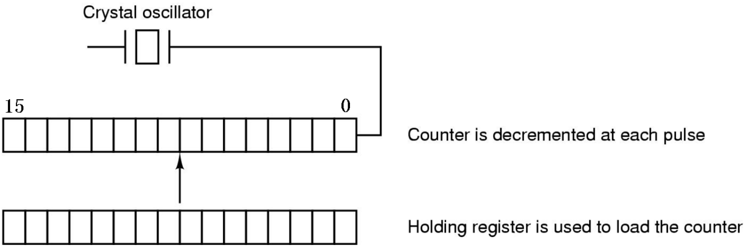

Crystal oscillator
15
0
Counter is decremented at each pulse
Holding register is used to load the counter

Timer



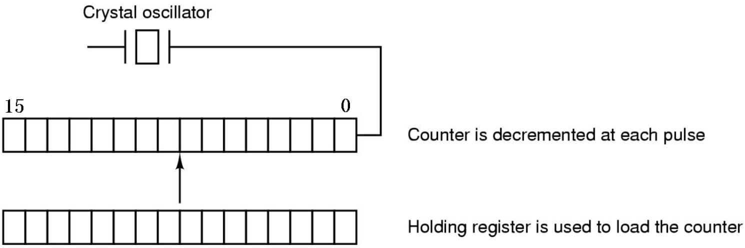
- The *counter register* is decremented by 1 at each pulse.

Timer



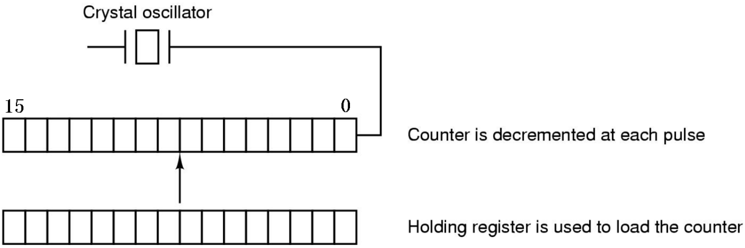
- The *counter register* is decremented by 1 at each pulse.
    - When the *counter register* reaches zero, the timer will interrupt CPU.

Timer



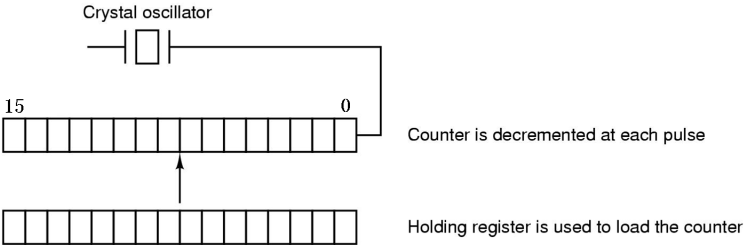
- The *counter register* is decremented by 1 at each pulse.
  - When the *counter register* reaches zero, the timer will interrupt CPU.
  - Then the *counter register* will be reloaded with the value of *holding register* and decrementing repeats.

Timer



- The *counter register* is decremented by 1 at each pulse.
  - When the *counter register* reaches zero, the timer will interrupt CPU.
  - Then the *counter register* will be reloaded with the value of *holding register* and decrementing repeats.
- Example: Timer in IBM-PC

Timer



- The *counter register* is decremented by 1 at each pulse.
  - When the *counter register* reaches zero, the timer will interrupt CPU.
  - Then the *counter register* will be reloaded with the value of *holding register* and decrementing repeats.
- Example: Timer in IBM-PC
  - An INTEL i8253 programmable interval timer with 16-bit counter and holding registers and pulses reach at 1193182Hz.

Questions

- Any questions?

System call

System call

- The operating system does nothing useful itself.

System call

- The operating system does nothing useful itself.
    - But it provides some useful services to the user programs, such as reading a file from disk and sending data to remote host via network adapter.

System call

- The operating system does nothing useful itself.
  - But it provides some useful services to the user programs, such as reading a file from disk and sending data to remote host via network adapter.
  - How does operating system provide these services?

