Introduction to
Operating Systems

LearnOA®

Computer-System Organization

Computer-System Architecture

Why do you need an Operating System? – Slide plan

- Flavour of computing environment components - Slides 4-7
- Defining an application, mapping its requirements – Slides 8- 12
- Moving to multiple applications – Slide 13-15
- Processor architecture – Slides 16-18
- Control Flow, Exceptions and Interrupts – Slides 19-21
- IO architecture – Slides 22-24
- Putting it all together: The need for an OS – Slide 25

How does a basic Computing Environment look like?

- A computing device (CPU) and its peripherals:
  - Some memory (RAM)
  - A storage device (Hard disk/Flash drive)
  - A display terminal
  - A printer

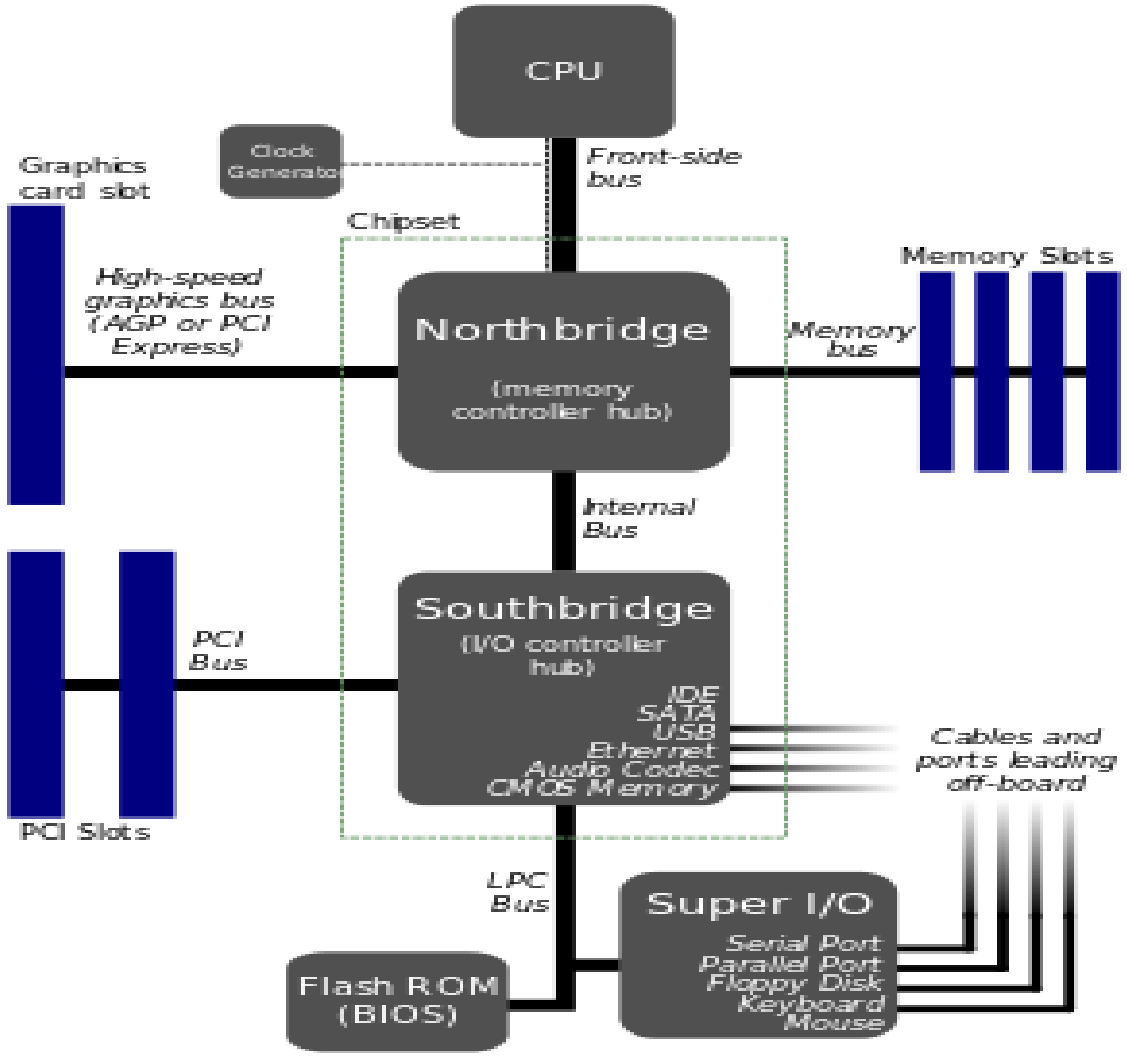
What do you need next?

- Electrical connections so that these components can talk - BUSES

Who talks to whom?

- Quite apart from the fact that the peripherals do not have any data worth talking about
- They cannot talk to each other
- It's the CPU that is the data supplier and is the communications path provider

So how does this comms path look like?



What is an application?

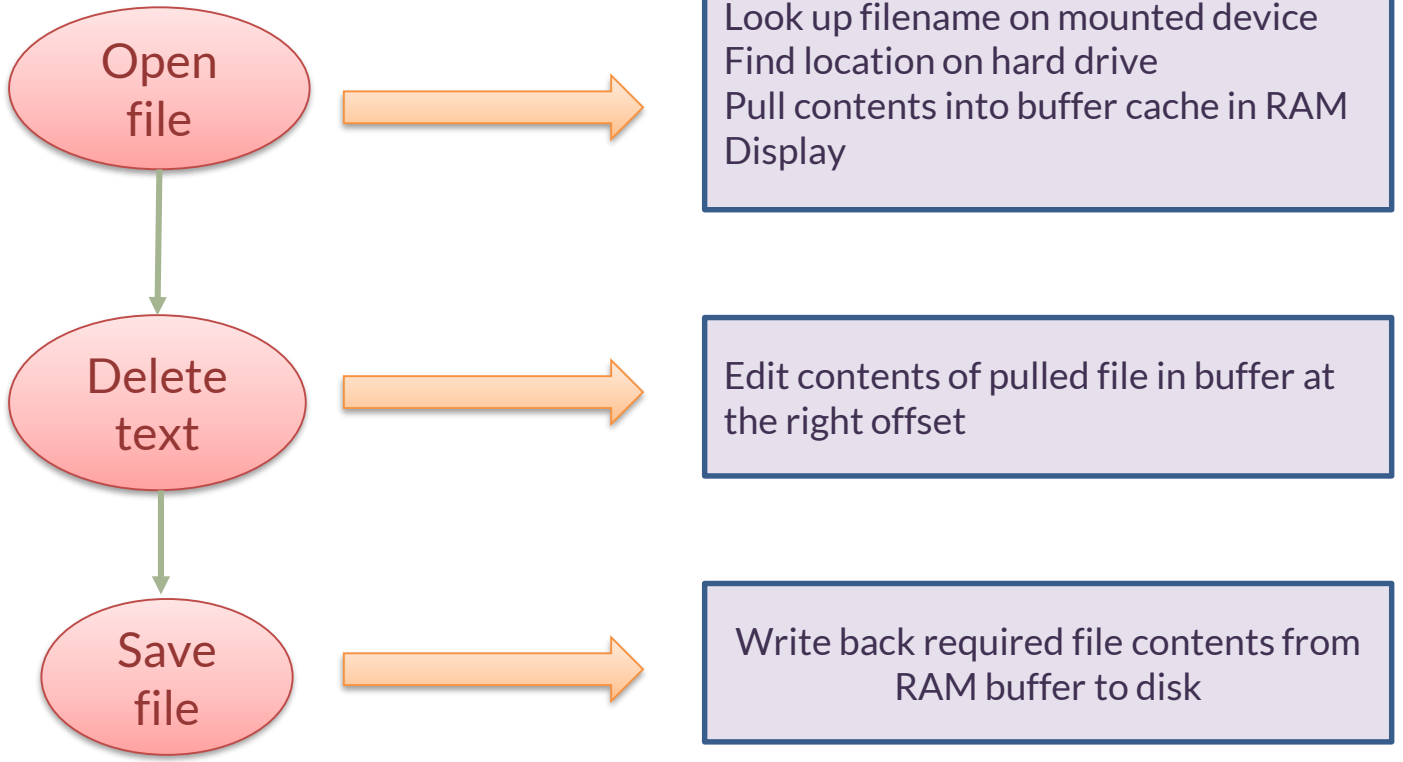
- An application is a software module that implements user-end functionality

- Examples include:
  - A text editor
  - An image viewer
  - A video player
  - A web interface
  - A social media application

What does a text editor do?

