INFS803
Cloud Computing

Virtualization

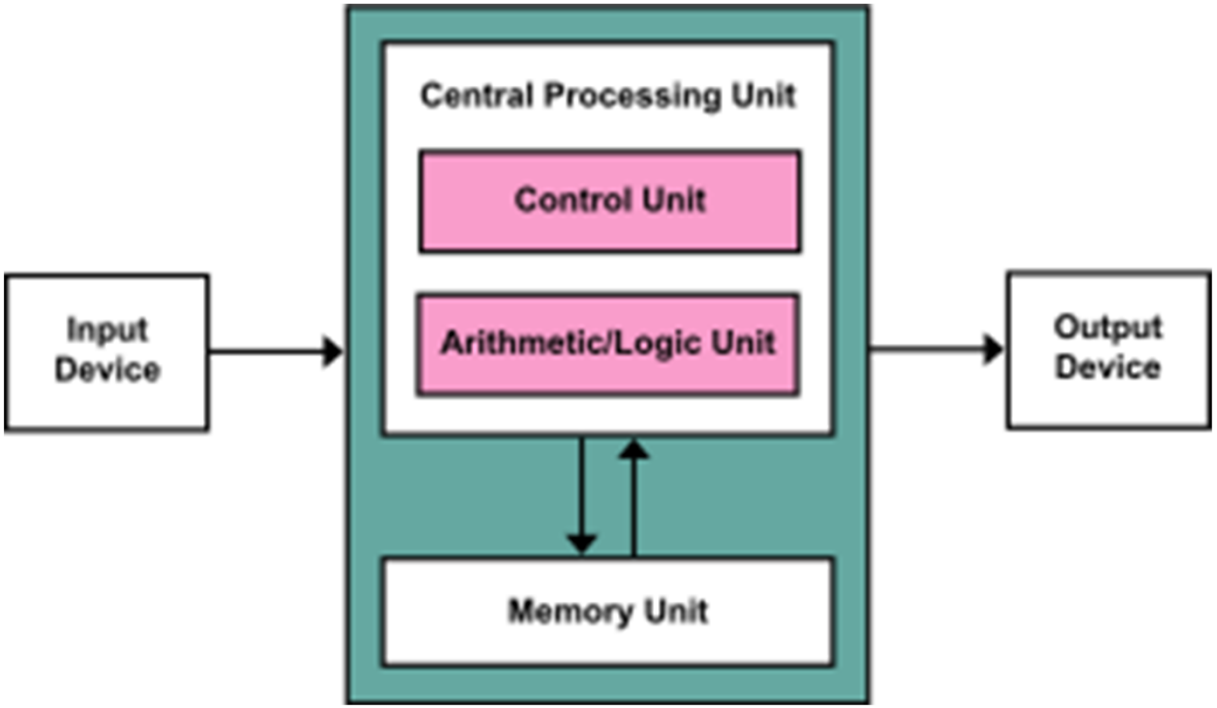
Contents

- Virtualization: motivation and concept
- Layering and virtualization.
- Dual mode execution and Processes
- Virtual memory and page replacement
- Virtual machine monitor.
- Virtual machine.
- x86 support for virtualization.
- Full and paravirtualization.
- Xen.

Motivation

- Three fundamental abstractions are necessary to describe the operation of a computing systems:

  (1) interpreters/processors, (2) memory, (3) communications links



Von Neumann architecture

Motivation (cont'd)

- As the scale of a system and the size of its users grows, it becomes very challenging to manage its recourses (see three points above)

- Resource management issues:
  - provision for peak demands
  - **heterogeneity** of hardware and software
  - machine failures
- **Virtualization is a basic enabler of Cloud Computing, it simplifies the management of physical resources for the three abstractions**

- For example, the state of a virtual machine (VM) running under a virtual machine monitor (VMM) can de saved and migrated to another server to balance the load
- For example, virtualization allows users to operate in environments they are familiar with, rather than forcing them to specific ones

Motivation (cont' d)

- ***"Virtualization**, in computing, refers to the act of creating a **virtual (rather than actual) version of something**, including but not limited to a virtual computer hardware platform, operating system (OS), memory, storage device, or computer network resources."* from Wikipedia

- Virtualization abstracts the underlying resources; simplifies their use; isolates users from one another; and supports replication which increases the elasticity of a system

Motivation (cont' d)

- Cloud resource virtualization is important for:
  - Performance isolation
    - as we can dynamically assign and account for resources across different applications
  - System security:
    - as it allows isolation of services running on the same hardware
  - Performance and reliability:
    - as it allows applications to migrate from one platform to another
  - The development and management of services offered by a provider

Virtualization

- Virtualization simulates the interface to a physical object by:
  - Multiplexing: creates multiple virtual objects from one instance of a physical object. Many virtual objects to one physical. Example - a processor is multiplexed among a number of processes or threads.
  - Aggregation: creates one virtual object from multiple physical objects. One virtual object to many physical objects. Example - a number of physical disks are aggregated into a RAID disk.
  - Emulation: constructs a virtual object of a certain type from a different type of a physical object. Example - a physical disk emulates a Random Access Memory (RAM).
  - Multiplexing and emulation. Examples - virtual memory with paging multiplexes real memory and disk; a virtual address emulates a real address.

Layering and Virtualization

- Layering – a common approach to manage system complexity:
  - Simplifies the description of the subsystems; each subsystem is abstracted through its *interfaces* with the other subsystems
  - Minimises the interactions among the subsystems of a complex system
  - With layering we are able to design, implement, and modify the individual subsystems independently
- Layering in a computer system:
  - Hardware
  - Software
    - Operating system
    - Libraries
    - Applications
- Layering in cloud service models

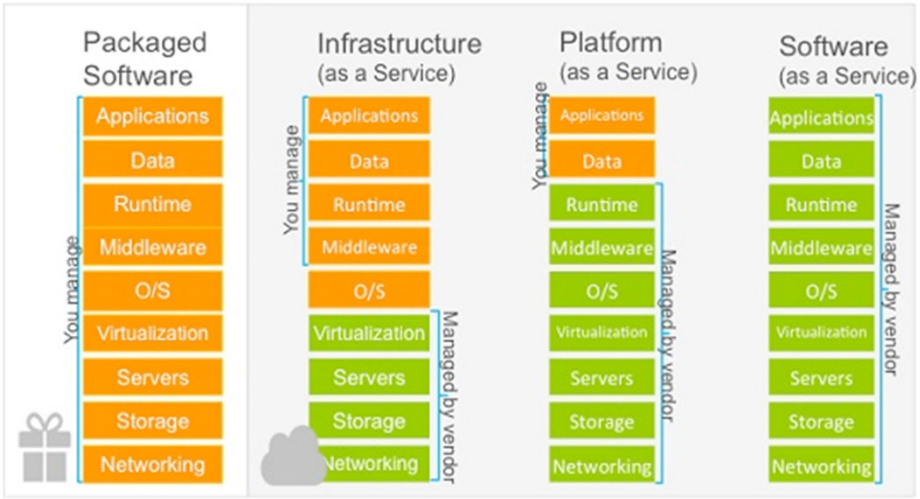
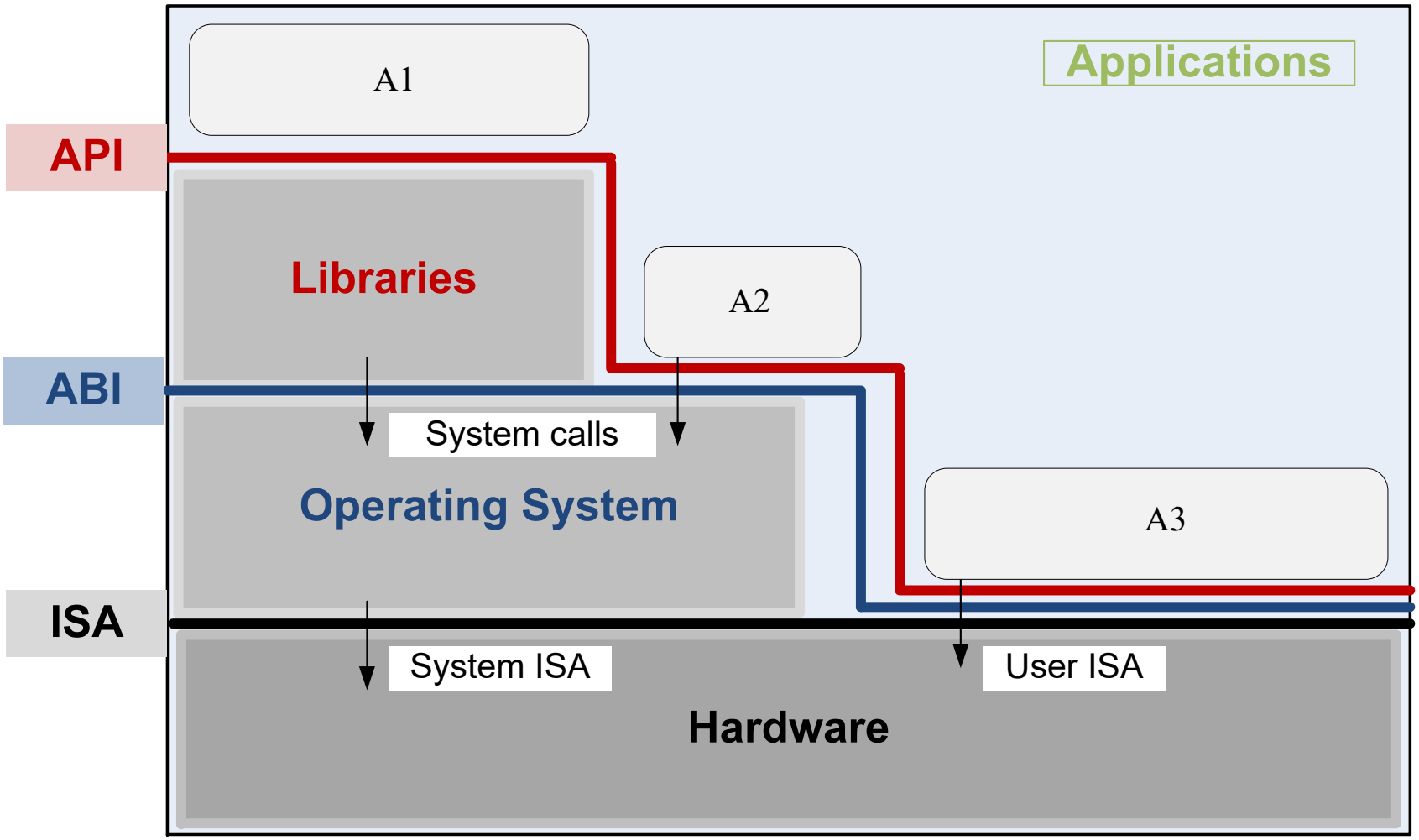


Figure 1. Source: Microsoft Azure

Layering and Interfaces



Application Programming Interface (API, OS&hardware-independent), Application Binary Interface (ABI, OS-dependent), and Instruction Set Architecture (ISA). An application uses library functions (A1), makes system calls (A2), and executes machine instructions (A3)

Interfaces

- **Instruction Set Architecture** (ISA) – at the boundary between hardware and software.