System call

- The operating system does nothing useful itself.
  - But it provides some useful services to the user programs, such as reading a file from disk and sending data to remote host via network adapter.
  - How does operating system provide these services?
- It's the **system call**

System call

- The operating system does nothing useful itself.
    - But it provides some useful services to the user programs, such as reading a file from disk and sending data to remote host via network adapter.
    - How does operating system provide these services?
- It's the **system call** - the (well-defined) **interface** between the operating system and the user programs.

System call

- The operating system does nothing useful itself.
  - But it provides some useful services to the user programs, such as reading a file from disk and sending data to remote host via network adapter.
  - How does operating system provide these services?
- It's the **system call** - the (well-defined) **interface** between the operating system and the user programs.
  - User programs can **ONLY** request services provided by the operating system via system call.

System call

- The operating system does nothing useful itself.
  - But it provides some useful services to the user programs, such as reading a file from disk and sending data to remote host via network adapter.
  - How does operating system provide these services?
- It's the **system call** - the (well-defined) **interface** between the operating system and the user programs.
  - User programs can **ONLY** request services provided by the operating system via system call.
  - The system calls in the interface vary from operating system to operating system.

System call

- The operating system does nothing useful itself.
  - But it provides some useful services to the user programs, such as reading a file from disk and sending data to remote host via network adapter.
  - How does operating system provide these services?
- It's the **system call** - the (well-defined) **interface** between the operating system and the user programs.
  - User programs can **ONLY** request services provided by the operating system via system call.
  - The system calls in the interface vary from operating system to operating system.
  - Also known as **supervisor call**.

Example

Example

- Most operating systems provide the service that can read some bytes from a file: *read*

Example

- Most operating systems provide the service that can read some bytes from a file: *read*
  - $count = read(fd, buffer, nbytes);$

Example

- Most operating systems provide the service that can read some bytes from a file: *read*
  - *count = read(fd, buffer, nbytes);*
  - This system call reads *nbytes* data from file *fd* to the specified *buffer* and returns the number of bytes actually read in *count*.

Procedure (1/2)

Procedure (1/2)

1-3 Prepare parameters;

Procedure (1/2)

1-3 Prepare parameters;

4 Call the wrapper (written in assembly language) of system call;

Procedure (1/2)

1-3 Prepare parameters;

4 Call the wrapper (written in assembly language) of system call;

5 Store the **system call number** of *read* into a register;

Procedure (1/2)

1-3 Prepare parameters;

4 Call the wrapper (written in assembly language) of system call;

5 Store the **system call number** of *read* into a register;

6 Trap into the operating system;

Procedure (1/2)

1-3 Prepare parameters;

4 Call the wrapper (written in assembly language) of system call;

5 Store the **system call number** of *read* into a register;

6 Trap into the operating system;

7 Get **system call service routine** for *read* by indexing a **system call table** using system call number;

Procedure (1/2)

1-3 Prepare parameters;

4 Call the wrapper (written in assembly language) of system call;

5 Store the **system call number** of *read* into a register;

6 Trap into the operating system;

7 Get **system call service routine** for *read* by indexing a **system call table** using system call number;

8-11 System call service routine runs and returns to user programs on completion.

Procedure (1/2)

1-3 Prepare parameters;

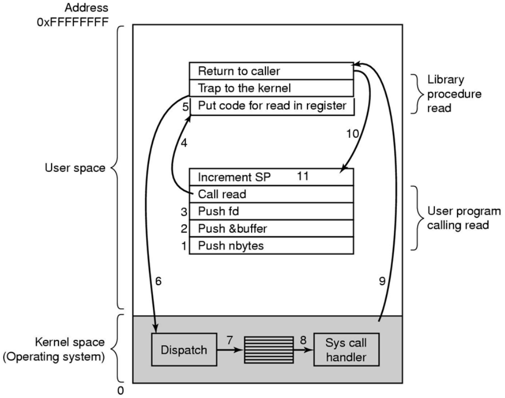
4 Call the wrapper (written in assembly language) of system call;

5 Store the **system call number** of *read* into a register;

6 Trap into the operating system;

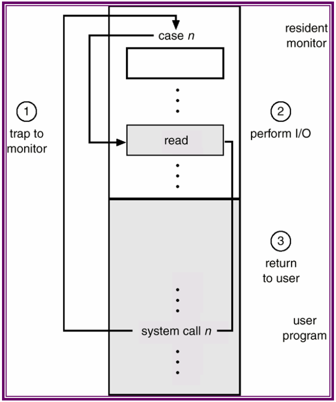
7 Get **system call service routine** for *read* by indexing a **system call table** using system call number;

8-11 System call service routine runs and returns to user programs on completion.