- A text editor allows you to read and write text files
- It needs to present the user with a humanly readable file and allow the user to insert/delete text from appropriate points in this file
- In other words, it needs to :
  - ✓ Access data from within a file stored on a mass storage device, push it into a text editor window on the display terminal/screen
  - ✓ Allow user inputs – insertions/deletions - at appropriate points in the file via an editor window

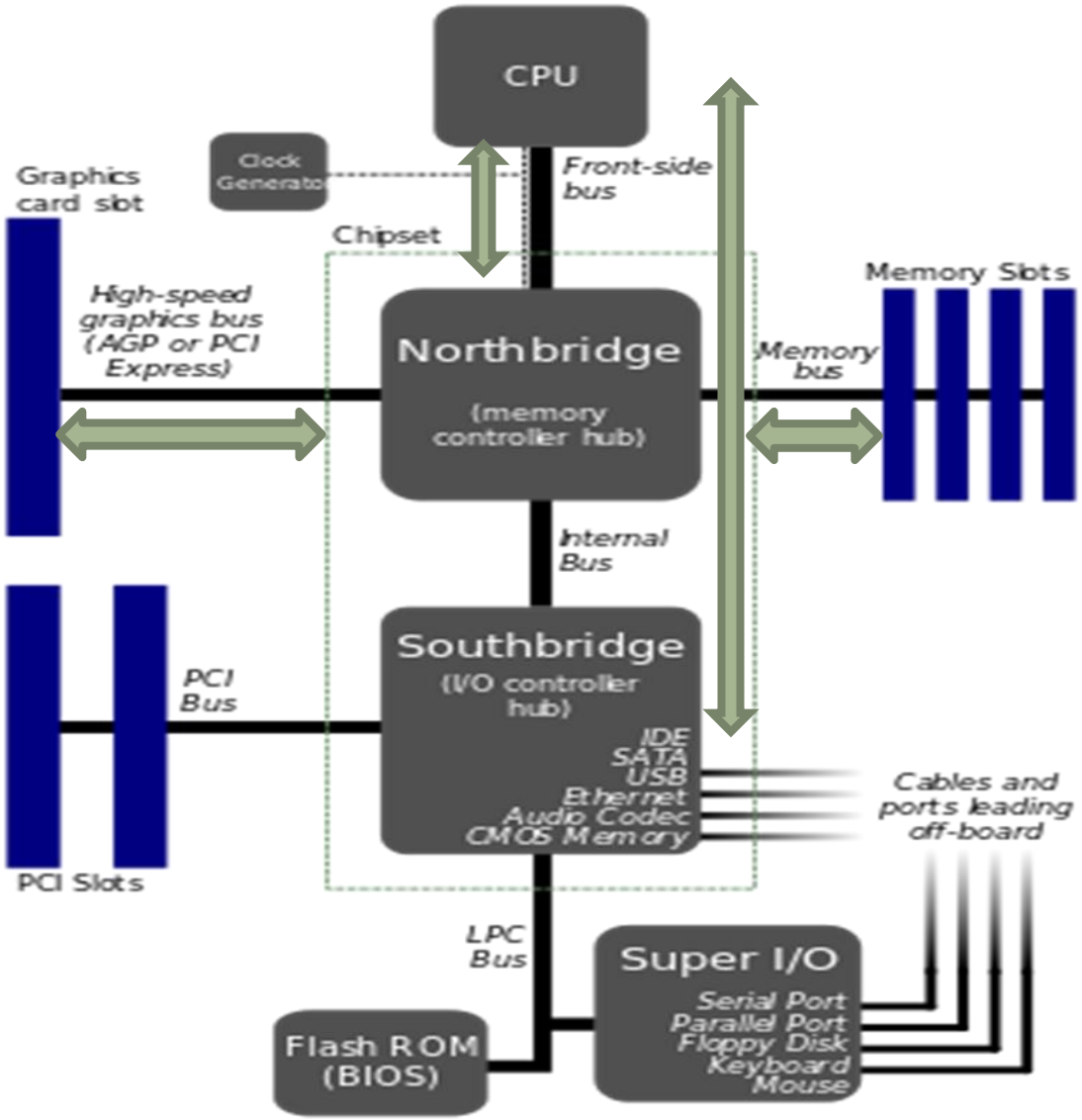
What does a text editor do? (contd.)



How does this map to hardware?

- Look up filename on mounted device, find location on hard drive
- Pull contents into disk buffer in RAM
- Display

* Refer the diagram on the next page


CPU
Clock Generator
Front-side bus
Graphics card slot
Chipset
High-speed graphics bus (AGP or PCI Express)
Northbridge
(memory controller hub)
Memory Slots
Memory bus
Internal Bus
Southbridge
(I/O controller hub)
IDE
SATA
USB
Ethernet
Audio Codec
CMOS Memory
Cables and ports leading off-board
PCI Bus
PCI Slots
LPC Bus
Super I/O
Serial Port
Parallel Port
Floppy Disk
Keyboard
Mouse
Flash ROM (BIOS)

What does the application(text editor) want? – High-level view

- The application wants to read/write specific data to this file/peripheral
- It wants a simple read/write routine to read/write specific areas on the peripheral
- It doesn't want to be bogged down with details on how this is done in hardware
- It wants this to be done in a reproducibly correct manner in a reproducibly short enough interval of time
- It wants to be informed of rare failures

What does an Application want? – Low-level view

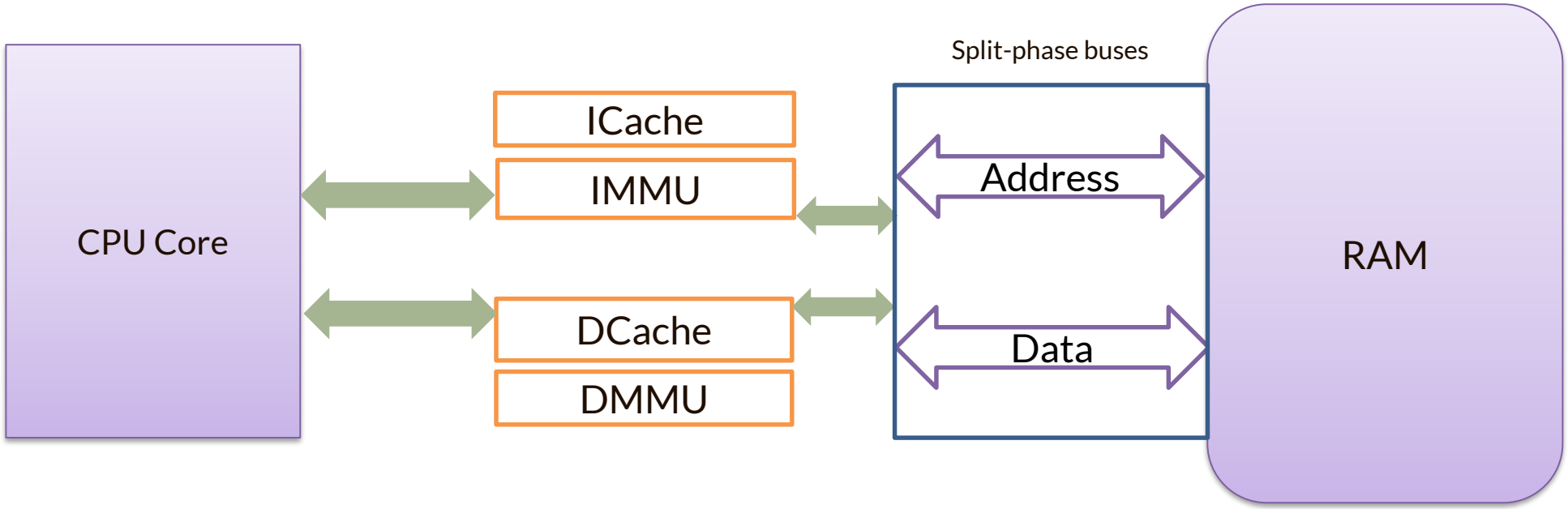
- “Private” access to processor registers to do operations on its variables – processor context maintenance
- Reasonable execution time on the CPU
- A separate code address space to load and run the instructions that make up its code
- A separate data address space to store its global variables, a separate data address space to store its local variables – all static memory
- Dynamic memory allocation for dynamic memory

What does an Application want? – Low-level view

- Computation-intensive, IO-intensive – the need for scheduling
- If one of these applications is computation-intensive and the other IO-intensive, the original need for multitasking so that you can run these tasks in parallel
- Expected application responses (latency) and throughputs vary depending on application – a logging task can be a background activity, a response to a network event might need to be more immediate
- Load balancing - if there are multiple cores
- Concurrency control
- When you have two or more applications vying for the same resource, the same file for instance, you need some means to synchronize/serialize access

How does basic Processor Architecture look like?