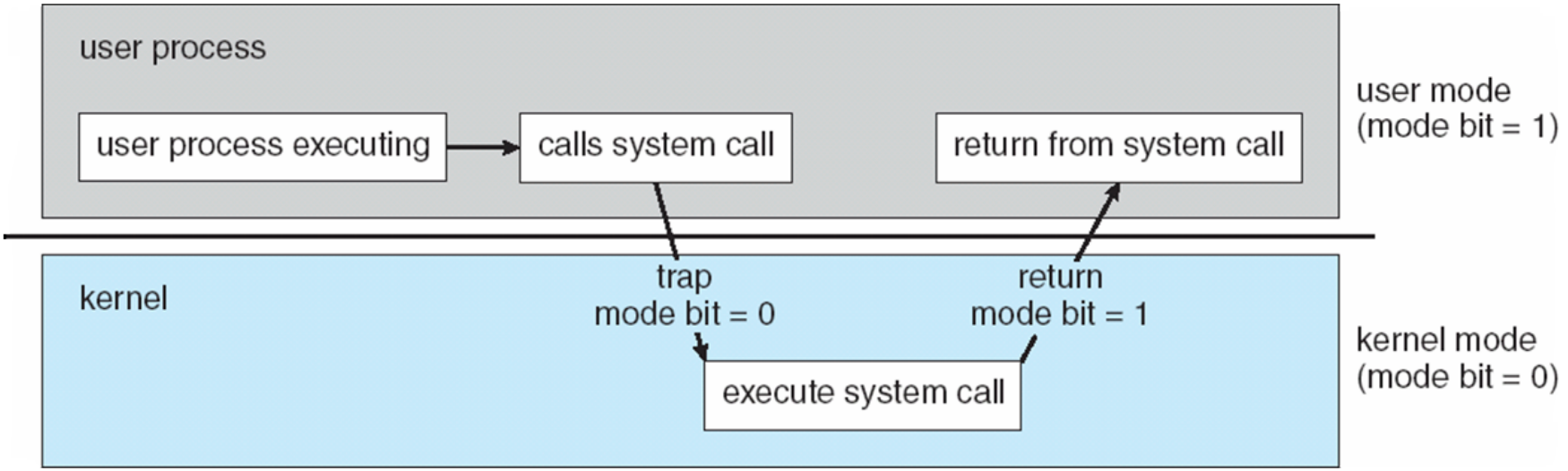
- **Application Binary Interface** (ABI) – allows the ensemble consisting of the application and the library modules to access the hardware; the ABI does not include *privileged* system instructions, instead it invokes system calls.
- **ABI is the projection of the computer system seen by the process.**

- **Application Program Interface** (API) - defines the set of instructions the hardware was designed to execute and gives the application access to the ISA; it includes high-level language (HLL) library calls which often invoke system calls

API is language-dependent, how about Web API/service?

Dual-Mode Operation

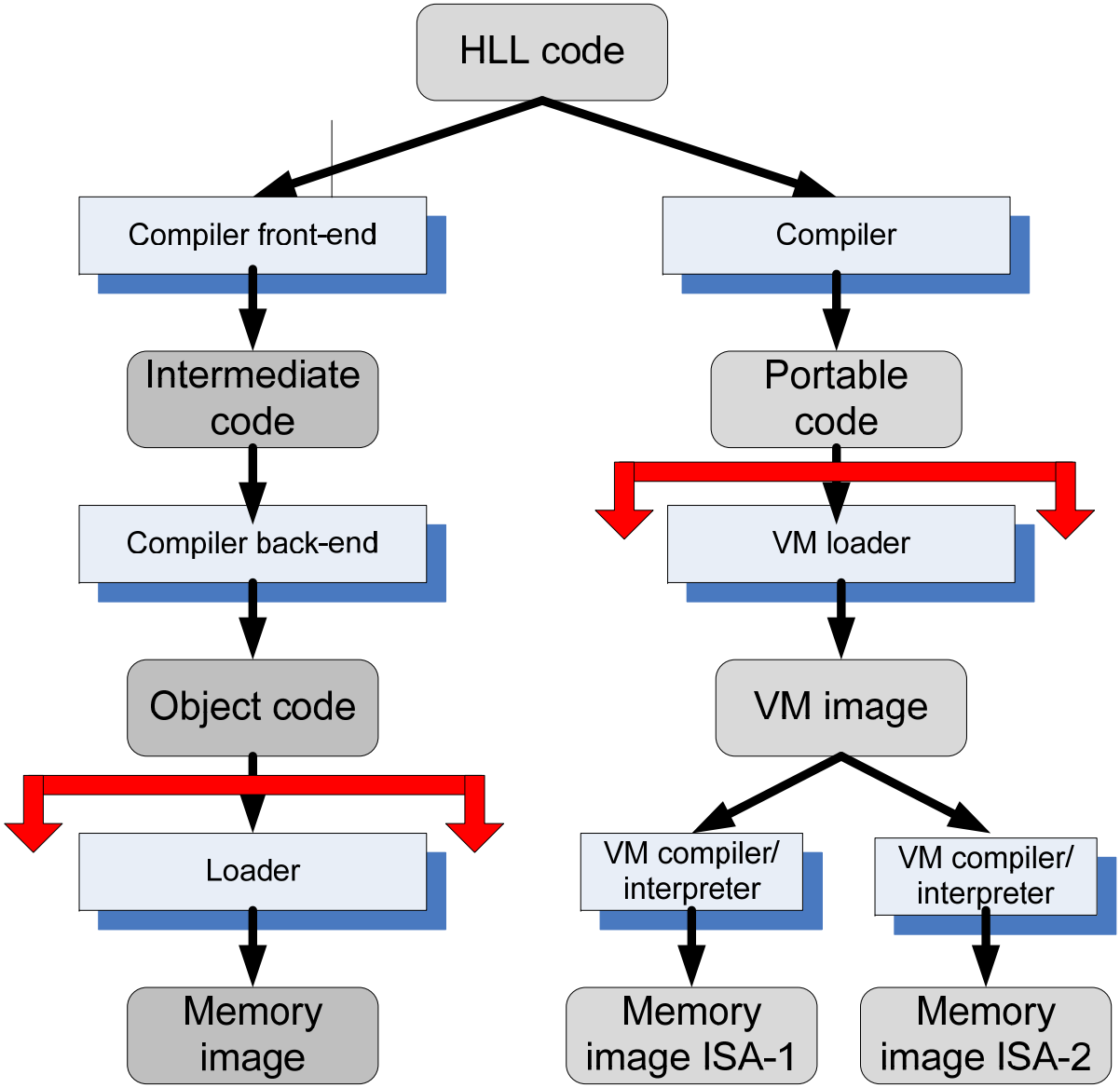
- Dual-mode operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - Mode bit provided by hardware
    - Ability to distinguish when system is running user or kernel code
    - Some instructions are **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return resets it to user



User-mode vs Kernel-mode

- Kernel-code (in particular, interrupt handlers) runs in kernel mode
  - the hardware allows all machine instructions to be executed and allows unrestricted access to memory and I/O ports
- Everything else runs in user mode
- The OS relies very heavily on this hardware-enforced protection mechanism

HLL Language Translations



Code portability

- Binaries created by a compiler for a specific ISA and a specific operating systems are not portable

- It is possible, though, to compile a HLL program for a virtual machine (VM) environment where portable code is produced and distributed and then converted by binary translators to the ISA of the host system

- A **dynamic binary translation** converts blocks of guest instructions from the portable code to the host instruction and leads to a significant performance improvement, as such blocks are cached and reused

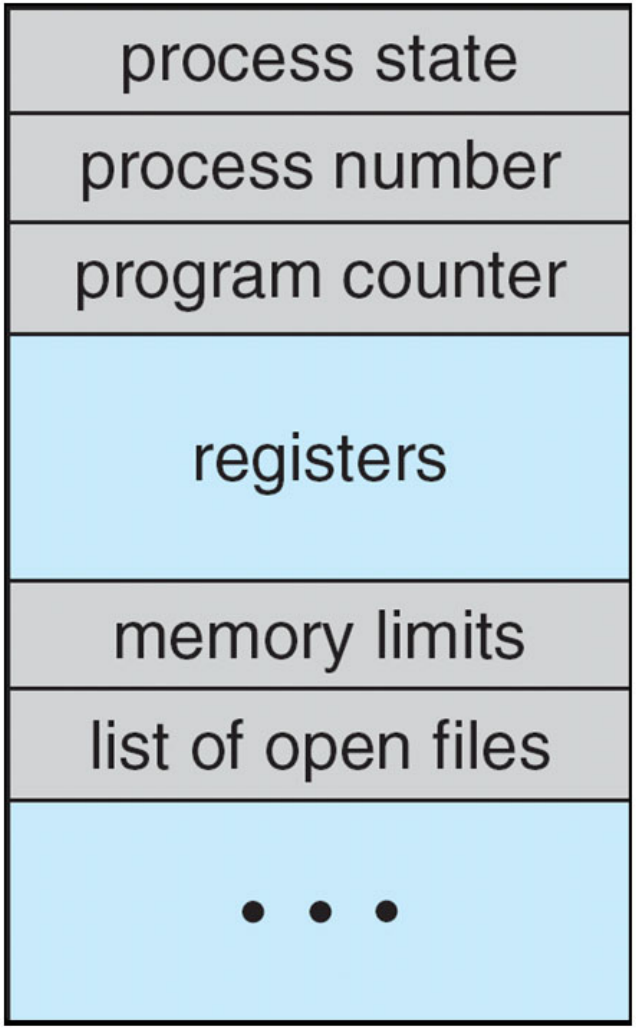
Processes

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process Control Block (PCB): Information about a process

Information associated with each process (also called **task control block**)

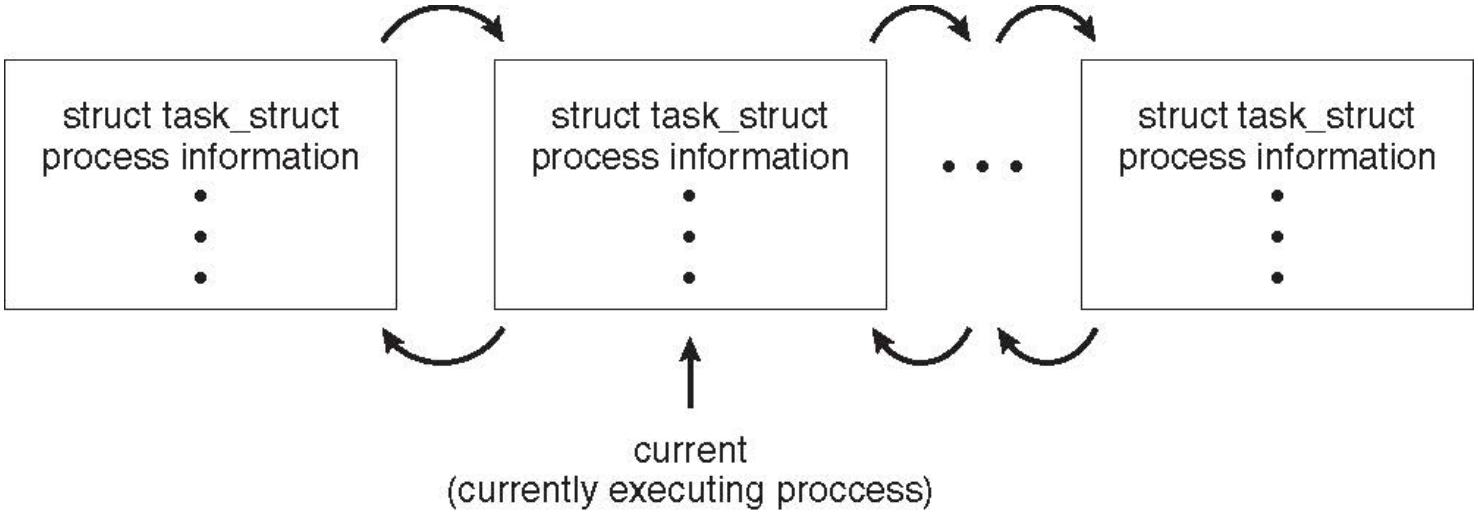
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Process Representation in Linux