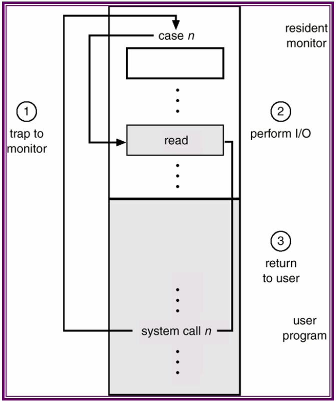


Procedure (2/2)

Procedure (2/2)

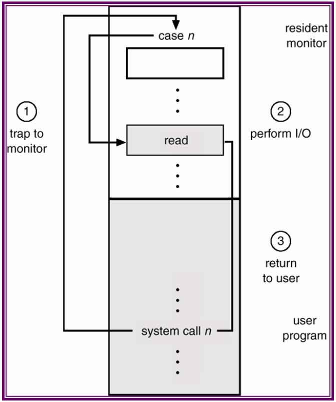


Procedure (2/2)



- Resident monitor (or simply monitor) here means the operating system and

Procedure (2/2)



- Resident monitor (or simply monitor) here means the operating system and
- *n* is the system call number.

Trap into the operating system (1/2)

Trap into the operating system (1/2)

- The user programs cannot trap into the operating system directly.

Trap into the operating system (1/2)

- The user programs cannot trap into the operating system directly.
- How to trap into the operating system?

Trap into the operating system (1/2)

- The user programs cannot trap into the operating system directly.
- How to trap into the operating system?
  - Method 1: **Exception (software-generated interrupt)**.

Trap into the operating system (1/2)

- The user programs cannot trap into the operating system directly.
- How to trap into the operating system?
    - Method 1: **Exception (software-generated interrupt)**.
    - Method 2: Special instruction.

Trap into the operating system (2/2)

[3] Short for SoftWare Interrupt.

Trap into the operating system (2/2)

- Exception

[3] Short for SoftWare Interrupt.

Trap into the operating system (2/2)

- Exception
  - INTEL IA-32 provides an instruction to trigger a exception, *INT*.

[3] Short for SoftWare Interrupt.

Trap into the operating system (2/2)

- Exception
  - INTEL IA-32 provides an instruction to trigger a exception, *INT*.
    - For example, Linux/FreeBSD uses *INT 0x80* to trap into the operating system and Windows NT/XP uses *INT 0x2e*.

[3] Short for SoftWare Interrupt.

Trap into the operating system (2/2)

- Exception
  - INTEL IA-32 provides an instruction to trigger a exception, *INT*.
    - For example, Linux/FreeBSD uses *INT 0x80* to trap into the operating system and Windows NT/XP uses *INT 0x2e*.
- Special instruction

[3] Short for SoftWare Interrupt.

Trap into the operating system (2/2)

- Exception
  - INTEL IA-32 provides an instruction to trigger a exception, *INT*.
    - For example, Linux/FreeBSD uses *INT 0x80* to trap into the operating system and Windows NT/XP uses *INT 0x2e*.
- Special instruction
  - In addition, INTEL IA-32 provides two special instructions to trap into the operating system: *SYSENTER and SYSEXIT* because of the extra overhead of *INT* instruction.

[3] Short for SoftWare Interrupt.

Trap into the operating system (2/2)

- Exception
  - INTEL IA-32 provides an instruction to trigger a exception, *INT*.
    - For example, Linux/FreeBSD uses *INT 0x80* to trap into the operating system and Windows NT/XP uses *INT 0x2e*.
- Special instruction
  - In addition, INTEL IA-32 provides two special instructions to trap into the operating system: *SYSENTER and SYSEXIT* because of the extra overhead of *INT* instruction.
    - Only supported on processors after Pentium II, i.e., Family 6, Model 3, Stepping 3.

[3] Short for SoftWare Interrupt.

