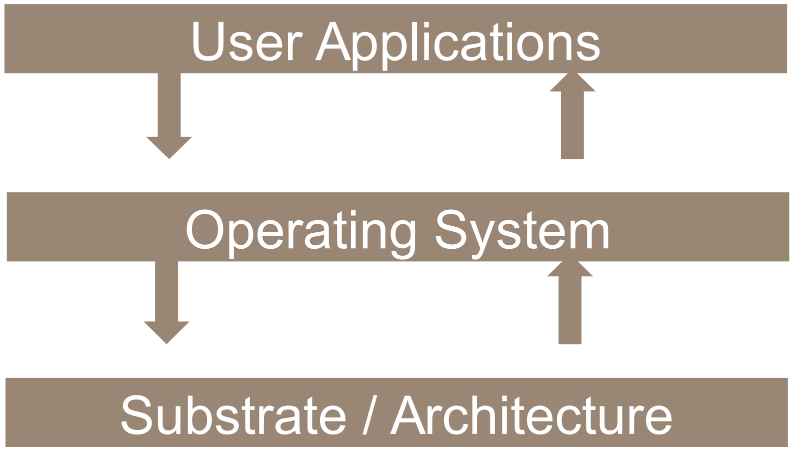
Operating System 101: The Evolution of Operating Systems

Tag: CTSS, Multics, Unix, BSD, Linux, Android, mbed

Jim Huang (黃敬群) <jserv>

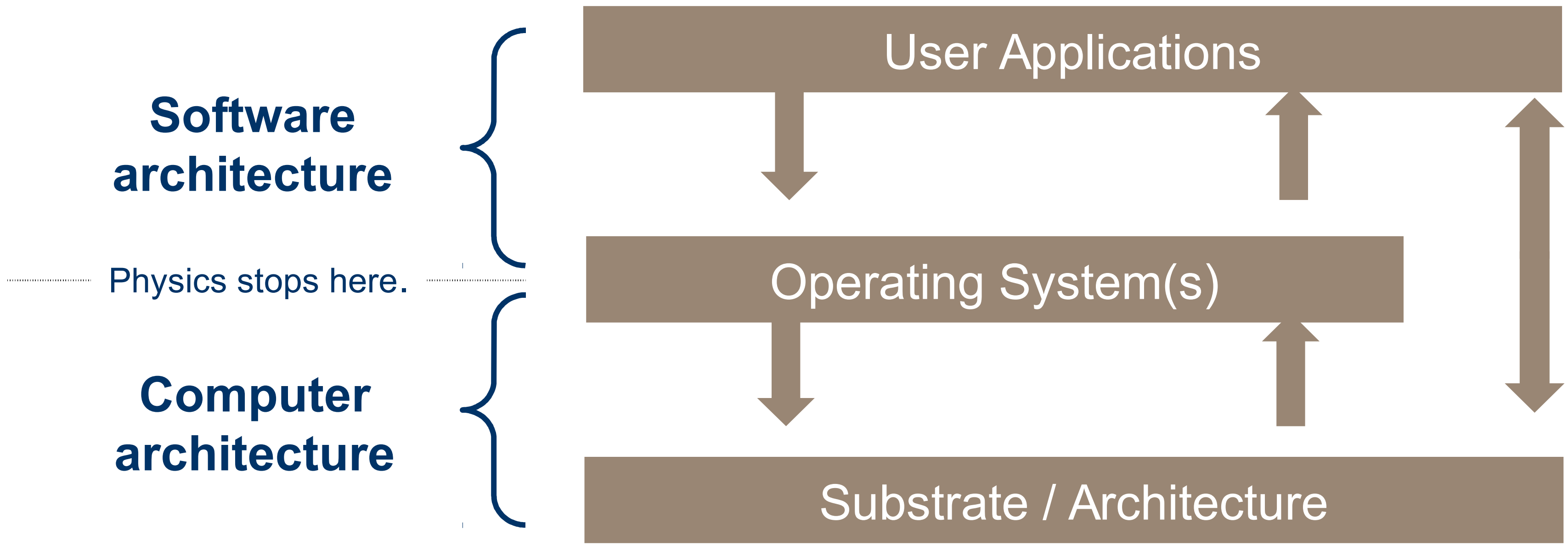
What are we about to reach?

- **Programs**
- **Platforms**
- **Performance**
- **…**



“The system is all the code your program uses that you didn’t have to write.”

“Software Architecture”



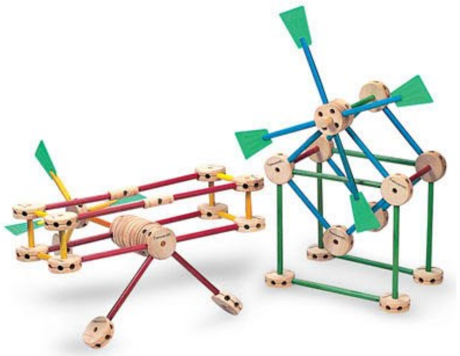
Comparative architecture: what works

Reusable / recurring design patterns

- **Used in OS**
- **Supported by OS**

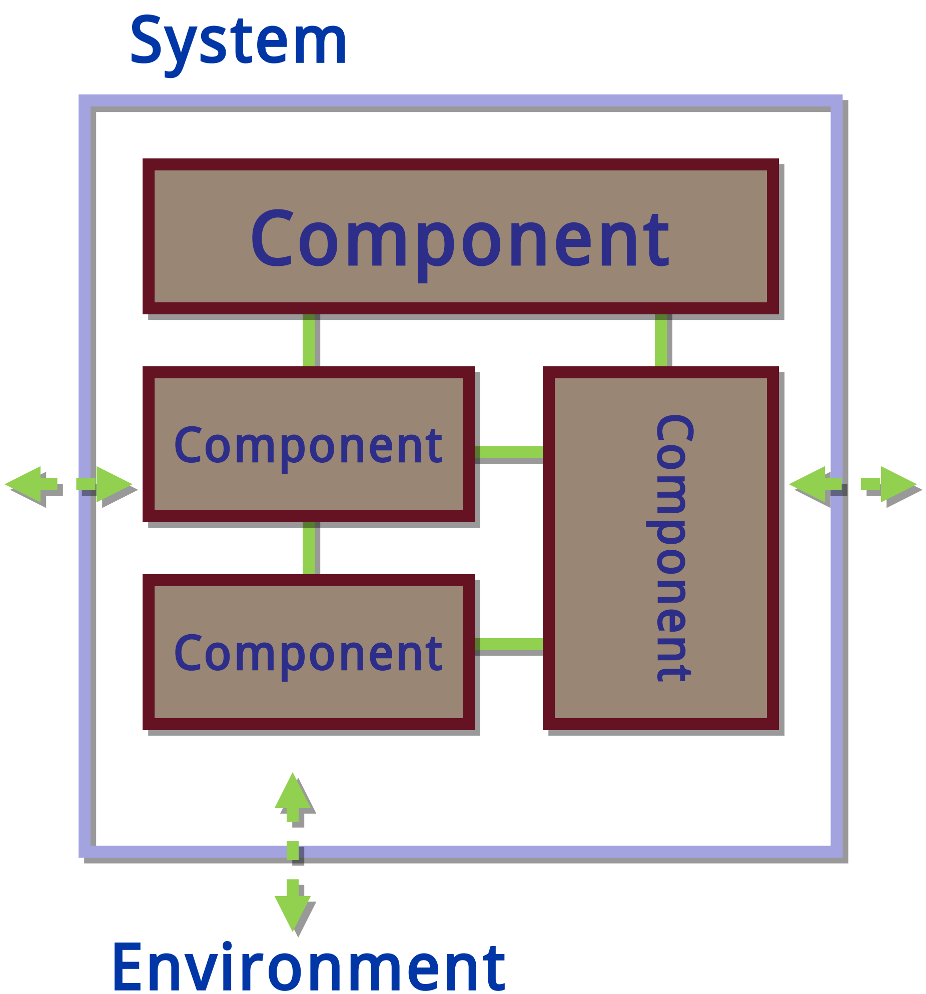
Platform abstractions

- Platforms provide “building blocks”…
- …and APIs to use them to construct software
  - Instantiate/create/allocate
  - Manipulate/configure
  - Attach/detach
  - Combine in uniform ways
  - Release/destroy
- Abstractions are layered.
  - What to expose? What to hide?



The choice of abstractions reflects a philosophy of how to build and organize software systems.

Managing Complexity



Systems are built from components.

Operating systems define styles of software components and how they interact.

OS maps components onto the underlying machine.

…and makes it all work together.

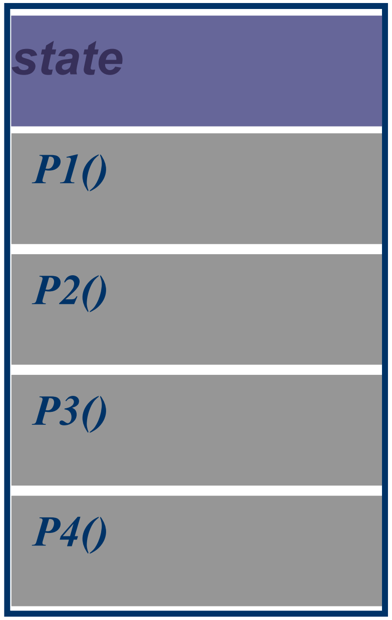
Comparative software architecture



Large, long-lived software systems are like buildings.
They are built by workers using standard design patterns.
They depend on some underlying infrastructure.
But they can evolve and are not limited by the laws of physics.

A simple module

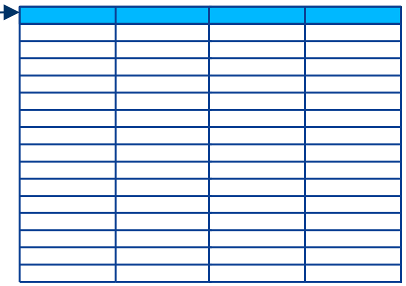
- A set of procedures/functions/methods.
- An interface (API) that defines a template for how to call/invoke the procedures.
- State (data) maintained and accessed by the procedures.
- A module may be a **class** that defines a template (type) for a data structure, which may have multiple instances (**objects**).



Abstract Data Type (ADT): the module's **state** is manipulated only through its **API** (Application Programming Interface).

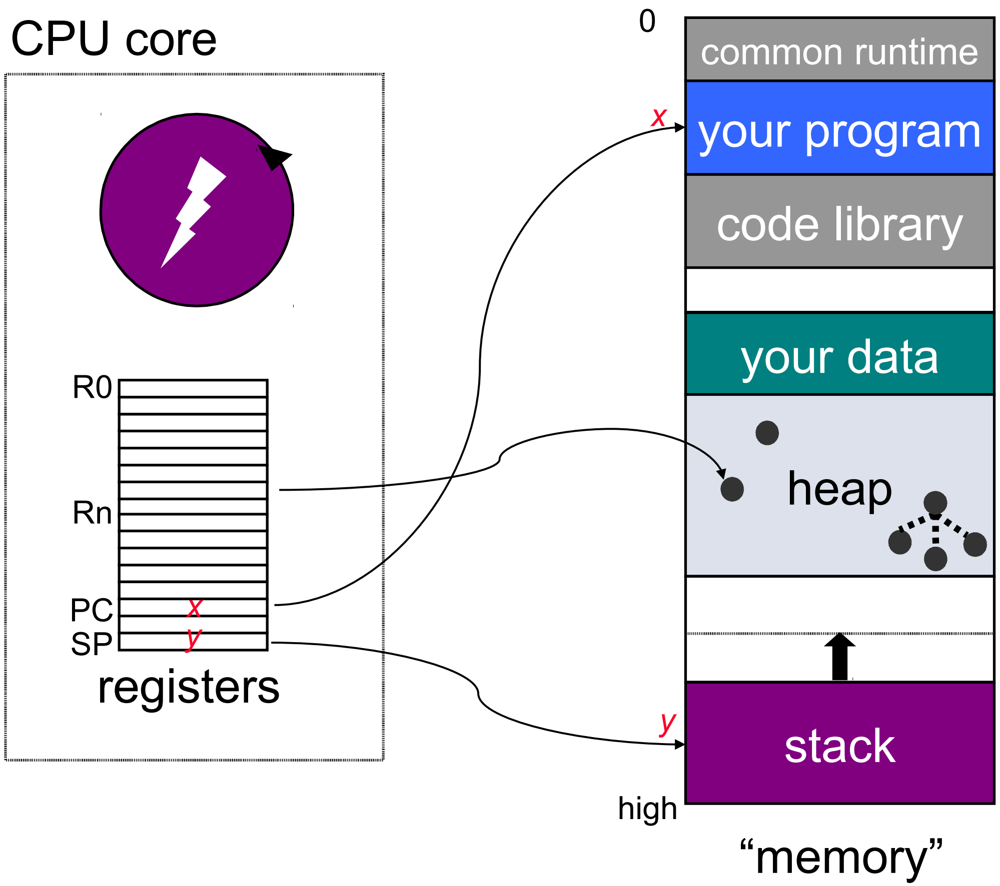
Code: instructions in memory

```
_p1:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $1, %eax
    movq    %rdi, -8(%rbp)
    popq    %rbp
    ret
```



```
load      _x, R2        ; load global variable x
add R2, 1, R2       ; increment: x = x + 1
store     R2, _x        ; store global variable x
```

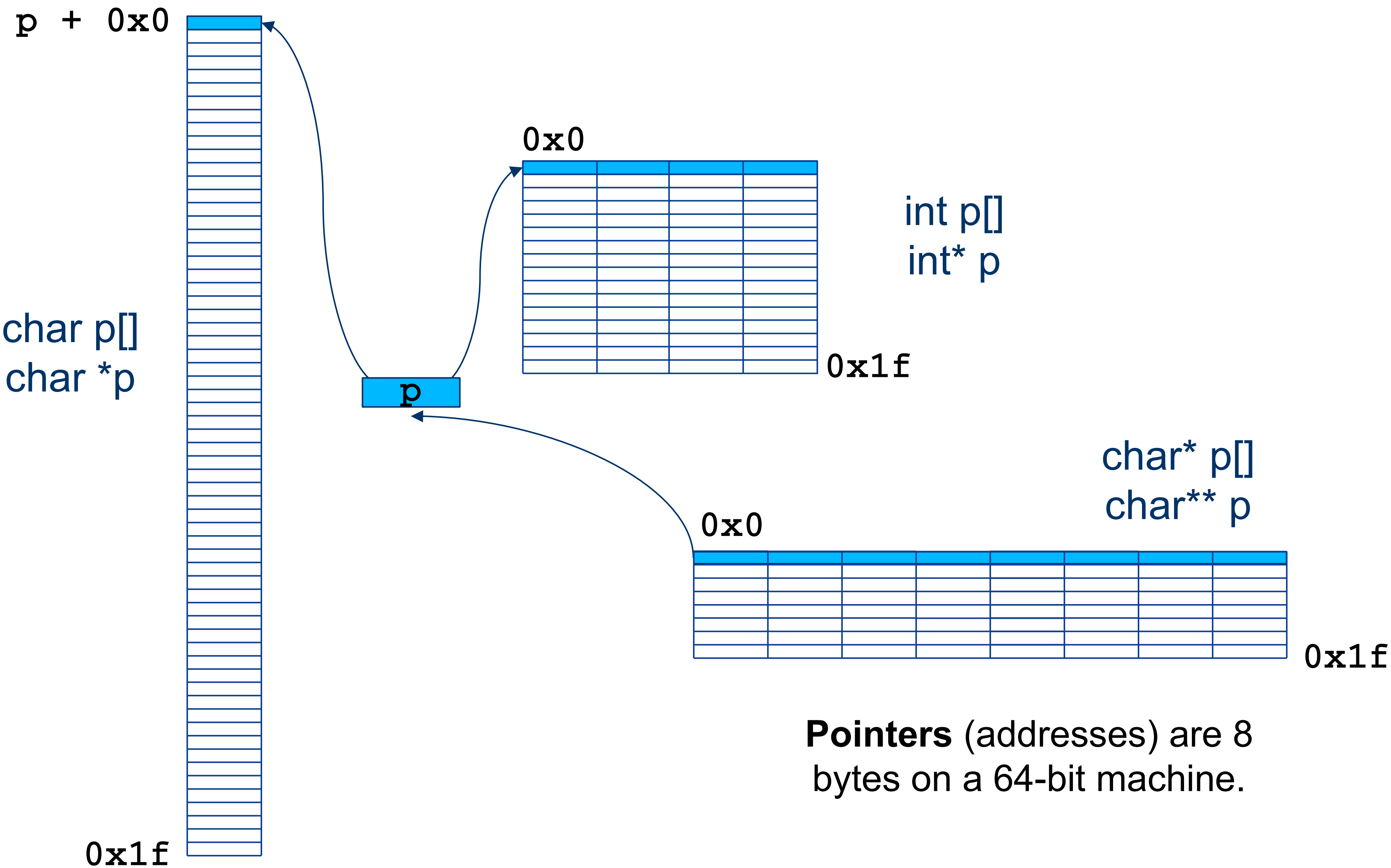
A Peek Inside a Running Program



address space
(virtual or physical)

e.g., a **virtual memory** for a running program (process)

Data in memory
64 bytes: 3 ways



Pointers (addresses) are 8 bytes on a 64-bit machine.

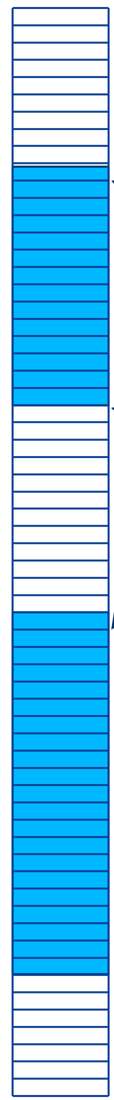
Heap: dynamic memory

The "**heap**" is an ADT in a runtime library: the code to maintain the heap is a **heap manager**.

It allocates a contiguous slab of virtual memory from the OS kernel, then "carves it up" as needed.

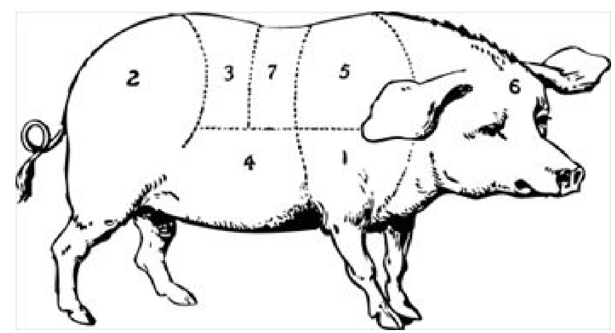
It enables the programming language environment, to store dynamic objects.

E.g., with Unix *malloc* and *free* library calls.