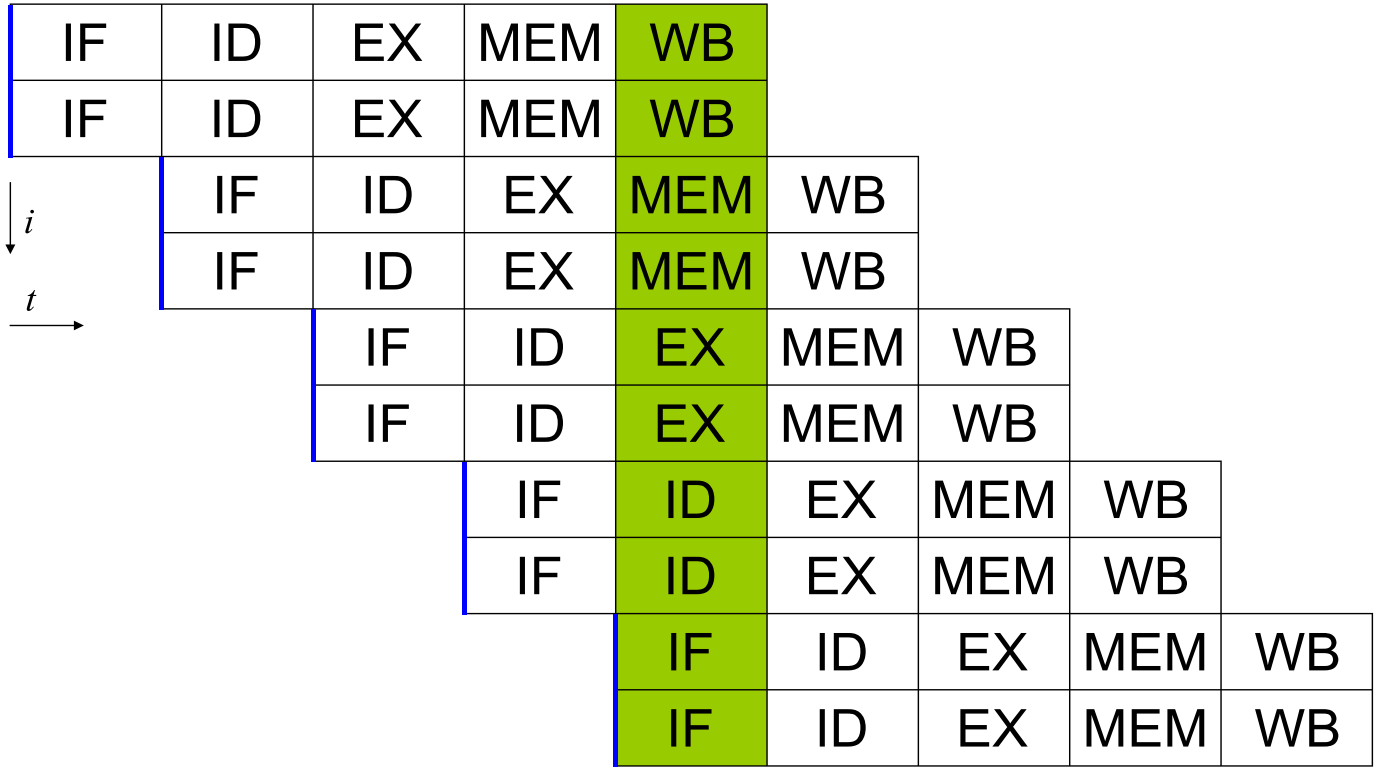
Modified Harvard Architecture



Processor Architecture (contd.)

- **Most modern processors are:**
  - ✓ Control flow, load-store processors
  - ✓ Superscalar pipelined
  - ✓ They support:
  - ✓ Out-of-order execution, In-order completion
  - ✓ Precise exceptions

Processor Architecture – pipeline stages



What Exactly is Control Flow?

- Instruction fetching generally happens incrementally in the form ProgramCounter = ProgramCounter + 4
  - Except upon execution of branch/trap/sc instructions
  - Except upon receiving processor exceptions and hardware interrupts
- What is a practical effect of this? Instruction caching is beneficial.

What are Exceptions?

- An “involuntary” error during any of the pipeline stages of instruction processing
- Bus error, machine check, protection violation, illegal opcode

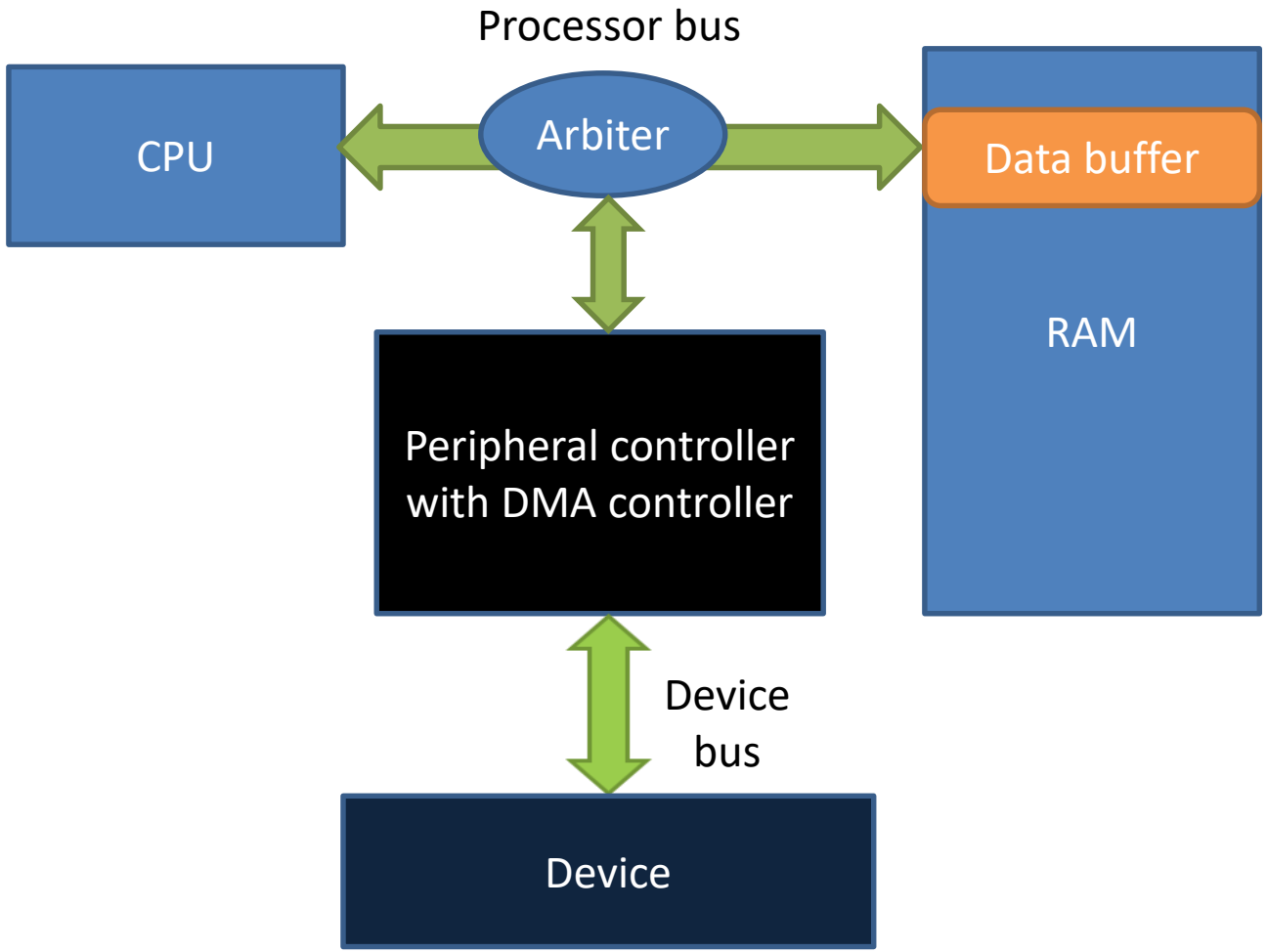
- Voluntary (program-intended) execution of a trap/system call instruction

- What is the effect of an exception?
  - Control flow jumps to the first instruction of an exception handler
  - Processor execution mode switches to protected/kernel mode
  - Depending on the nature of the exception, control flow is either handed back to the instruction causing the exception or the instruction after it
  - In some cases of serious errors, the exception handler has the option of resetting the processor

What are Interrupts?