Trap into the operating system (2/2)

- Exception
  - INTEL IA-32 provides an instruction to trigger a exception, *INT*.
    - For example, Linux/FreeBSD uses *INT 0x80* to trap into the operating system and Windows NT/XP uses *INT 0x2e*.
- Special instruction
  - In addition, INTEL IA-32 provides two special instructions to trap into the operating system: *SYSENTER and SYSEXIT* because of the extra overhead of *INT* instruction.
    - Only supported on processors after Pentium II, i.e., Family 6, Model 3, Stepping 3.
  - ARM processors use *swi* [3] to trap into the operating system.

[3] Short for SoftWare Interrupt.

System call v.s. library function

System call v.s. library function

- System call will trap into the OS kernel; while library function does not.

System call v.s. library function

- System call will trap into the OS kernel; while library function does not.
    - So, system call is MUCH more slow than library function.

System call v.s. library function

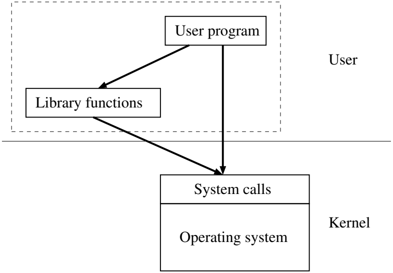
- System call will trap into the OS kernel; while library function does not.
  - So, system call is MUCH more slow than library function.
- Library function is the same as the user-defined function. We can replace an existing library function with our own versions, but we can't replace a system call.

System call v.s. library function

- System call will trap into the OS kernel; while library function does not.
    - So, system call is MUCH more slow than library function.
- Library function is the same as the user-defined function. We can replace an existing library function with our own versions, but we can't replace a system call.
- A system call in one operating system may become a library function in another operating system and vice versa.

System call v.s. library function

- System call will trap into the OS kernel; while library function does not.
  - So, system call is MUCH more slow than library function.
- Library function is the same as the user-defined function. We can replace an existing library function with our own versions, but we can't replace a system call.
- A system call in one operating system may become a library function in another operating system and vice versa.



Questions

- Any questions?

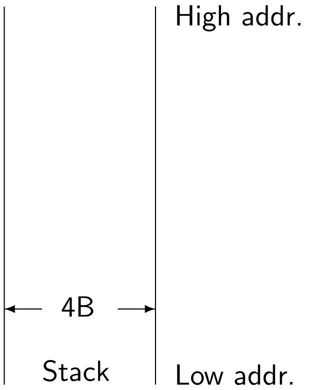
Function calling convention in C

Function calling convention in C

```
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```

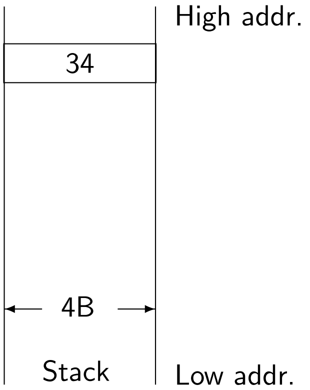
Function calling convention in C

```c
int foo(int a, int b)
{
    char buf[4];
    return a+b;
}
int main()
{
    int i;
    i = foo(12, 34);
    return i;
}
```



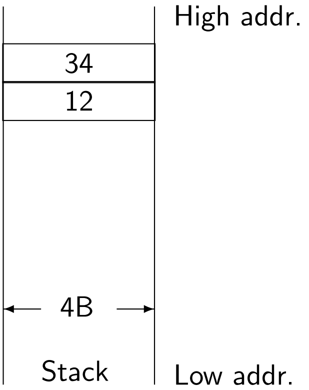
Function calling convention in C

```c
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```



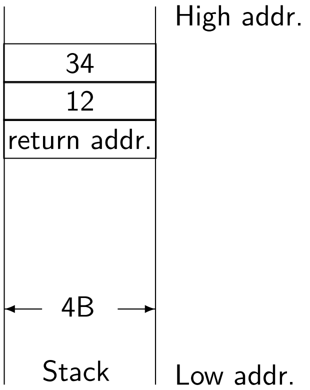
Function calling convention in C

```c
int foo(int a, int b)
{
     char buf[4];
     return a+b;
}
int main()
{
     int i;
     i = foo(12, 34);
     return i;
}
```