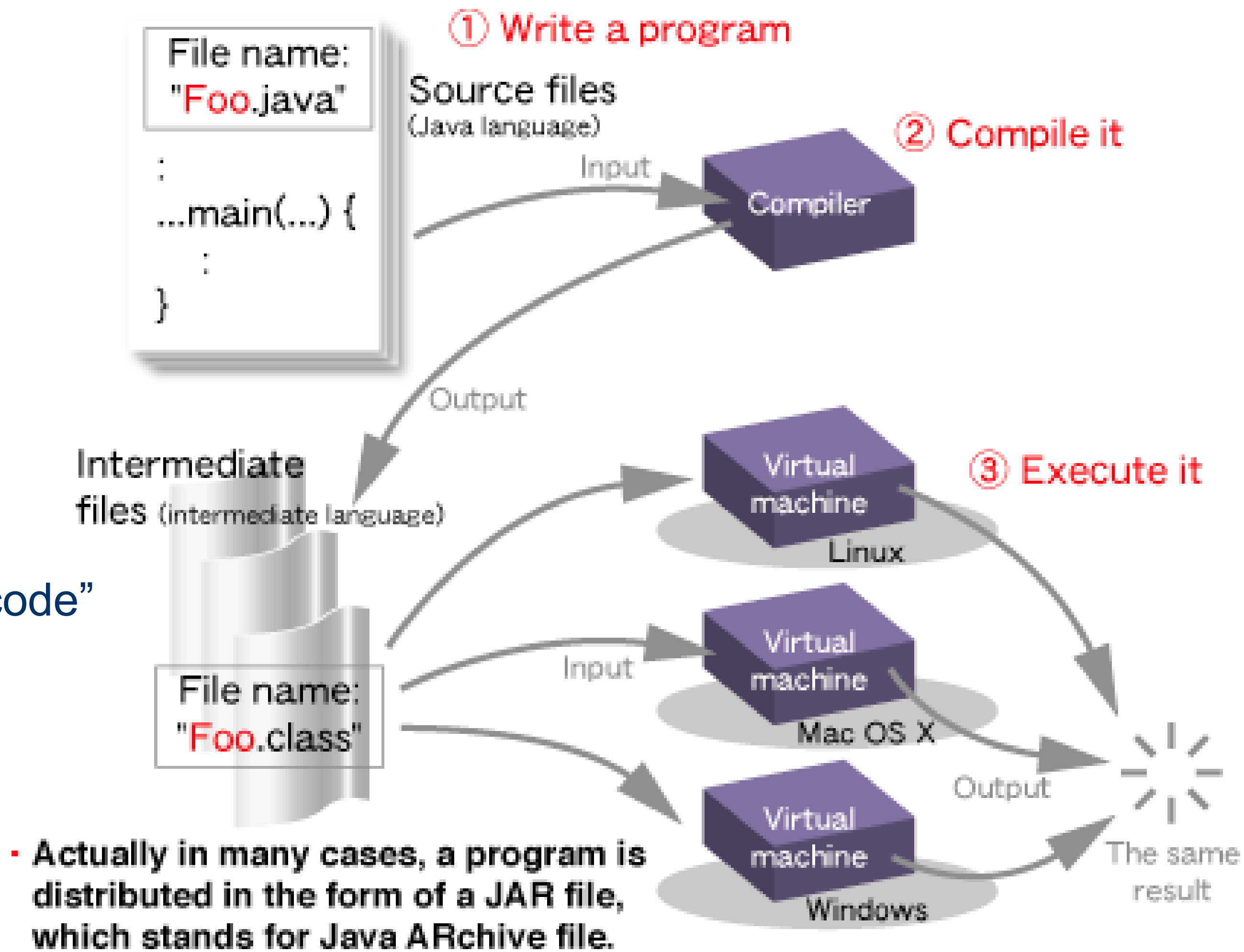
Allocated heap blocks for structs or objects. Align!

Free block



But some programs are interpreted

They run on an “abstract machine” (e.g., JVM) implemented in software.



”bytecode”

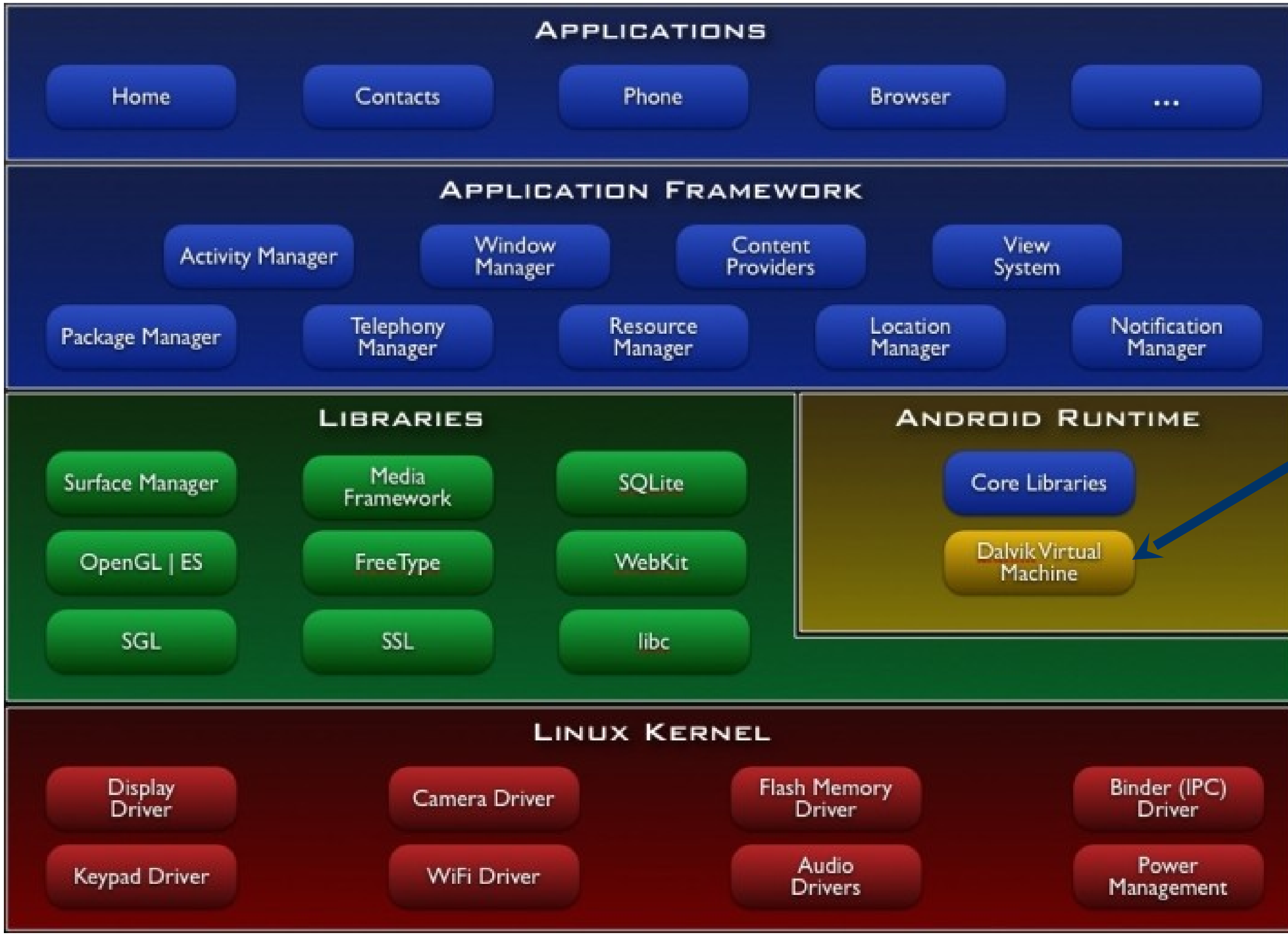
http://www.media-art-online.org/java/help/how-it-works.html

Platforms are layered/nested

Some lessons of history

- At the time it was created, Unix was the "simplest multi-user OS people could imagine."
  - It's in the name: Unix vs. Multics
- Simple abstractions can deliver a lot of power.
  - Many people have been inspired by the power of Unix.
- The community spent four decades making Unix complex again....but the essence is unchanged.
- Unix is a simple context to study core issues for classical OS design. "It's in there."
- Unix variants continue to be in wide use.

ANDROID DEVELOPER CHALLENGE



Virtual Machine

“Classical OS”

Reloaded.

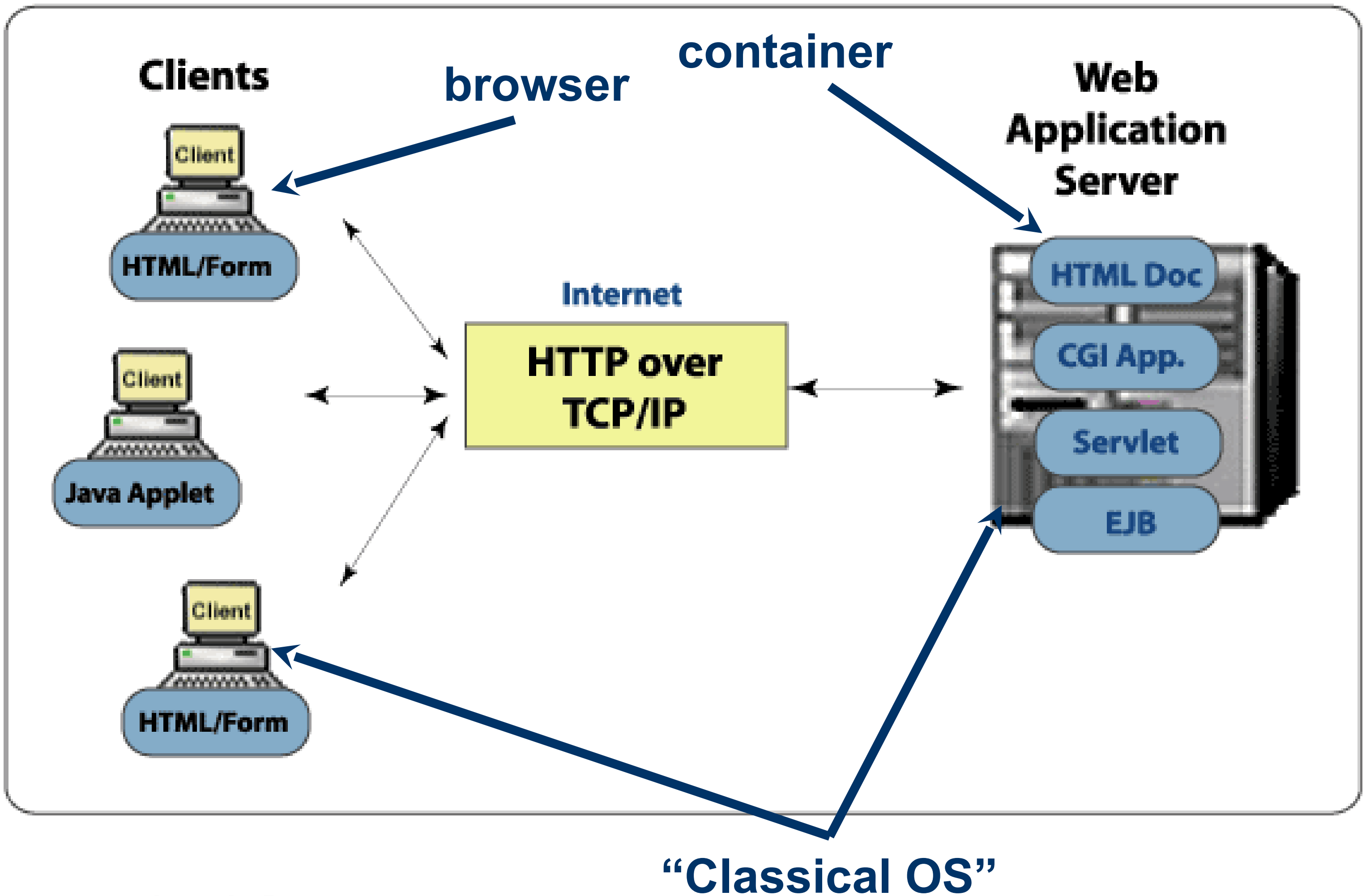
[http://www.android.com]

End-to-end application delivery

Where is your application?
Where is your data?
Where is your OS?

<u>Cloud and Software-as-a-Service (SaaS)</u>
Rapid evolution, no user upgrade, no user data management.
Agile/elastic deployment on virtual infrastructure.

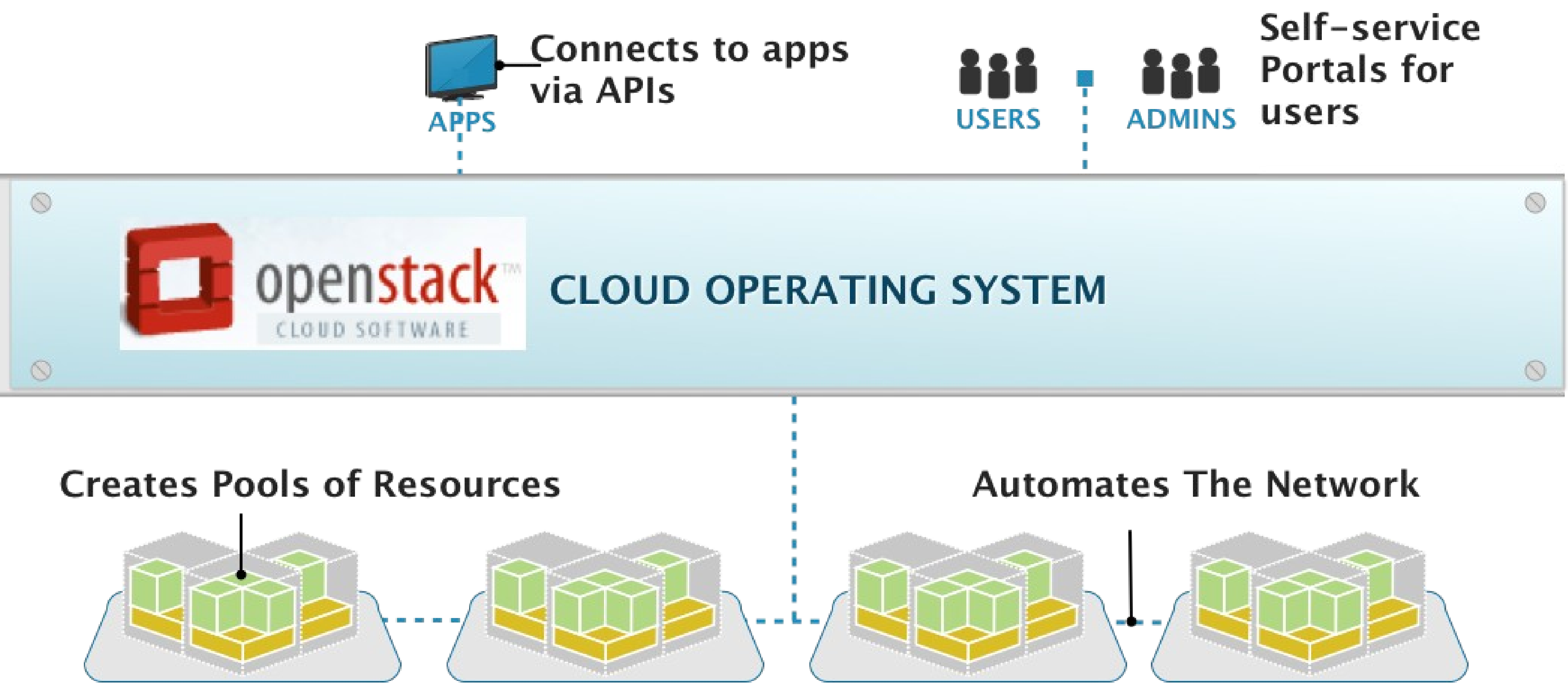
SaaS platform elements



[wiki.eeng.dcu.ie]

OpenStack, the Cloud Operating System

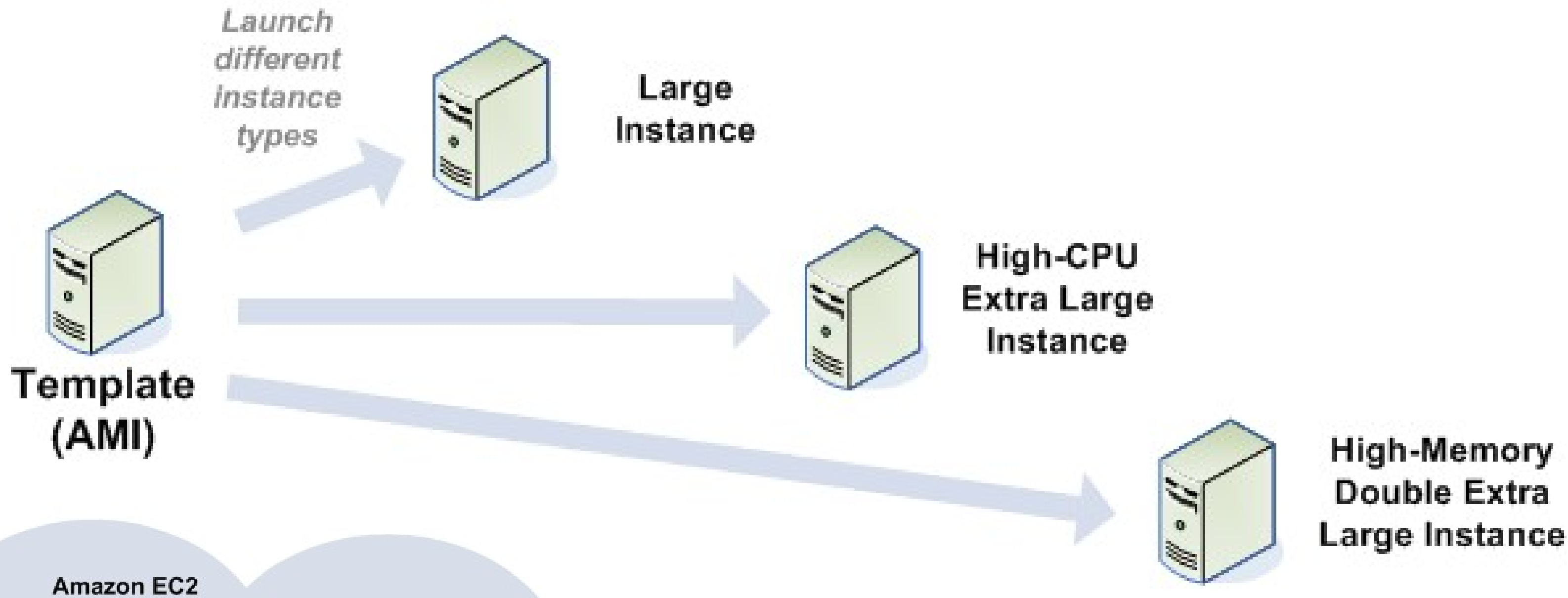
Management Layer That Adds Automation & Control

