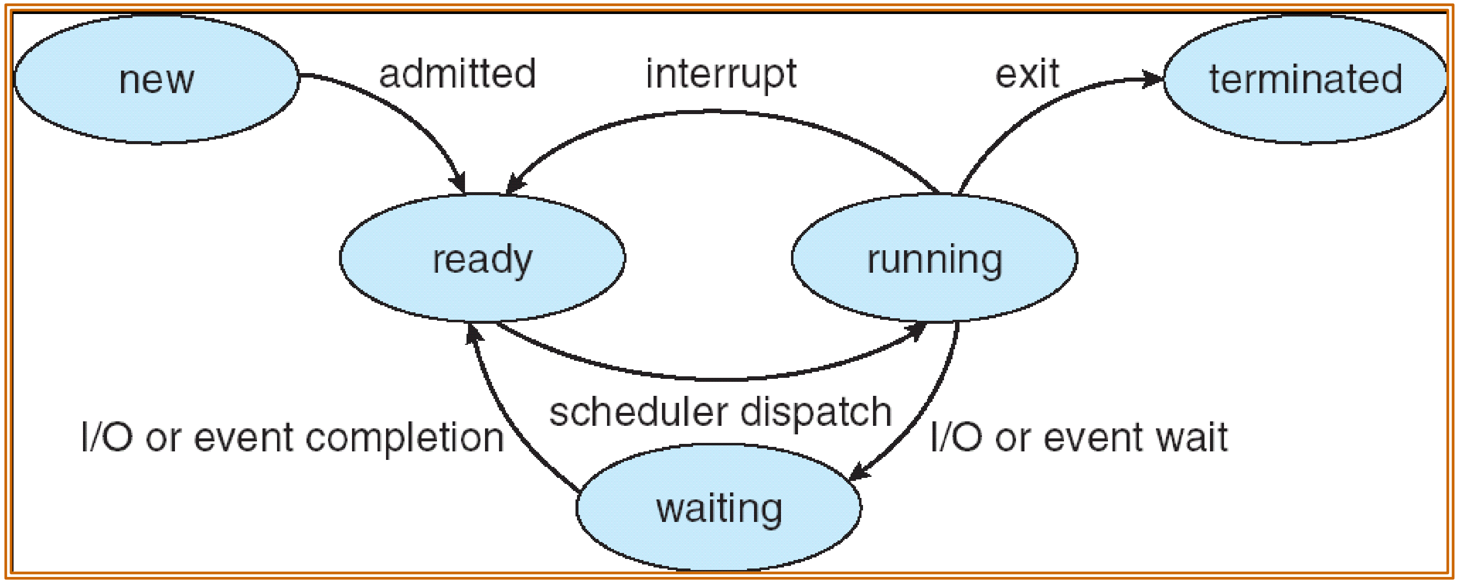
# Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