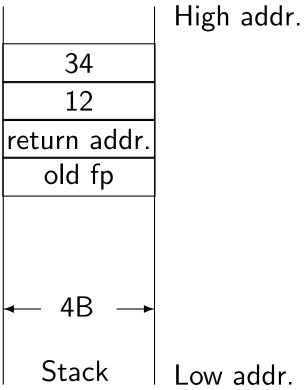
Function calling convention in C

```c
int foo(int a, int b)
{
    char buf[4];
    return a+b;
}
int main()
{
    int i;
    i = foo(12, 34);
    return i;
}
```



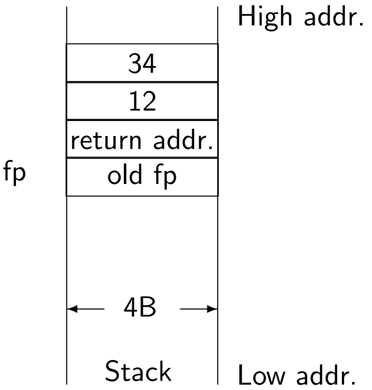
Function calling convention in C

```c
int foo(int a, int b)
{
    char buf[4];
    return a+b;
}
int main()
{
    int i;
    i = foo(12, 34);
    return i;
}
```



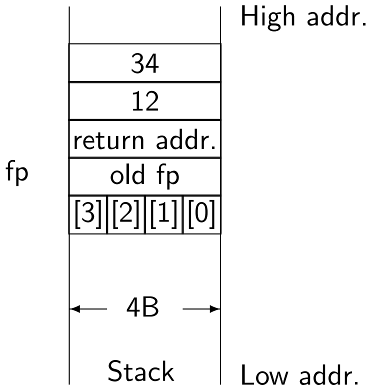
Function calling convention in C

```c
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```



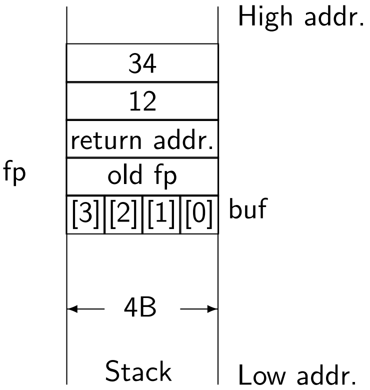
Function calling convention in C

```c
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```



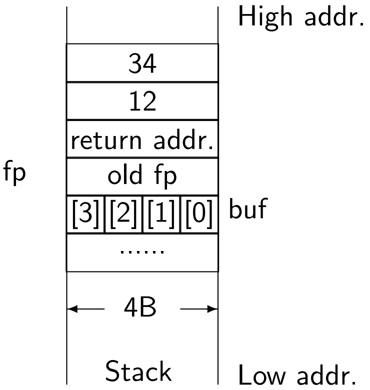
Function calling convention in C

```c
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```

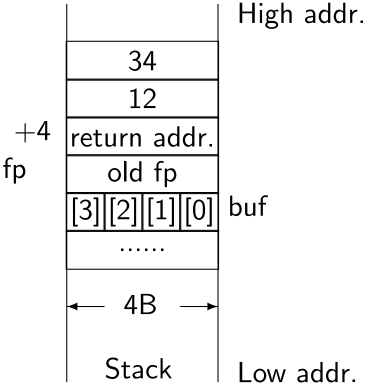


Function calling convention in C

```
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```

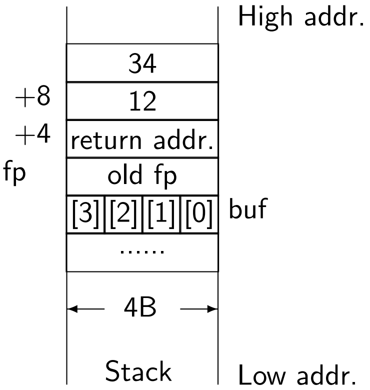


```c
int foo(int a, int b)
{
    char buf[4];
    return a+b;
}
int main()
{
    int i;
    i = foo(12, 34);
    return i;
}
```



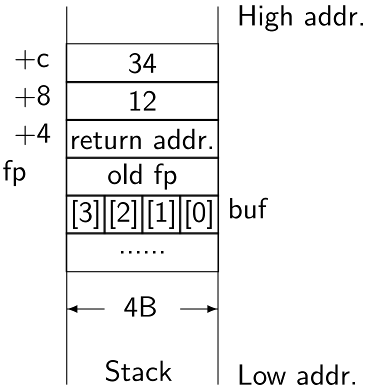
Function calling convention in C

```c
int foo(int a, int b)
{
    char buf[4];
    return a+b;
}
int main()
{
    int i;
    i = foo(12, 34);
    return i;
}
```