Represented by the C structure `task_struct`

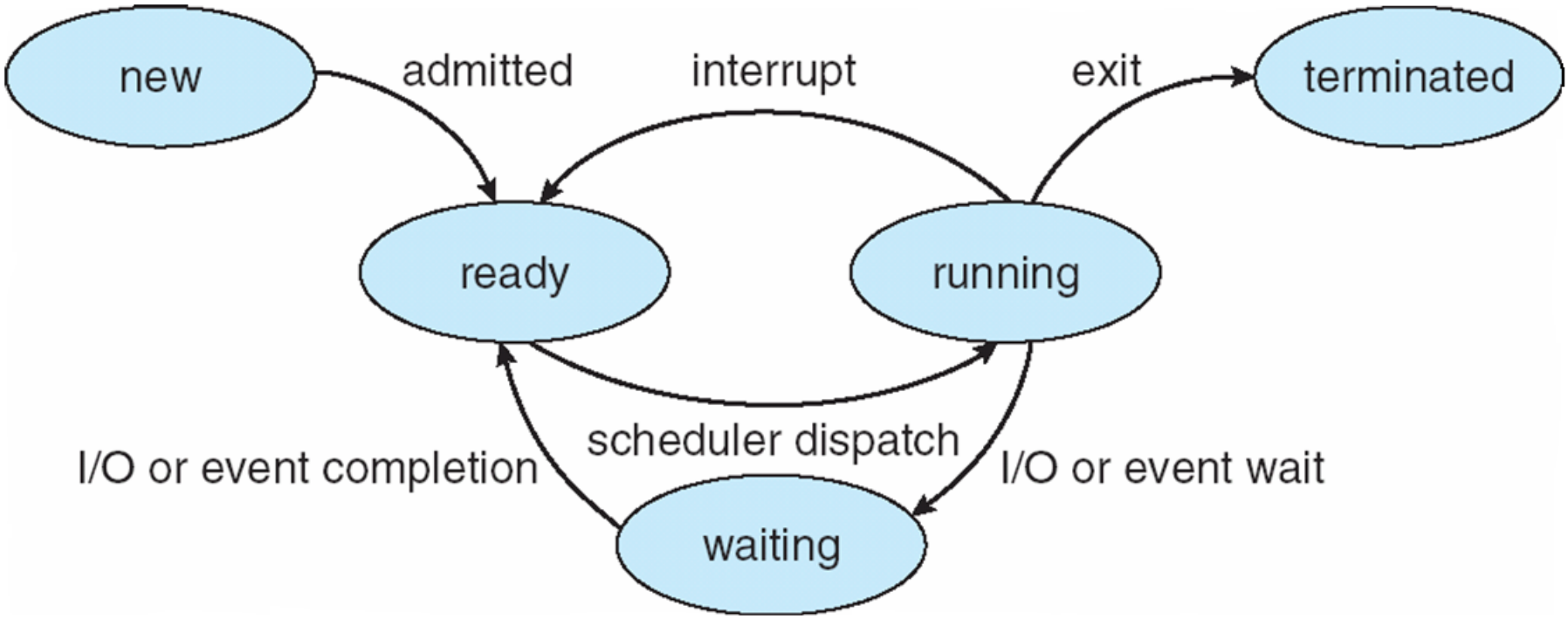
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process
*/
```



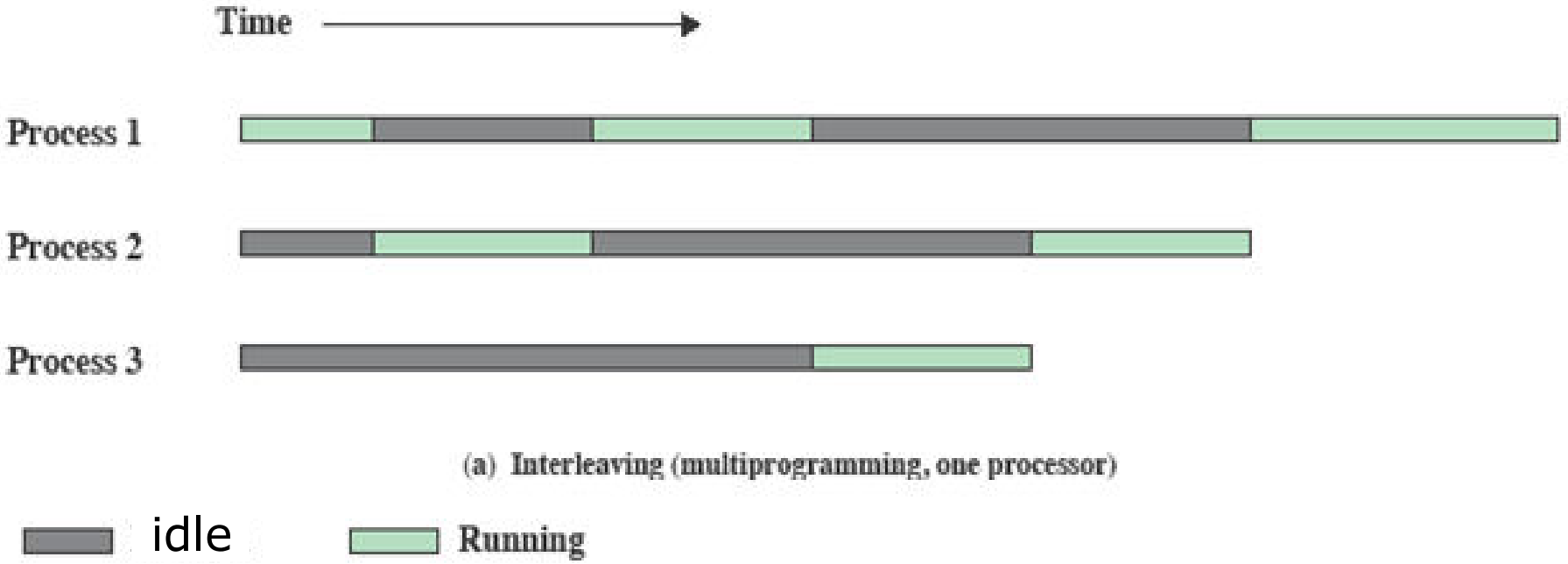
Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

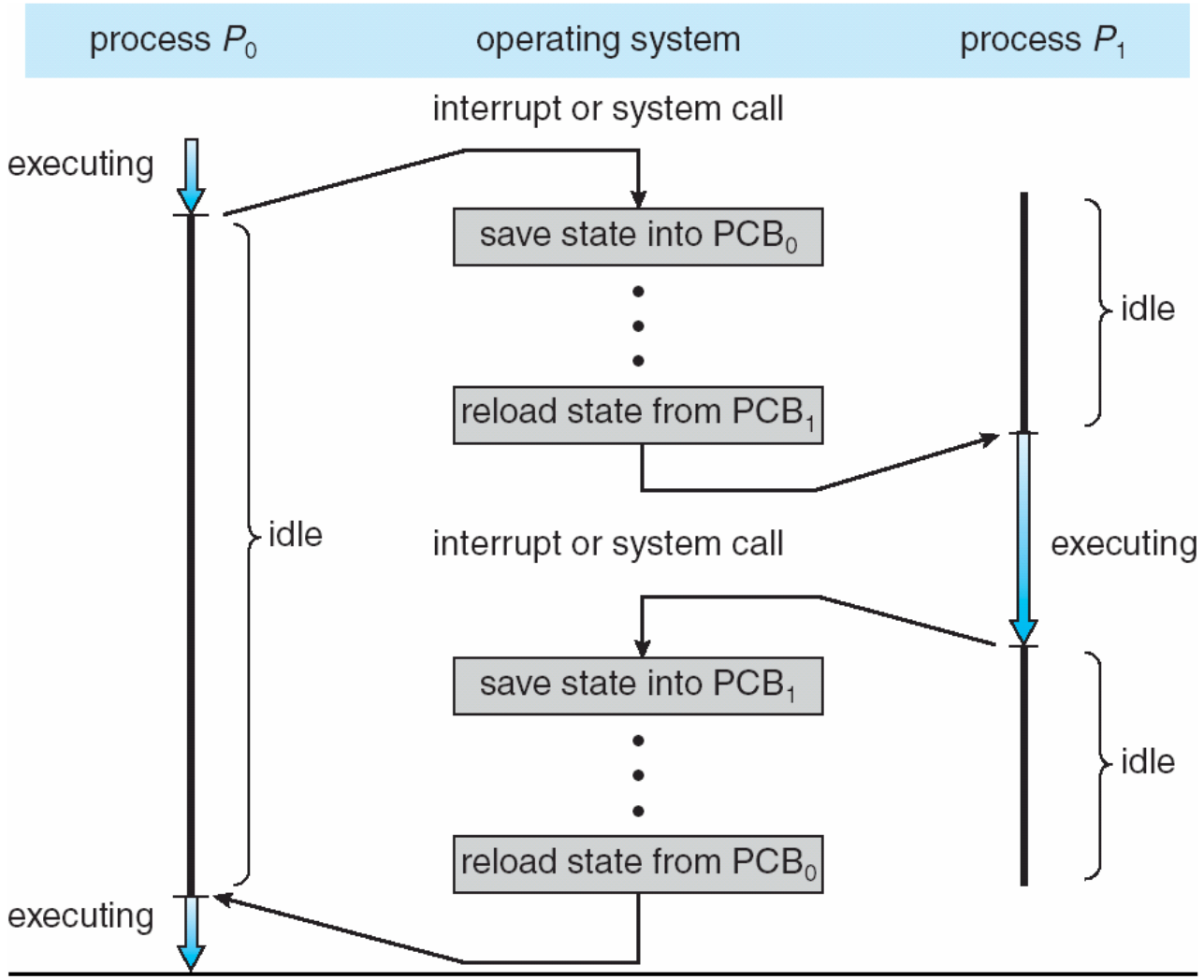


A multi-tasking environment



One CPU, Three Processes

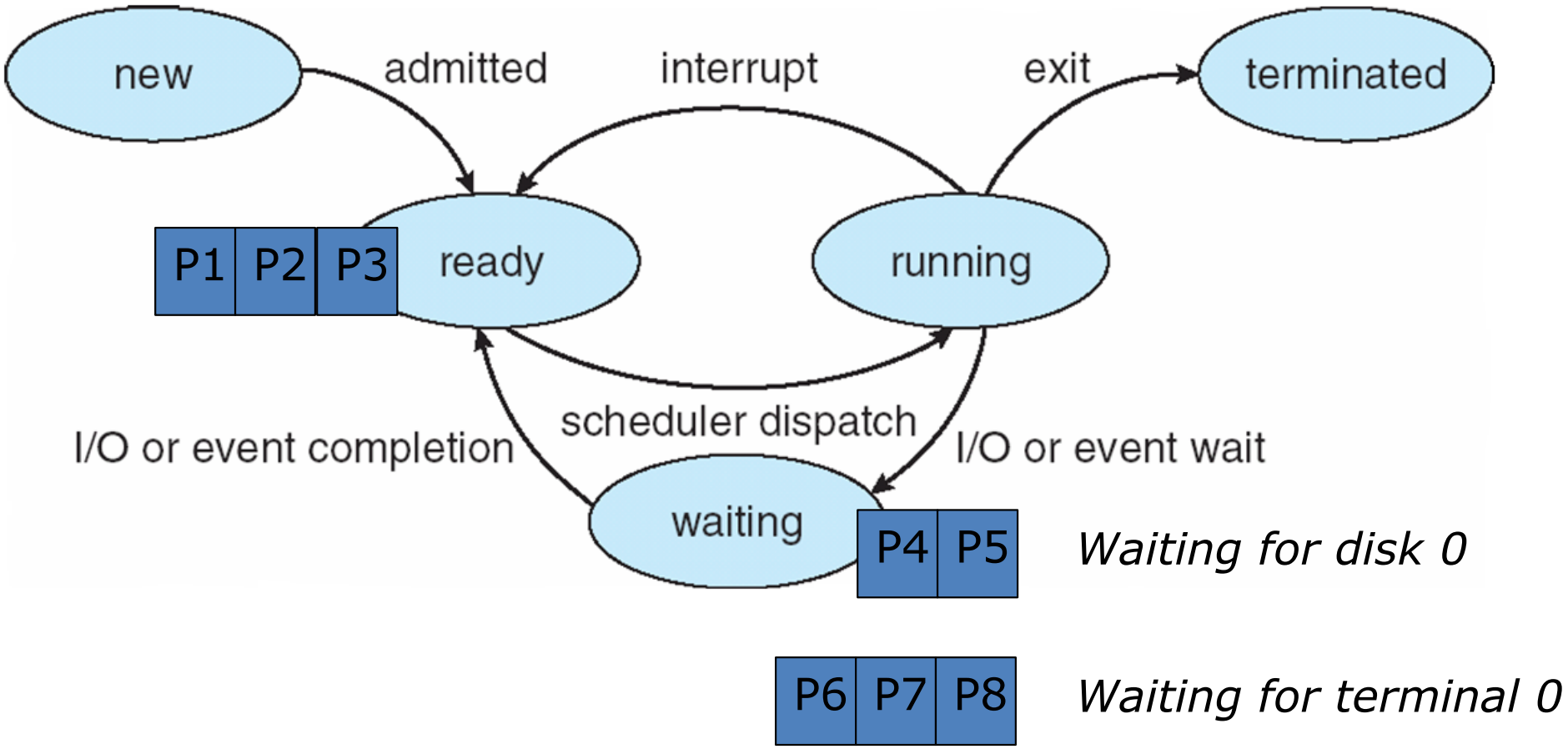
CPU Switch From Process to Process



Process Scheduling

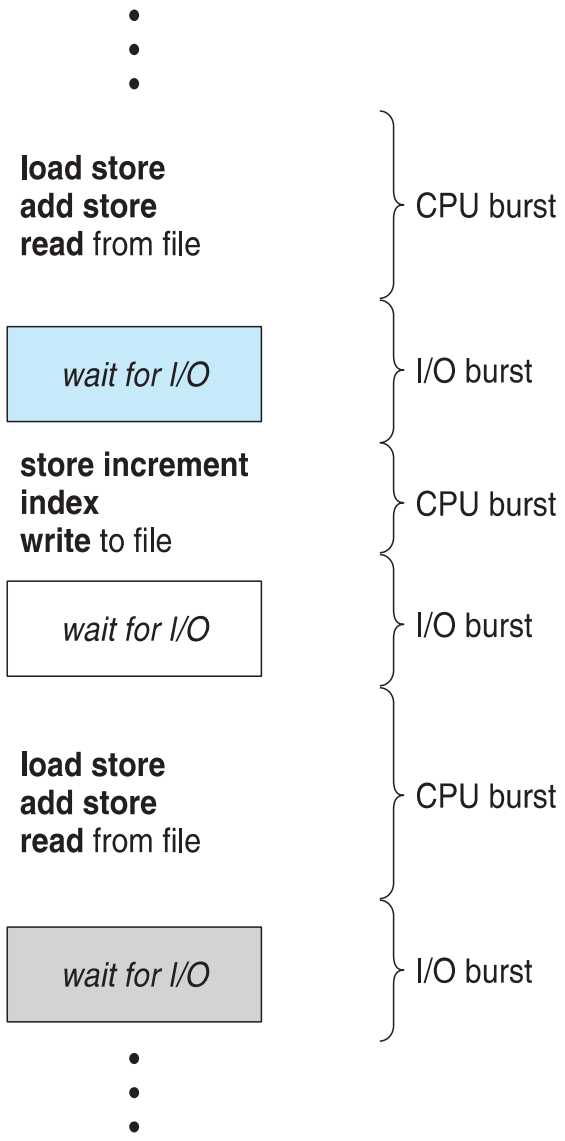
- Maximize CPU use, quickly switch processes onto CPU for time sharing
  - E.g., if a process is waiting for I/O, CPU can be utilized by other processes
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Diagram of Process State



CPU burst and I/O burst

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



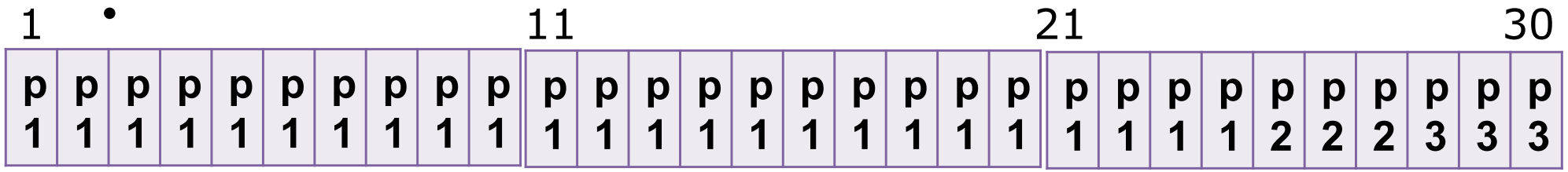
Alternative sequence of CPU and I/O burst

First- Come, First-Served (FCFS) Scheduling

For simplicity of illustration, we only consider one CPU burst per process

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

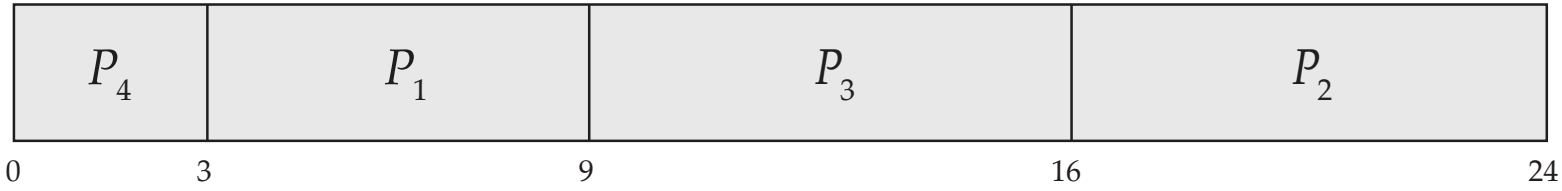


- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Shortest-Job-First (SJF) Scheduling

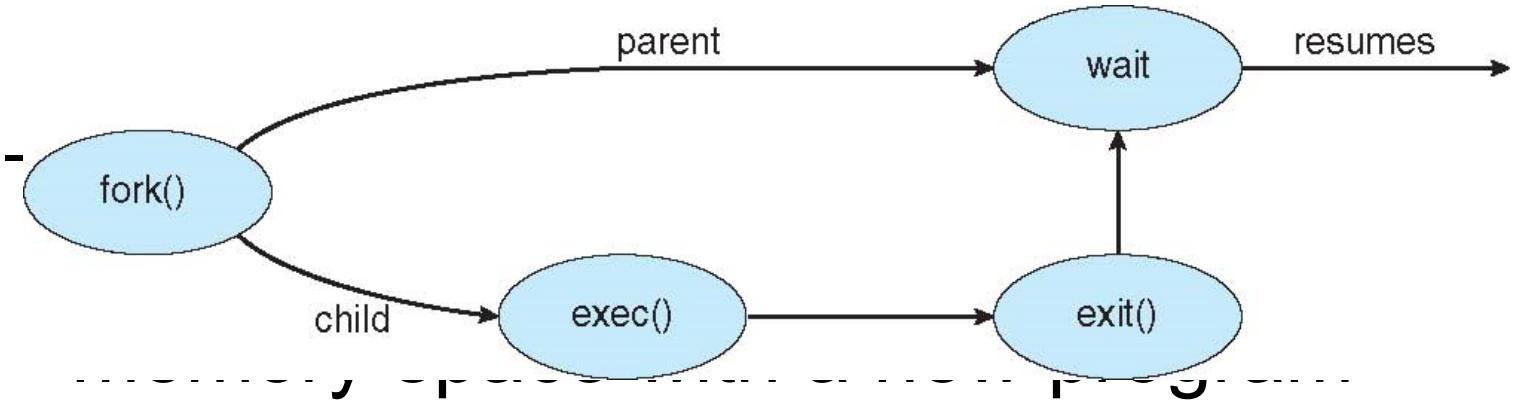
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF



Process Creation

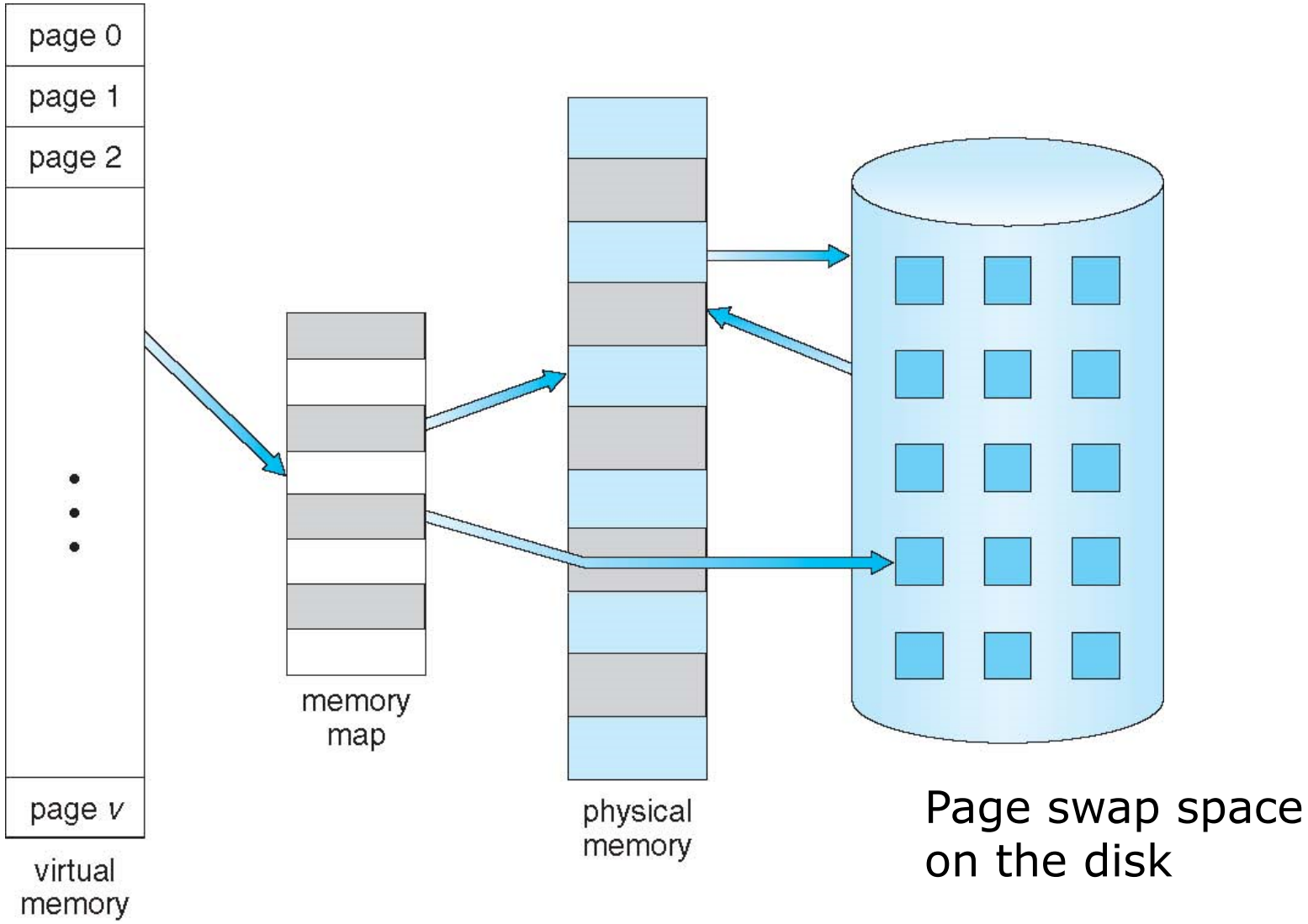
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new



Process Termination

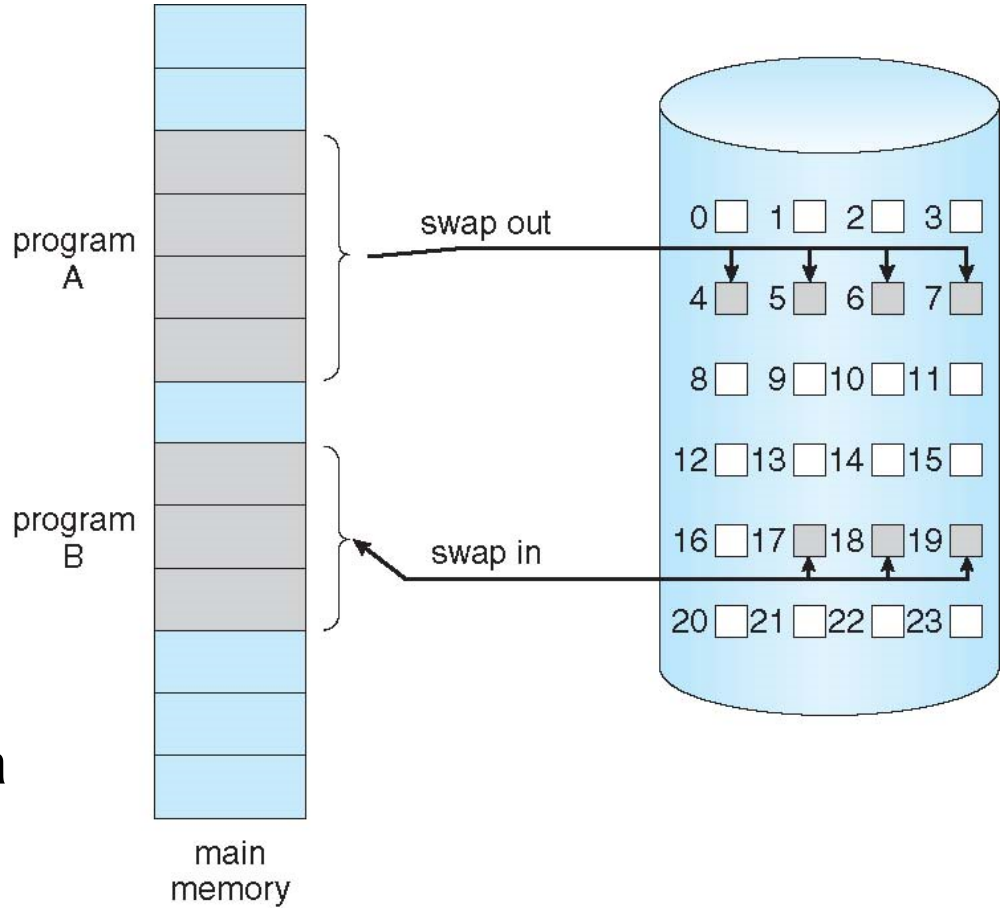
- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Virtual Memory That is Larger Than Physical Memory

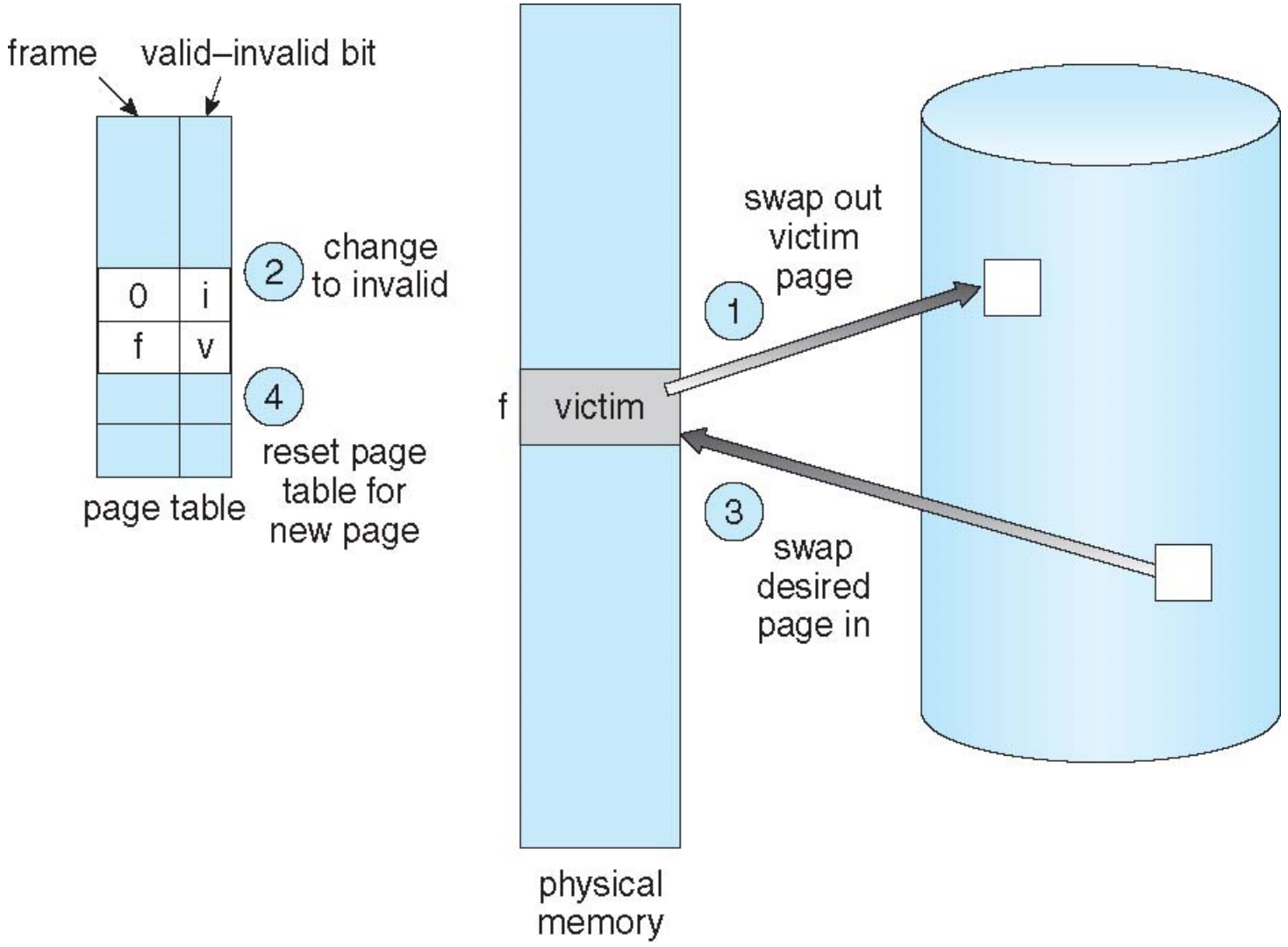


Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed



Page Replacement



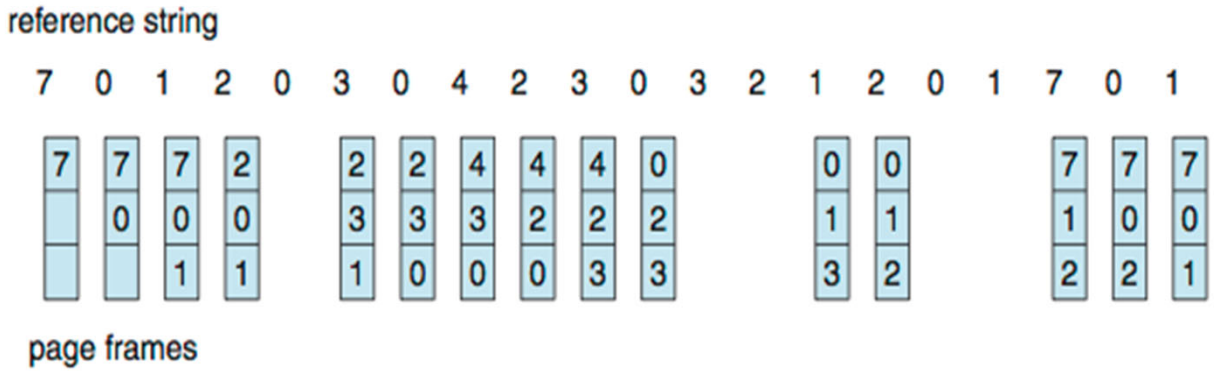
Page Replacement Algorithms

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

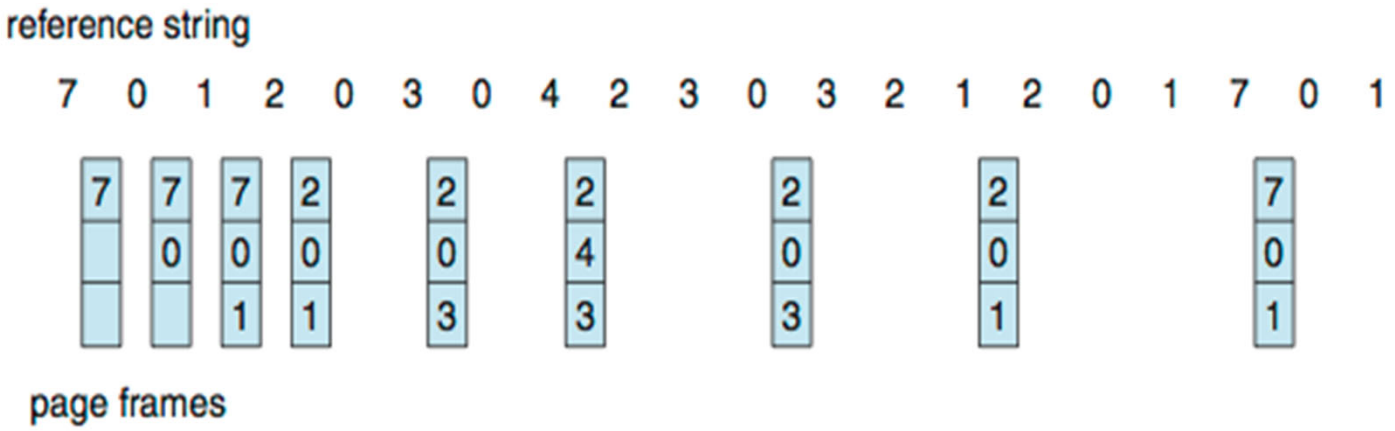


15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

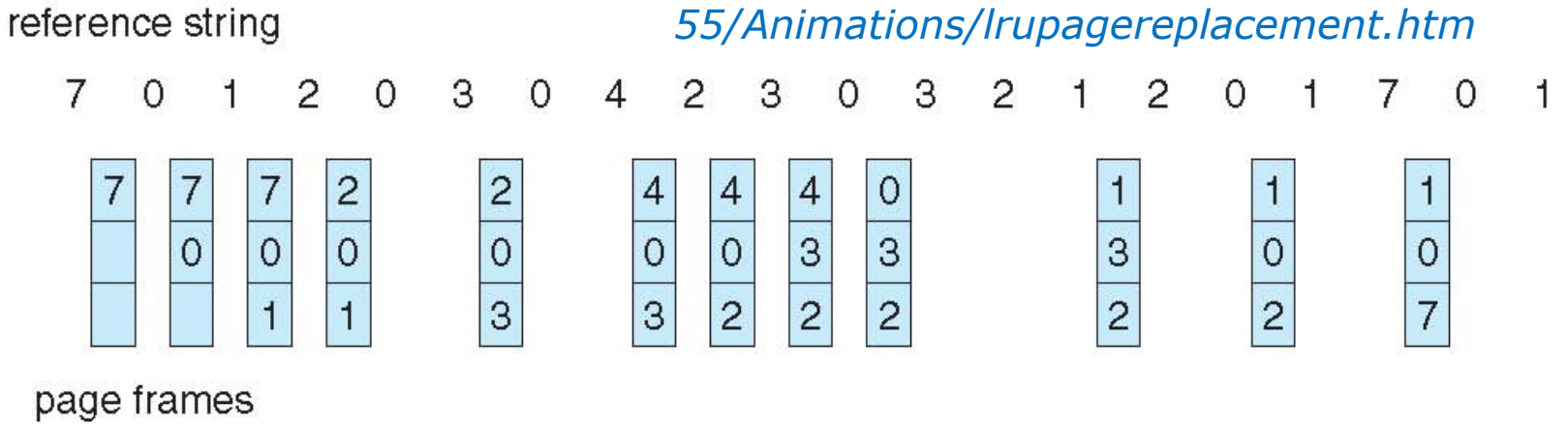


Move forward along the reference string, whenever a number on the string matches a number in the memory, mark the number as **stay in the memory** until only one number in the memory is not mark, then this number is the victim

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/lrupagereplacement.htm



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

Move backward along the reference string, whenever a number on the string matches a number in the memory, mark the number as **stay in the memory** until only one number in the memory is not mark, then this number is the victim

Exercise

1,2,3,4,1,2,5,1,2,3,4,5

Calculate the number of page faults using FIFO, optimal, and LRU algorithms

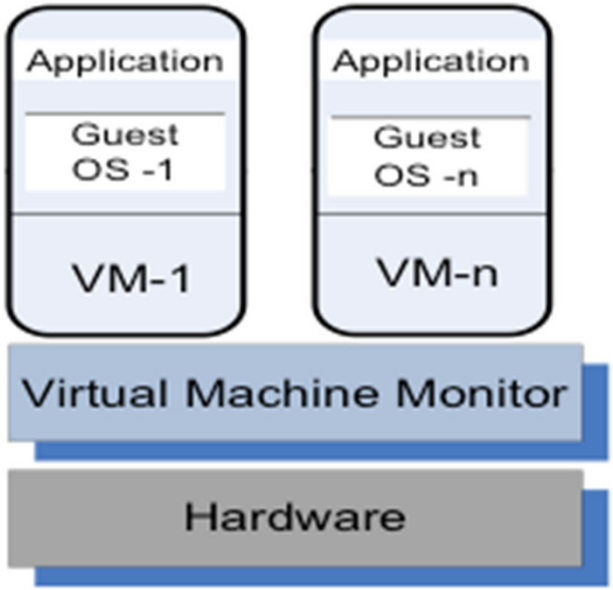
History of Virtualization

(from "Modern Operating Systems" 4th Edition, p474 by Tanenbaum and Bos)

- **1960's, IBM: CP/CMS** control program: a virtual machine operating system for the IBM System/360 Model 67