# 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

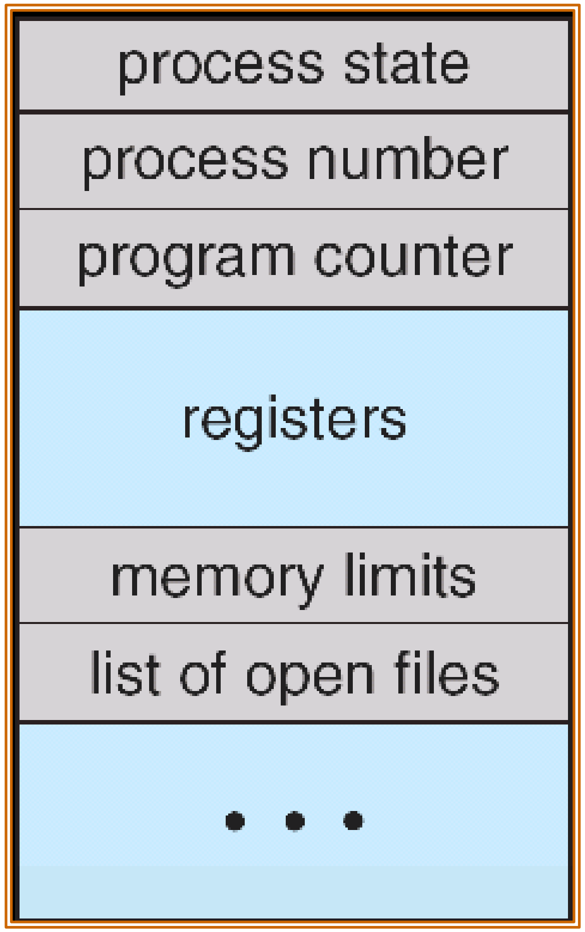


# Process Control Block (PCB)

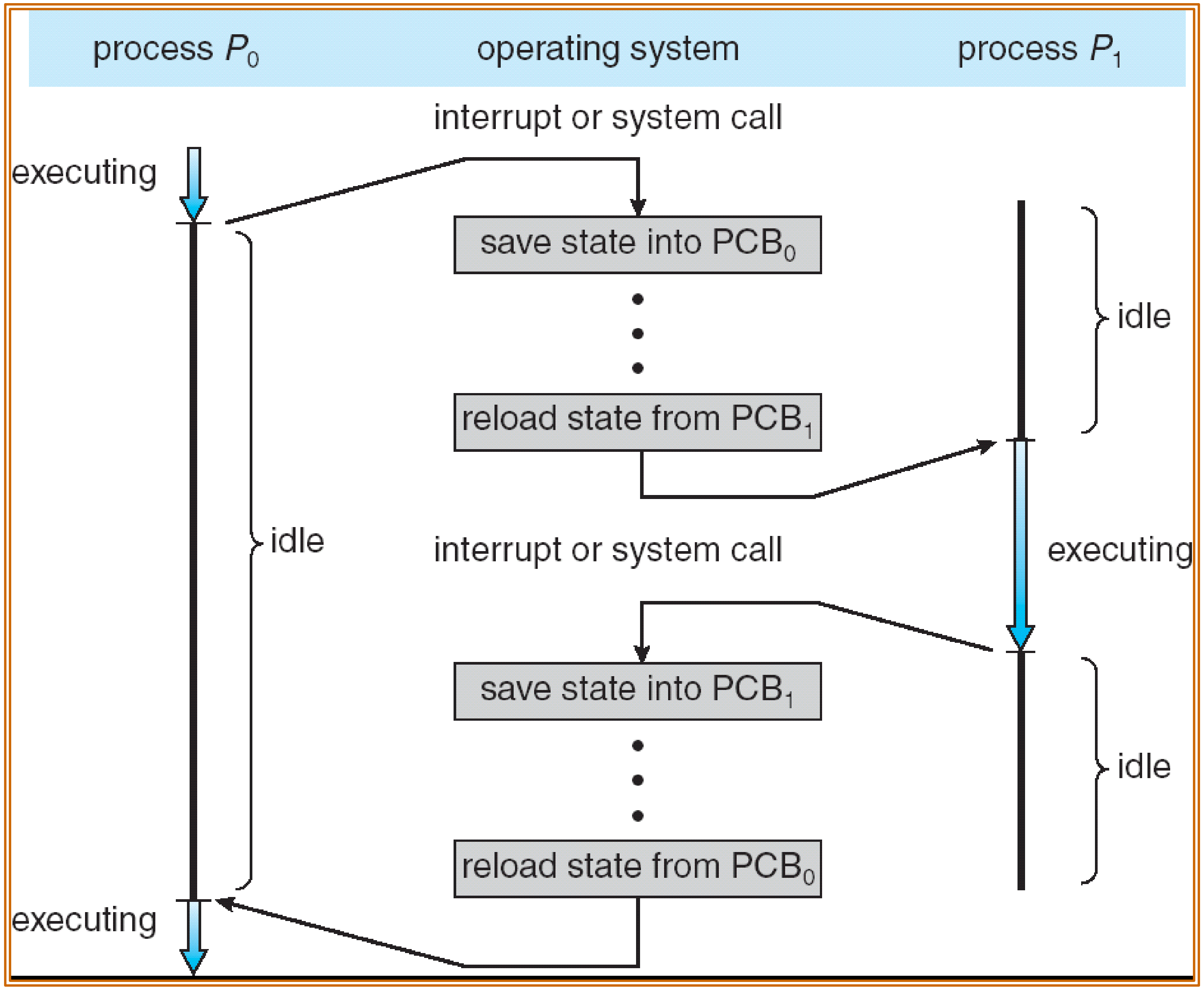
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