- Interrupts are hardware events that result in interruption of regular control flow
- Like exceptions, control flow jumps to the first instruction of the interrupt handler routine
- Processor execution mode switches to protected/kernel mode
- Most interrupts are caused by hardware to trigger operating system-level software responses that process data generated by hardware and pass this post-processed data to application software for end-user consumption

IO Hardware Architecture



How does the processor view all of this?

- We will assume like most modern architectures that IO is memory mapped
- In the processor's view, therefore, a peripheral is in effect, a specific range of addresses that it can read to and write from, i.e. it can perform loads and stores

What happens during IO device interactions with the core?

- The need for buffering – to cover for data bursts/relative device speed differences/free up user/kernel copy space both in the incoming and outgoing directions
- DMA – the Direct Memory Access controller has direct access to memory, i.e. it can master the processor-memory bus and transfer data to/from RAM to/from the peripheral controller
- Processor writes to the buffer are normally followed by a DMA transfer command that sets the DMA in action. When the DMA controller is done, it sets a bit that will indicate the transfer is complete, or an error bit if there was an error.
- Similarly, for reads, processor sends a read request to the DMA, and the reader thread blocks. The DMA when done with the read from the device controller, responds by setting a read done bit that interrupts the processor where, via a copy from the DMA-ed buffer to user buffer, the application read is satisfied.

Putting it all together – The need for an OS

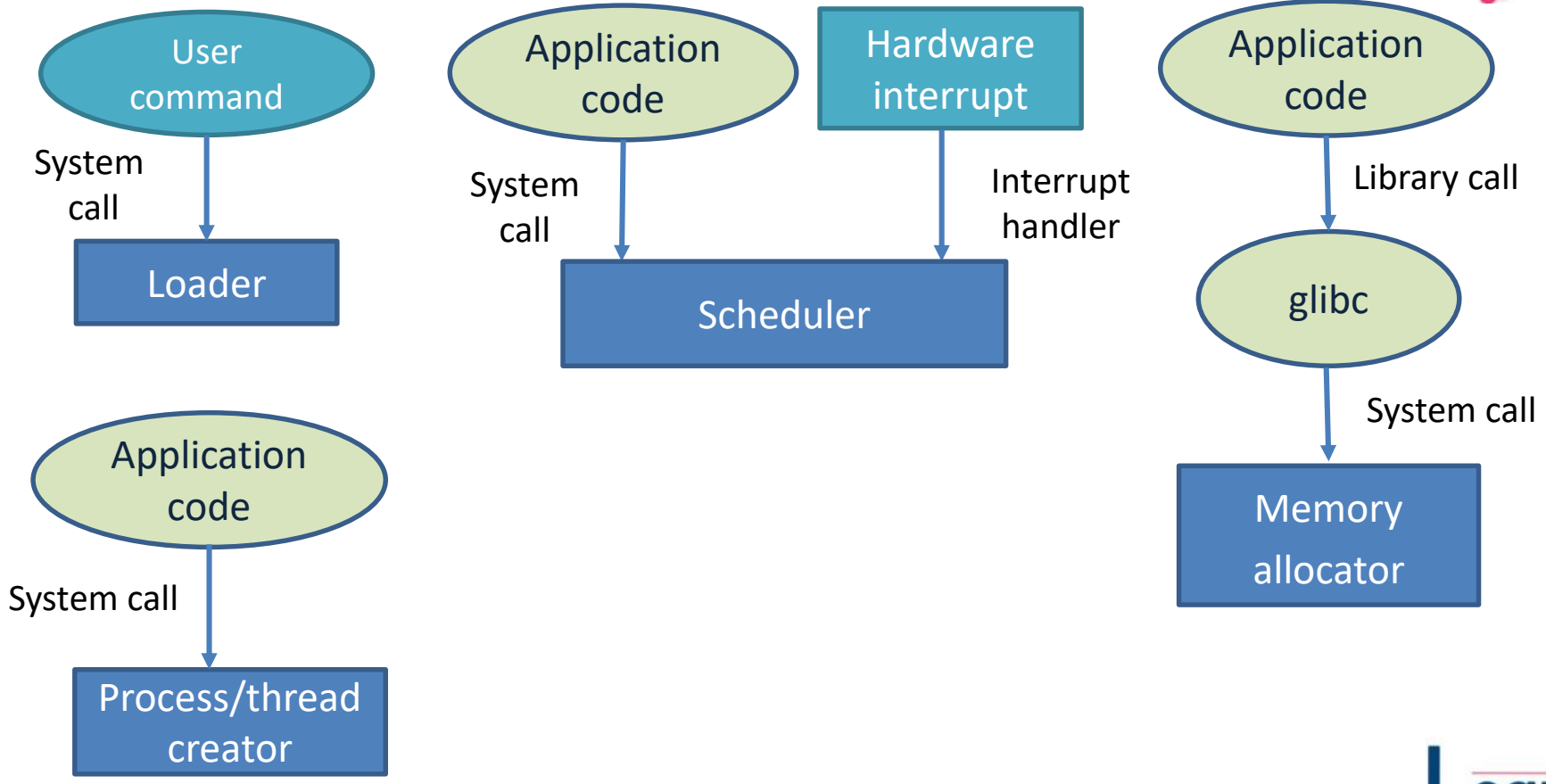
- Let's now look at the various application requirements (high-level, low-level and IO access) that drive the need for an OS:
  - Maintaining all contexts (high-level, low-level application and IO access) in a time-bound manner
    - ✓ Space-multiplexing – address spaces, dynamic memory allocation, concurrency control
    - ✓ Time-multiplexing – interrupts, task scheduling
  - Protecting the kernel from application – kernel mode execution and separation of application and kernel address spaces
  - Providing generic system call library interfaces for ease of application programming and code maintenance with kernel software and hardware details hidden in the implementation
  - Reporting of failures/faults and fault tolerance – interaction with processor exceptions and resultant action