- **2000, IBM: z-series** with 64-bit virtual address spaces and backward compatible with the System/360
- **1974: Popek and Golberg** from UCLA published *"Formal Requirements for Virtualizable Third Generation Architectures"* where they listed the conditions a computer architecture should satisfy to support virtualization efficiently. The popular x86 architecture that originated in the 1970s did not support these requirements for decades.
- **1990's, Stanford researchers, VMware:** Researchers developed a new hypervisor and founded VMware, the biggest virtualization company of today's. First virtualization solution was is 1999 for x86.
- Today many virtualization solutions: Xen from Cambridge, KVM, Hyper-V, …
- IBM was the first to produce and sell virtualization for the mainframe. But, VMware popularised virtualization for the masses.

Virtual Machine Monitor (VMM / Hypervisor)

- A **virtual machine monitor (VMM/hypervisor)** partitions the resources of computer system into one or more **virtual machines (VMs)**. Allows several operating systems to run concurrently on a single hardware platform
- A VM is an execution environment that runs an OS
- VM – an isolated environment that appears to be a whole computer, but actually only has access to a portion of the computer resources
- A VMM allows:
  - Multiple services to share the same platform
  - Live migration - the movement of a server from one platform to another
  - System modification while maintaining backward compatibility with the original system
  - Enforces isolation among the systems, thus security



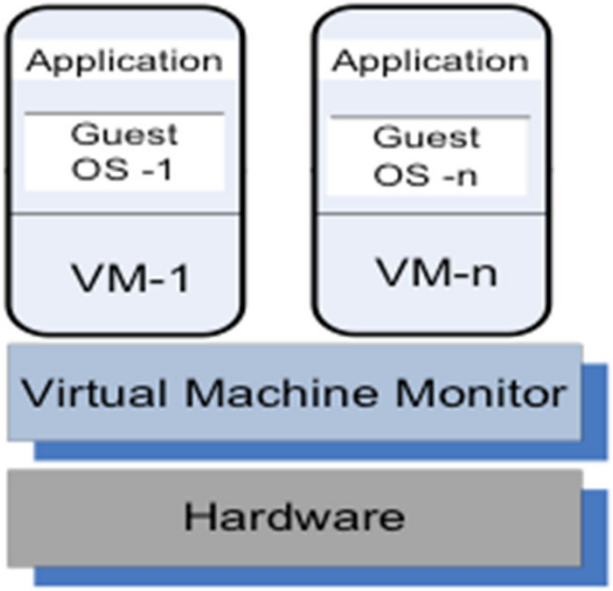
- A **guest operating system** is an OS that runs in a VM under the control of the VMM.

VMM Virtualizes the CPU and the Memory

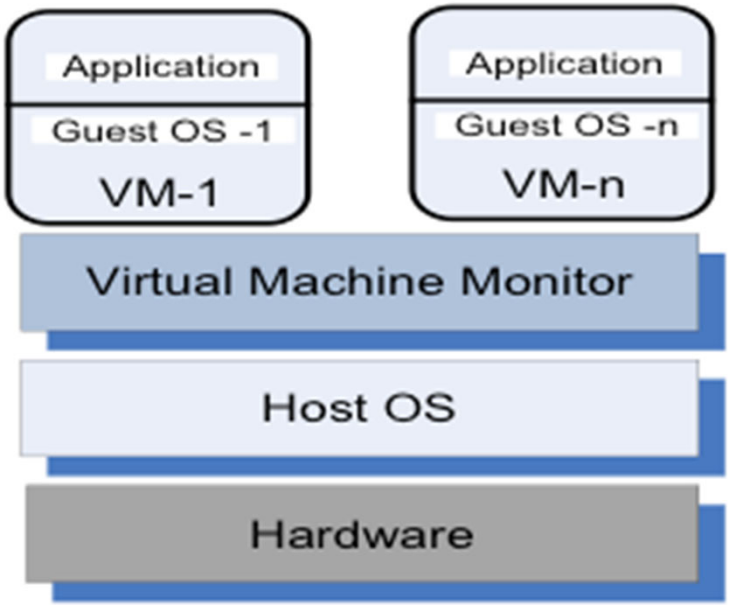
- A VMM (also hypervisor) (howto):
  - Traps the privileged instructions executed by a guest OS and enforces the correctness and safety of the operation
  - Traps interrupts and dispatches them to the individual guest operating systems
  - Controls the virtual memory management
  - Maintains a shadow page table for each guest OS and replicates any modification made by the guest OS in its own shadow page table. This shadow page table points to the actual page frame and it is used by the Memory Management Unit (MMU) for dynamic address translation.
  - Monitors the system performance and takes corrective actions to avoid performance degradation. For example, the VMM may swap out a VM to avoid thrashing (page fault rate is too high).

Type 1 and 2 Hypervisors

Type 1 Hypervisor



Type 2 Hypervisor



- Taxonomy of VMMs:
  1. Type 1 Hypervisor (bare metal, native): supports multiple virtual machines and runs directly on the hardware (e.g., VMware ESX , Xen, Denali)
  2. Type 2 Hypervisor (hosted) VM - runs under a host operating system (e.g., user-mode Linux)

Examples of Hypervisors