# Process Control Block (PCB)



# CPU Switch From Process to Process

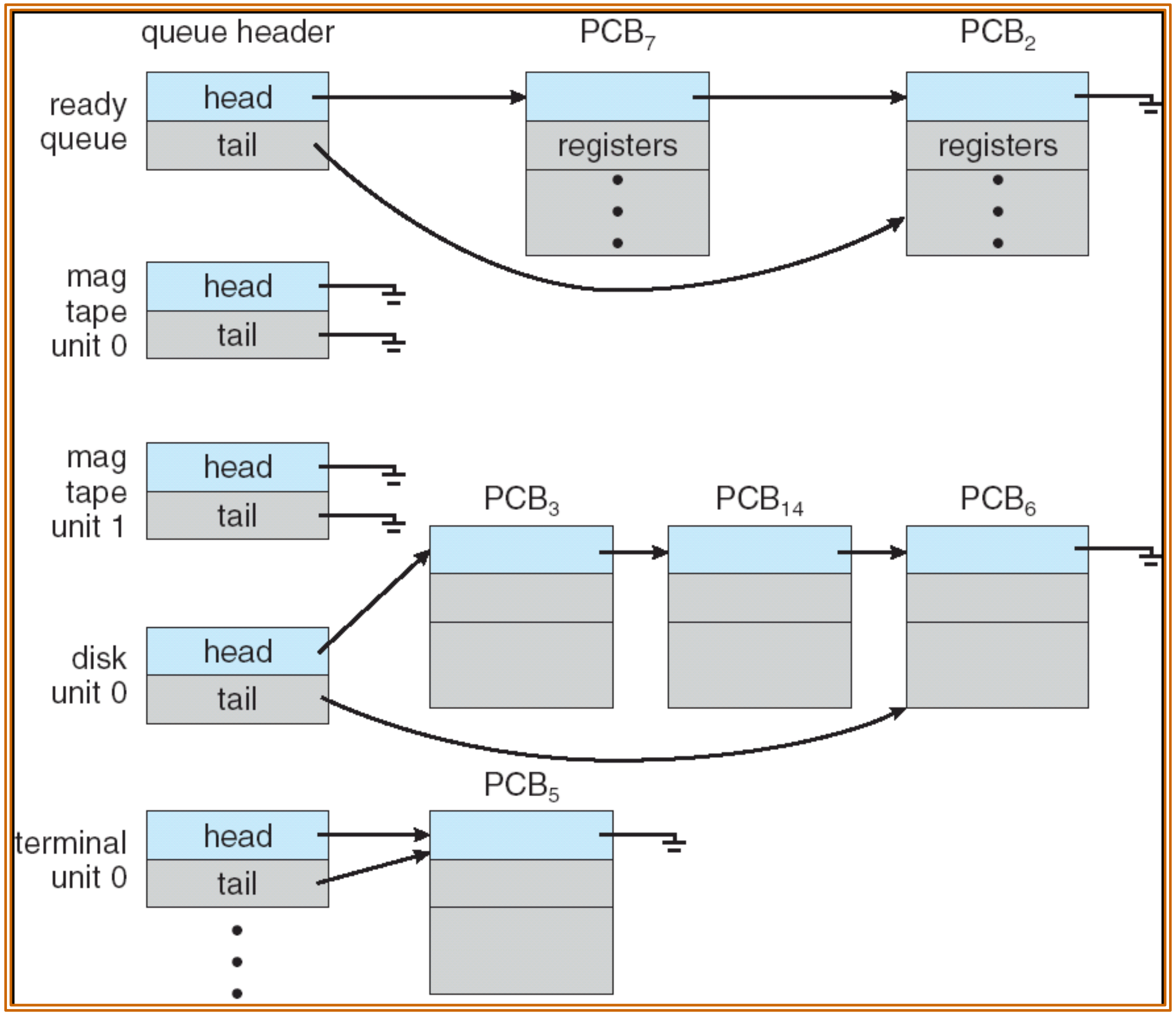


## Process Scheduling Queues

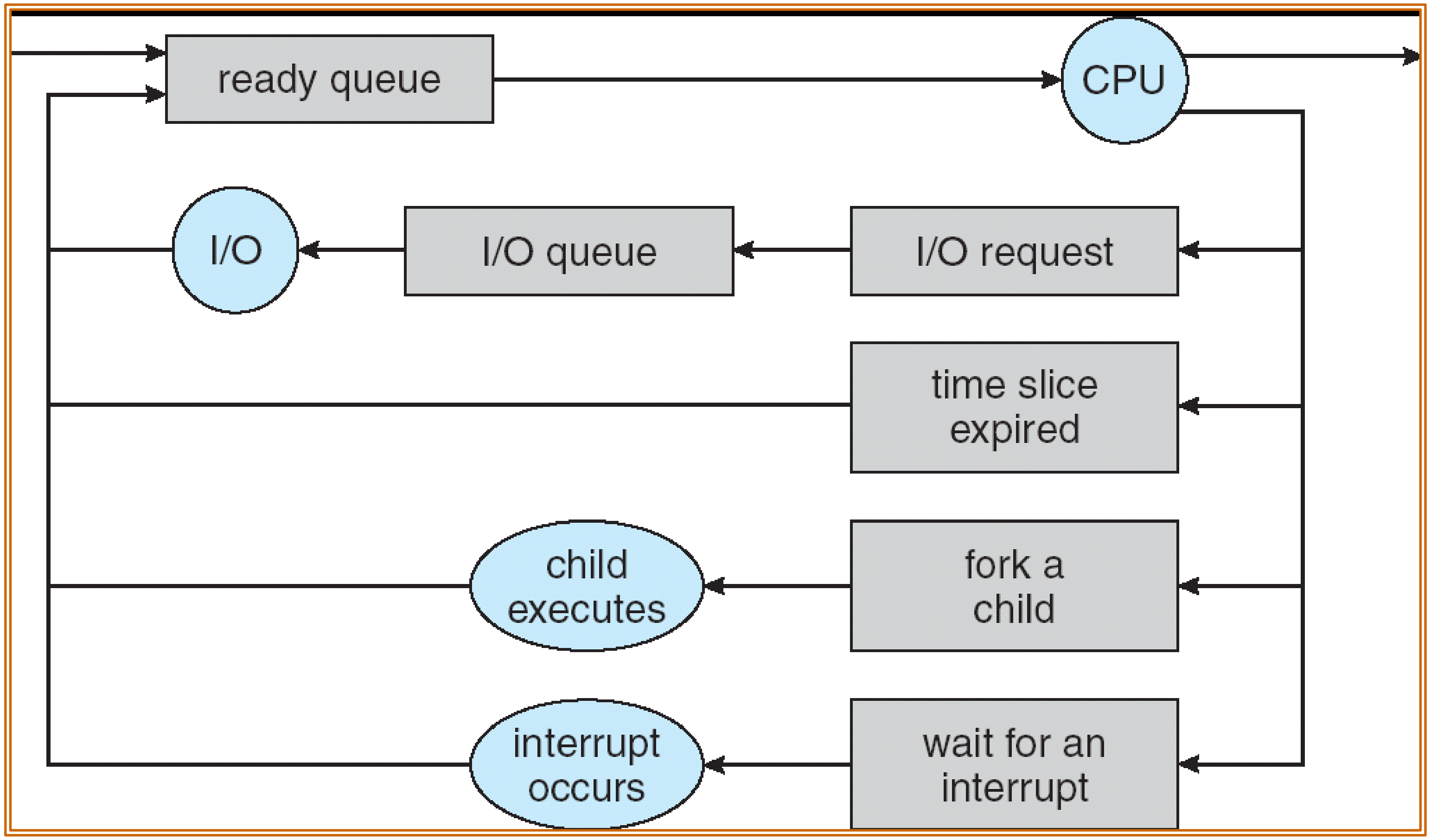
- *Job queue* – set of all processes in the system
- *Ready queue* – set of all processes ready and waiting to execute
- *Device queues* – set of processes waiting for an I/O device
- Process migration between the various queues