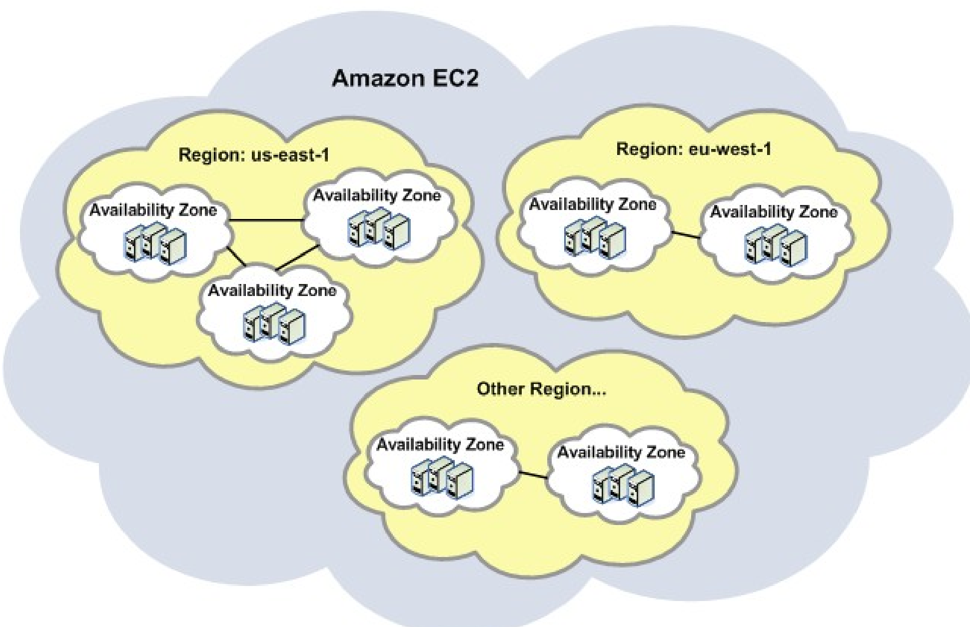


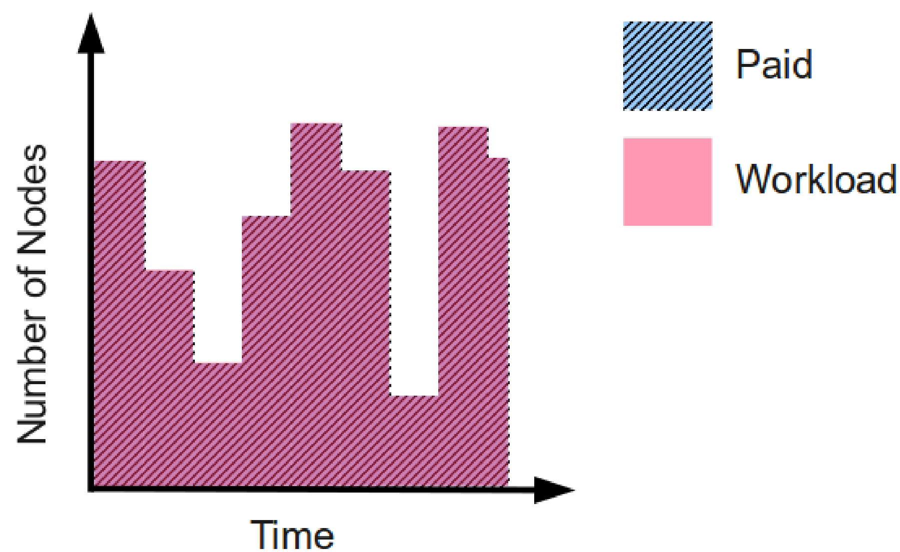
[Anthony Young @ Rackspace]

EC2
The canonical public cloud

Virtual Appliance Image



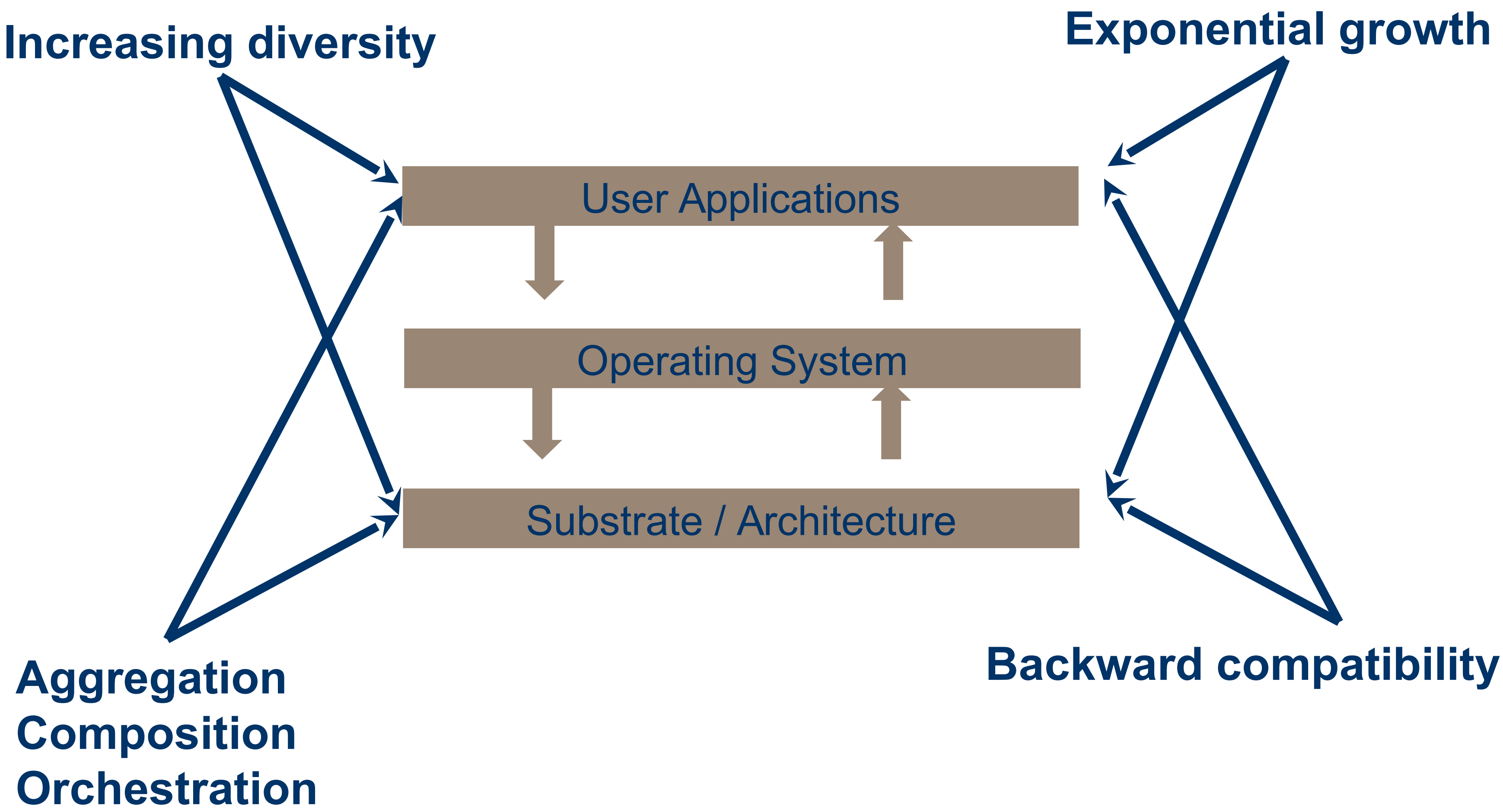




Canonical OS Example: “Classical OS”

- **Unix/Linux, Windows, Mac OS X**
- **Research systems**
  - **Multics**
  - **Mach**
  - **Minix**
  - **…**

Drivers of Change



Broad view: smartphones to servers, web, and cloud.

Key Interfaces

- Instruction set architecture (ISA)
- Application binary interface (ABI)
- Application programming interface (API)

Operating System as Software

- Functions in the same way as ordinary computer software
- Program, or suite of programs, executed by the processor
- Frequently relinquishes control and must depend on the processor to allow it to regain control

Operating System as Resource Manager

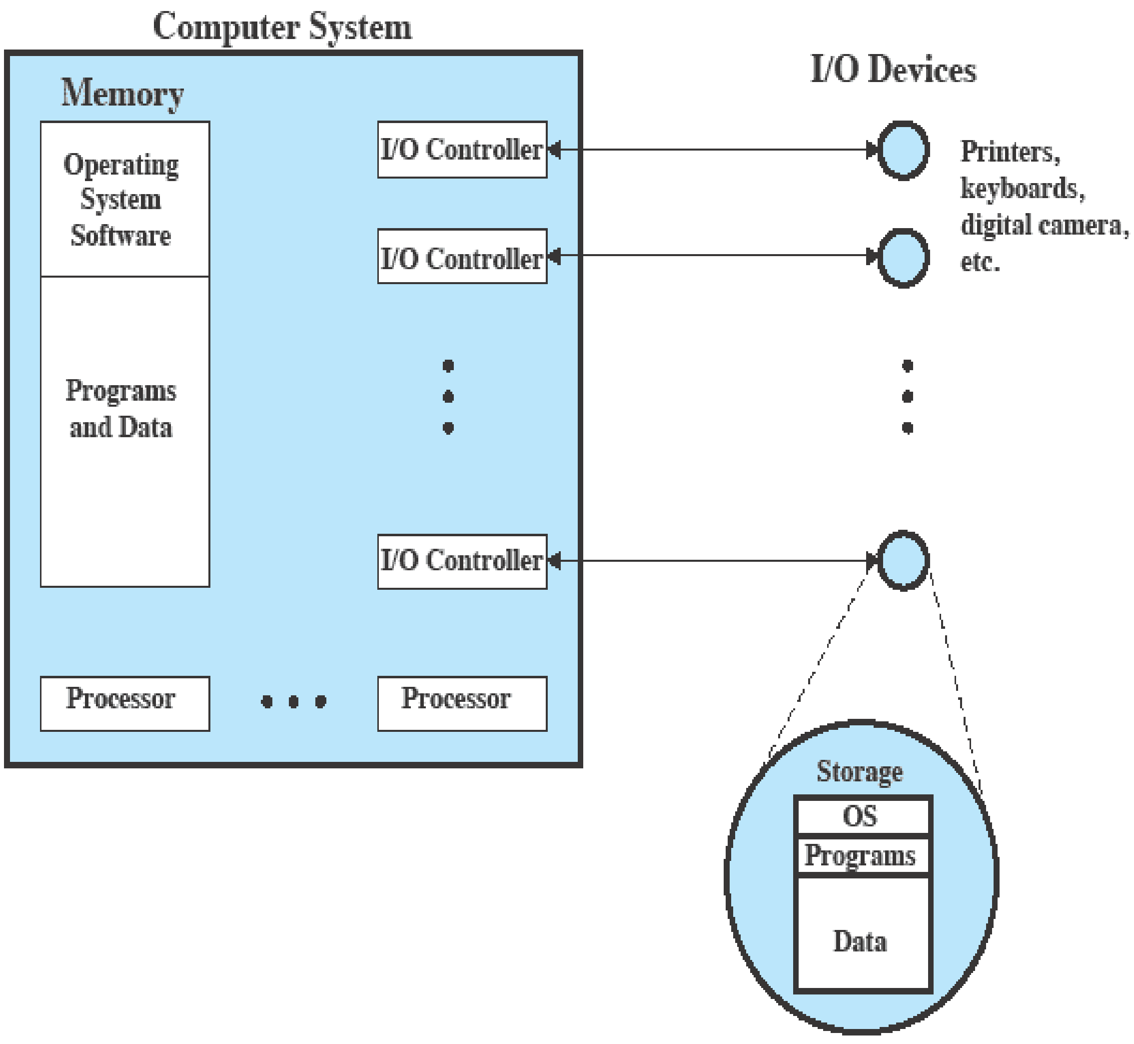


Figure 2.2 The Operating System as Resource Manager

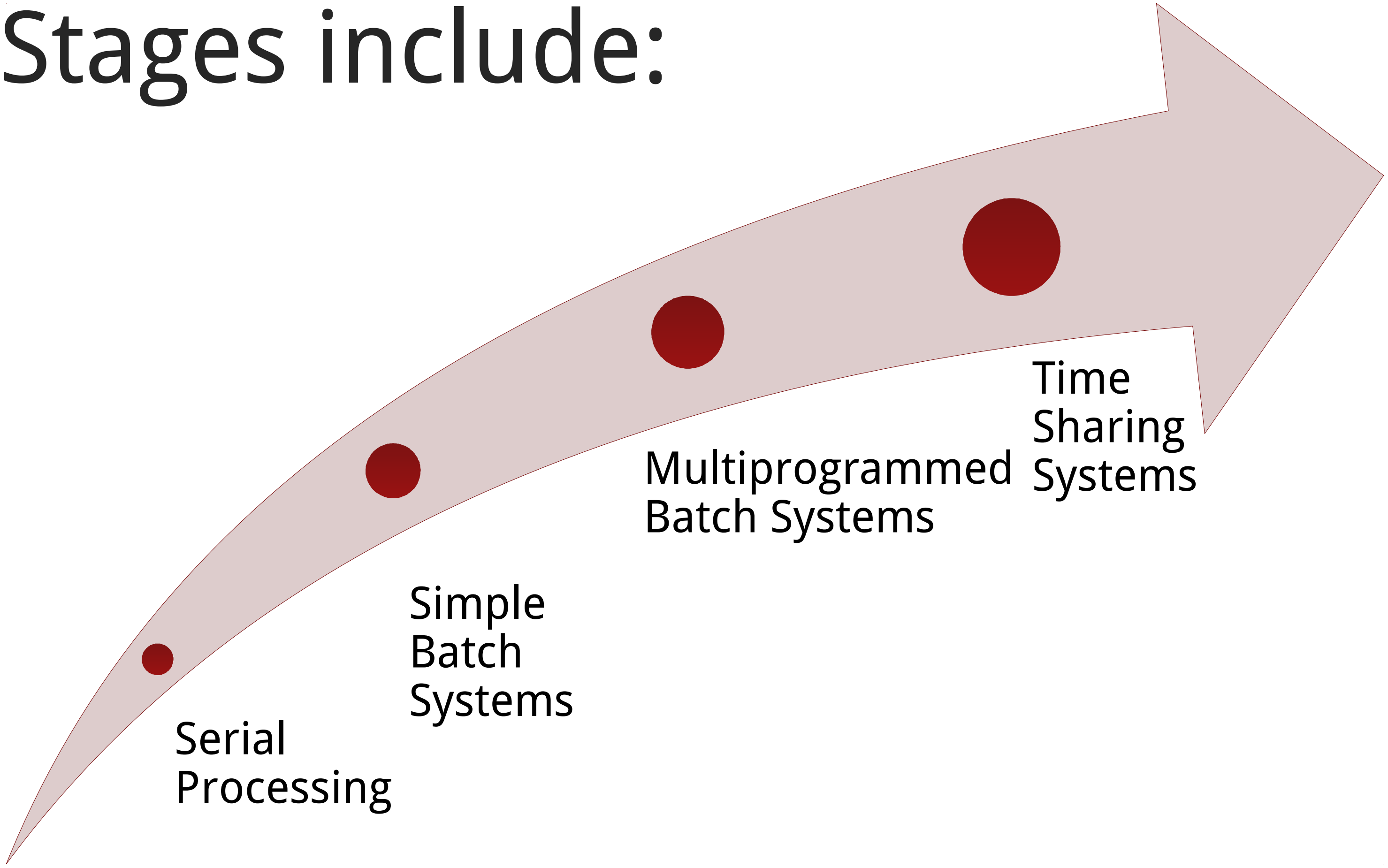
Evolution of Operating Systems

- A major OS will evolve over time for a number of reasons:

  - Hardware upgrades
  - New types of hardware
  - New services
  - Fixes

Evolution of Operating Systems

- Stages include:



Desirable Hardware Features

Memory protection for monitor

- while the user program is executing, it must not alter the memory area containing the monitor

Timer

- prevents a job from monopolizing the system

Privileged instructions

- can <u>**only**</u> be executed by the monitor

Interrupts

- gives OS more flexibility in controlling user programs

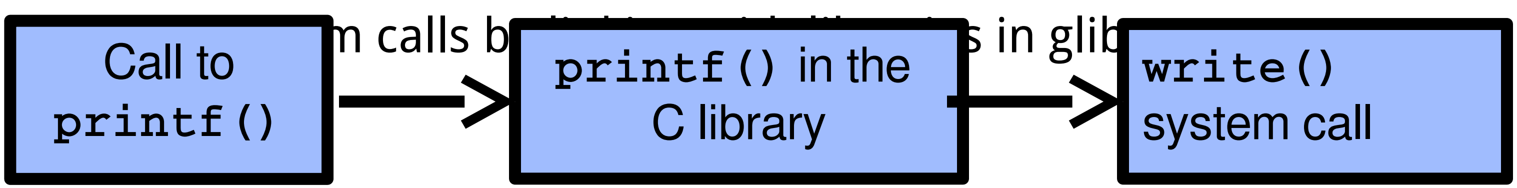
Modes of Operation

User Mode
- user program executes in user mode
- certain areas of memory are protected from user access
- certain instructions may not be executed

Kernel Mode
- monitor executes in kernel mode
- privileged instructions may be executed
- protected areas of memory may be accessed

System Call

- **Challenge: Interaction Despite Isolation**
  - How to isolate processes and their resources...
    - » While permitting them to request help from the kernel
    - » Processes interact while maintaining policies such as security, QoS, etc
  - Letting processes interact with one another in a controlled way
    - » Through messages, shared memory, etc
- **Enter the System Call interface**
  - Layer between the hardware and user-space processes
  - Programming interface to the services provided by the OS
- **Mostly accessed by programs via a high-level Application Program Interface (API) rather than directly**
  - m calls b s in glib



Modes of Operation

User Mode

- user program executes in user mode
- certain areas of memory are protected from user access
- certain instructions may not be executed