Name	Host ISA	Guest ISA	Host OS	guest OS	Company
Integrity VM	*x86*-64	*x86*-64	HP-Unix	Linux,Windows HP Unix	HP
Power VM	Power	Power	No host OS	Linux, AIX	IBM
z/VM	z-ISA	z-ISA	No host OS	Linux on z-ISA	IBM
Lynx Secure	*x86*	*x86*	No host OS	Linux, Windows	LinuxWorks
Hyper-V Server	*x86*-64	*x86*-64	Windows	Windows	Microsoft
Oracle VM	*x86*, *x86*-64	*x86*, *x86*-64	No host OS	Linux, Windows	Oracle
RTS Hypervisor	*x86*	*x86*	No host OS	Linux, Windows	Real Time Systems
SUN xVM	*x86*, SPARC	same as host	No host OS	Linux, Windows	SUN
VMware EX Server	*x86*, *x86*-64	*x86*, *x86*-64	No host OS	Linux, Windows Solaris, FreeBSD	VMware
VMware Fusion	*x86*, *x86*-64	*x86*, *x86*-64	MAC OS *x86*	Linux, Windows Solaris, FreeBSD	VMware
VMware Server	*x86*, *x86*-64	*x86*, *x86*-64	Linux, Windows	Linux, Windows Solaris, FreeBSD	VMware
VMware Workstation	*x86*, *x86*-64	*x86*, *x86*-64	Linux, Windows	Linux, Windows Solaris, FreeBSD	VMware
VMware Player	*x86*, *x86*-64	*x86*, *x86*-64	Linux Windows	Linux, Windows Solaris, FreeBSD	VMware
Denali	*x86*	*x86*	Denali	ILVACO, NetBSD	University of Washington
Xen	*x86*, *x86*-64	*x86*, *x86*-64	Linux Solaris	Linux, Solaris NetBSD	University of Cambridge

Performance and Security Isolation

- The run-time behavior of an application is affected by other applications running concurrently on the same platform and competing for CPU cycles, cache, main memory, disk and network access. Thus, it is difficult to predict the completion time!

- Performance isolation - a critical condition for QoS guarantees in shared computing environments

- A VMM is a much simpler and better specified system than a traditional operating system. Example - Xen has approximately 60,000 lines of code; Denali has only about half: 30,000

- The security vulnerability of VMMs is considerably reduced as the systems expose a much smaller number of privileged functions. For example, Xen VMM has 28 hypercalls while Linux has 100s of system calls

Conditions for Efficient Virtualization (from Popek and Goldberg):

- Conditions for efficient virtualization (from Popek and Goldberg):
  1. A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
  2. The VMM should be in complete control of the virtualized resources.
  3. A statistically significant fraction of machine instructions must be executed without the intervention of the VMM.

Challenges of x86 CPU Virtualization

- Four layers of privilege execution → rings
  - User applications run in ring 3
  - OS runs in ring 0
- In which ring should the VMM run?
  - In ring 0, then, same privileges as an OS → wrong
  - In rings 1,2,3, then OS has higher privileges → wrong
  - Move the OS to ring 1 and the VMM in ring 0 → OK
- Three classes of machine instructions:
  1. **privileged instructions** can be executed in kernel mode. When attempted to be executed in user mode, they cause a *trap* and so executed in kernel mode.