Operating System
Structure

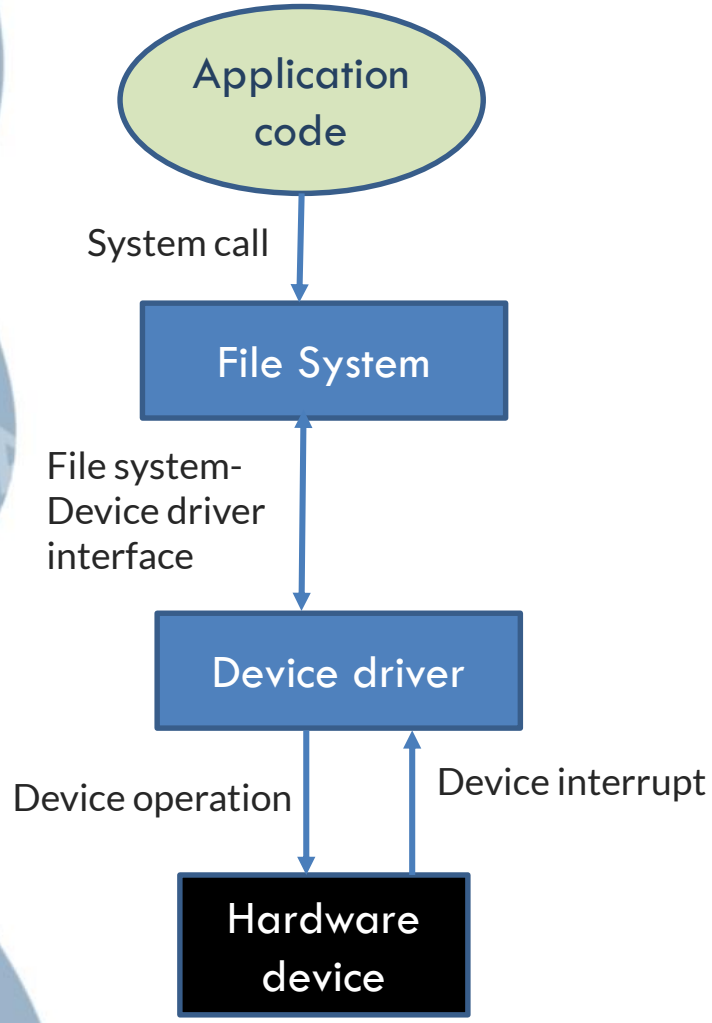
Operating System
Operations

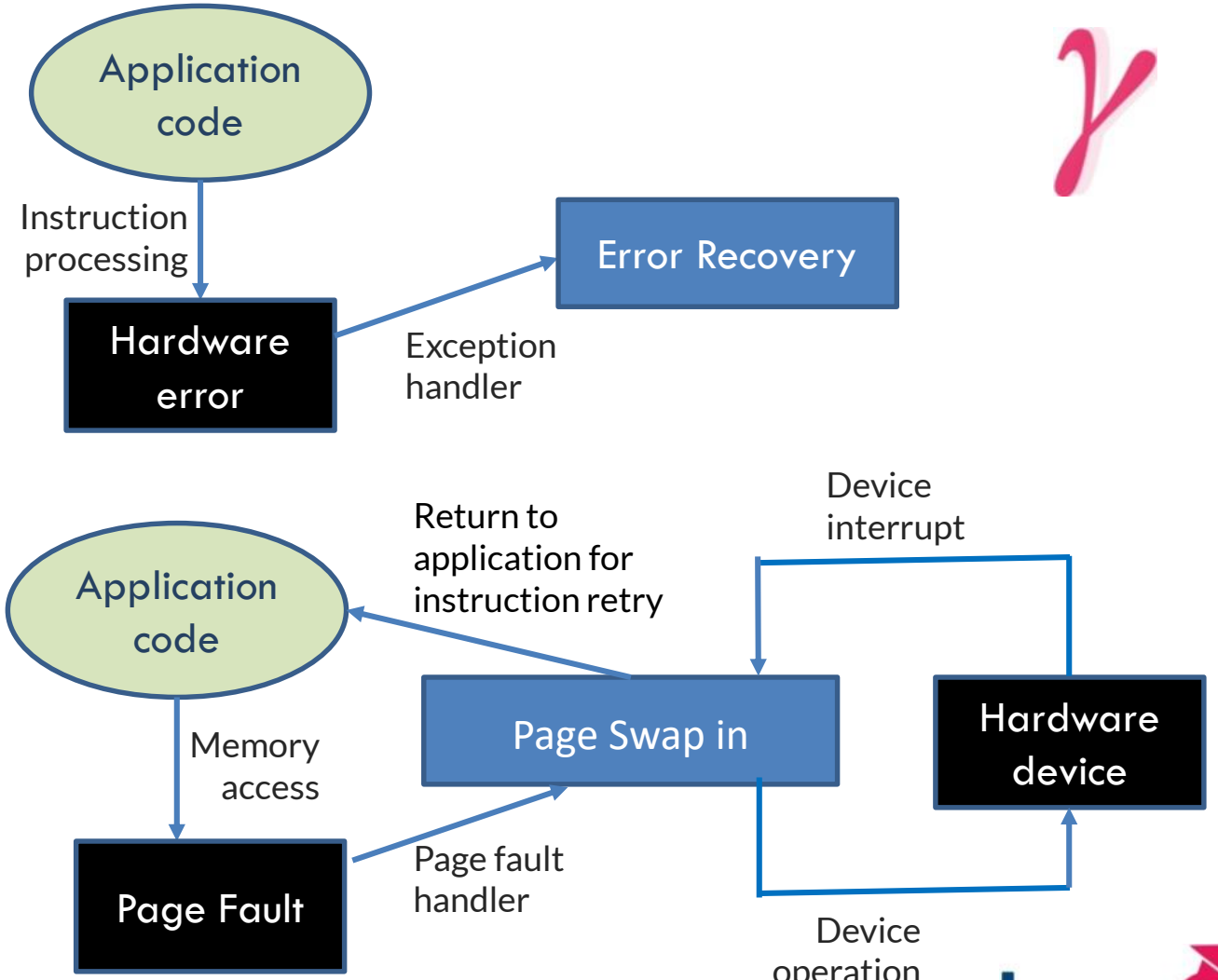
Select Operating System Internals and their application/hardware interfaces - slides 27, 28

What do the internals of an OS look like? How do applications/hardware interface with these internals? -- Snapshot view



Internals of an OS and interfaces





A Categorisation of Operating Systems

- Operating Systems can be classified based on the scheduling policies they implement:
  - Fair-share/time-sharing (Unix, Linux, Solaris)
  - Soft real time (VxWorks, QNX)
  - Hard real time (RTEMS)

Scheduling Policies of Operating Systems

- Fair-share/time-sharing: Priority of task is given importance in task selection, but time-slicing is implemented so that there is an upper limit for how long a task can run continuously. Quick voluntary swap outs are rewarded with an increase in priority.

- Soft real time: Priority of task is given the most importance in task selection. In a few OS's, proportional scheduling is use. But there are no hard deadlines for task execution.

- Hard real time: Various deadline first policies are used.

A Categorisation of Operating Systems (contd.)

- Operating Systems can also be classified based on their software architecture:
  - ✓ Monolithic (Unix, Linux, Solaris, VxWorks)
  - ✓ Microkernel (QNX, LynxOS, Mach)

A Categorisation of Operating Systems (contd.)

- **Monolithic architecture:**
  - Monolithic: As the name indicates, all the OS kernel code is pushed into one monolithic object binary

- **Microkernel architecture:**
  - Only the mandatory parts of the kernel (like the scheduler, and memory management system) are in a monolithic binary, modules like the File Manager (user-level), network stack, character device manager (kernel-level) run as OS daemons/system tasks, and services are provided by message passing

Why Multiprocessing?

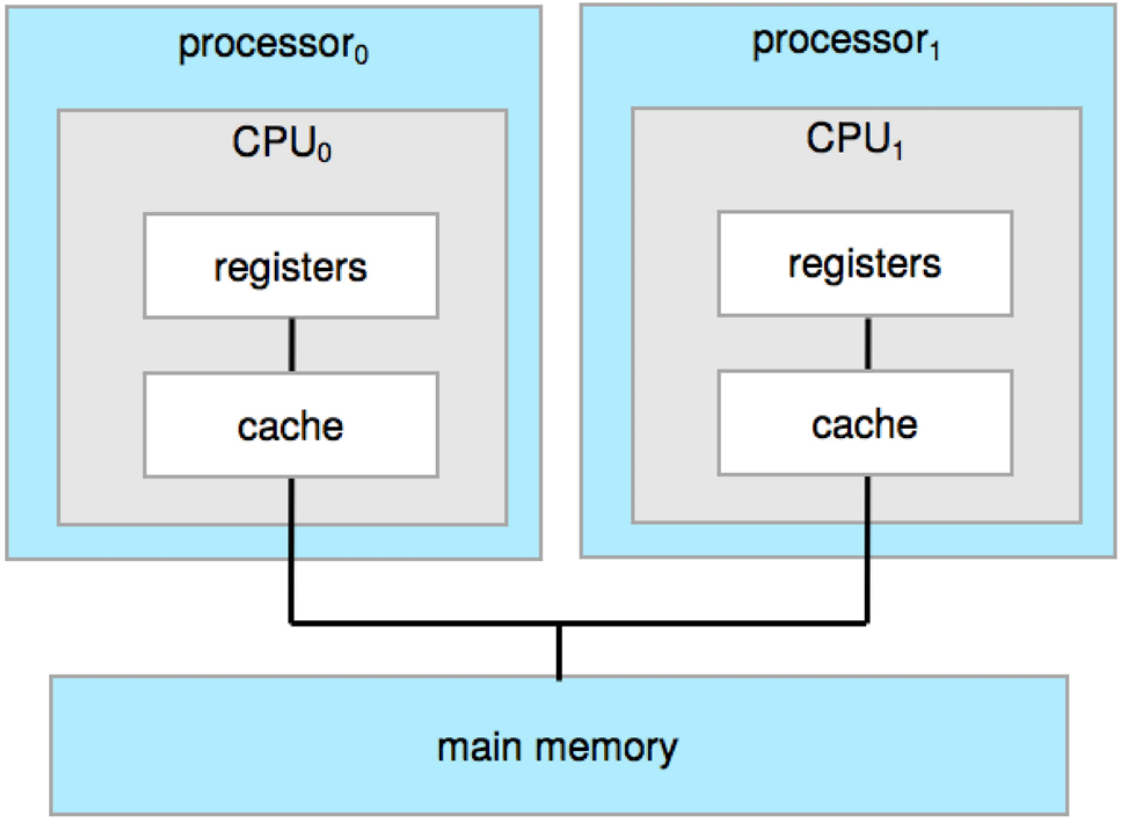
- Three motivations for multiprocessing:
  - True thread-level parallelism, especially for independent threads, is fully exploited only using parallel processing
  - Uni-processing has hit a physical speed wall
  - Fault tolerance – failure of a core/node can be tolerated by another core/node picking up work until the failed core/node is replaced