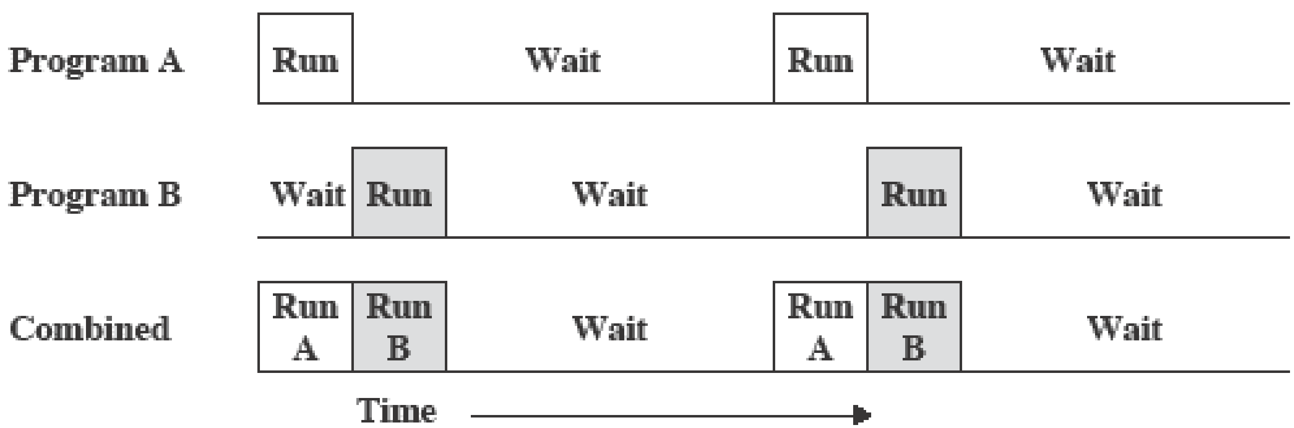
Kernel Mode

- monitor executes in kernel mode
- privileged instructions may be executed
- protected areas of memory may be accessed

Uniprogramming

(a) Uniprogramming

Multiprogramming



(b) Multiprogramming with two programs

Effects on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

Table 2.2 Effects of Multiprogramming on Resource Utilization

Utilization Histograms

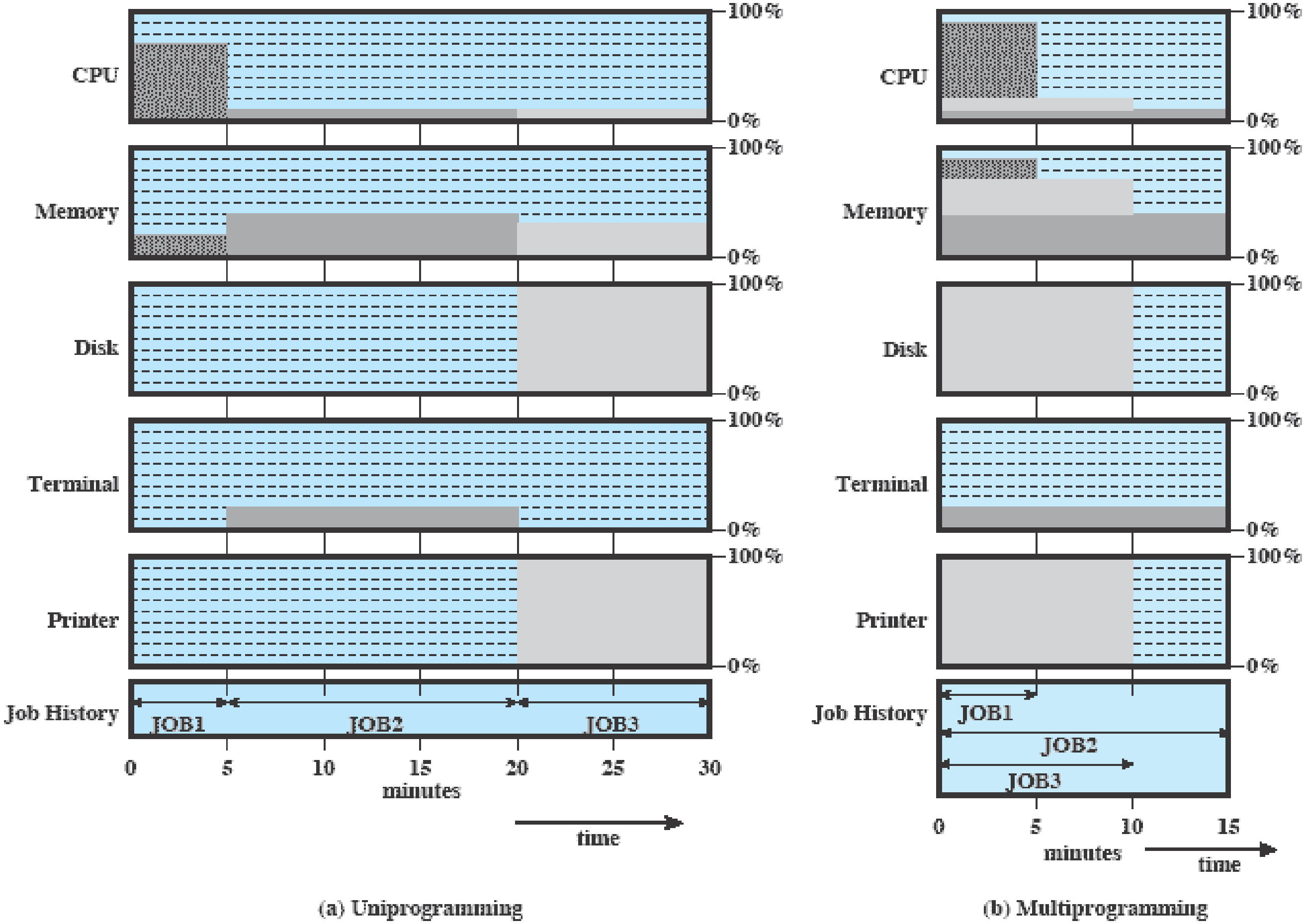


Figure 2.6 Utilization Histograms

Compatible Time-Sharing Systems

CTSS

- One of the first time-sharing operating systems
- Developed at MIT by a group known as Project MAC
- Ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that
- To simplify both the monitor and memory management a program was always loaded to start at the location of the 5000th word

Time Slicing

- System clock generates interrupts at a rate of approximately one every 0.2 seconds
- At each interrupt OS regained control and could assign processor to another user
- At regular time intervals the current user would be preempted and another user loaded in
- Old user programs and data were written out to disk
- Old user program code and data were restored in main memory when that program was next given a turn

CTSS Operation

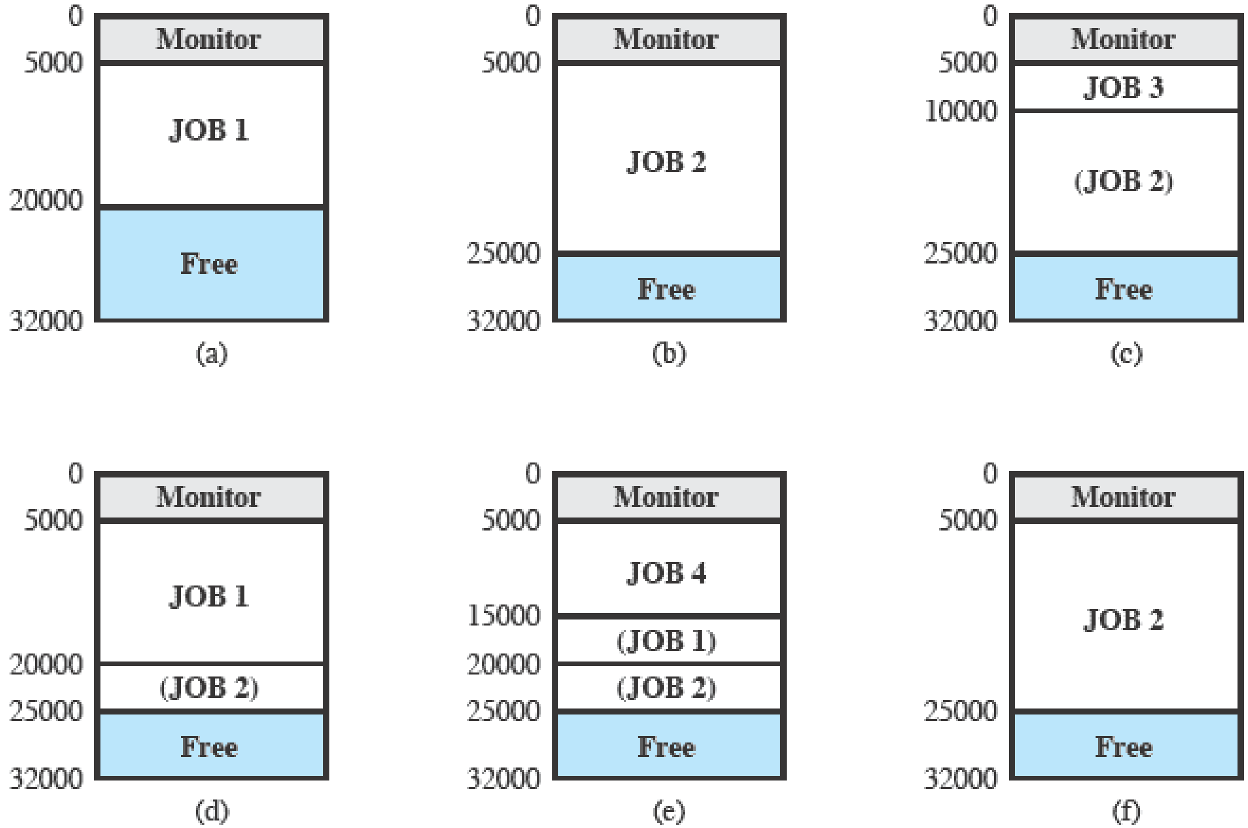


Figure 2.7 CTSS Operation

Major Advances

- Operating Systems are among the most complex pieces of software ever developed

Major advances in development include:

- Processes
- Memory management
- Information protection and security
- Scheduling and resource management
- System structure

Process

- Fundamental to the structure of operating systems

A *process* can be defined as:

- a program in execution
- an instance of a running program
- the entity that can be assigned to, and executed on, a processor
- a unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

Development of the Process

- Three major lines of computer system development created problems in timing and synchronization that contributed to the development:

multiprogramming batch operation

- processor is switched among the various programs residing in main memory

time sharing

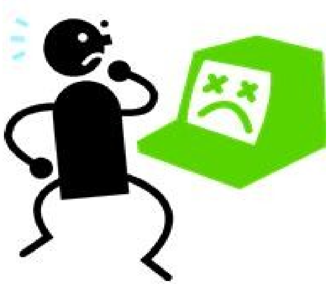
- be responsive to the individual user but be able to support many users simultaneously

real-time transaction systems

- users are entering queries or updates against a database

Causes of Errors

- **Improper synchronization**
  - a program must wait until the data are available in a buffer
  - improper design of the signaling mechanism can result in loss or duplication
- **Failed mutual exclusion**
  - more than one user or program attempts to make use of a shared resource at the same time
  - only one routine at at time allowed to perform an update against the file



- **Nondeterminate program operation**
  - program execution is interleaved by the processor when memory is shared
  - the order in which programs are scheduled may affect their outcome
- **Deadlocks**
  - it is possible for two or more programs to be hung up waiting for each other
  - may depend on the chance timing of resource allocation and release

Components of a Process

- A process contains three components:
  - an executable program
  - the associated data needed by the program (variables, work space, buffers, etc.)
  - the execution context (or “process state”) of the program

- The execution context is essential:
  - it is the internal data by which the OS is able to supervise and control the process
  - includes the contents of the various process registers
  - includes information such as the priority of the process and whether the process is waiting for the completion of a particular I/O event

Process Management

- The entire state of the process at any instant is contained in its context
- New features can be designed and incorporated into the OS by expanding the context to include any new information needed to support the feature

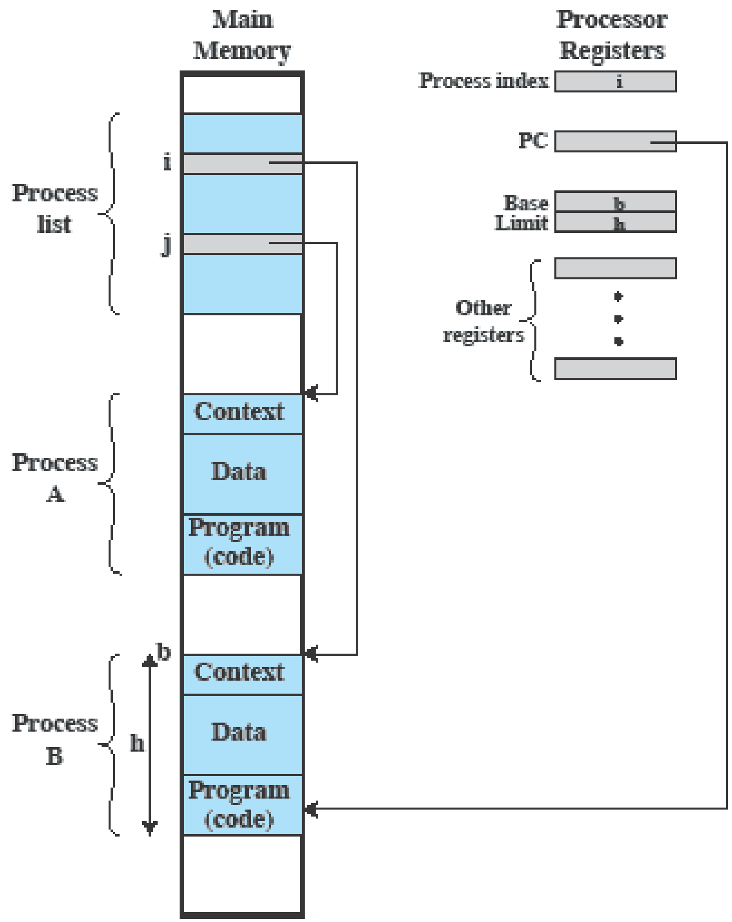


Figure 2.8 Typical Process Implementation

Memory Management

- The OS has five principal storage management responsibilities:

- process isolation
- automatic allocation and management
- support of modular programming
- protection and access control
- long-term storage

Virtual Memory

- A facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available
- Conceived to meet the requirement of having multiple user jobs reside in main memory concurrently

Paging

- Allows processes to be comprised of a number of fixed-size blocks, called pages
- Program references a word by means of a virtual address
    - consists of a page number and an offset within the page
    - each page may be located anywhere in main memory
- Provides for a dynamic mapping between the virtual address used in the program and a real (or physical) address in main memory

Virtual Memory

- A facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available
- Conceived to meet the requirement of having multiple user jobs reside in main memory concurrently

Virtual Memory Addressing

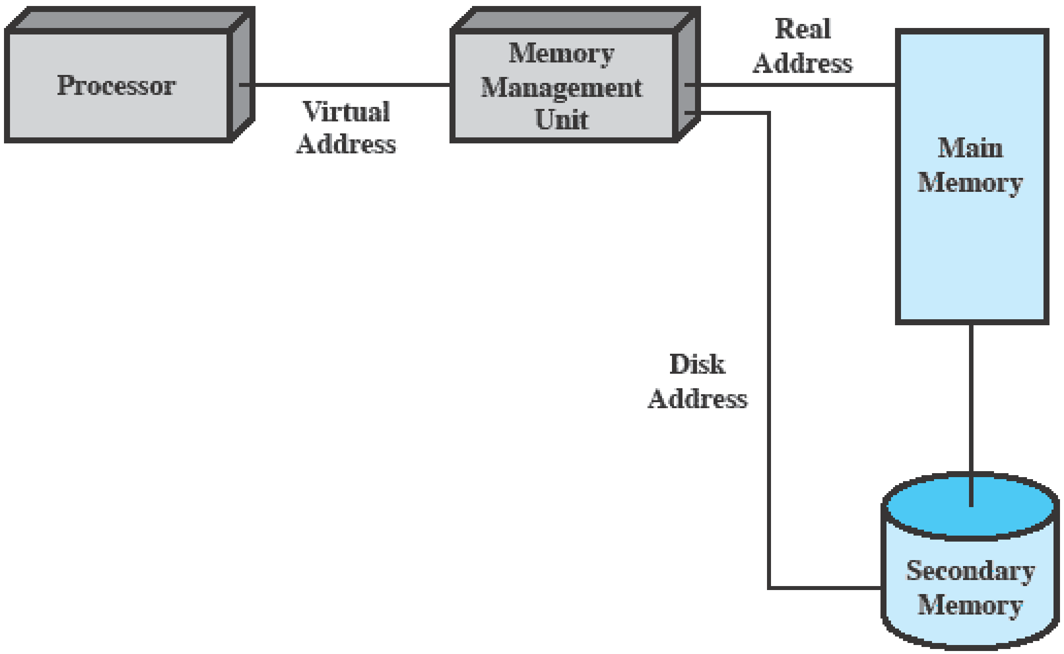
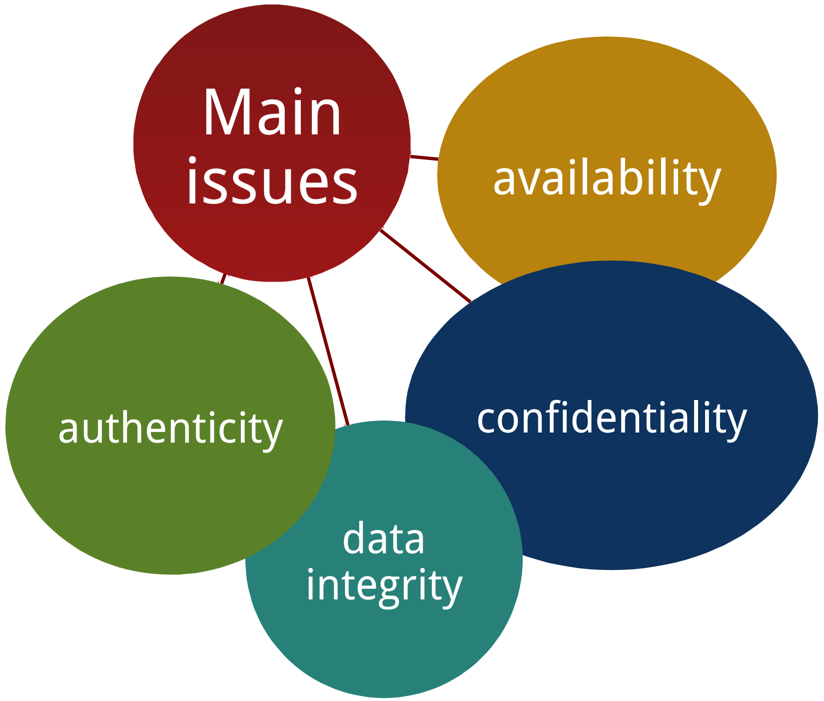


Figure 2.10 Virtual Memory Addressing

Information Protection and Security

- The nature of the threat that concerns an organization will vary greatly depending on the circumstances
- The problem involves controlling access to computer systems and the information stored in them