CODERINDEED

# Ready Queue And Various I/O Device Queues



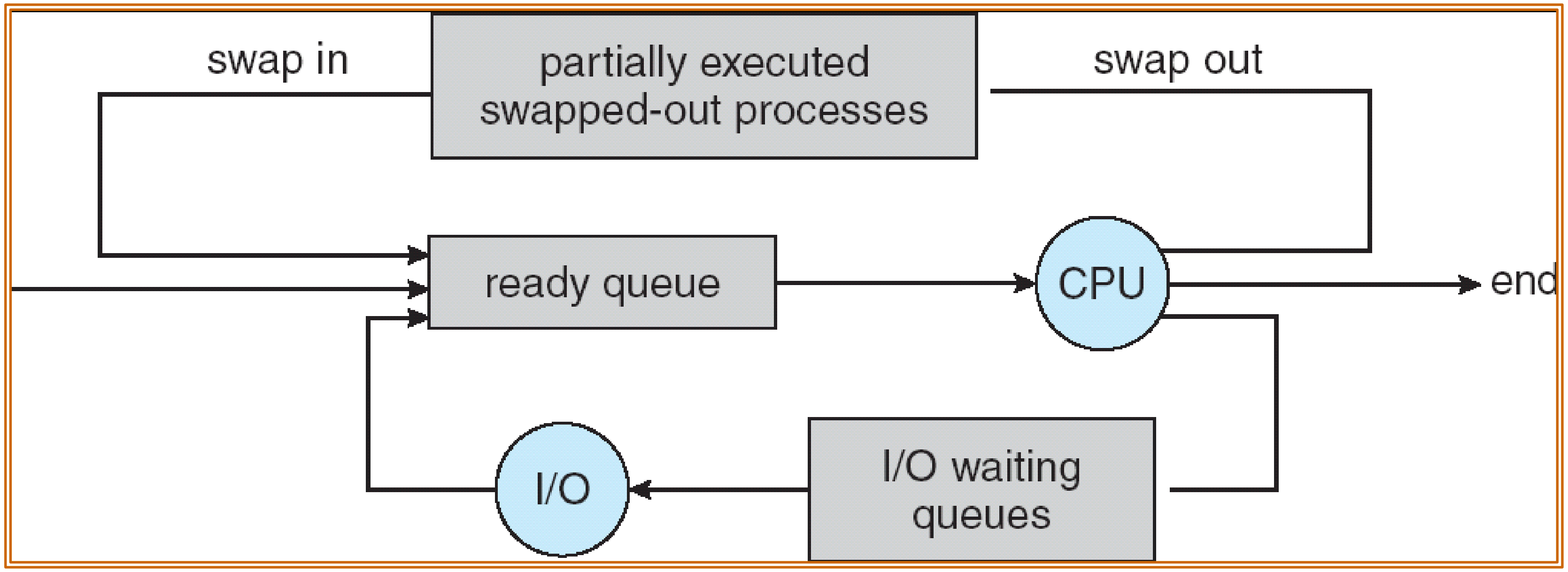
# Representation of Process Scheduling



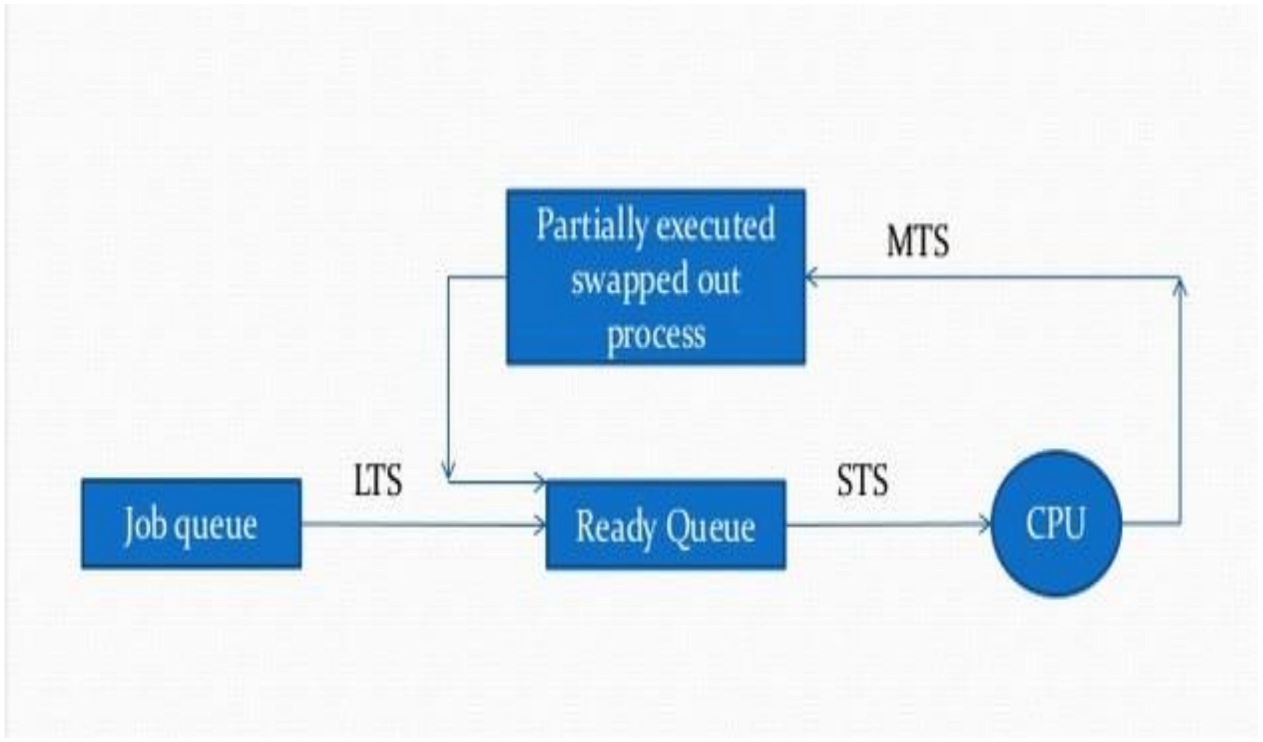
## Schedulers

- *Long-term scheduler* (or job scheduler) – selects which processes should be brought into the ready queue
- *Short-term scheduler* (or CPU scheduler) – selects which process should be executed next.

# Addition of Medium Term Scheduling



# Scheduling


Partially executed swapped out process
MTS
LTS
STS
Job queue
Ready Queue
CPU

## Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts
  - *CPU-bound process* – spends more time doing computations; few very long CPU bursts

## Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Dependent on hardware support