  2. **nonprivileged instructions** the ones that can be executed in user mode
  3. **sensitive instructions** can be executed in either kernel or user but they behave differently. Sensitive instructions require special precautions at execution time.
  4. Instructions that are both sensitive and nonprivileged are hard to virtualize

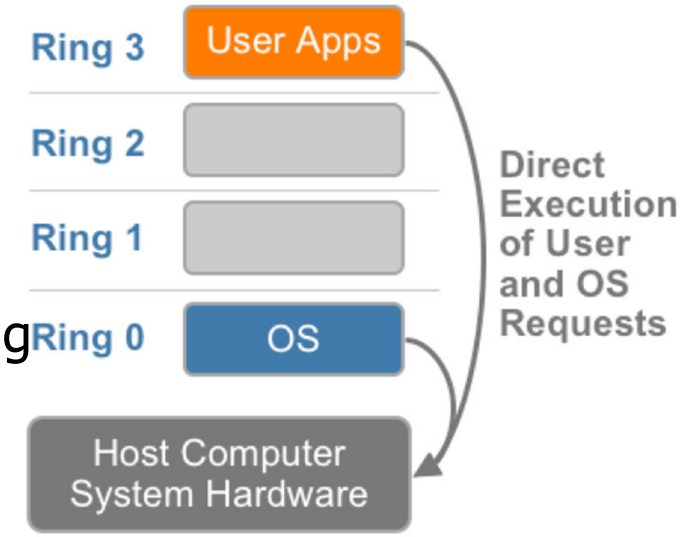


Figure 4 – x86 privilege level architecture without virtualization

Techniques for Virtualizing CPU on x86

1. **Full virtualization with binary translation**
2. **OS-assisted Virtualization or Paravirtualization**
3. **Hardware assisted virtualization**

Techniques for Virtualizing CPU on x86

Full virtualization – a guest OS can run unchanged under the VMM as if it was running directly on the hardware platform. Each VM runs an exact copy of the actual hardware.

- **Binary translation** rewrites parts of the code on the fly to replace sensitive but not privileged instructions with safe code to emulate the original instruction
- "*The hypervisor translates all operating system instructions on the fly and caches the results for future use, while user level instructions run unmodified at native speed.*" (from VMware paper)
- Examples: VMware, Microsoft Virtual Server
- Advantages:
  - No hardware assistance,
  - No modifications of the guest OS
  - Isolation, Security
- Disadvantages:
  - Speed of execution

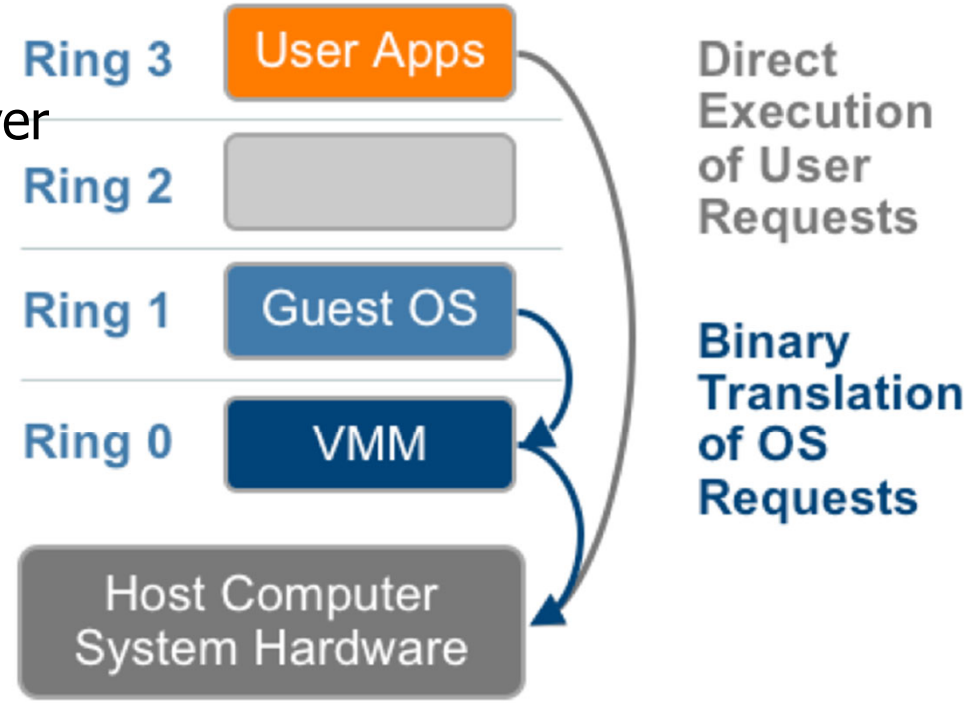


Figure 5 – The binary translation approach to x86 virtualization

Techniques for Virtualizing CPU on x86

Paravirtualization – "*involves modifying the OS kernel to replace non-virtualizable instructions with hypercalls that communicate directly with the virtualization layer hypervisor. The hypervisor also provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping.*" (from VMware paper)

- Advantage: faster execution, lower virtualization overhead
- Disadvantage: poor portability
- Examples: Xen, Denali

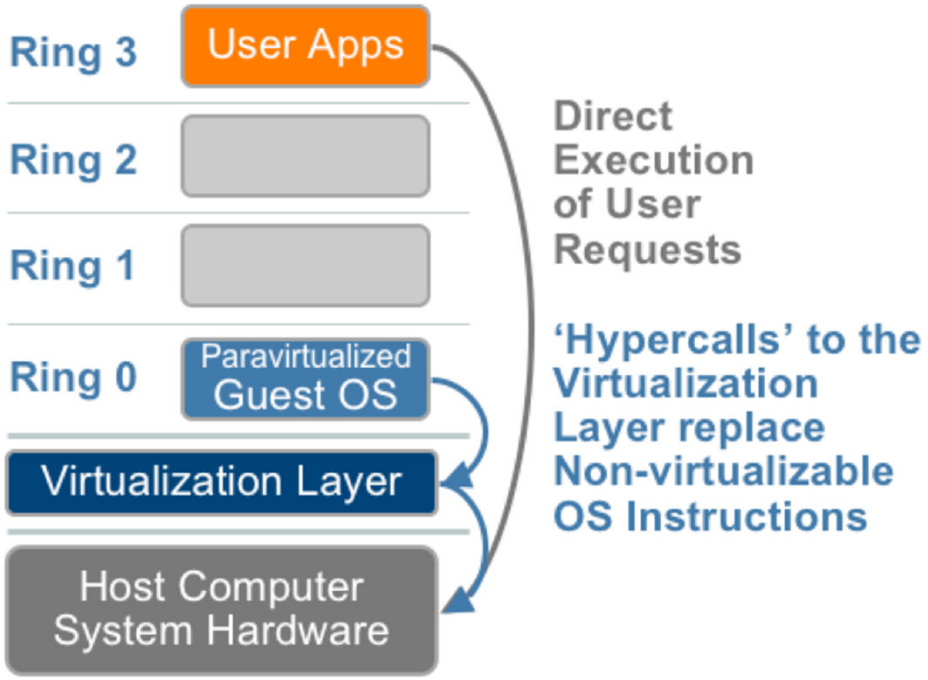
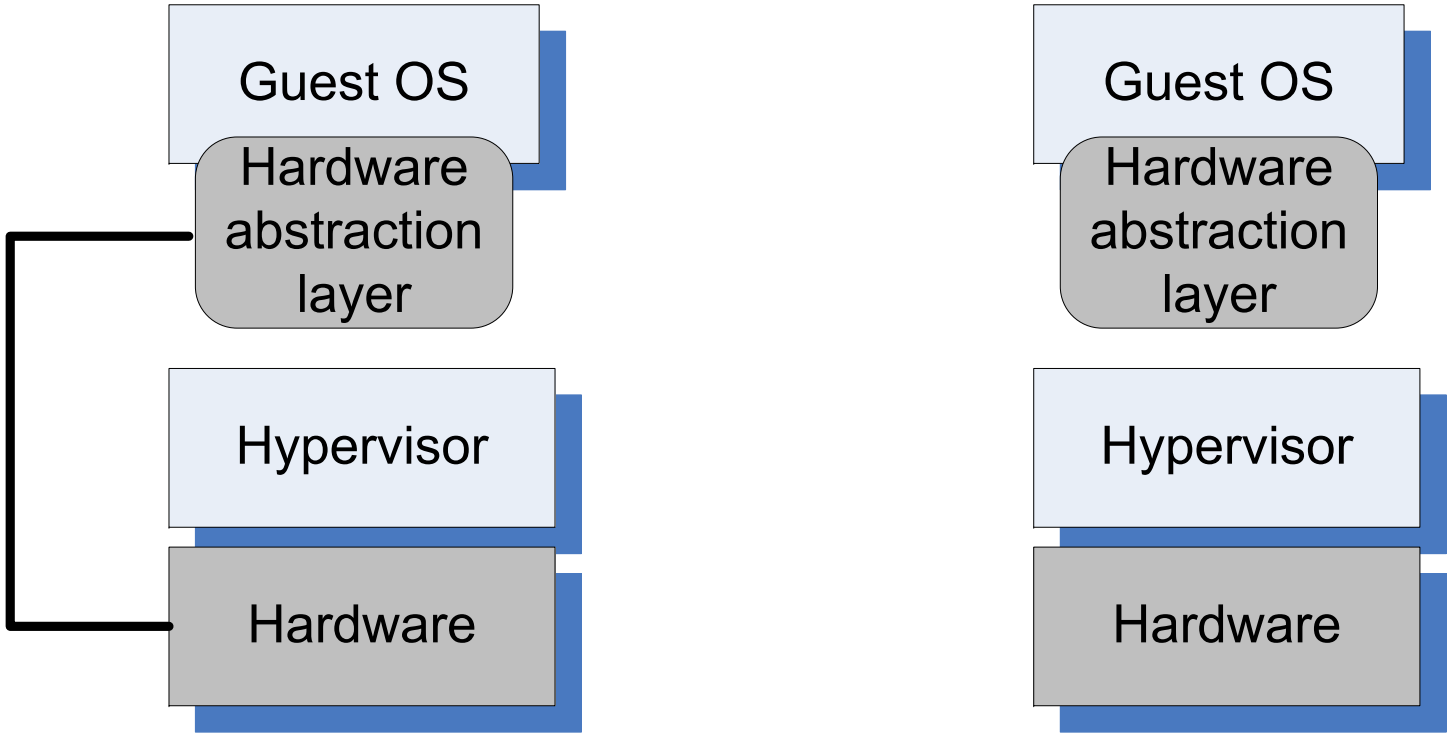


Figure 6 – The Paravirtualization approach to x86 Virtualization

Full Virtualization and Paravirtualization



(a) Full virtualization

(b) Paravirtualization

Techniques for Virtualizing CPU on x86

- **Hardware Assisted Virtualization** – "*a new CPU execution mode feature that allows the VMM to run in a new root mode below ring 0. As depicted in Figure 7, privileged and sensitive calls are set to automatically trap to the hypervisor, removing the need for either binary translation or paravirtualization*" (from VMware paper)

- Advantage: even faster execution
- Examples: Intel VT-x, Xen 3.x

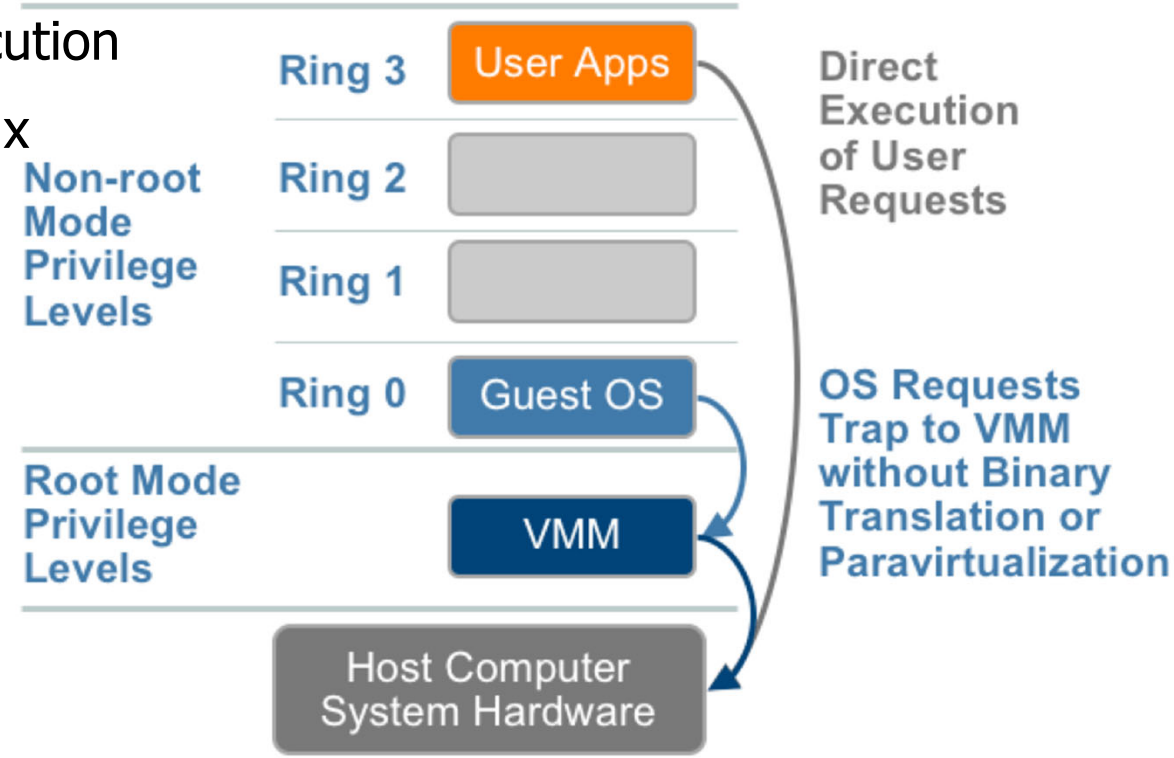
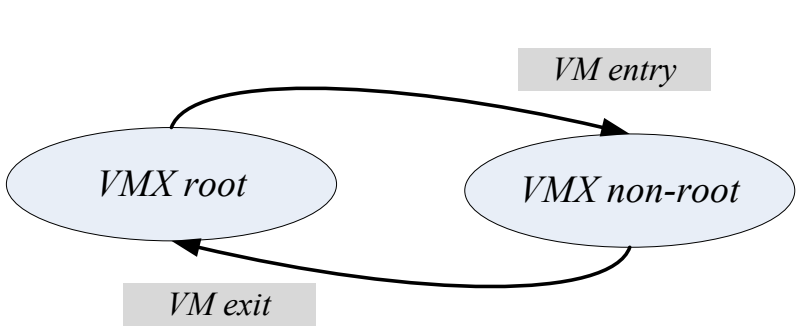


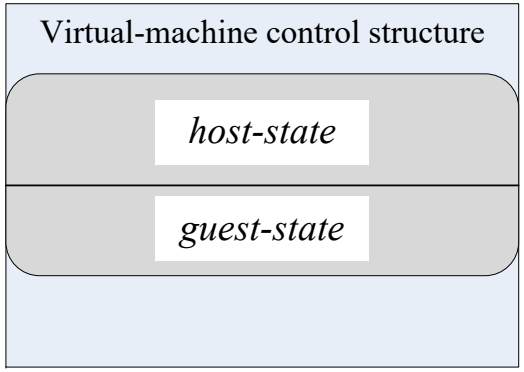
Figure 7 – The hardware assist approach to x86 virtualization

VT-x, a Major Architectural Enhancement

- In 2005 Intel released two Pentium 4 models supporting VT-x.