Types of Multiprocessing – A hardware architecture based categorisation

- Shared memory multiprocessing
  - ✓ Shared bus
  - ✓ Distributed shared memory
- Message passing multiprocessing
- Hybrids
- Graphics Processing Unit (GPU) Cores

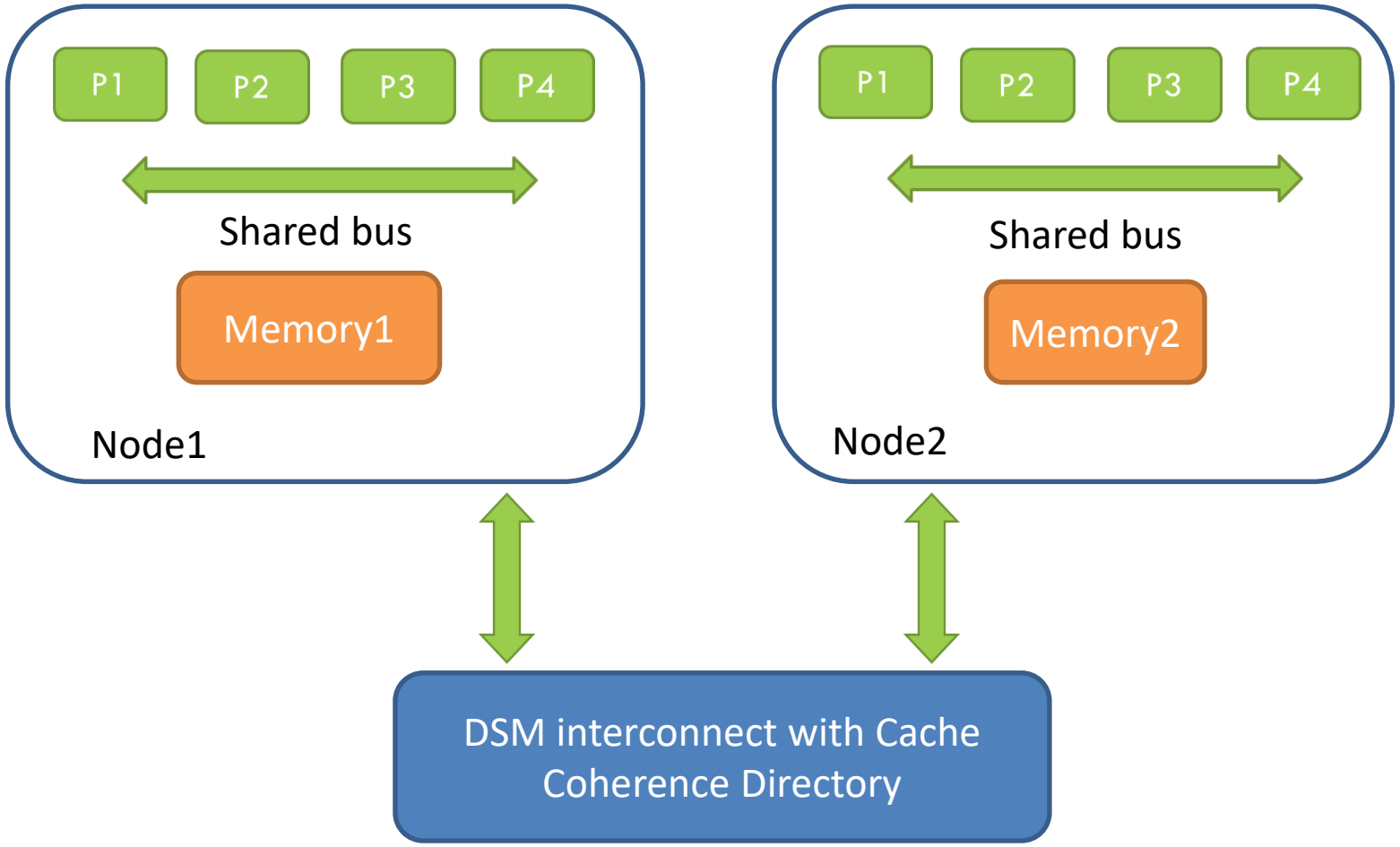
Types of Multiprocessing (contd.)

Shared bus architecture



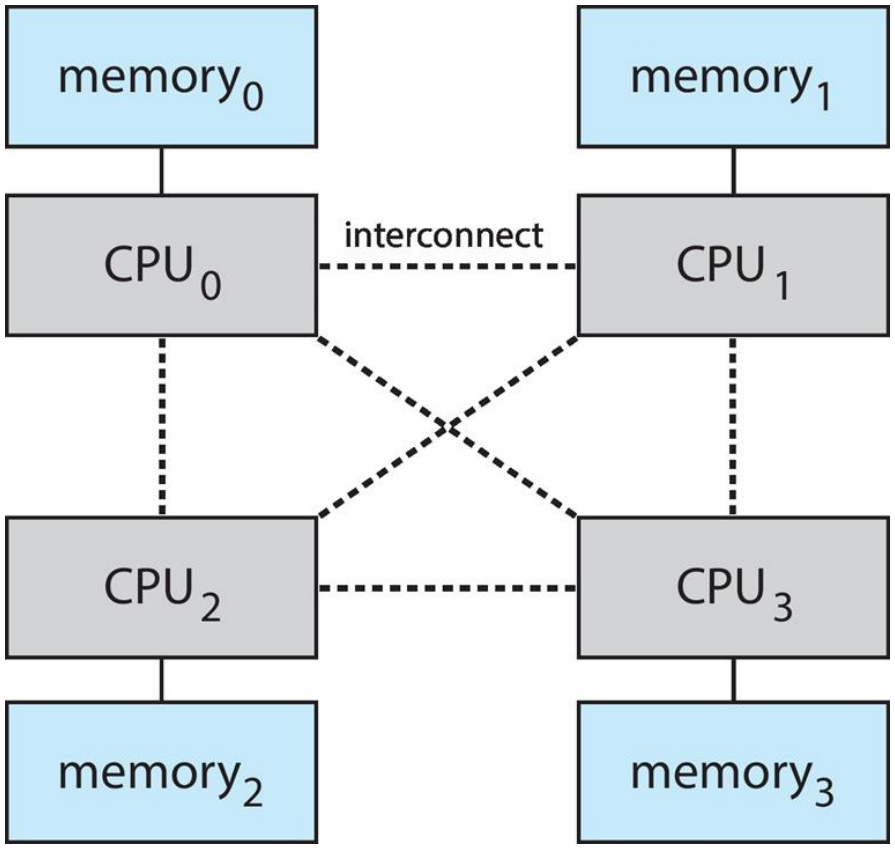
Types of Multiprocessing (contd.)

Distributed shared memory architecture/NUMA with cache-coherent interconnect



Types of Multiprocessing (contd.)

Message passing architecture



Types of Multiprocessing (contd.)