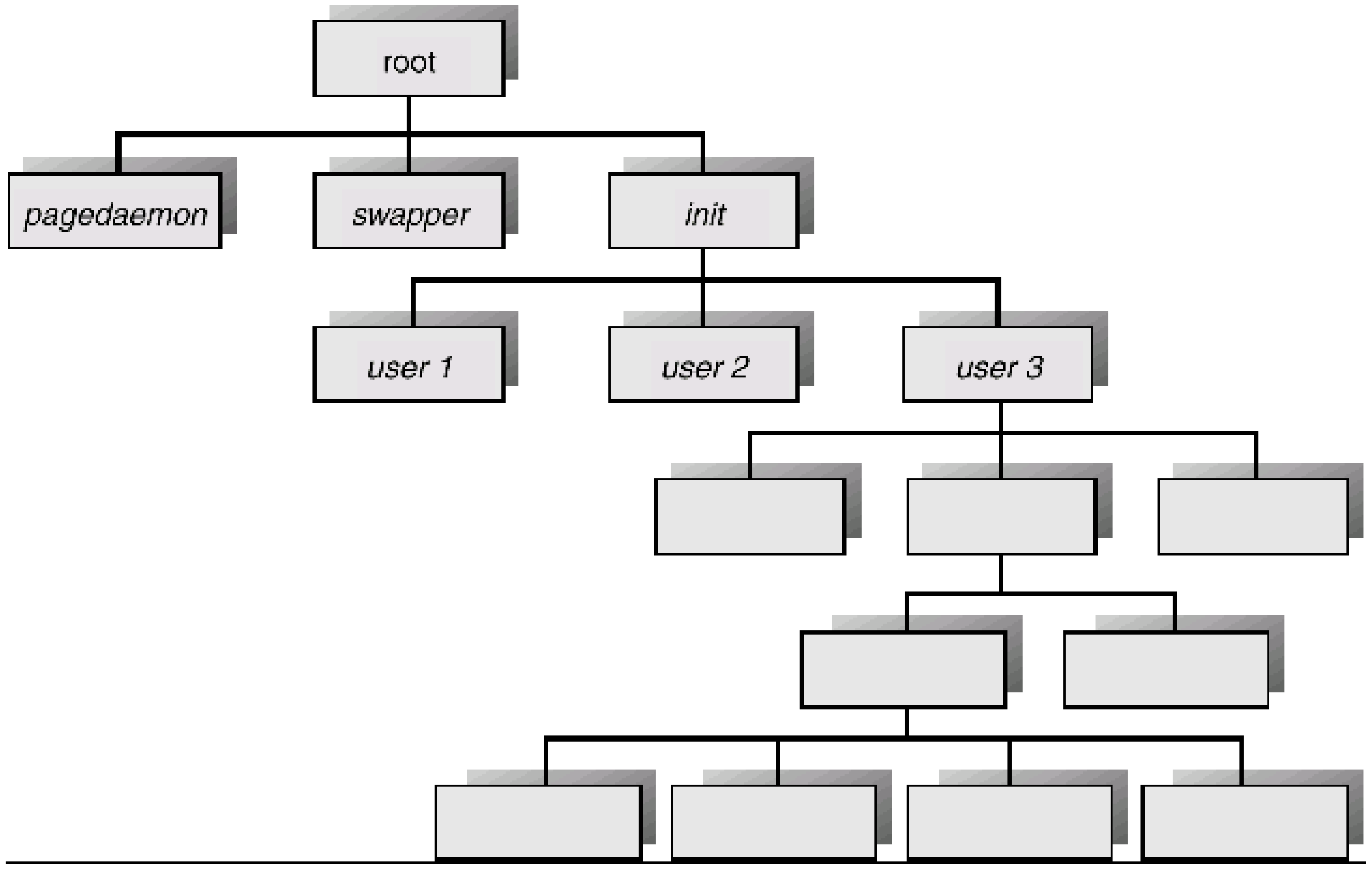
## Process Creation

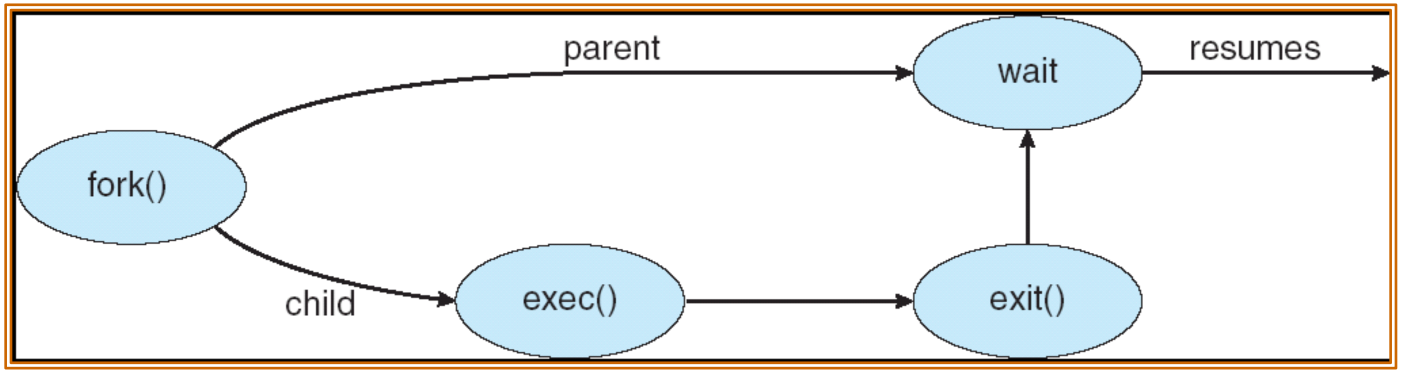
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

## Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

## A Tree of Processes On A Typical UNIX System




parent
wait
resumes
fork()
exec()
exit()
child

# Process Termination

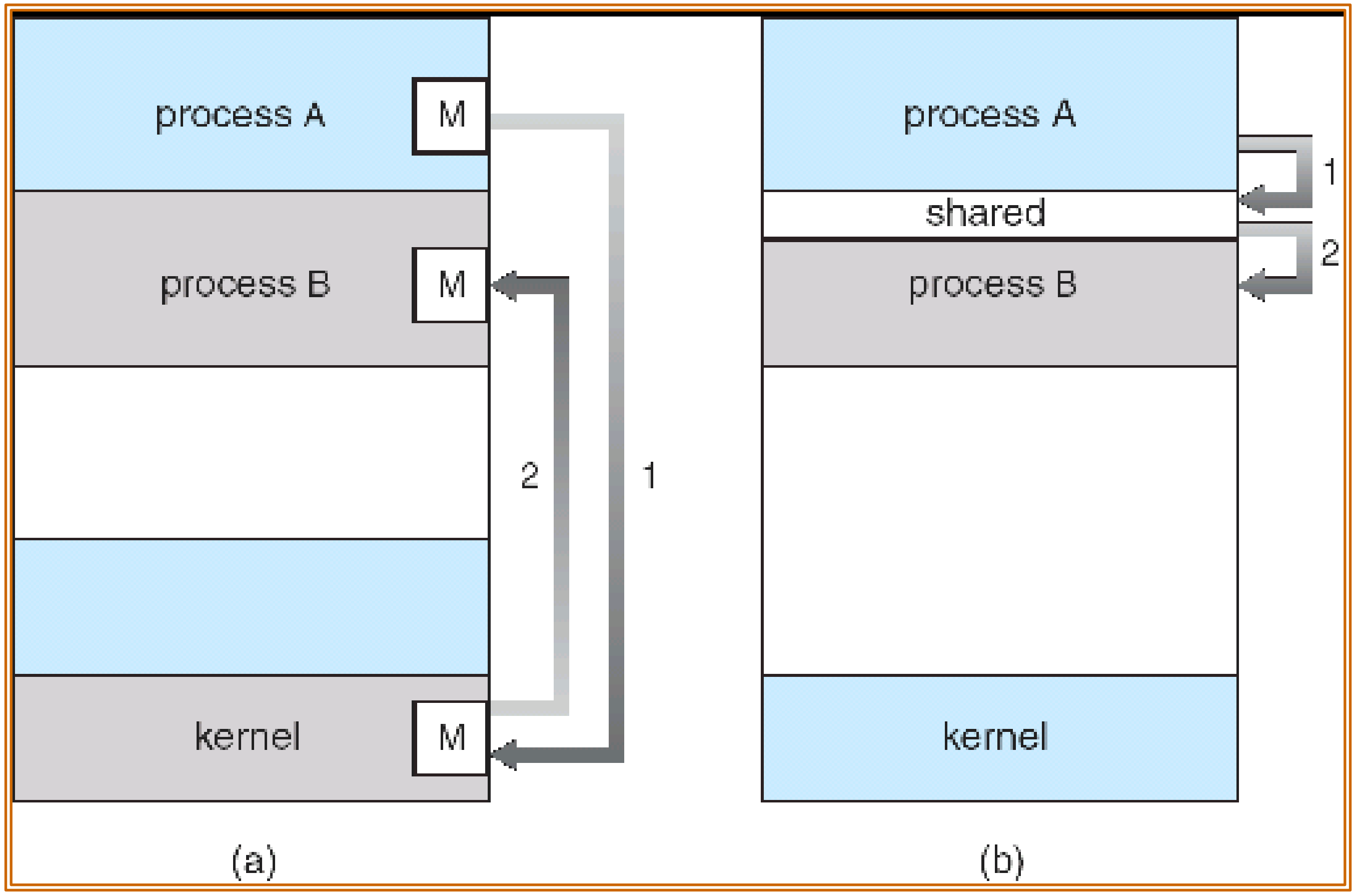
- Process executes last statement and asks the operating system to terminate it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

## Types of Processes

- *Independent* process cannot affect or be affected by the execution of another process
- *Cooperating* process can affect or be affected by the execution of another process
- Reasons for providing an environment that allows process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

## Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer. Consumer waits
  - *bounded-buffer* assumes that there is a fixed buffer size. Producer waits.


process A
M
process B
M
kernel
M
2
1
(a)
process A
shared
process B
kernel
1
2
(b)

## Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other by passing of messages.
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)

  Messages can be of 2 types: Fixed length, variable length.
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

- Several methods for logically implementing a link and the send/receive operations. For example:
  - Direct or indirect communication
  - Symmetric or asymmetric communication

## Implementation Questions

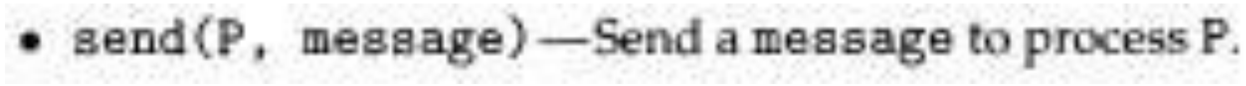
- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

## Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

- A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender.

- receive (id, message) - Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.



- `send(P, message)`—Send a message to process P.

# Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

## Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  **send**(*A, message*) – send a message to mailbox A