- VT-x supports two modes of operations (Figure (a)):
  1. VMX root - for VMM operations.
  2. VMX non-root - support a VM.
  - And a new data structure called the **Virtual Machine Control Structure** including *host-state* and *guest-state* areas (Figure (b)).
  - VM entry - the processor state is loaded from the guest-state of the VM scheduled to run; then the control is transferred from VMM to the VM.
  - VM exit - saves the processor state in the guest-state area of the running VM; then it loads the processor state from the host-state area, finally transfers control to the VMM.



(a)

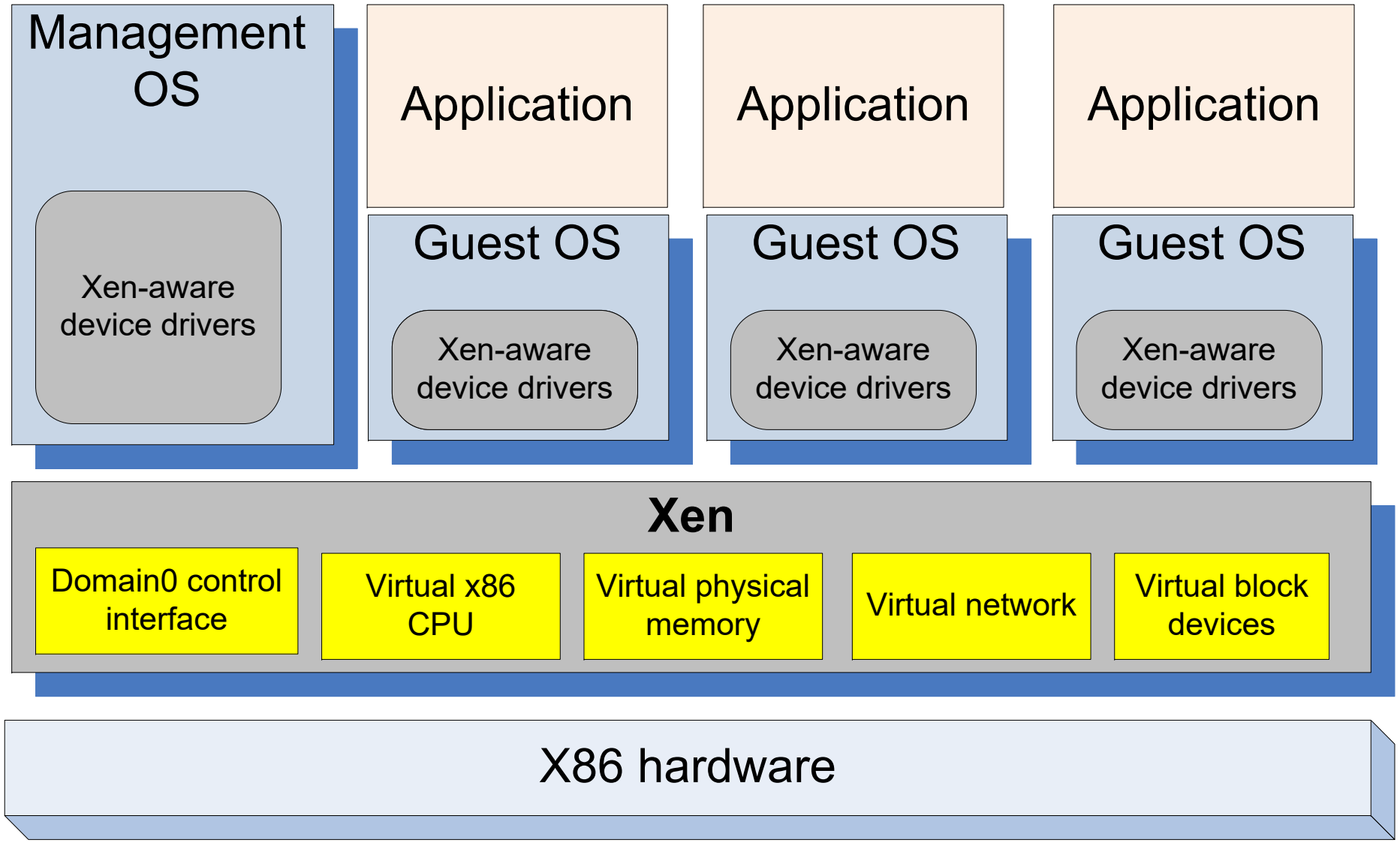


(b)

Xen - a VMM based on Paravirtualization

- The goal of the Cambridge group - design a VMM capable of scaling to about 100 VMs running standard applications and services without any modifications to the Application Binary Interface (ABI). (2003, Computing Laboratory, Cambridge University)
- Linux, Minix, NetBSD, FreeBSD and others can operate as paravirtualized Xen guest OS running on x86, x86-64, Itanium, and ARM architectures.
- Xen domain - ensemble of address spaces hosting a guest OS and applications running under the guest OS. Runs on a virtual CPU.
  - Dom0 - dedicated to execution of Xen control functions and privileged instructions.
  - DomU - a user domain.
- Applications make system calls using hypercalls processed by Xen; privileged instructions issued by a guest OS are paravirtualized and must be validated by Xen.

Xen


Management OS
Xen-aware device drivers
Application
Guest OS
Xen-aware device drivers
Application
Guest OS
Xen-aware device drivers
Application
Guest OS
Xen-aware device drivers
Xen
Domain0 control interface
Virtual x86 CPU
Virtual physical memory
Virtual network
Virtual block devices
X86 hardware

Dom0 Components

- XenStore – a Dom0 process.
  - Supports a system-wide registry and naming service.
  - Implemented as a hierarchical key-value storage.
  - A watch function informs listeners of changes of the key in storage they have subscribed to.
  - Communicates with guest VMs via shared memory using Dom0 privileges.
- Toolstack - responsible for creating, destroying, and managing the resources and privileges of VMs.
  - To create a new VM, a user provides a configuration file describing memory and CPU allocations and device configurations.
  - Toolstack parses this file and writes this information in XenStore.
  - Takes advantage of Dom0 privileges to map guest memory, to load a kernel and virtual BIOS and to set up initial communication channels with XenStore and with the virtual console when a new VM is created.

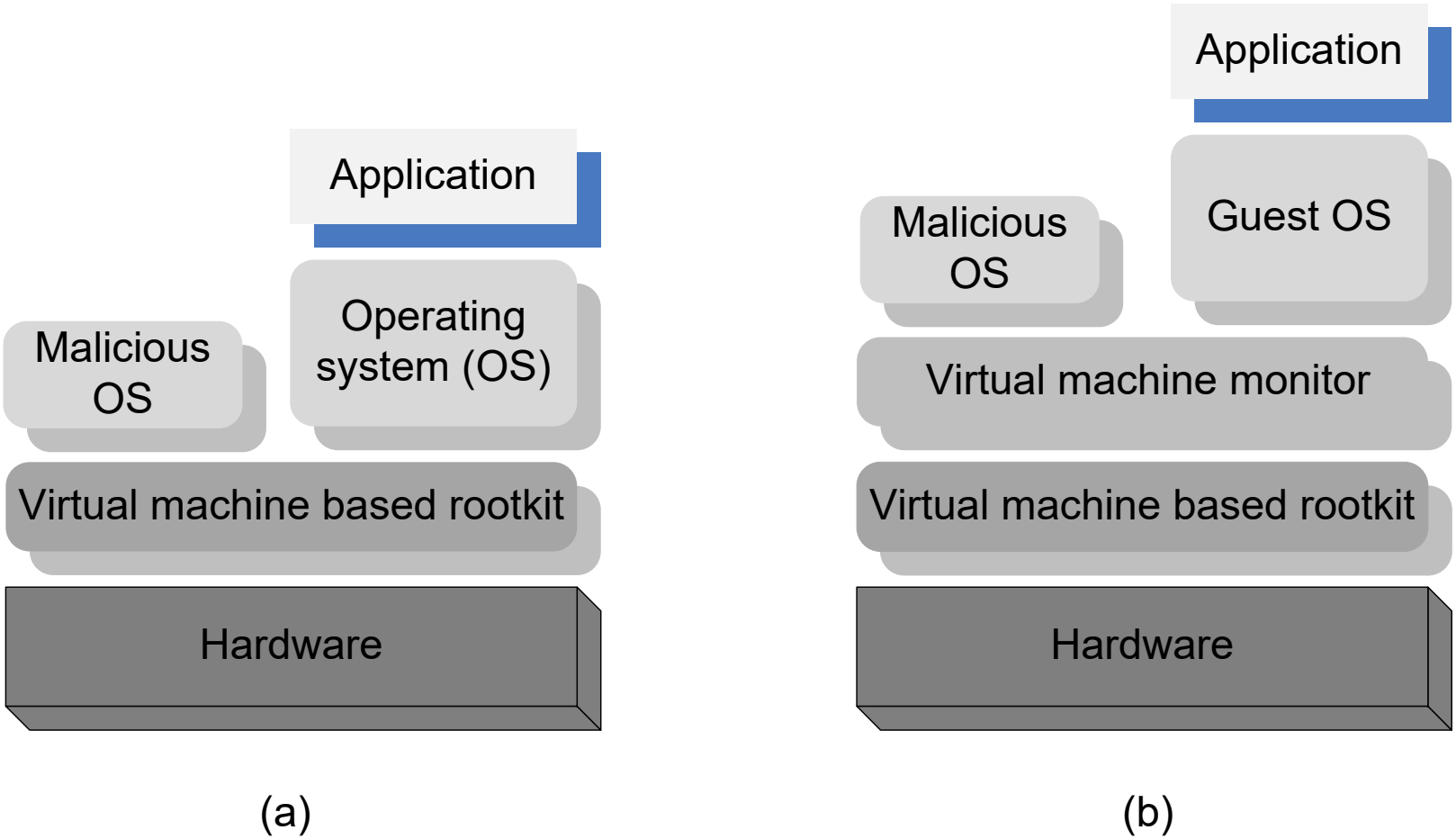
Strategies for virtual memory management, CPU multiplexing, and I/O devices

Function	Strategy
Paging	A domain may be allocated discontinuous pages. A guest OS has direct access to page tables and handles pages faults directly for efficiency; page table updates are batched for performance and validated by *Xen* for safety.