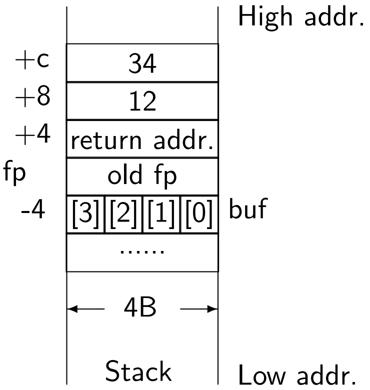
Function calling convention in C

```c
int foo(int a, int b)
{
	char buf[4];
	return a+b;
}
int main()
{
	int i;
	i = foo(12, 34);
	return i;
}
```



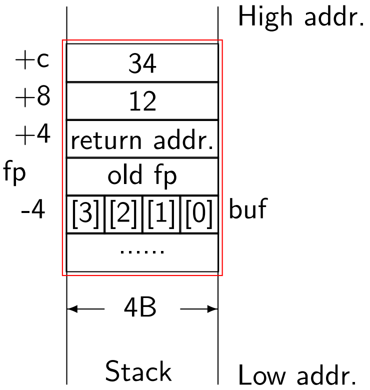
Function calling convention in C

```c
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```



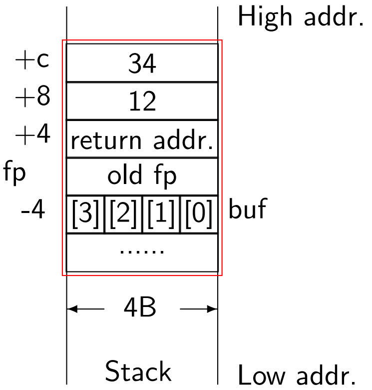
Function calling convention in C

```c
int foo(int a, int b)
{
    char buf[4];
    return a+b;
}
int main()
{
    int i;
    i = foo(12, 34);
    return i;
}
```



Function calling convention in C

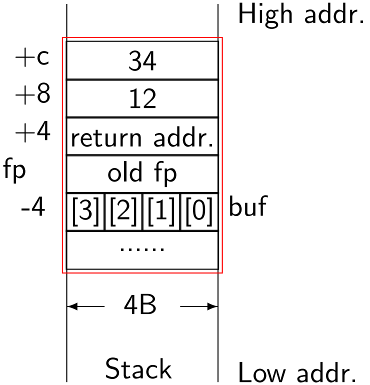
```
int foo(int a, int b)
{
      char buf[4];
      return a+b;
}
int main()
{
      int i;
      i = foo(12, 34);
      return i;
}
```



- The area enclosed by red rectangle is called **stack frame**, or simply **frame**.

Function calling convention in C

```c
int foo(int a, int b)
{
    char buf[4];
    return a+b;
}
int main()
{
    int i;
    i = foo(12, 34);
    return i;
}
```



- The area enclosed by red rectangle is called **stack frame**, or simply **frame**.
  - Stack frames are chained into a singly-linked list via **fp** (short for **frame pointer**).

Stack frame

Stack frame

- The stack frame (also known as **activation record**) is used to record the information of function calling.

Stack frame

- The stack frame (also known as **activation record**) is used to record the information of function calling.
  - Parameters

Stack frame

- The stack frame (also known as **activation record**) is used to record the information of function calling.
  - Parameters
  - Return address

Stack frame

- The stack frame (also known as **activation record**) is used to record the information of function calling.
  - Parameters
  - Return address
  - Local variables allocation and

Stack frame

- The stack frame (also known as **activation record**) is used to record the information of function calling.
  - Parameters
  - Return address
  - Local variables allocation and
  - a “prev” pointer to previous stack frame.

Stack frame

- The stack frame (also known as **activation record**) is used to record the information of function calling.
  - Parameters
  - Return address
  - Local variables allocation and
  - a "prev" pointer to previous stack frame.
- It's stored in the **stack** of the running program.

Questions

- Any questions?