  **receive**(*A, message*) – receive a message from mailbox A

## Indirect Communication

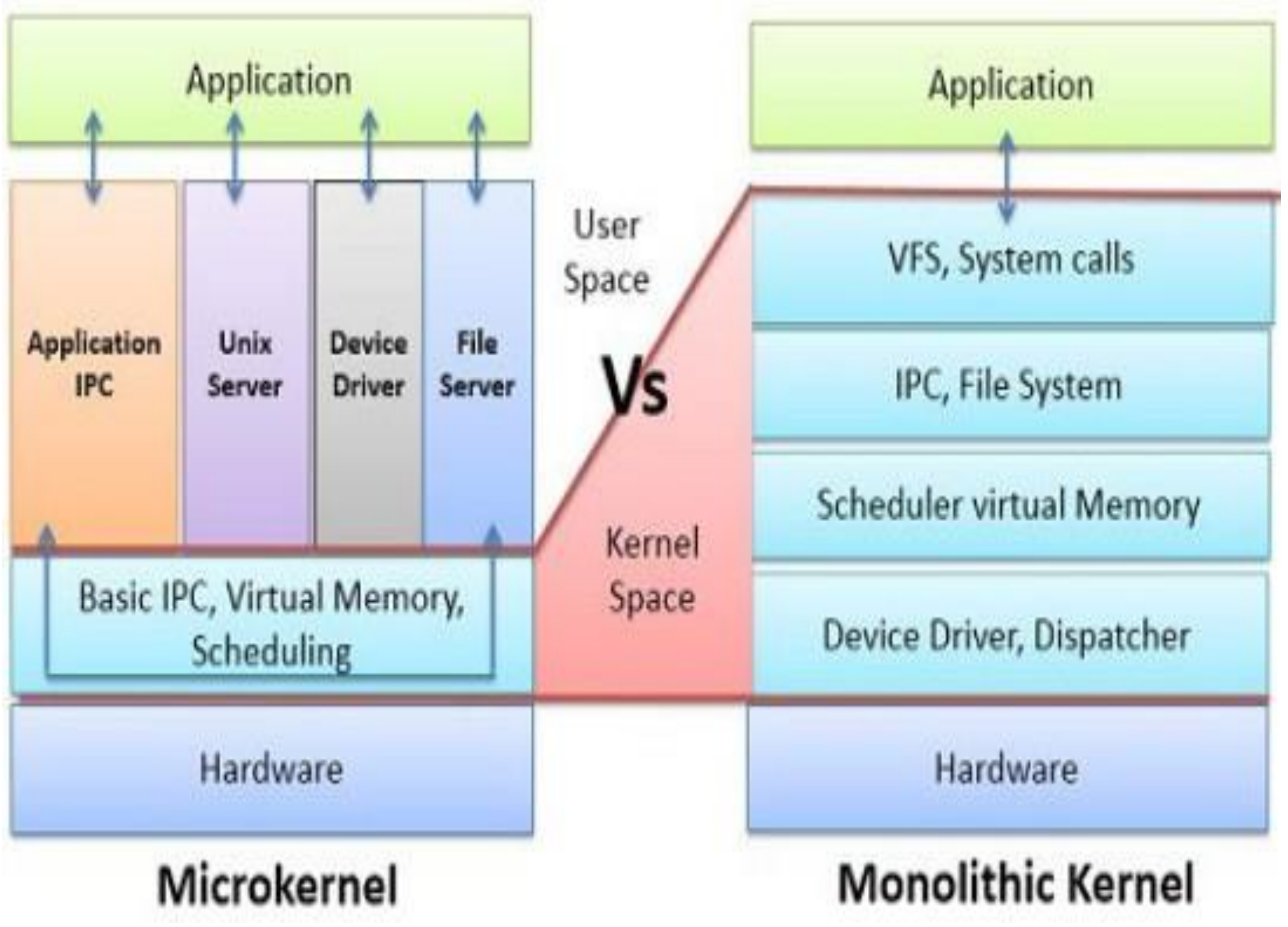
- Mailbox sharing
  - $P_1$, $P_2$, and $P_3$ share mailbox A
  - $P_1$, sends; $P_2$ and $P_3$ receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver.  Sender is notified who the receiver was.

## Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
    - **Blocking send** has the sender block until the message is received
    - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
    - **Non-blocking** send as the sender sends the message and continue
    - **Non-blocking** receive has the receiver receive a valid message or null

## Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits


Application
Application IPC
Unix Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware
Microkernel
User Space
Vs
Kernel Space
Application
VFS, System calls
IPC, File System
Scheduler virtual Memory
Device Driver, Dispatcher
Hardware
Monolithic Kernel

## Key Differences Between Microkernel and Monolithic Kernel

- The basic point on which microkernel and monolithic kernel is distinguished is that microkernel implement user services and kernel services in different address spaces and monolithic kernel implement both user services and kernel services under same address space
- The size of microkernel is small as only kernel services reside in the kernel address space. However, the size of monolithic kernel is comparatively larger than microkernel because both kernel services and user services reside in the same address space.

- **The execution of monolithic kernel is faster as the communication between application and hardware is established using the system call. On the other hands, the execution of microkernel is slow as the communication between application and hardware of the system is established through message passing**
- **It is easy to extend microkernel because new service is to be added in user address space that is isolated from kernel space, so the kernel does not require to be modified. Opposite is the case with monolithic kernel if a new service is to be added in monolithic**

- **Microkernel is more secure than monolithic kernel as if a service fails in microkernel the operating system remain unaffected. On the other hands, if a service fails in monolithic kernel entire system fails.**
- **Monolithic kernel designing requires less code, which further leads to fewer bugs. On the other hands, microkernel designing needs more code which further leads to more bugs.**