Key Elements of an Operating System

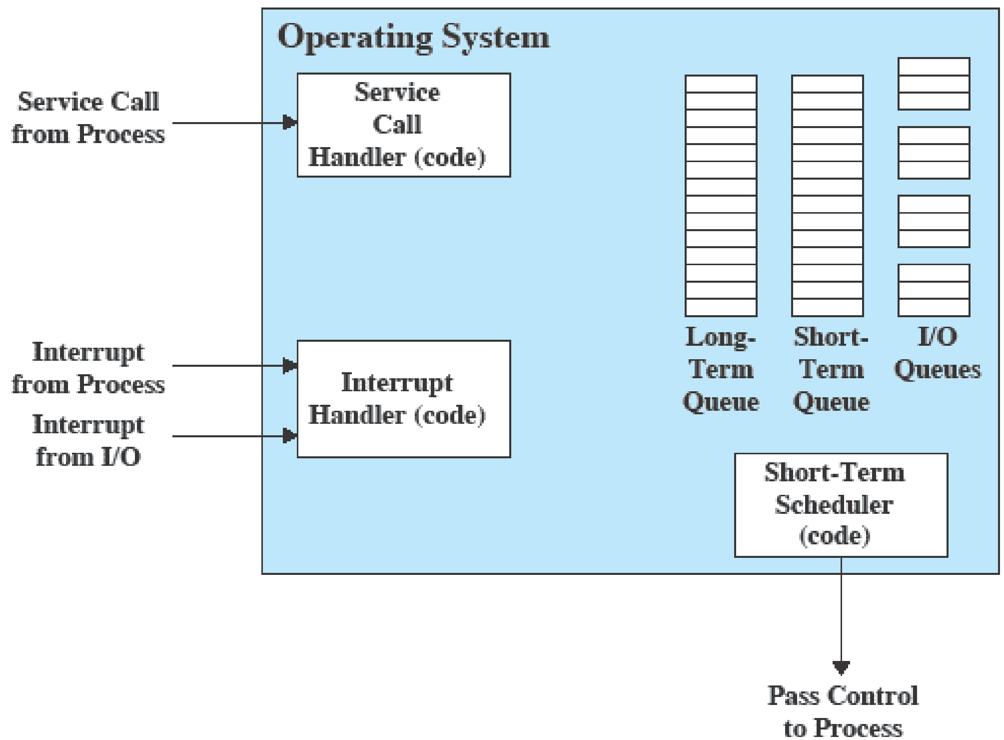


Figure 2.11 Key Elements of an Operating System for Multiprogramming

Different Architectural Approaches

- Demands on operating systems require new ways of organizing the OS

Different approaches and design elements have been tried:

- Microkernel architecture
- Multithreading
- Symmetric multiprocessing
- Distributed operating systems
- Object-oriented design

Microkernel Architecture

- Assigns only a few essential functions to kernel:
  - address spaces
  - interprocess communication (IPC)
  - basic scheduling
- The approach:
  - simplifies implementation
  - provides flexibility
  - is well suited to a distributed environment

Multithreading

- Technique in which a process, executing an application, is divided into threads that can run concurrently

Thread

- dispatchable unit of work
- includes a processor context and its own data area to enable subroutine branching
- executes sequentially and is interruptible

Process

- a collection of one or more threads and associated system resources
- programmer has greater control over the modularity of the application and the timing of application related events

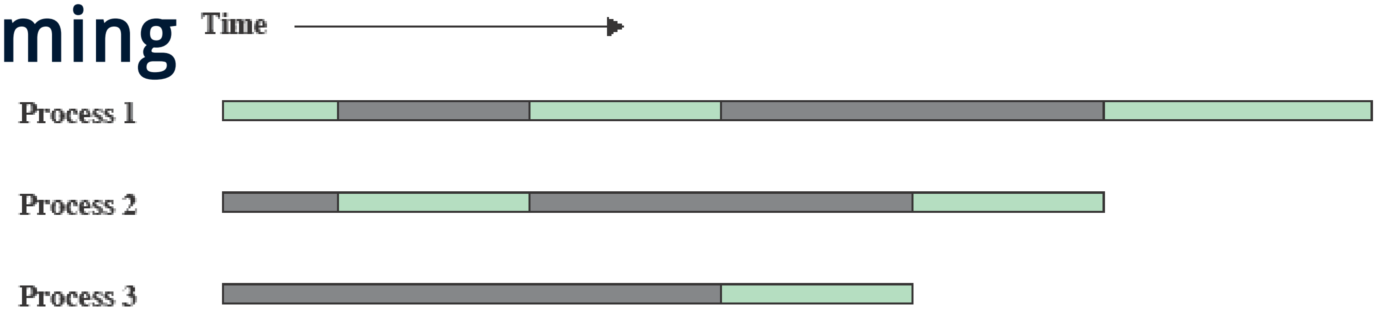
Symmetric Multiprocessing (SMP)

- Term that refers to a computer hardware architecture and also to the OS behavior that exploits that architecture
- Several processes can run in parallel
- Multiple processors are transparent to the user
  - these processors share same main memory and I/O facilities
  - all processors can perform the same functions
- The OS takes care of scheduling of threads or processes on individual processors and of synchronization among processors

SMP Advantages

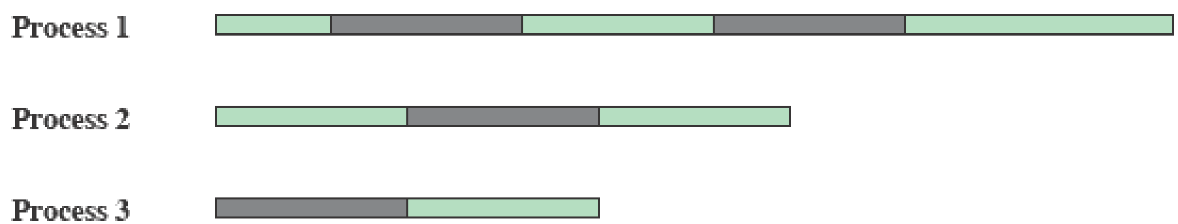
Performance	more than one process can be running simultaneously, each on a different processor
Availability	failure of a single process does not halt the system
Incremental Growth	performance of a system can be enhanced by adding an additional processor
Scaling	vendors can offer a range of products based on the number of processors configured in the system

Multiprogramming



(a) Interleaving (multiprogramming, one processor)

Multiprocessing



(b) Interleaving and overlapping (multiprocessing; two processors)

Figure 2.12 Multiprogramming and Multiprocessing

OS Design

Distributed Operating System

- Provides the illusion of
  - a single main memory space
  - single secondary memory space
  - unified access facilities
- State of the art for distributed operating systems lags that of uniprocessor and SMP operating systems

Object-Oriented Design

- Used for adding modular extensions to a small kernel
- Enables programmers to customize an operating system without disrupting system integrity
- Eases the development of distributed tools and full-blown distributed operating systems

Virtual Machines and Virtualization

- Virtualization
  - enables a single PC or server to simultaneously run multiple operating systems or multiple sessions of a single OS
  - a machine can host numerous applications, including those that run on different operating systems, on a single platform
  - host operating system can support a number of virtual machines (VM)
    - each has the characteristics of a particular OS and, in some versions of virtualization, the characteristics of a particular hardware platform

Virtual Memory Concept

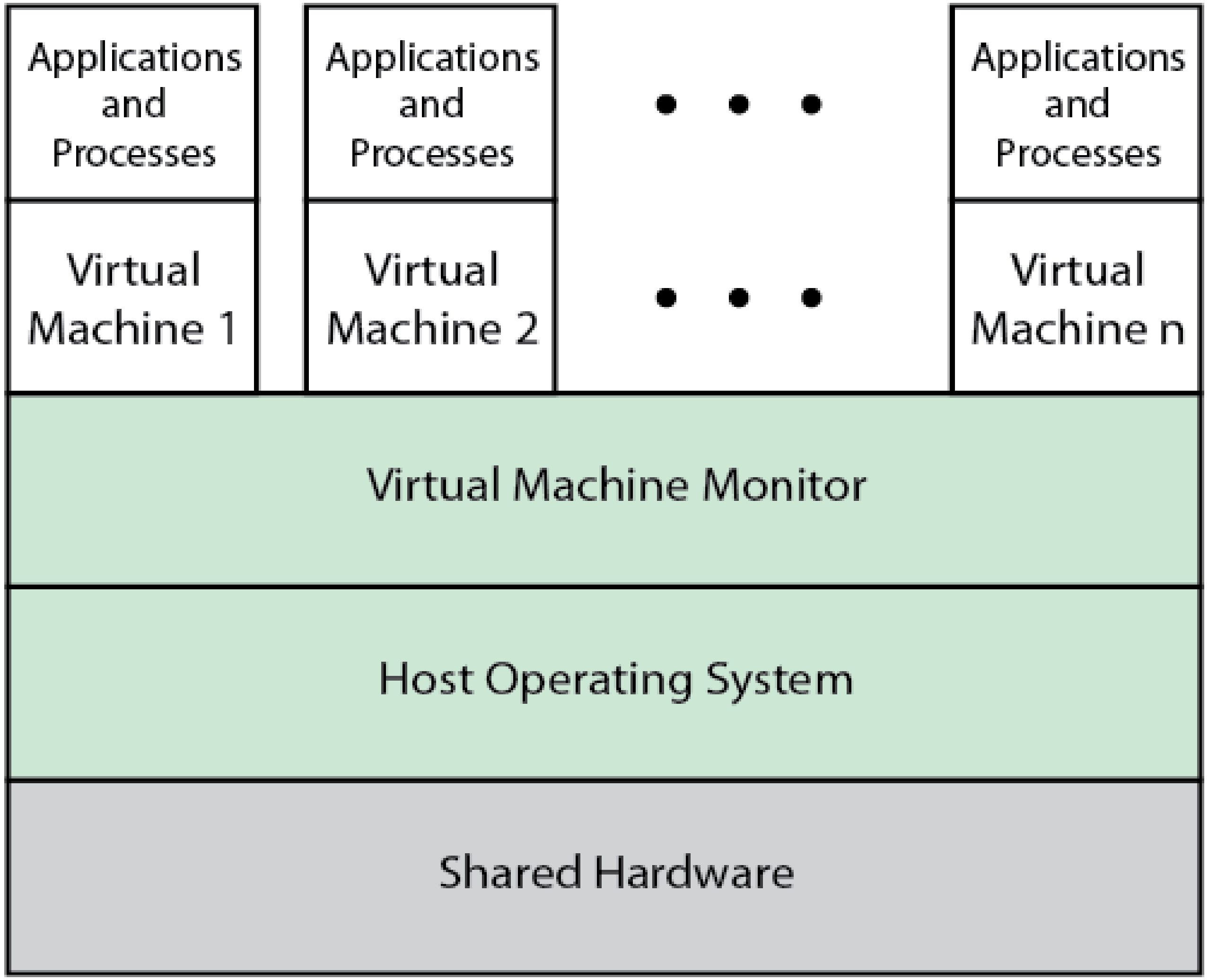


Figure 2.13 Virtual Memory Concept

Virtual Machine Architecture

Process perspective:

- the machine on which it executes consists of the virtual memory space assigned to the process
- the processor registers it may use
- the user-level machine instructions it may execute
- OS system calls it may invoke for I/O
- ABI defines the machine as seen by a process

Application perspective:

- machine characteristics are specified by high-level language capabilities and OS system library calls
- API defines the machine for an application

OS perspective:

- processes share a file system and other I/O resources
- system allocates real memory and I/O resources to the processes
- ISA provides the interface between the system and machine

Process and System Virtual Machines

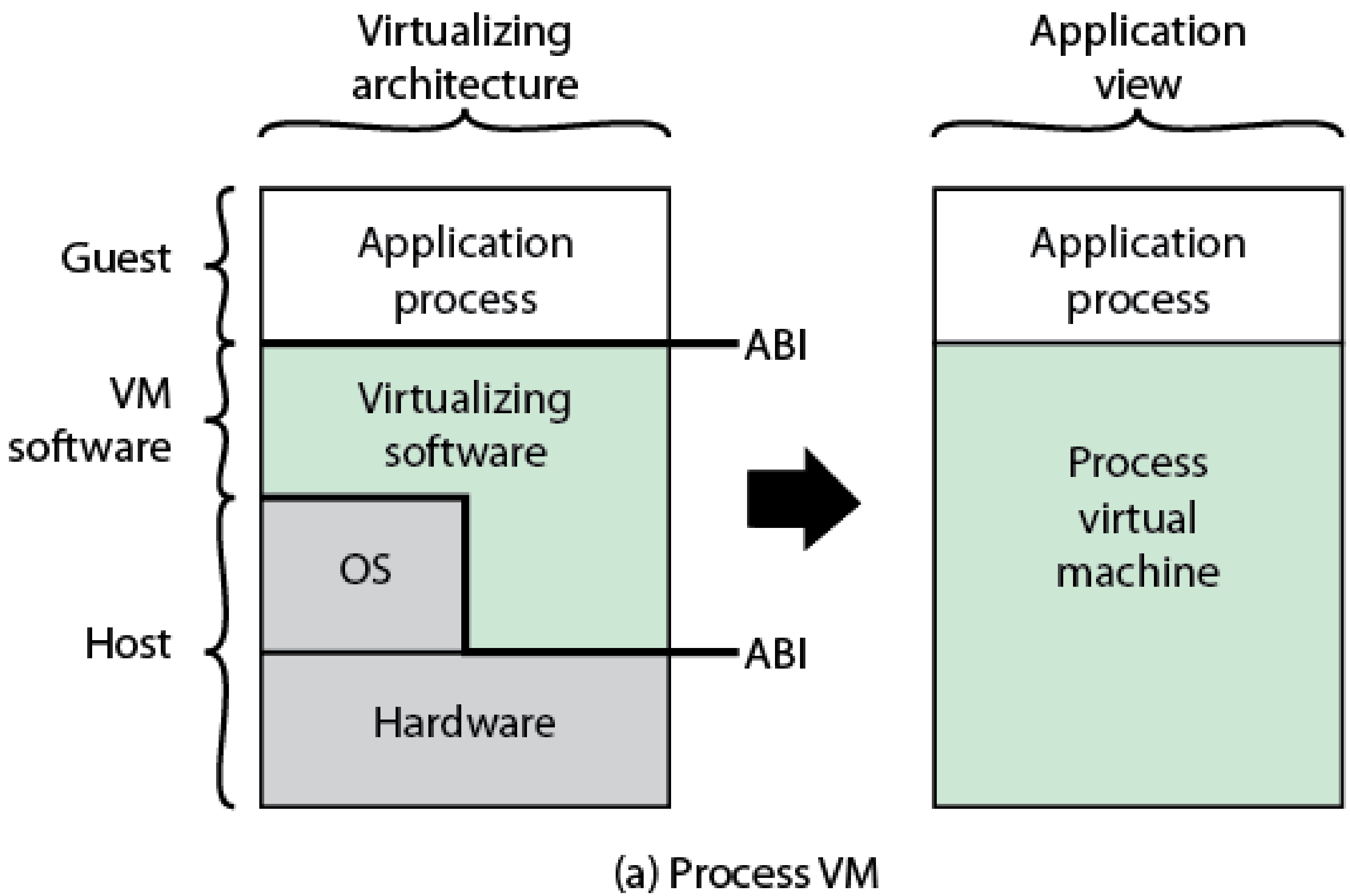
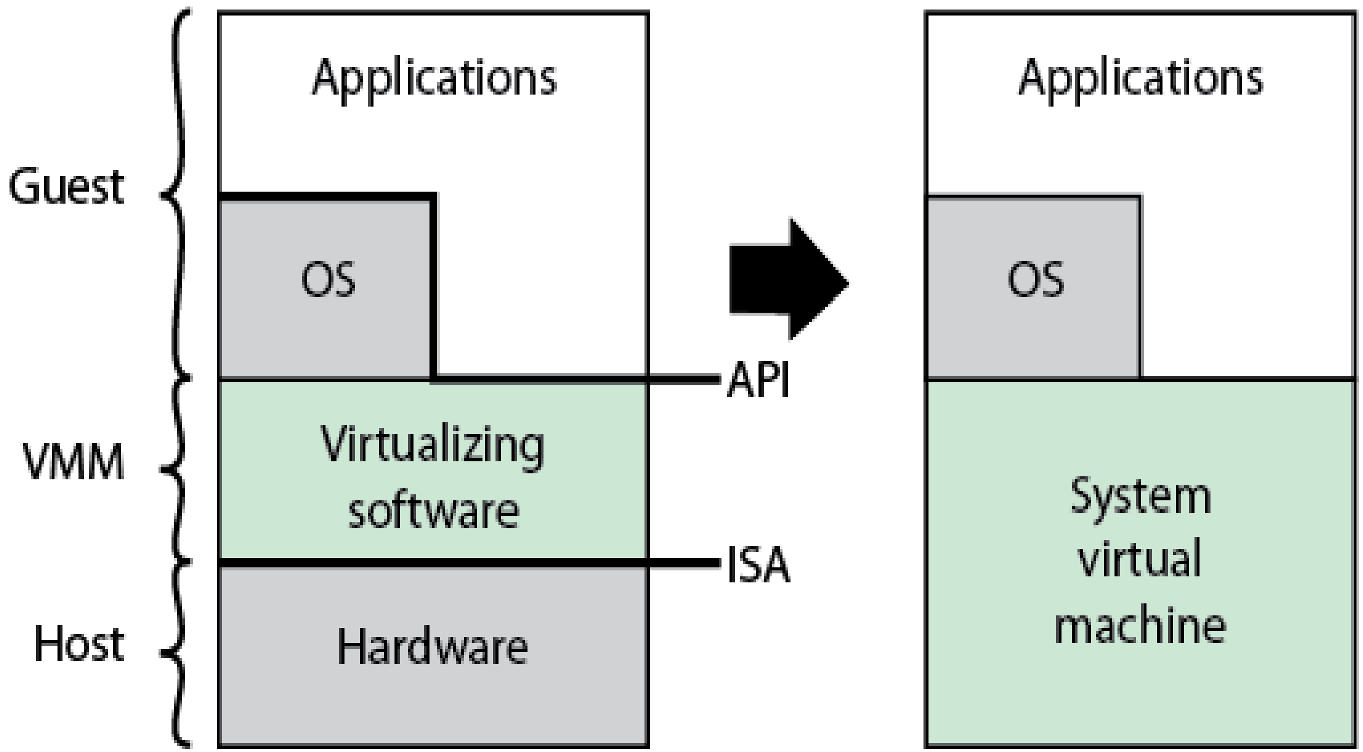