- **Advantages of shared memory multiprocessing:**
  - Can use the same uniprocessor programming model, hence porting an application to a multiprocessor system becomes relatively easy
- **Disadvantages:**
  - Shared bus bottleneck
- **Advantages of message passing multiprocessing:**
  - No shared bus bottleneck, hence more scalable with increase in number of cores
- **Disadvantages**:
  - Have to use a message passing programming model, hence porting an application to a multiprocessor system becomes relatively difficult

Types of Multiprocessing – a software architecture based categorisation

- **Symmetric multiprocessing**
  - All cores are identical as far as the software that is executed goes (kernel and user) – simplest and most popular model

- **Asymmetric multiprocessing**
  - One or more cores execute different applications and even in some cases, use different operating systems

  - In rare cases, there is a master/slave type asymmetry, where only one core executes most of the OS kernel code, and the other cores only execute a task pick up block

Computing System Components

Select hardware components and software components

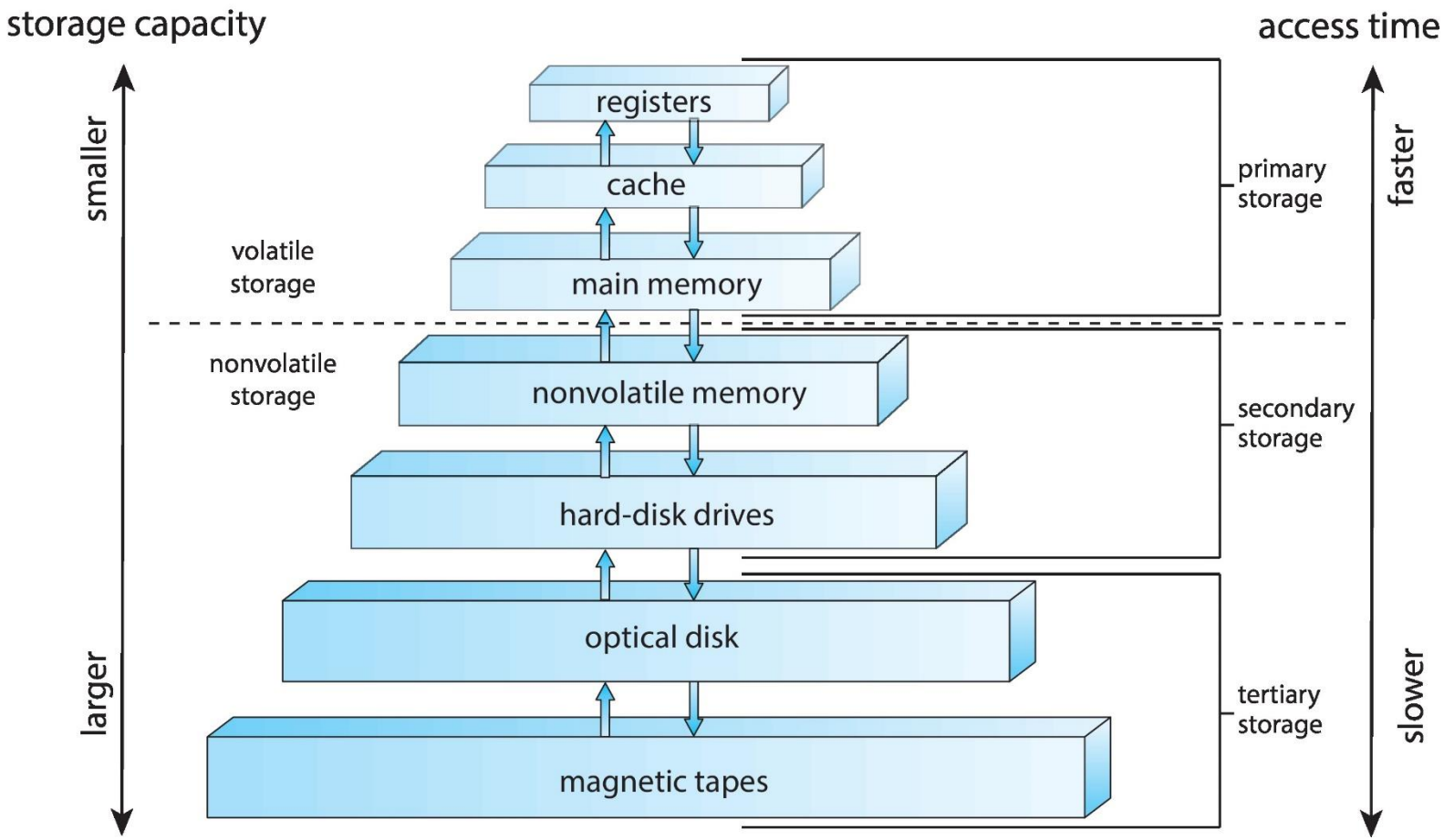
Review of relevant computer organization elements and OS software – slide plan

- Storage elements, storage hierarchies – Slides 42-43
- Memory addressing, byte ordering, 32-bit/64-bit processors, 32-bit/64-bit OSes – Slides 44-46
- Software hierarchy and OS services review – Slides 47-48

Storage Element Characteristics

- **Storage elements:**
  - **Primary storage (Main memory)**
    - This memory provides random access (RAM), volatile, semiconductor storage
    - Off-chip, but faster than secondary storage
    - Volatile – data needs power for persistence
  - **Primary storage (Cache)**
    - On-chip, faster, but more limited in space than RAM, content-addressable semi-conductor memory
    - Volatile like RAM
  - **Secondary storage**
    - Non-volatile, sequential access magnetic/semiconductor flash storage, slower than RAM
    - Read-only memory (BIOS) – flash

Storage Hierarchy



Memory Addressing and Byte Ordering

- Most modern memory architectures are byte-addressable
  - Smallest unit of memory addressable (accessible by a load/store instruction) is a byte
  - Byte ordering is different in different processors
    - ✓ Big-endian (most significant bytes of a 32-bit variable stored at lower address) – PowerPC
    - ✓ Little-endian (least significant bytes of a 32-bit variable stored at lower address)– Intel, ARM

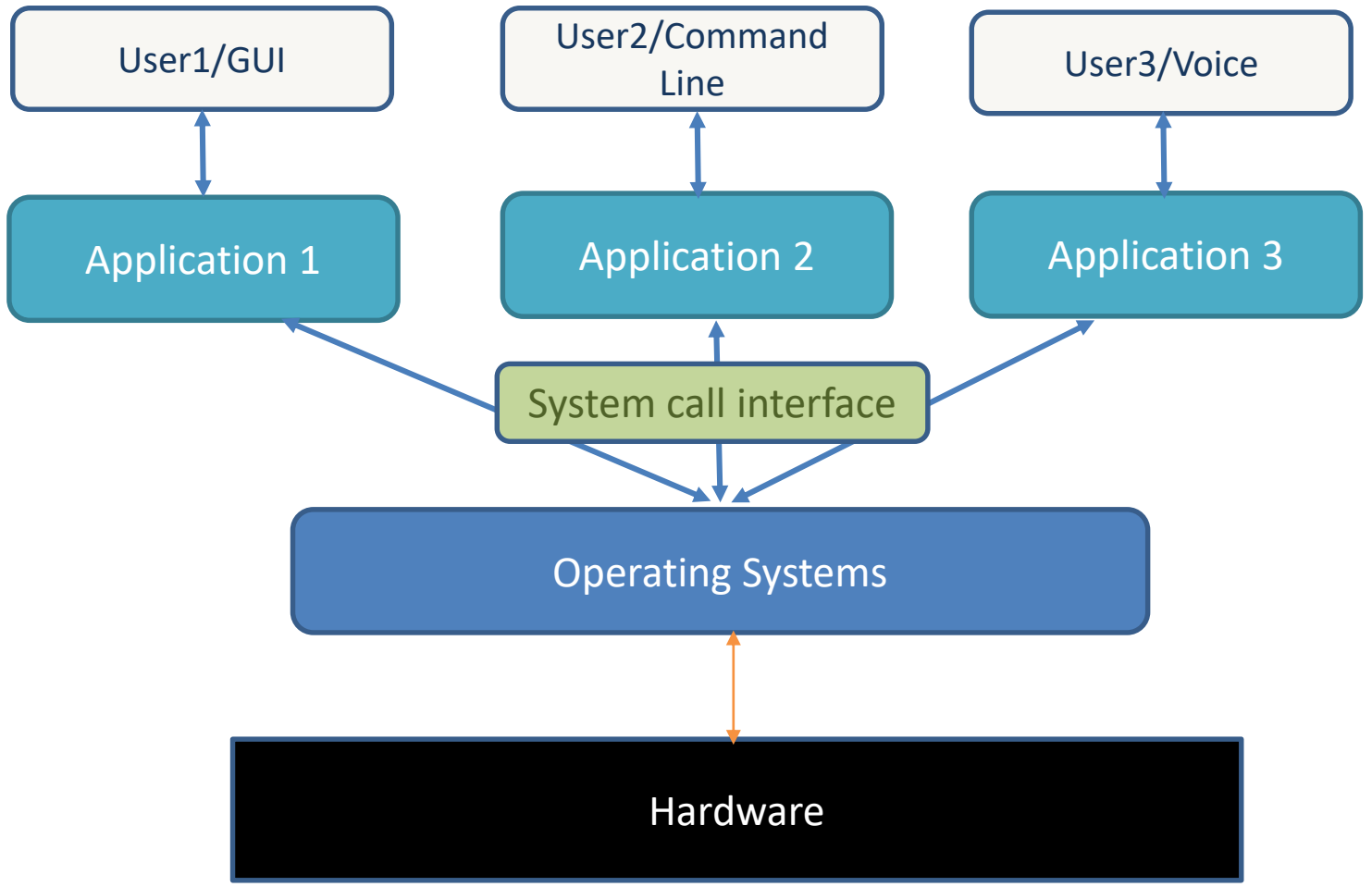
What are 32 bit and 64 bit **Processors** ?