Memory	Memory is statically partitioned between domains to provide strong isolation. `XenoLinux` implements a *balloon driver* to adjust domain memory.
Protection	A guest OS runs at a lower priority level, in ring 1, while *Xen* runs in ring 0.
Exceptions	A guest OS must register with *Xen* a description table with the addresses of exception handlers previously validated; exception handlers other than the page fault handler are identical with *x86* native exception handlers.
System calls	To increase efficiency, a guest OS must install a "fast" handler to allow system calls from an application to the guest OS and avoid indirection through *Xen*.
Interrupts	A lightweight event system replaces hardware interrupts; synchronous system calls from a domain to *Xen* use *hypercalls* and notifications are delivered using the asynchronous event system.
Multiplexing	A guest OS may run multiple applications.
Time	Each guest OS has a timer interface and is aware of "real" and "virtual" time.
Network and I/O devices	Data is transferred using asynchronous I/O rings; a ring is a circular queue of descriptors allocated by a domain and accessible within *Xen*.
Disk access	Only *Dom0* has direct access to IDE and SCSI disks; all other domains access persistent storage through the Virtual Block Device (VBD) abstraction.

The Darker Side of Virtualization

- In a layered structure, a defense mechanism at some layer can be disabled by malware running at a layer below it.
- It is feasible to insert a *rogue VMM,* a Virtual-Machine Based Rootkit (VMBR) between the physical hardware and an operating system.
- Rootkit - malware with a privileged access to a system.
- The VMBR can enable a separate malicious OS to run surreptitiously and make this malicious OS invisible to the guest OS and to the application running under it.
- Under the protection of the VMBR, the malicious OS could:
  - observe the data, the events, or the state of the target system.
  - run services, such as spam relays or distributed denial-of-service attacks.
  - interfere with the application.

The Darker Side of Virtualization (con't)



The insertion of a Virtual-Machine Based Rootkit (VMBR) as the lowest layer of the software stack running on the physical hardware; (a) below an operating system; (b) below a legitimate virtual machine monitor. The VMBR enables a malicious OS to run surreptitiously and makes it invisible to the genuine or the guest OS and to the application.

Summary

- Virtualization.
- Layering and virtualization.
- Processes and dual mode execution
- Virtual memory and page replacement
- Virtual machine monitor.
- Virtual machine.
- x86 support for virtualization.
- Full and paravirtualization.
- Xen.

References

- Cambridge Uni Cloud Computing Course by Dr Eva Kalyvianaki
- Cloud Computing Theory and Practice. By Dan C. Marinescu, Elsevier Science