Figure 2.14 Process and System Virtual Machines

Process and System Virtual Machines



(b) System VM

Figure 2.14 Process and System Virtual Machines

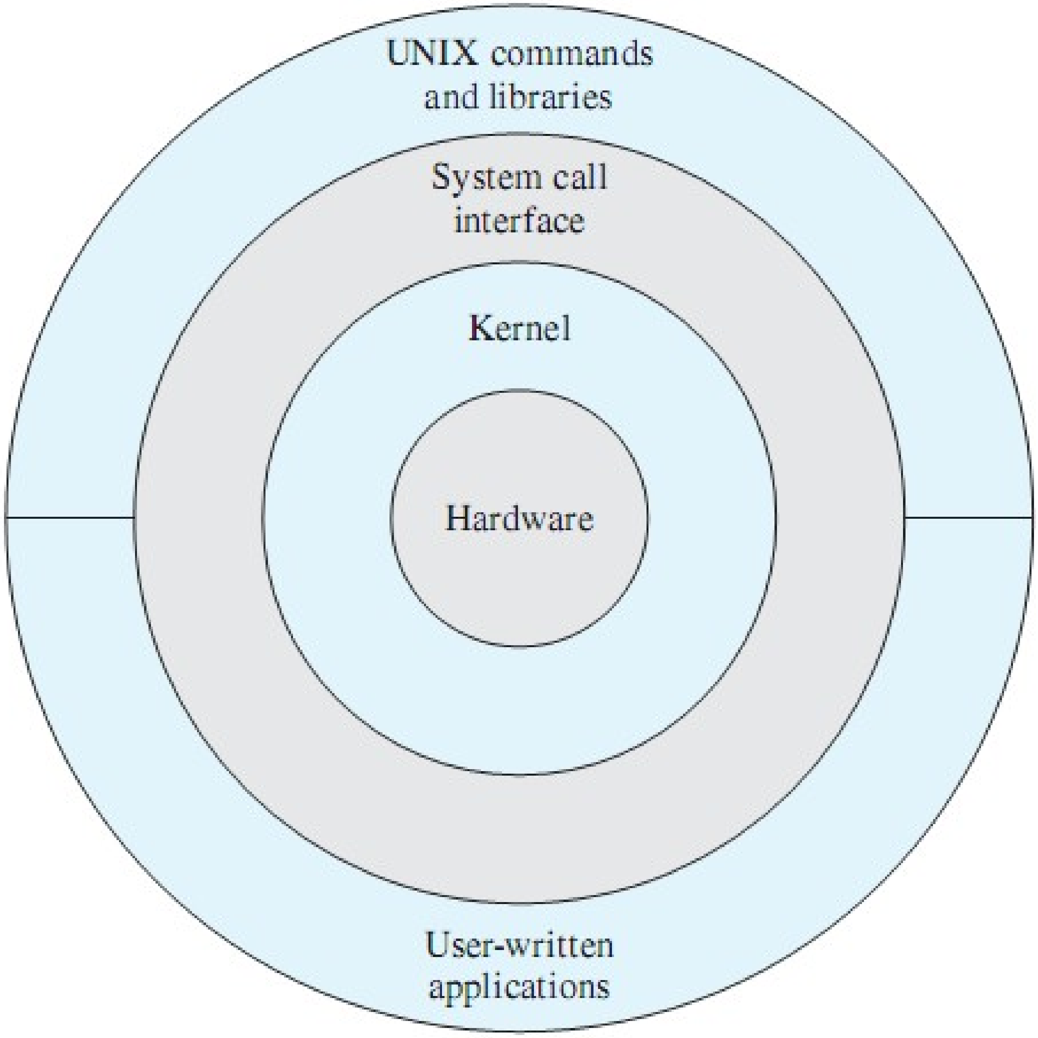
SMP OS Considerations

- A multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors
- **Key design issues:**
  - The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently

Virtual Machine Approach

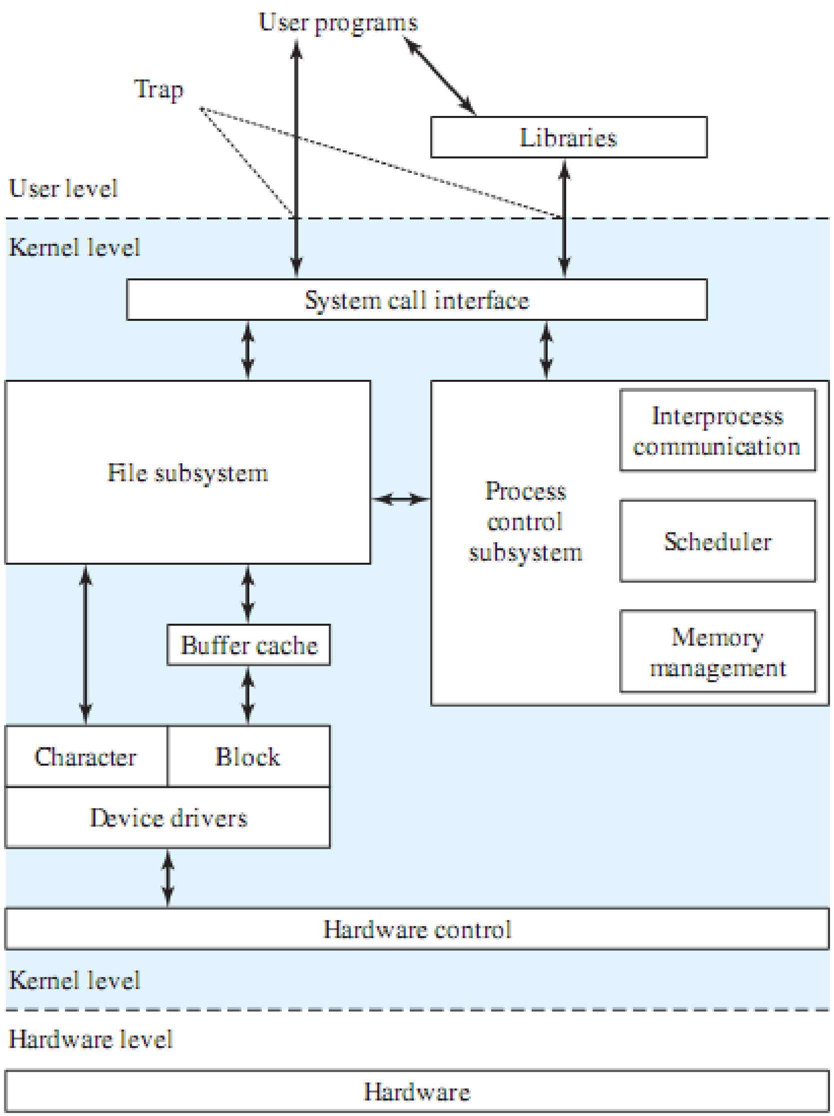
- Allows one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process
- Multicore OS could then act as a hypervisor that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that

Description of UNIX



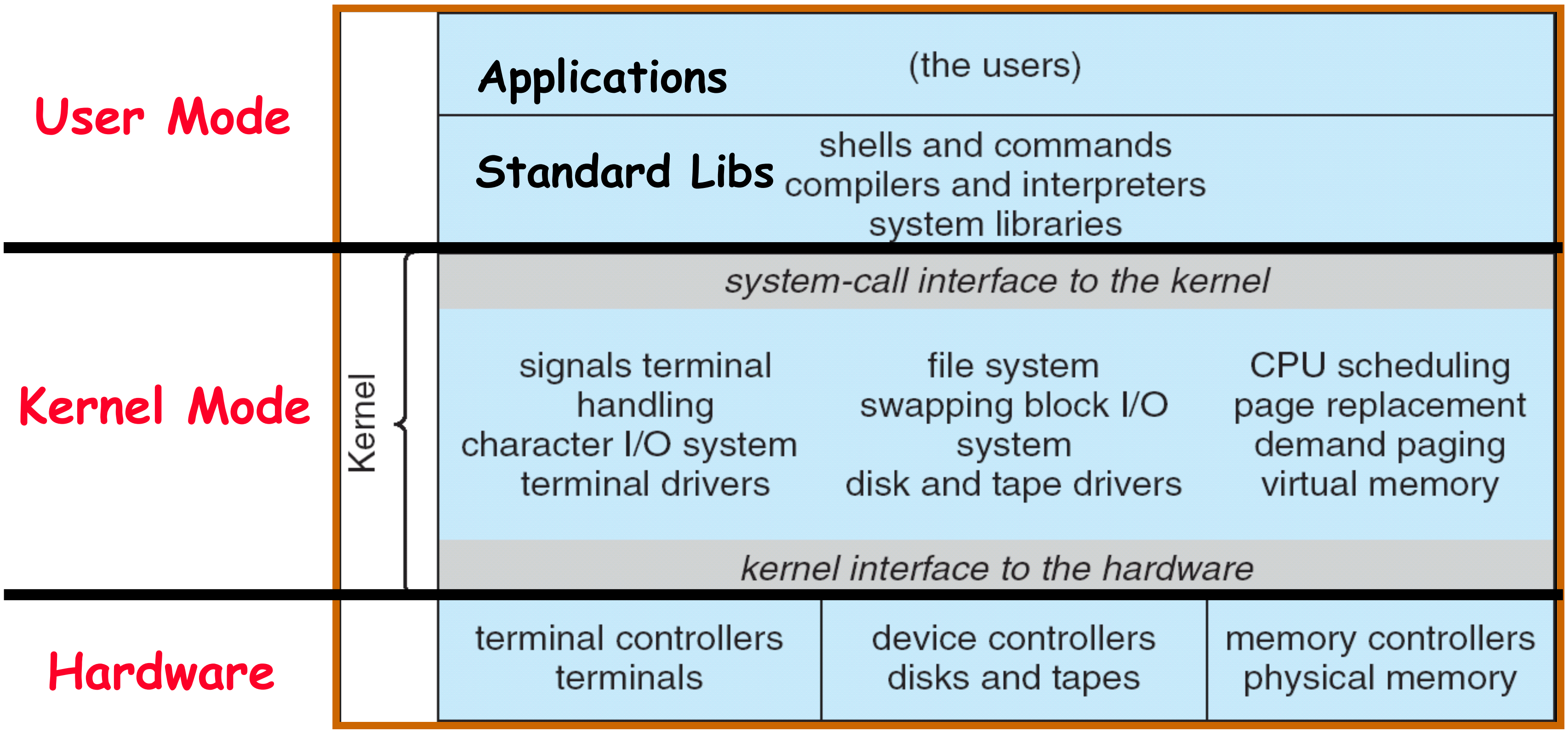
General UNIX Architecture

Traditional UNIX Kernel



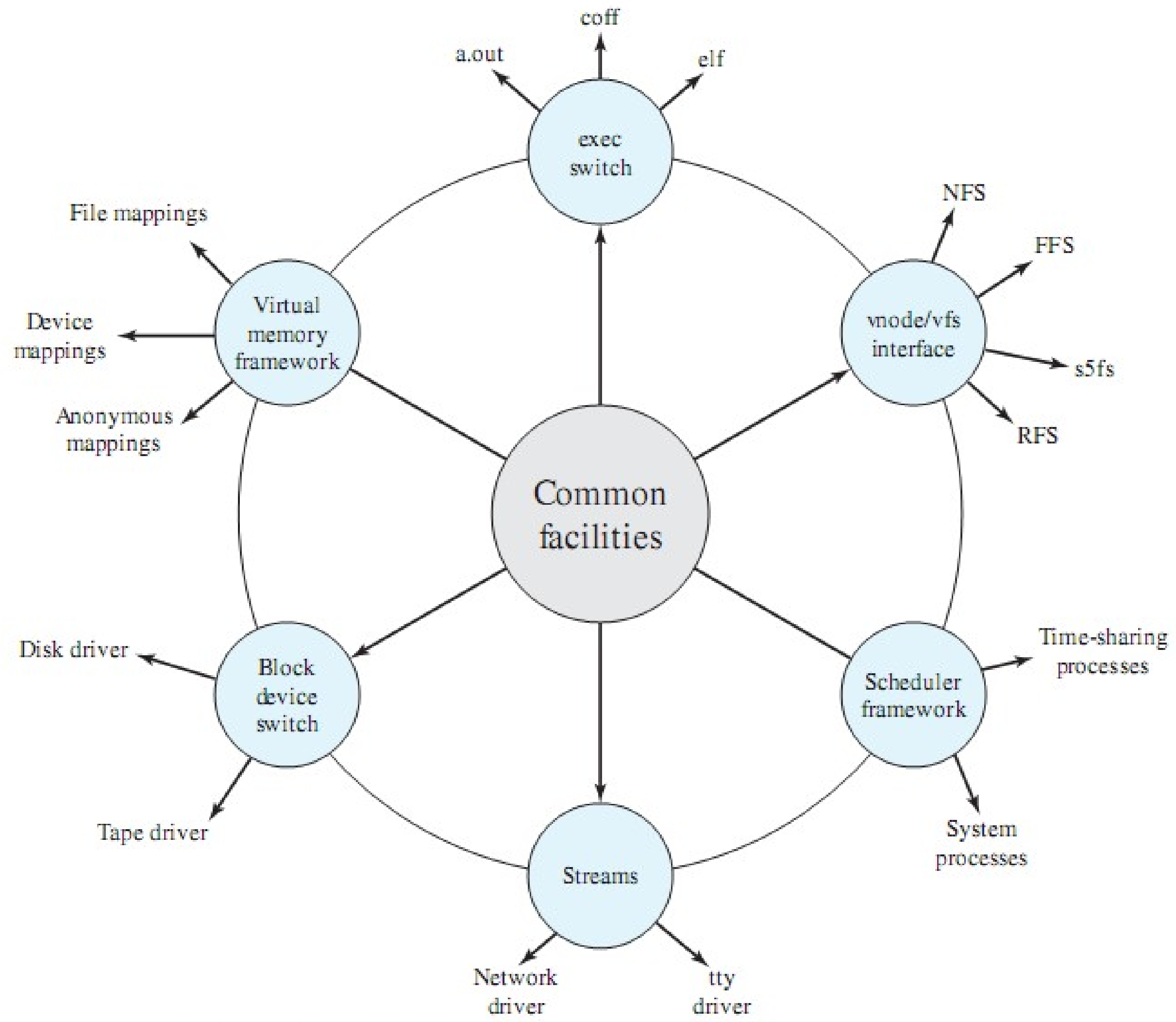
Traditional UNIX Kernel

Monolithic Structure: UNIX System Structure



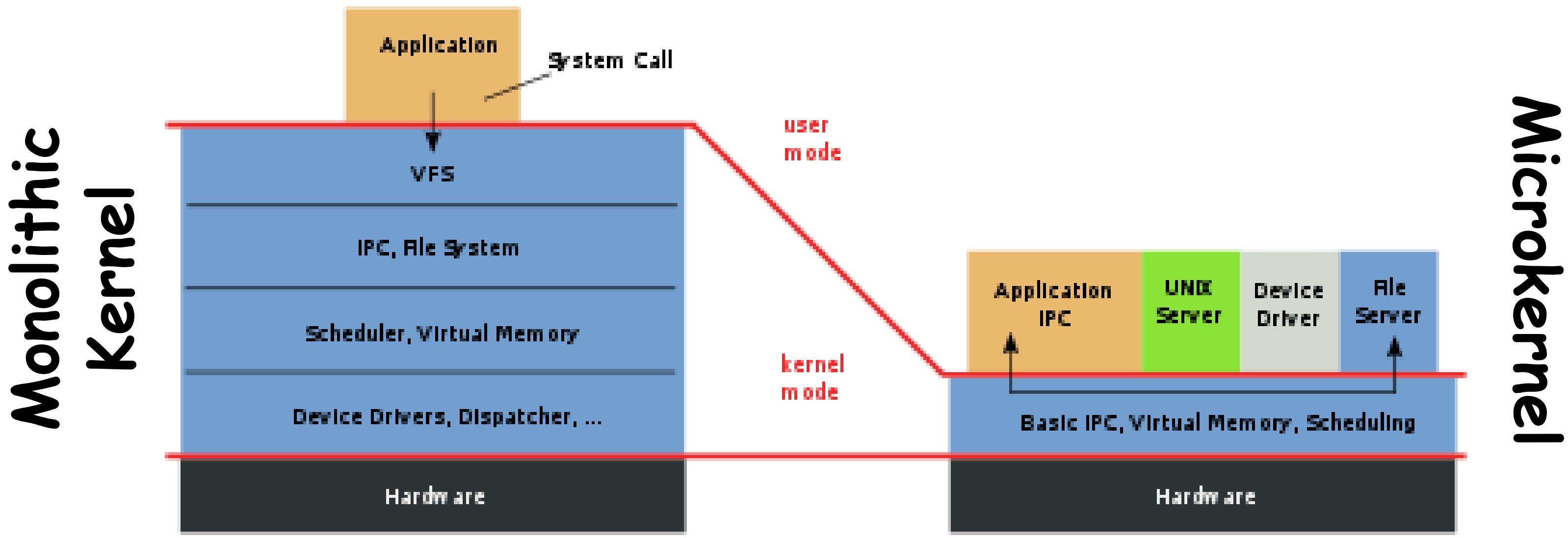
- Two-Layered Structure: User vs Kernel
  - All code representing protection and management of resources placed in same address space
  - Compromise of one component can compromise whole OS