- A 32-bit processor has general-purpose and special-purpose registers that are 32-bit wide, whereas a 64-bit processor has registers that are 64 bit wide
- Generally, on a 32-bit processor, a word is 32 bits, and on a 64-bit processor, a word is 64 bits
- The address bus on a 32-bit processor is 32 bits wide and on a 64-bit processor is 64 bits wide
- Therefore on a 32-bit processor, addressable memory is 2^32 = 4 Gigs, whereas on a 64-bit processor, it is in theory, 2^64 bits although as of now, MMU architectures are limited in practice to 2^36 bits
- Most 64-bit processors can operate in a 32-bit compatibility mode, which means the higher order 32 bits of the address bus and registers are ignored in hardware
- Data bus widths are not directly related to this:
  - ✓ On most 32-bit processors, they are 64 bits wide, promoting wider data fetches into the cache assuming spatial locality of program data access

What are 32 bit and 64 bit **Operating Systems** ?

- A 32 bit operating system is one whose software has been compiled using a 32-bit compiler – i.e. the logical/compile-time addresses generated are 32 bits in length
- Code that configures the MMU sets it up for 32 bit logical-to-physical address translation
- Correspondingly, a 64-bit operating system is one who software has been compiled using a 64-bit compiler, i.e. the logical addresses generated are 64 bits in length (48 bits sign-extended to 64 bits, since as of now, the MMU will set up only 48 bit translation)
- Code that configures the MMU sets it up for 48 bit logical-to-48 bit physical address translation
-
- You can run 32-bit applications on a 64-bit processor with a 64-bit OS, with the processor set up in 32-bit compatibility mode. Similarly, you can run a 32-bit OS on a 64-bit processor in compatibility mode

Software Component Hierarchy Review



Operating Systems services - Review

- User Interface
- Program loading and execution
- Process start up and scheduling
- Address space separation
- Inter-process communication
- Concurrency control
- Dynamic memory allocation
- File system manipulation
- Input / Output Operations
- Networking
- Fault detection and recovery
- Auditing of OS data structures
- Protection of kernel data

Operating Systems Design Issues

Select issues in Operating Systems design – Slides 50- 59

Select Operating Systems Design Issues

- **Program Loading and Execution:**
  - ✓ Fault tolerance requirements play a big role in deciding the design. Some carrier grade systems require run-time code patching for instance.
  - ✓ In carrier grade and other embedded systems that require highly reproducible behavior down to where your code and data are placed, you will opt for simplicity and predictability over flexibility. Hence, you will generally opt for non-relocatable code and data.
  - ✓ On systems with slightly looser constraints, you might opt for flexibility, with modules that contain relocatable code and data.

Select Operating Systems Design Issues

- **Process Start up and Scheduling, Address Space Separation**
  - ✓ Some very highly available systems cannot afford expensive process startups
  - ✓ Therefore, they opt for simplicity over some space savings, and go with logical address = physical address mappings and don't have fork type primitives
  - ✓ Real-time responsiveness required by applications will determine scheduling policy used

Select Operating Systems Design Issues

- **Inter-Process Communication (IPC)**
  - ✓ Nearly every OS will have message queues of different flavours as their principal IPC mechanism because message queues have a lot of useful in-built features:
  - ✓ Automatic synchronization between producer and consumer of message
  - ✓ The receiver can read messages of a certain type, from any point in the queue. Some implementations of message queues support message priorities. Message queues are therefore, not necessarily always FIFO
  - ✓ Shared memory is used a fair bit in carrier grade systems. It is faster than message queues, but will need explicit synchronization to ensure the same results as expected of serial operations.

Select Operating Systems Design Issues

- **Concurrency Control**
  - ✓ All OSes provide some form of concurrent control mechanisms because variables shared implicitly across threads or explicitly shared memory across processes need some form of concurrency control to get ensure the same results as expected of serial operations.
  - ✓ Semaphores and basic scheduler locks are provided by most OSes. Semaphores need to have priority inversion issues sorted out. Locks need to have bailout mechanisms in case locks are done for too long. More sophisticated locks are also provided by some OSes to provide as much concurrency as possible without affecting correctness of execution.

Select Operating Systems Design Issues

- **Dynamic Memory Allocation**
  - ✓ Kernel memory allocation – balancing act between fragmentation and performance. Fixed size – faster, but leads to fragmentation.
  - ✓ In highly available systems with tighter constraints, fragmentation is ignored and better performance is opted for. In systems with looser constraints, pre-allocated variable size blocks used with an optimal amount of coalescing.
  - ✓ In systems with tighter constraints, there is no distinction between kernel and user space allocation methods.
  - ✓ In systems with looser constraints, user allocation is allowed to be logically contiguous, but physically non-contiguous with virtual memory and swapping allowed.

Select Operating Systems Design Issues

- **File system manipulation**

  - ✓ Systems with tighter constraints are not allowed the luxury of more sophisticated file systems. But, active and passive partitions for live system upgrades are used. Error-checking is done on passive partitions, and only if error checks pass, partitions will be switched.

  - ✓ In systems with looser constraints, there is no sense of active/passive partitions, but more sophisticated (from a user point of view) file systems that support concurrent readers, but one writer are used. Journaling is used for error recovery.

Select Operating Systems Design Issues

- **Input / Output Operations**
  - ✓ Embedded systems with tight constraints generally support a lesser variety of bus protocols and have a set of very specific drivers as their requirements are very targeted.
  - ✓ More general purpose systems will support a much wider set of IO and issues of backwards compatibility are kept in mind.

Select Operating Systems Design Issues

- **Networking**
  - ✓ Similar to other forms of IO, a wider networking base is supported for more general purpose systems with looser constraints.

- **Supported debugging options**
  - ✓ Debug options range from core dumps of memory that allow for offline analysis to racepoints/breakpoints for more live debugging.
  - ✓ Various log files are also used to understand entire sequences of events that lead to a bug being reported.

Select Operating Systems Design Issues

- **Fault Detection and Recovery**

  - ✓ In very highly available systems, hardware design is tailored to recovering from even very rarely occurring hardware errors. And, this is supported in operating systems software.

  - ✓ In some systems, hardware is commodity hardware, but purely software concepts of task migration and virtual machine migration are supported.

  - ✓ In more general purpose systems, error recovery is much less sophisticated, and often restricted to software task crashes and general system resets in case of kernel crashes.

Select Operating Systems Design Issues

- **Auditing of OS Data Structures and Task Starvation Audits**
  - ✓ Self auditing of OS data structures like ready queues is done periodically and hardware/software watchdogs are implemented.
- **Protection of Kernel Code and Data**
  - ✓ In systems that can afford user-to-kernel mode switches, system calls are relied on.

Thank You!