Modern UNIX Kernel



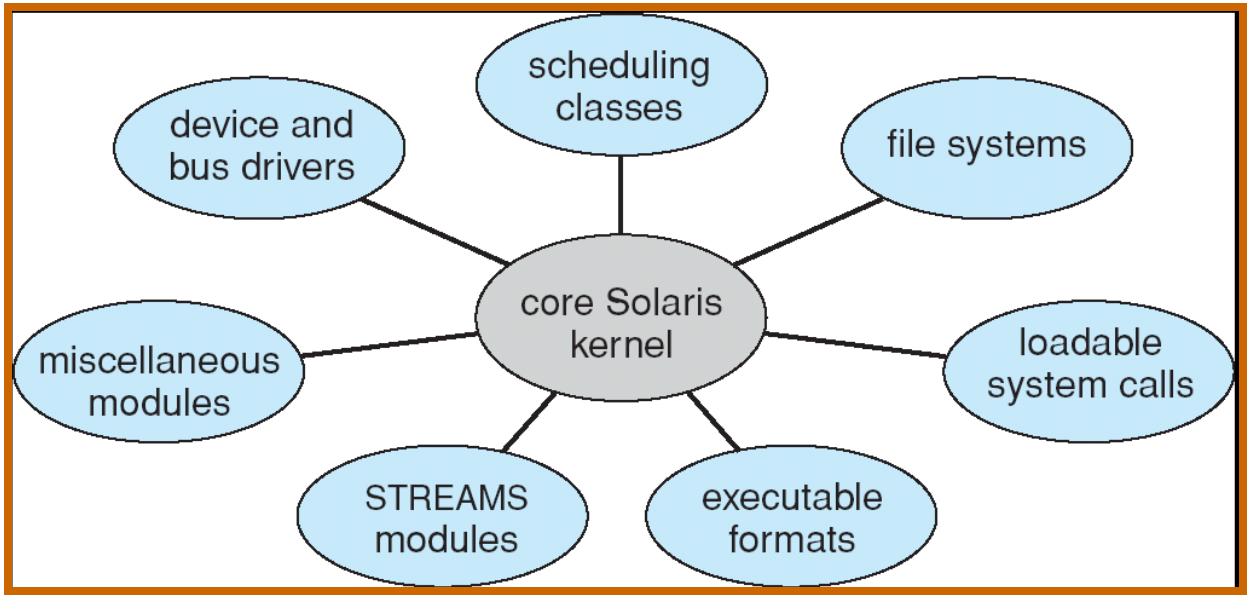
Modern UNIX Kernel

Microkernel Structure



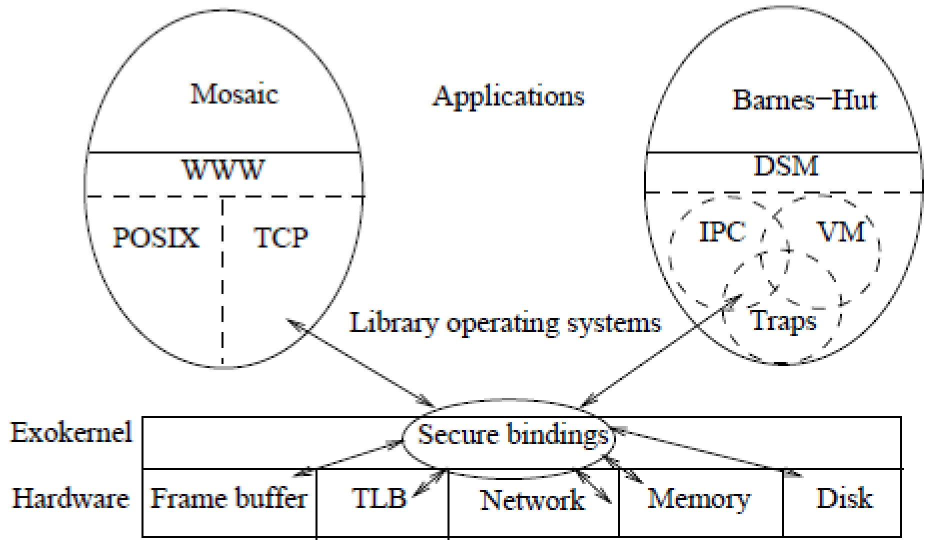
- **Moves functionality from the kernel into "*user*" space**
  - Small core OS running at kernel level
  - OS Services built from many independent user-level processes
  - Communication between modules with message passing
- **Benefits:**
  - Easier to extend
  - Easier to port OS to new architectures
  - More reliable (less code is running in kernel mode)
  - Fault Isolation (parts of kernel protected from other parts)
  - More secure
- **Detriments:**
  - **Performance overhead can be severe for naïve implementation**

Modules-based Structure



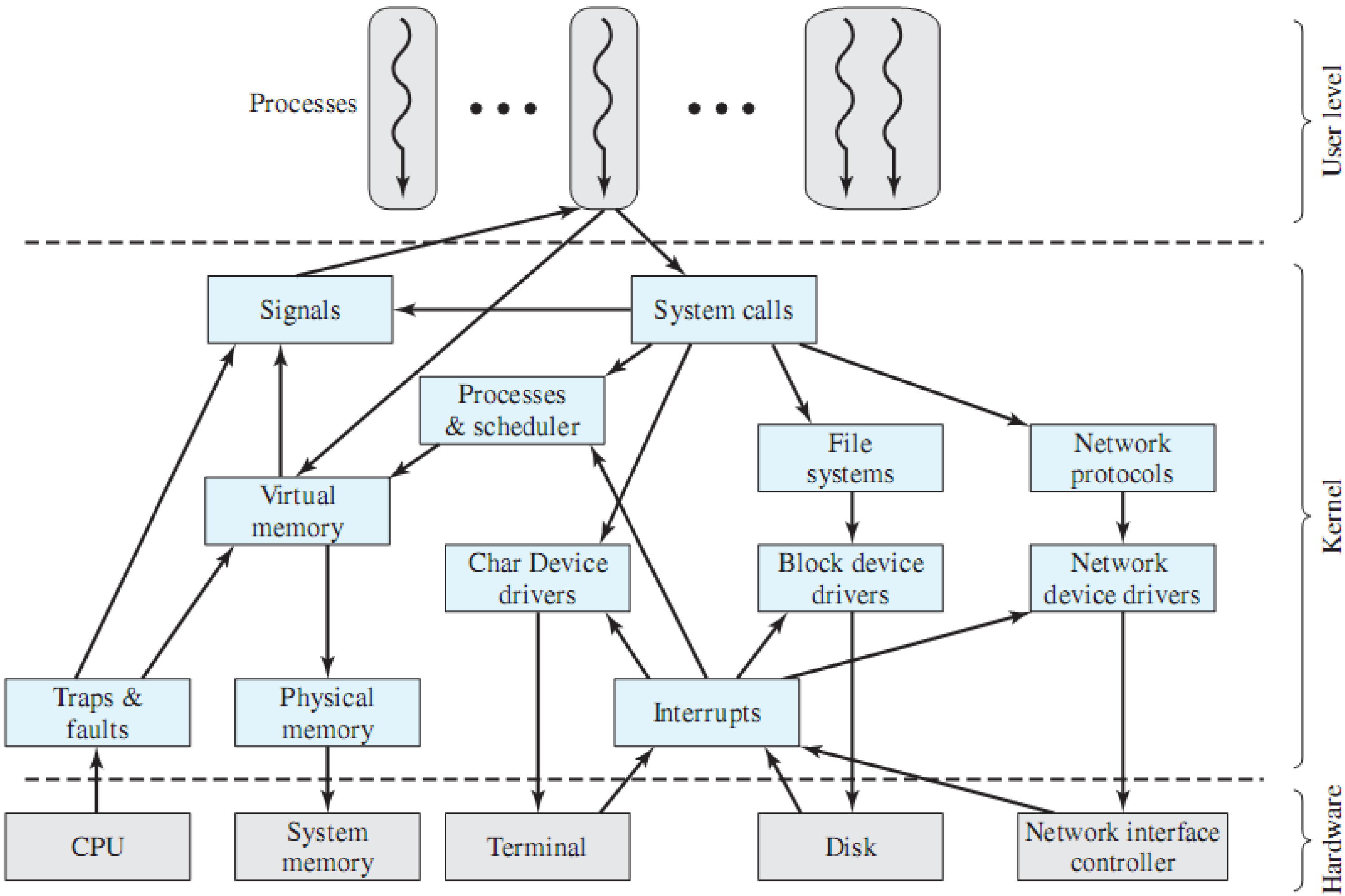
- **Most modern operating systems implement modules**
  - Uses object-oriented approach
    - » careful API design/Few if any global variables
- **Each core component is separate**
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- **Overall, similar to layers but with more flexible**
  - May or may not utilize hardware enforcement

ExoKernel: Separate Protection from Management



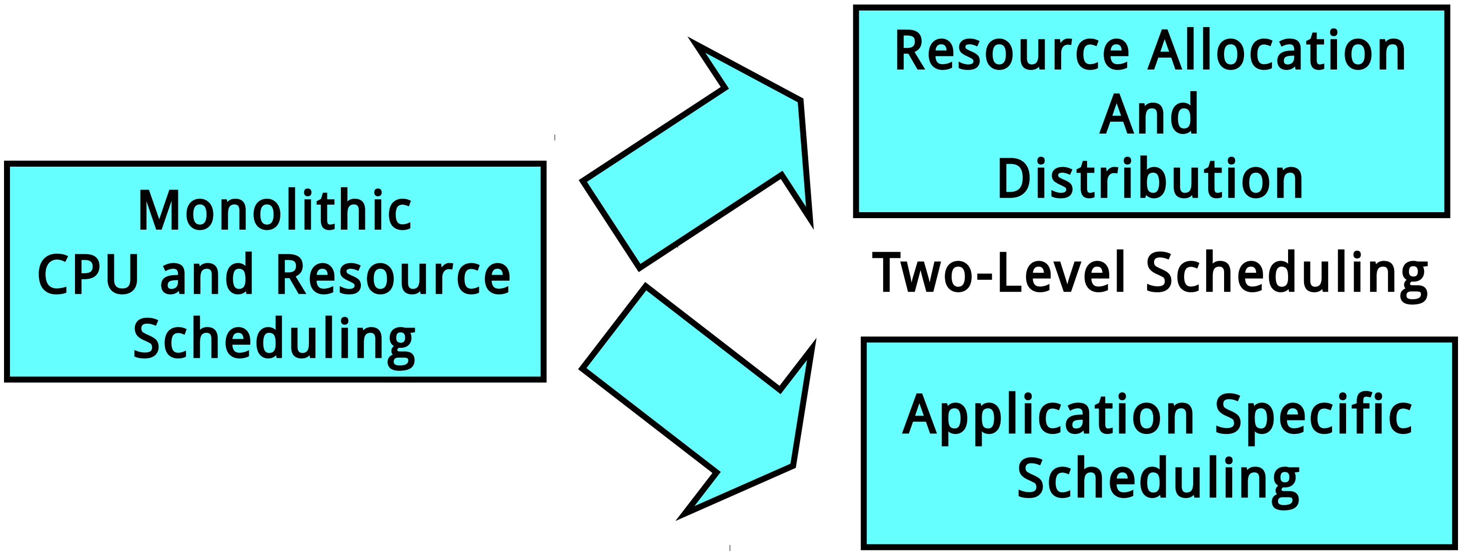
- **Thin layer exports hardware resources directly to users**
  - As little abstraction as possible
  - Secure Protection and Multiplexing of resources
- **LibraryOS: traditional OS functionality at User-Level**
  - Customize resource management for every application
  - Is this a practical approach?
- **Very low-level abstraction layer**
  - Need extremely specialized skills to develop LibraryOS

Linux Kernel Components



Linux Kernel Components

Two Level Scheduling

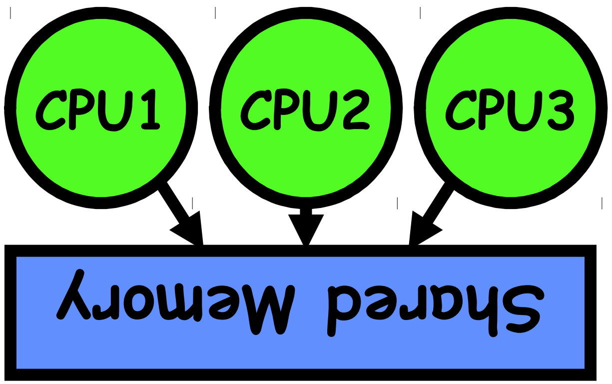


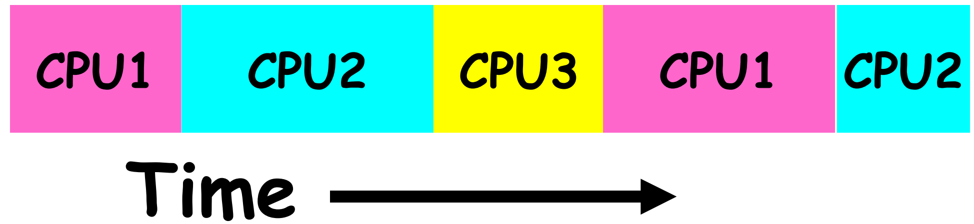
- Split monolithic scheduling into two pieces:
  - Course-Grained Resource Allocation and Distribution to Cells
    - Chunks of resources (CPUs, Memory Bandwidth, QoS to Services)
    - Ultimately a hierarchical process negotiated with service providers
  - Fine-Grained (User-Level) Application-Specific Scheduling
    - Applications allowed to utilize their resources in any way they see fit
    - Performance Isolation: Other components of the system cannot interfere with Cells use of resources

Concurrency

- **"Thread" of execution**
  - Independent Fetch/Decode/Execute loop
  - Operating in some Address space
- **Uniprogramming: *one thread at a time***
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Get rid concurrency by defining it away
  - Does this make sense for personal computers?
- **Multiprogramming: *more than one thread at a time***
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
  - Often called "multitasking", but multitasking has other meanings (talk about this later)

How can we give the illusion of multiple processors?



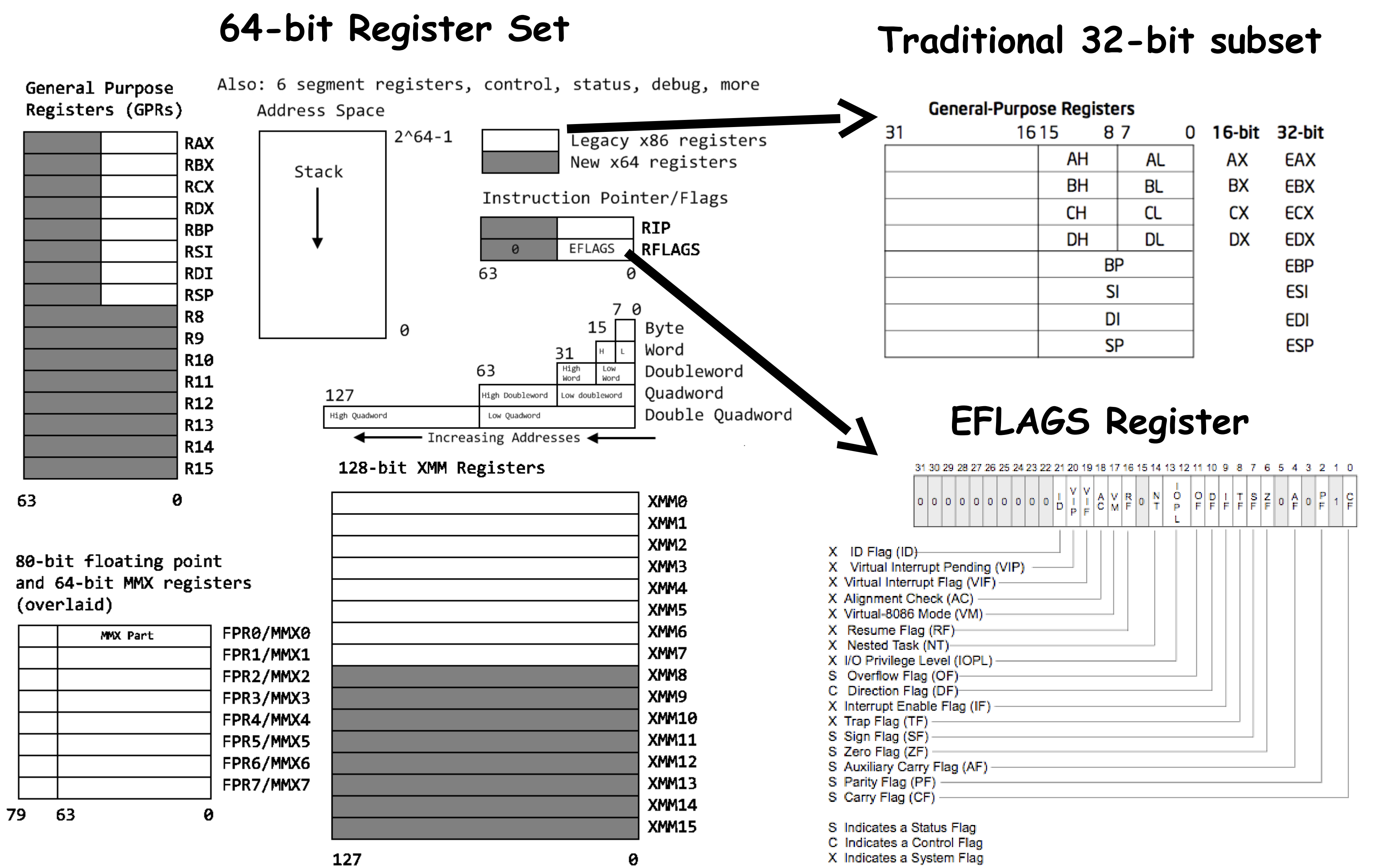


- **Assume a single processor. How do we provide the illusion of multiple processors?**
  – Multiplex in time!
- **Each virtual "CPU" needs a structure to hold:**
  – Program Counter (PC), Stack Pointer (SP)
  – Registers (Integer, Floating point, others…?)
  – Call result a "Thread" for now…
- **How switch from one CPU to the next?**
  – Save PC, SP, and registers in current state block
  – Load PC, SP, and registers from new state block
- **What triggers switch?**
  – Timer, voluntary yield, I/O, other things

Properties of this simple multiprogramming technique

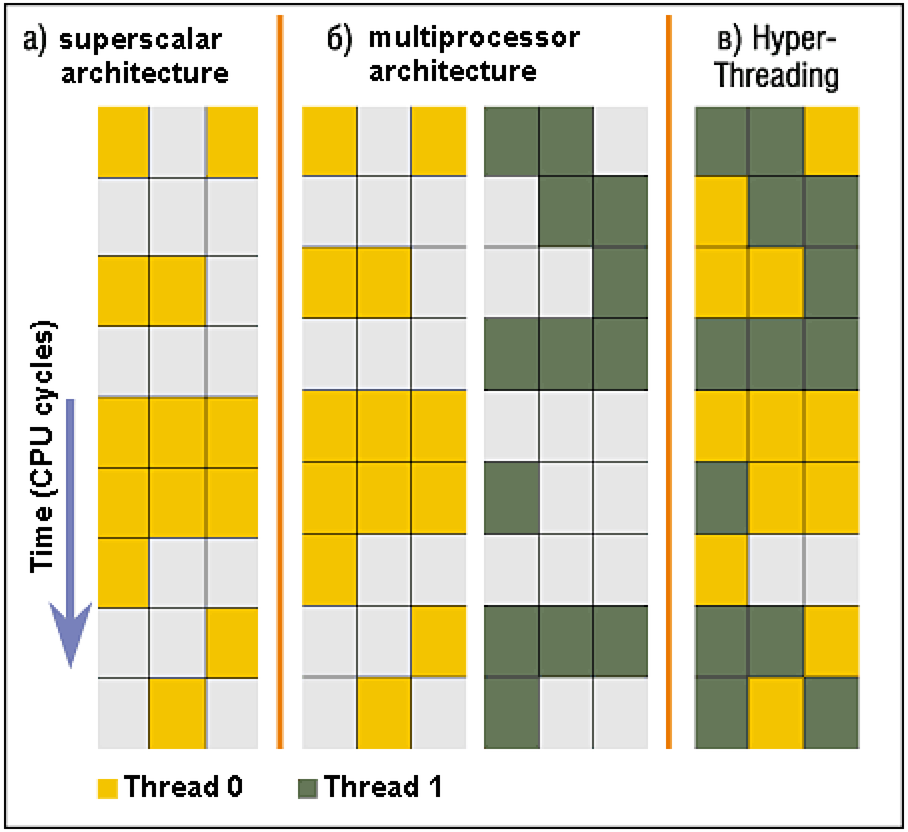
- **All virtual CPUs share same non-CPU resources**
  - I/O devices the same
  - Memory the same
- **Consequence of sharing:**
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- **This (unprotected) model common in:**
  - Embedded applications
  - Windows 3.1/Machintosh (switch only with yield)

What needs to be saved in Modern X86?



Modern Technique: SMT/Hyperthreading

- **Hardware technique**
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- **Can schedule each thread as if were separate CPU**
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first



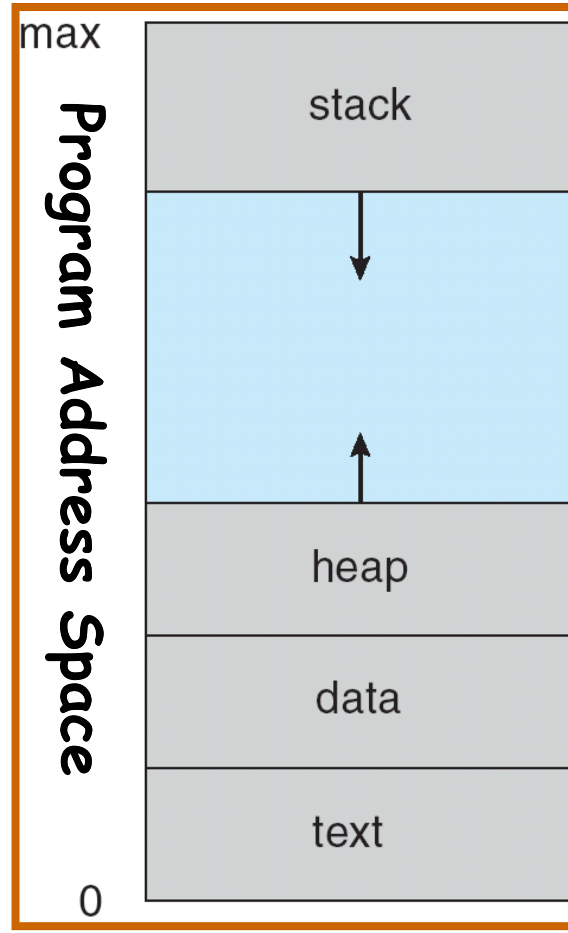
- **Original technique called "Simultaneous Multithreading"**
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5

How to protect threads from one another?

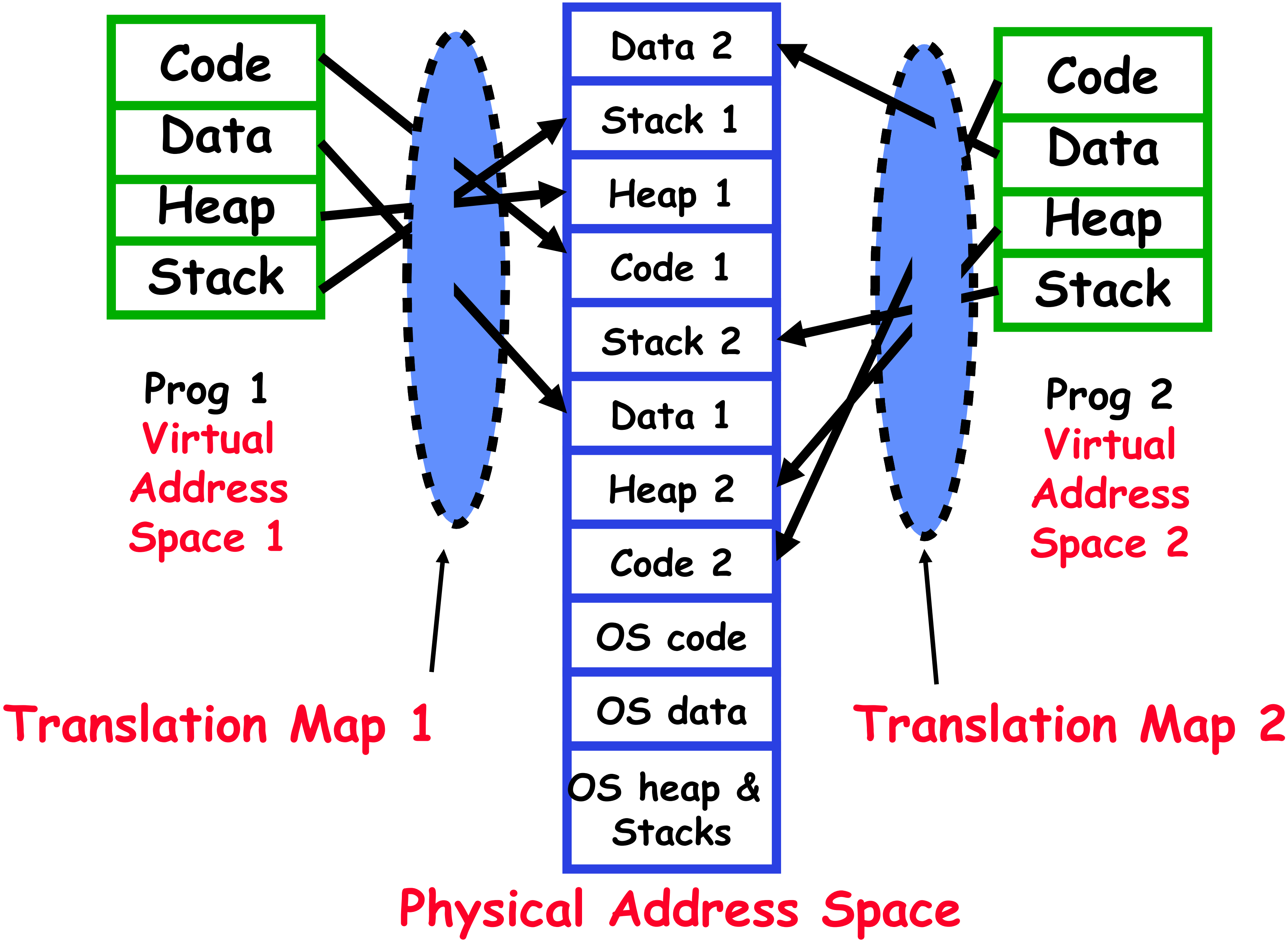
- Need three important things:
  1. Protection of memory
     - » Every task does not have access to all memory
  2. Protection of I/O devices
     - » Every task does not have access to every device
  3. Protection of Access to Processor: Preemptive switching from task to task
     - » Use of timer
     - » Must not be possible to disable timer from usercode

Program's Address Space

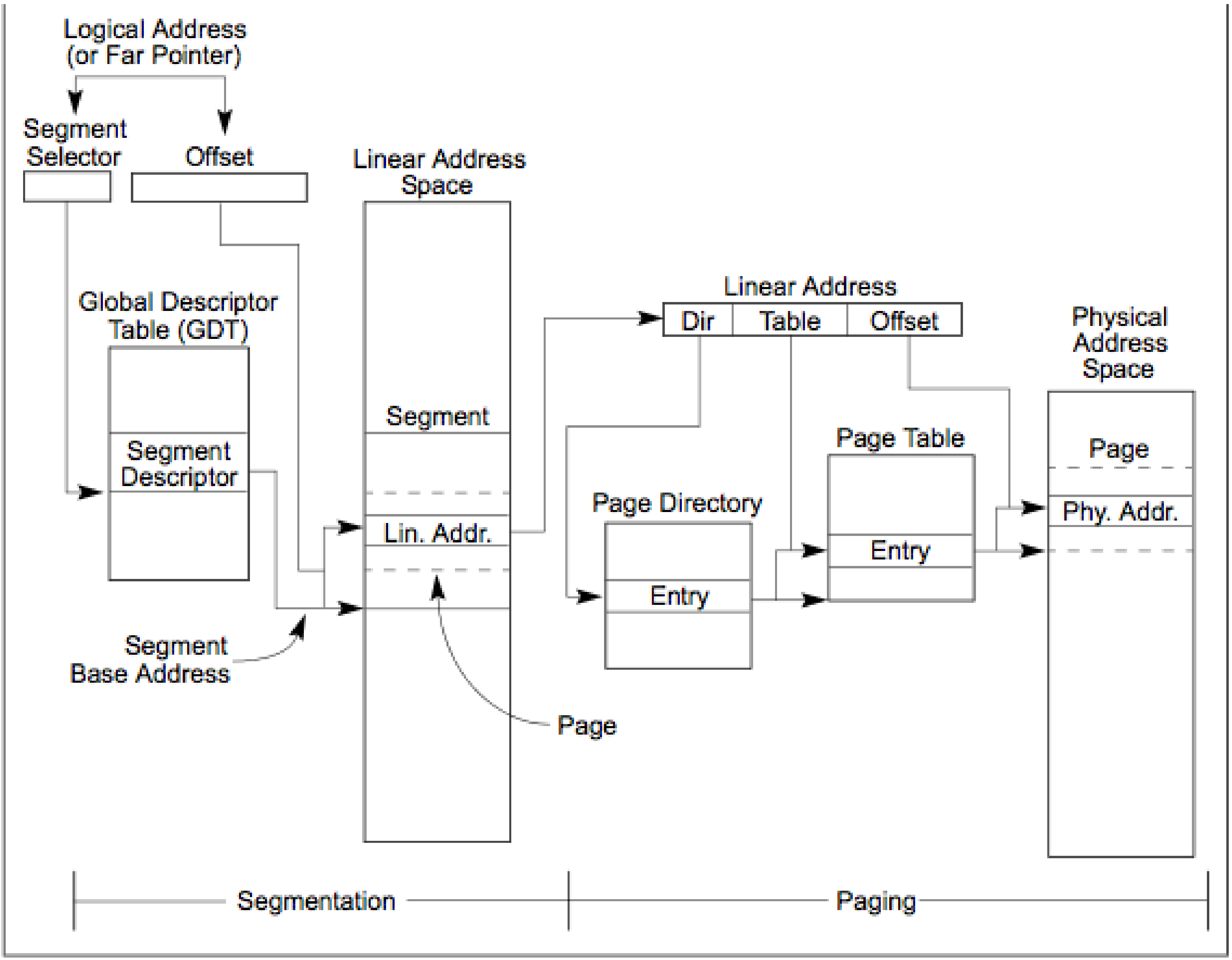
- **Address space ≡ the set of accessible addresses + state associated with them:**
  - For a 32-bit processor there are 2^{32} = 4 billion addresses
- **What happens when you read or write to an address?**
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Perhaps causes exception (fault)



Providing Illusion of Separate Address Space: Load new Translation Map on Switch

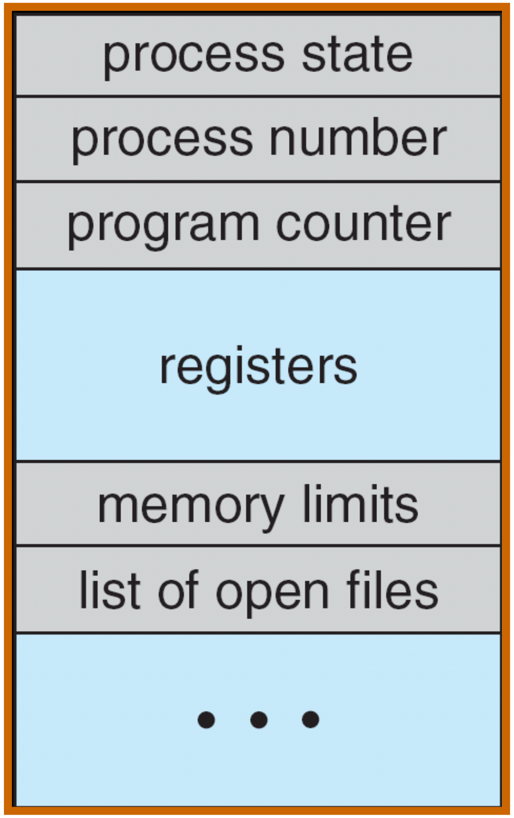


X86 Memory model with segmentation



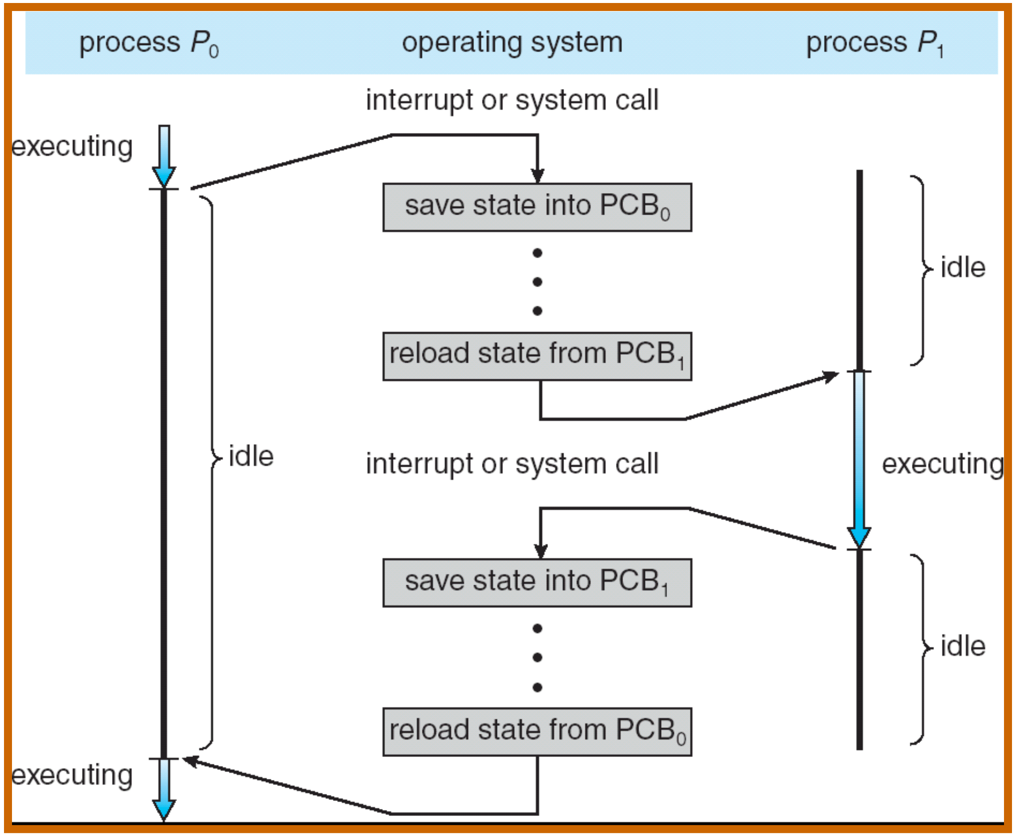
How do we multiplex processes?

- **The current state of process held in a process control block (PCB):**
  - This is a “snapshot” of the execution and protection environment
  - Only one PCB active at a time
- **Give out CPU time to different processes (Scheduling):**
  - Only one process “running” at a time
  - Give more time to important processes
- **Give pieces of resources to different processes (Protection):**
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



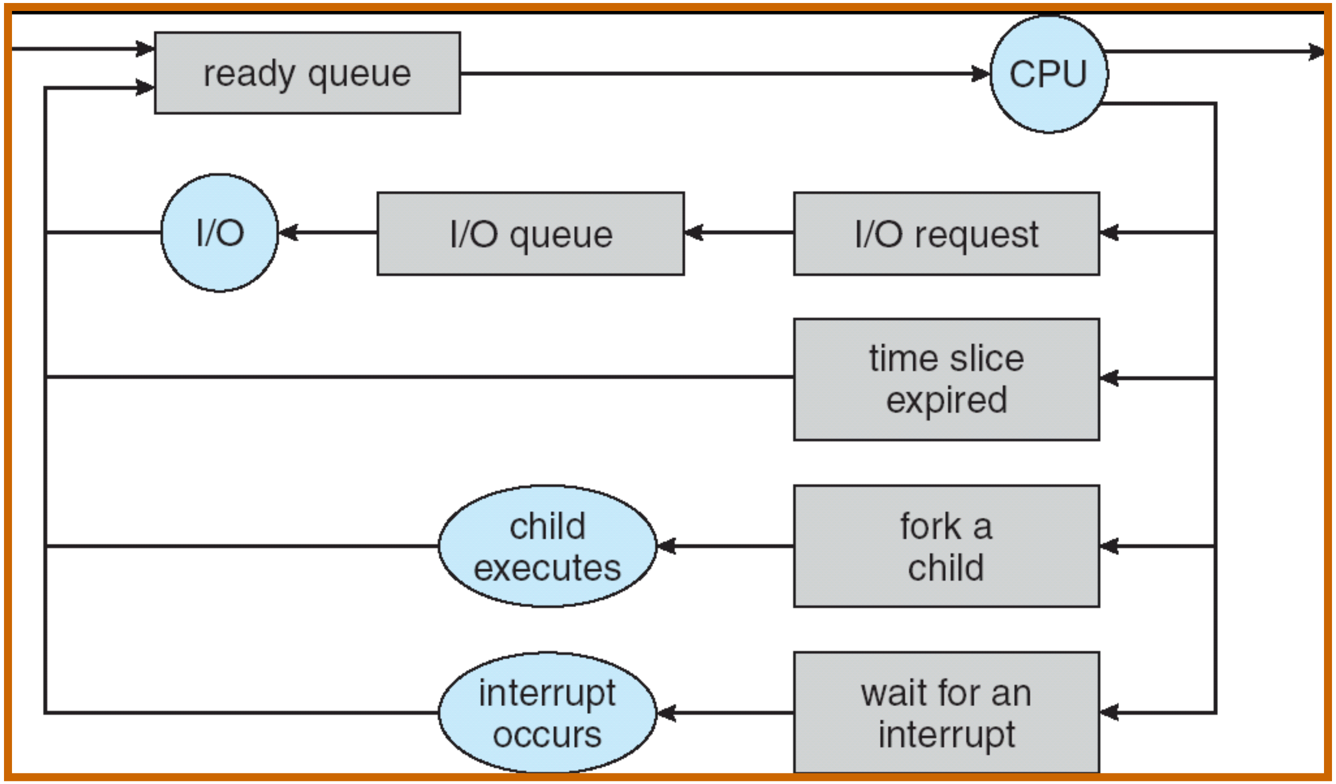
Process Control Block

CPU Switch From Process to Process



- **This is also called a "context switch"**
- **Code executed in kernel above is overhead**
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but... contention for resources instead

Process Scheduling



- **PCBs move from queue to queue as they change state**
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible