Chapter 3

Processes

Programs in execution.

Source: Slides from Silberchatz website

Chapter 3: Objectives

- To introduce the **notion of a process** -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including **scheduling, creation and termination, and communication**
- To explore **interprocess communication** using shared memory and message passing

Process Concept (1/1)

- An OS 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

Process Concept (1/2)

- Has **multiple parts:**
  - The **program code**, also called **text section**
  - Current activity including **program counter**, **processor registers**
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process Concept (1/3)

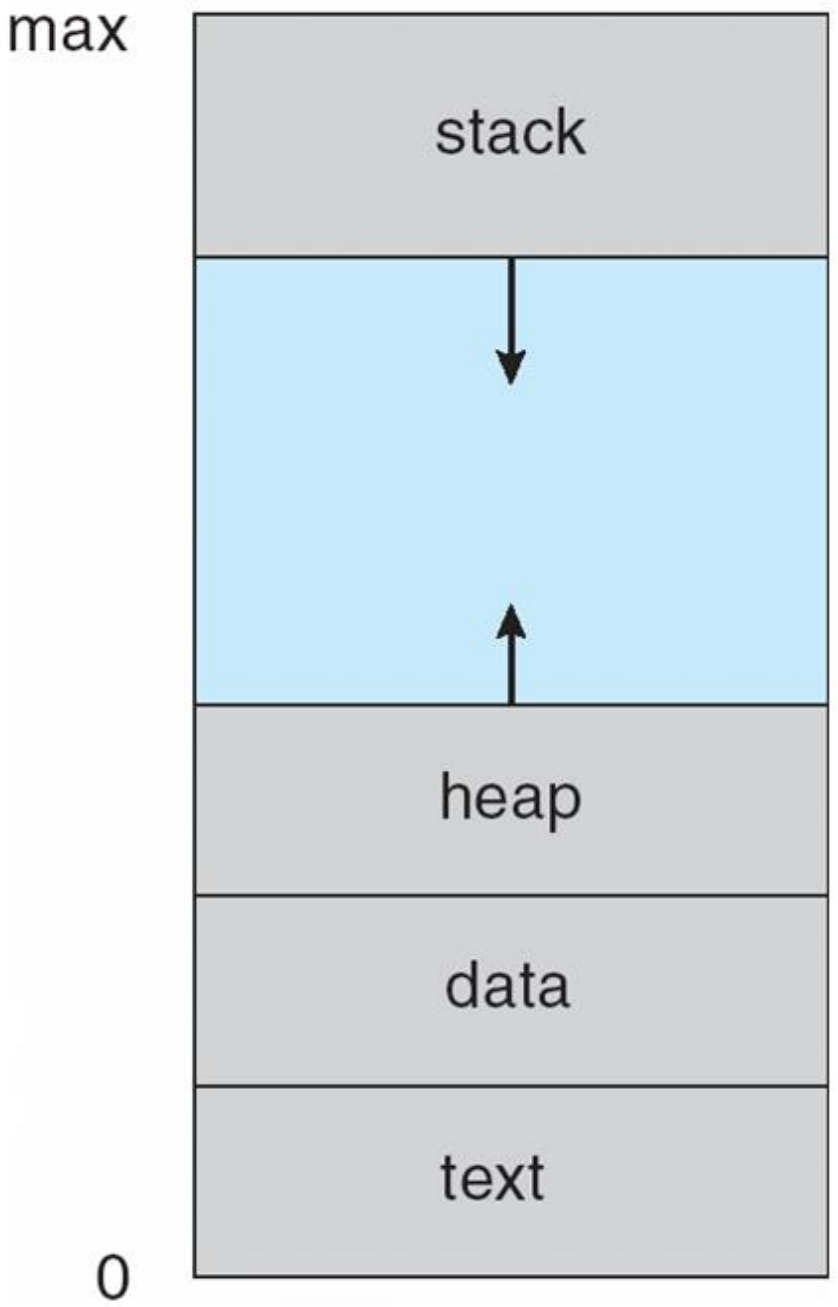
- **Program** is **passive entity** stored on disk (executable file)
- **Process** is **active entity.**
  - **Program becomes process when executable file loaded into memory**
- **Execution of program** started via GUI mouse clicks, command line entry of its name, etc
- **One program** can be **several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

BREAK.

Recap Questions (1): TRUE OR FALSE

1. Microkernel moves as functionalities into kernel space as possible. **False**
2. In layered approach of OS structure, bottom layer is called the hardware layer. **True**
3. Process is the passive entity. **False**
4. Heap are of memory is used for storing temporary data. **False**
5. Process is a program in execution. **True**

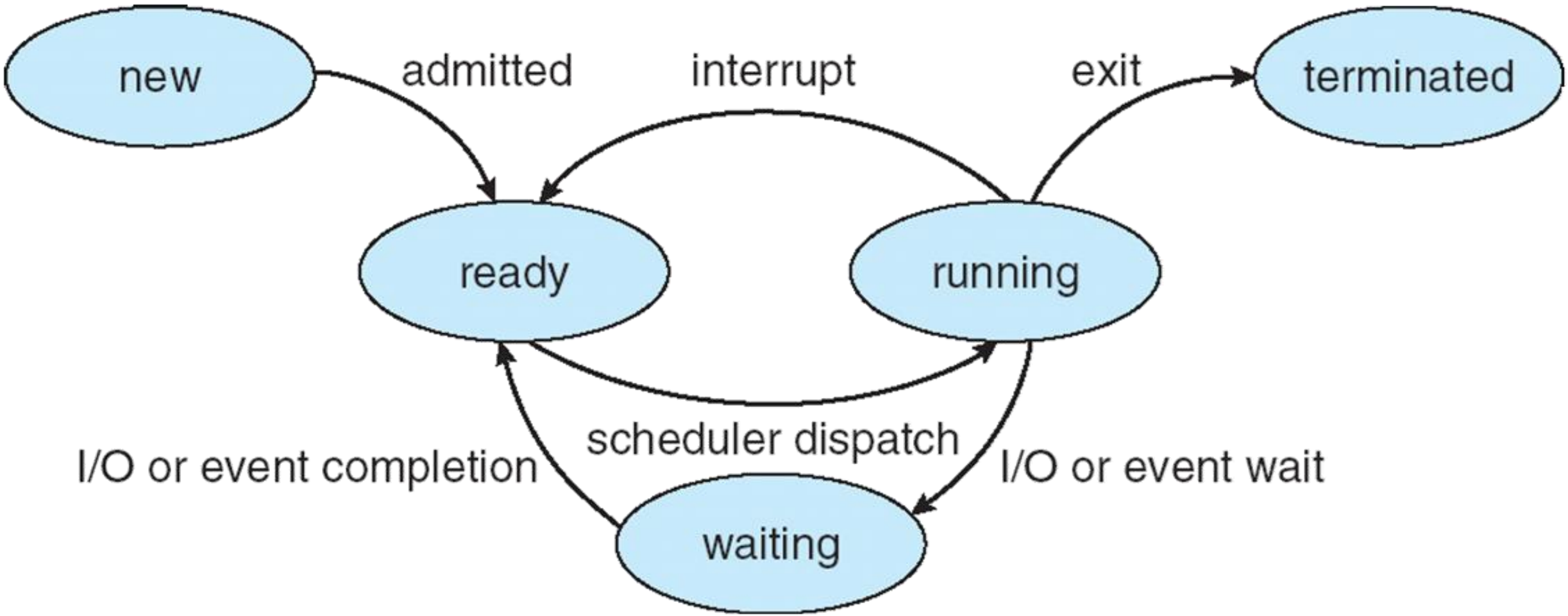
Process Concept (1/4)



Process State (2/1)

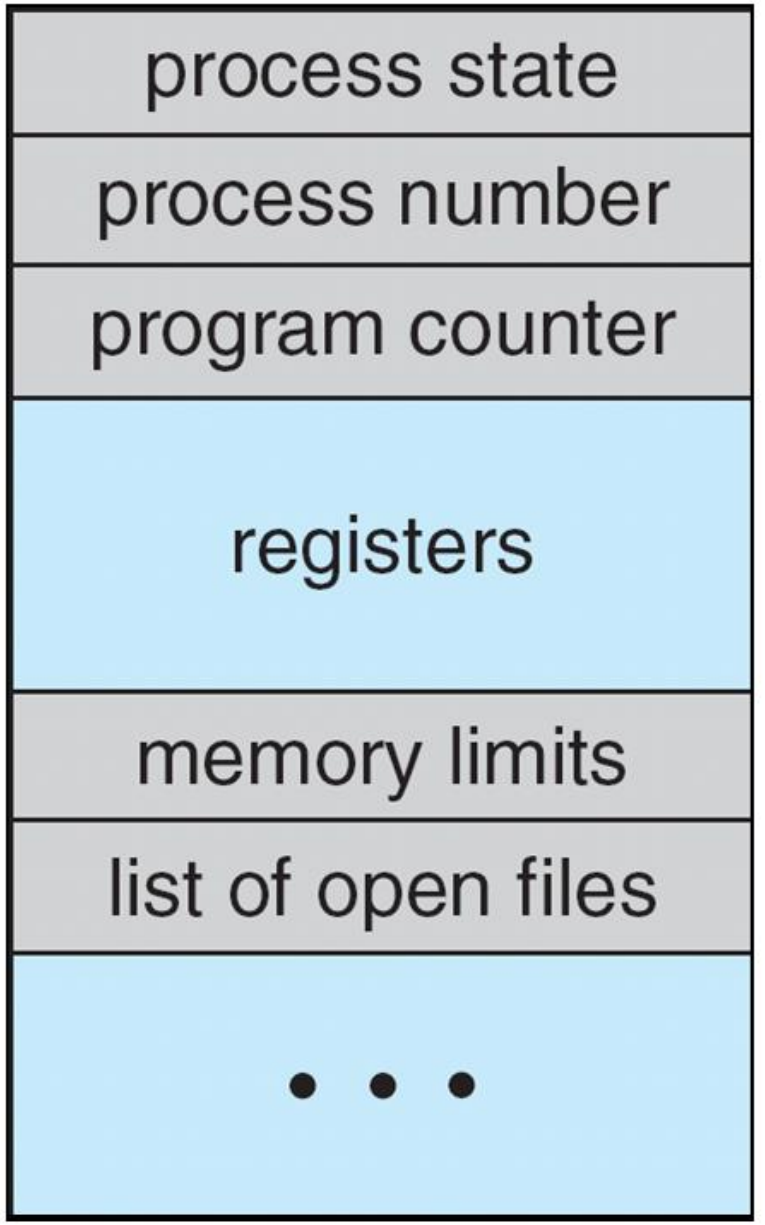
- As a **process *executes***, **it changes state**
  - **new:** The process is being created
  - **running:** Instructions are being executed
  - **waiting:** The process is waiting for some event to occur
  - **ready:** The process is waiting to be assigned to a processor
  - **terminated:** The process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

Process State (2/2)

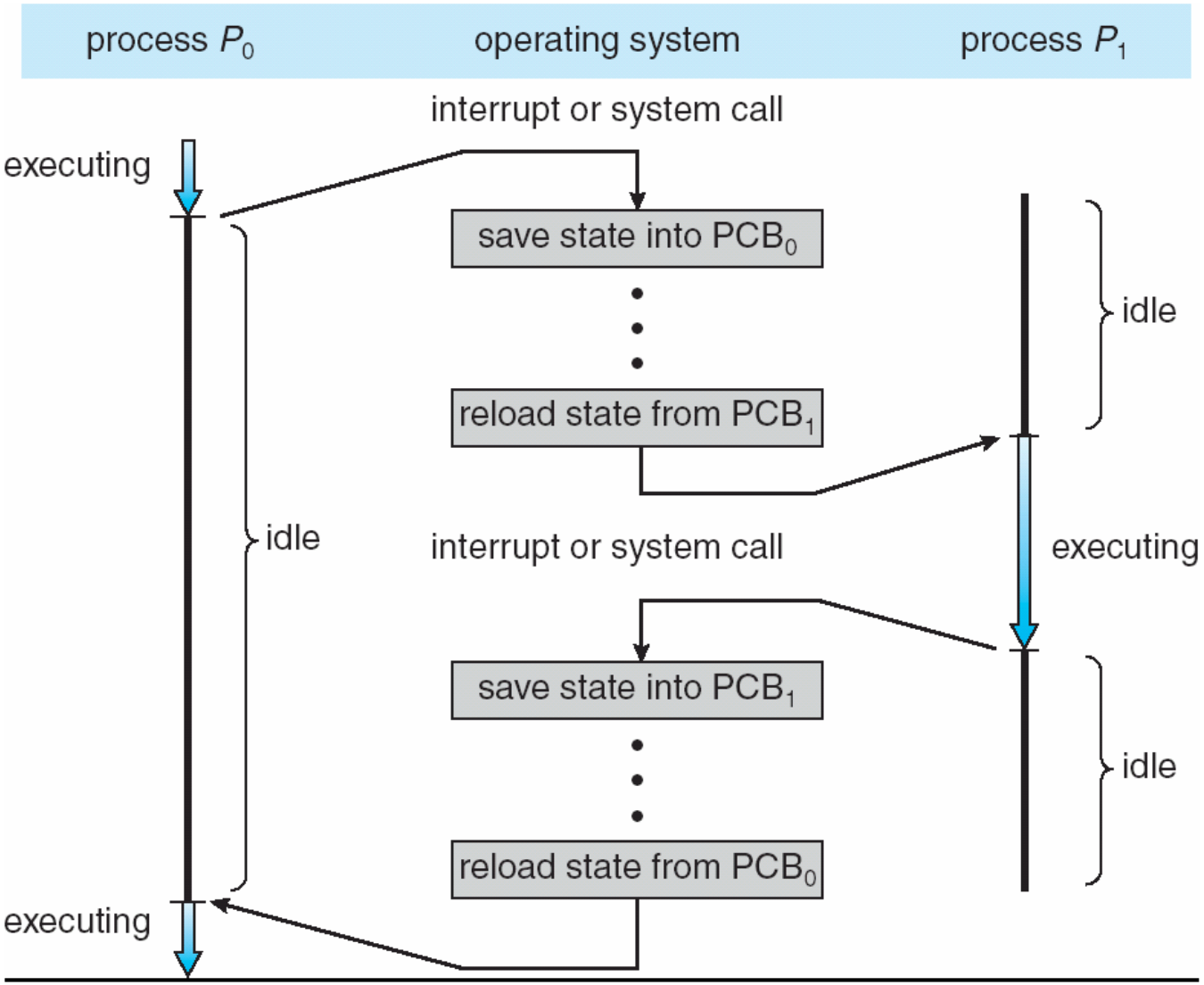


Process Control Block (PCB) (1/1)

- **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



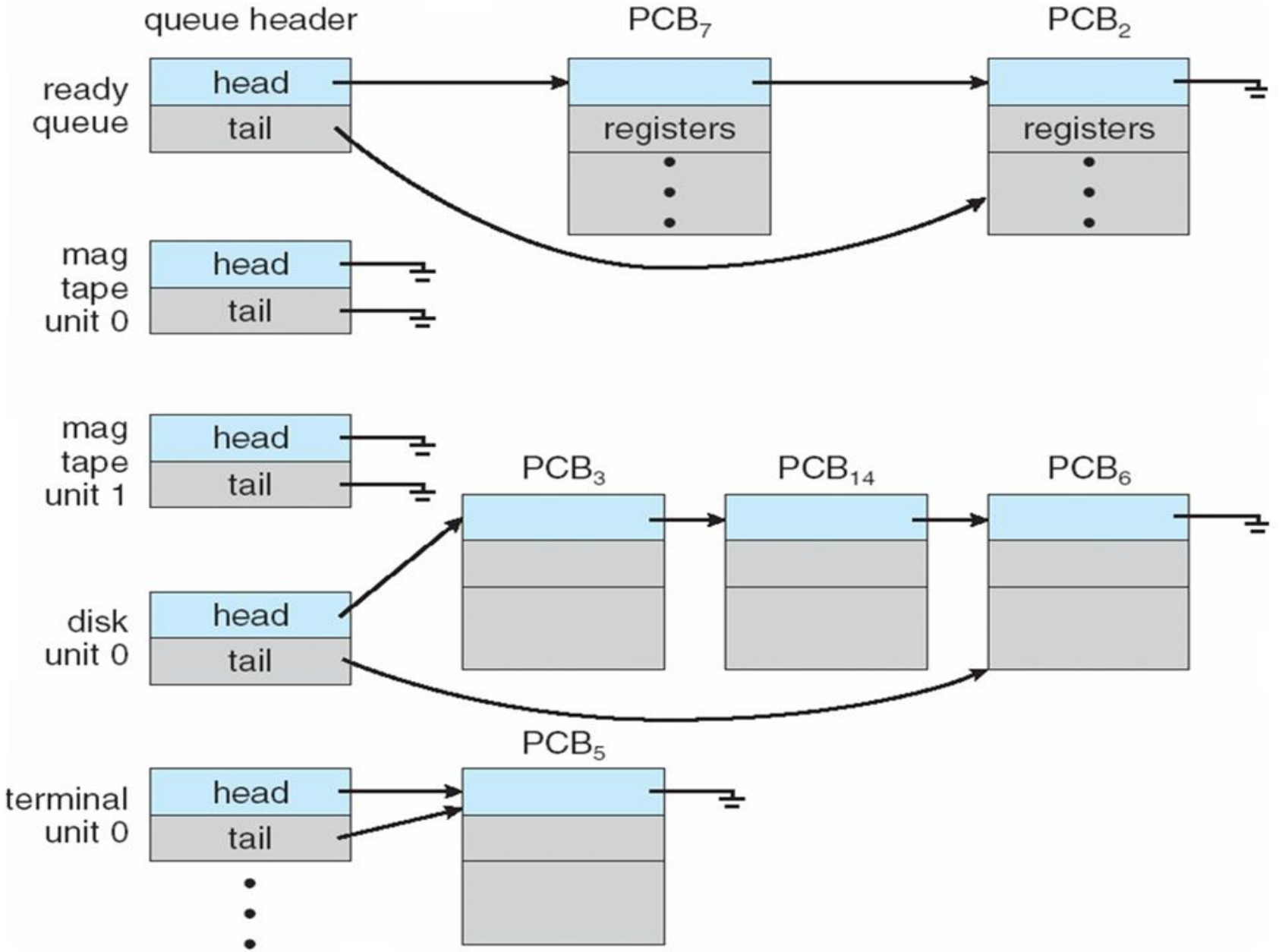
CPU Switch From Process to Process



Process Scheduling

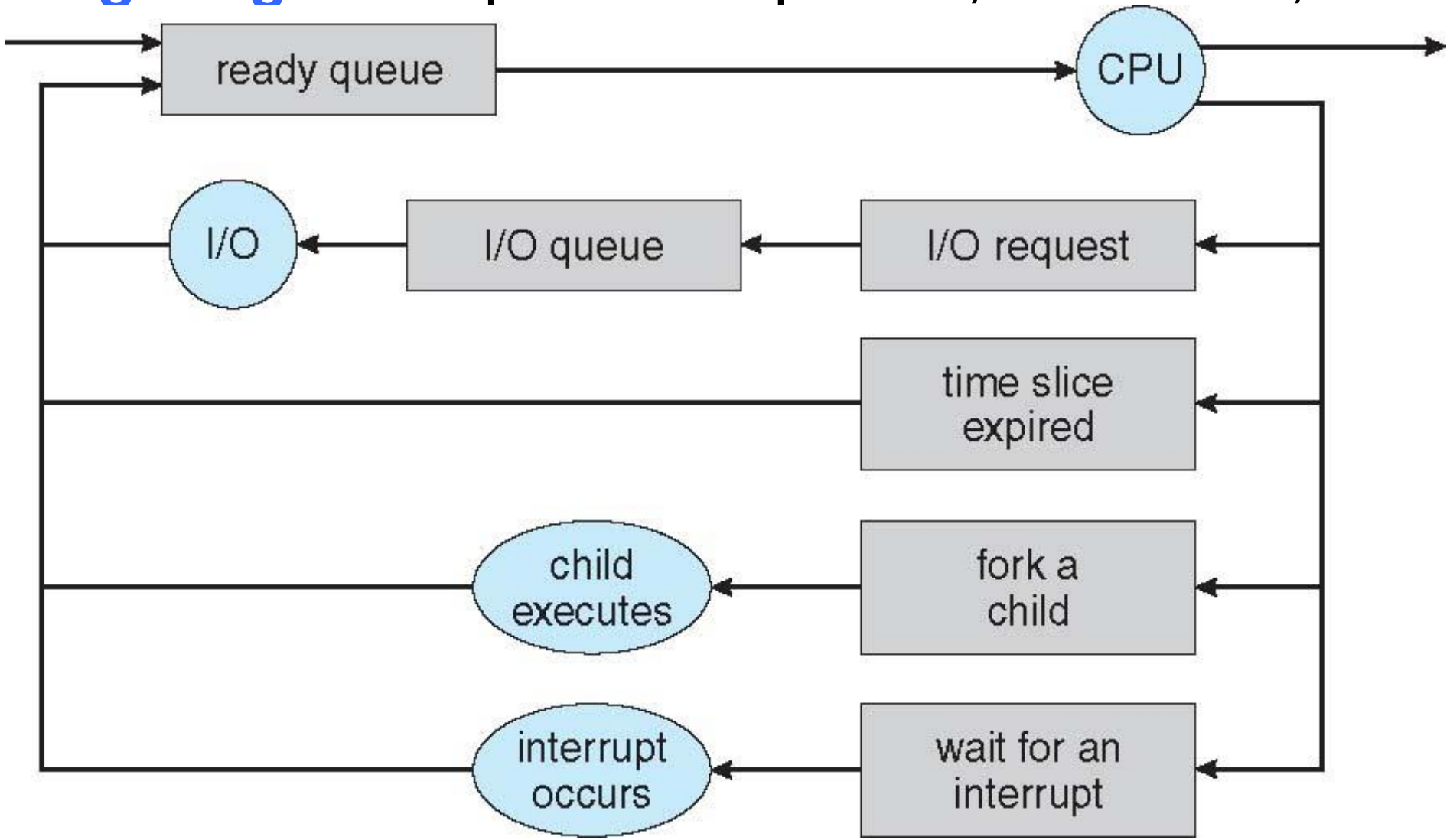
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler:** selects among available processes for next execution on CPU
- Maintains ***scheduling queues of processes***
  - **Job queue:** set of all processes in the system
  - **Ready queue:** set of all processes 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

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows



Schedulers

- **Short-term scheduler** (or ***CPU scheduler):*** selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) □ (must be fast)
- **Long-term scheduler** (***or job scheduler):*** selects which processes should be brought into the ready queue
  - Is invoked infrequently (seconds, minutes), so may be slow
  - Controls the degree of multiprogramming

Throughput: no. of processes completed in unit time

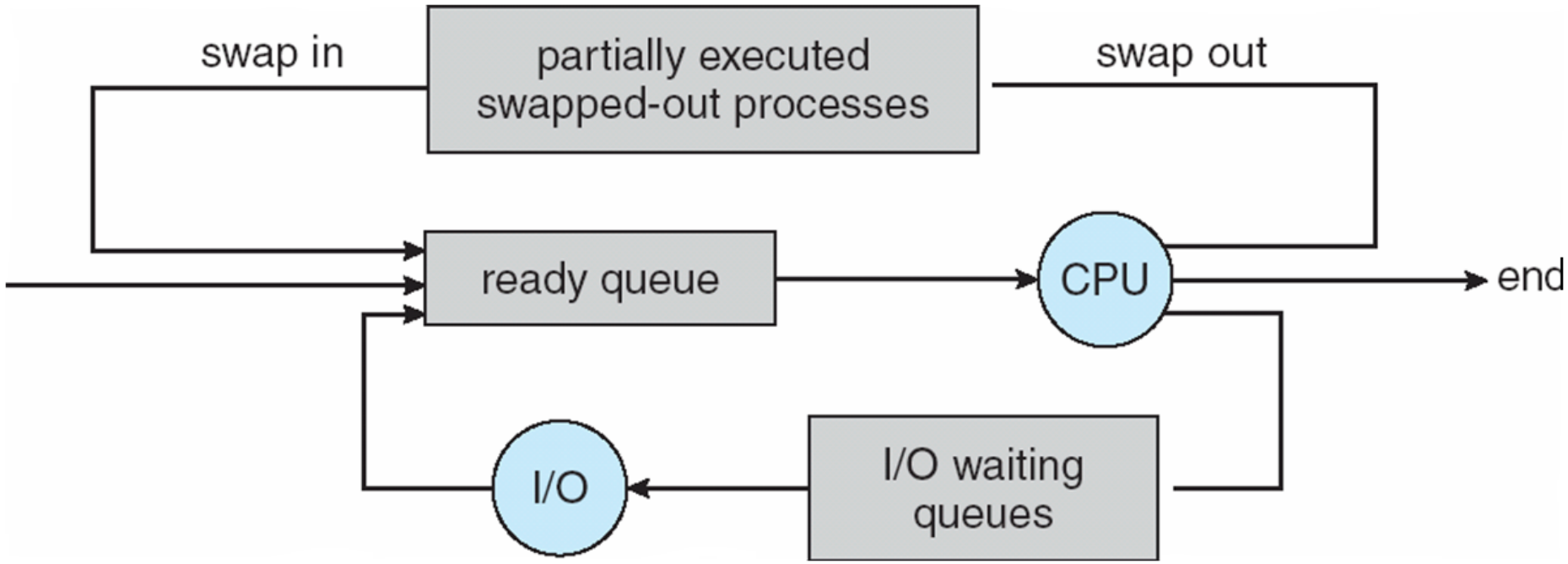
Schedulers

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - *Example: DBMS*
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
    - *Example: Weather*
- Long-term scheduler strives for good ***process mix***

Medium Term Scheduler (Swapper)

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

Role when throughput is less



Context Switching

- 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 via a context switch
- **Context** of a process represented in the PCB
- **Context-switch time is overhead;** the system does no useful work while switching:
  - The more complex the OS and the PCB ==> the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU => multiple contexts loaded at once

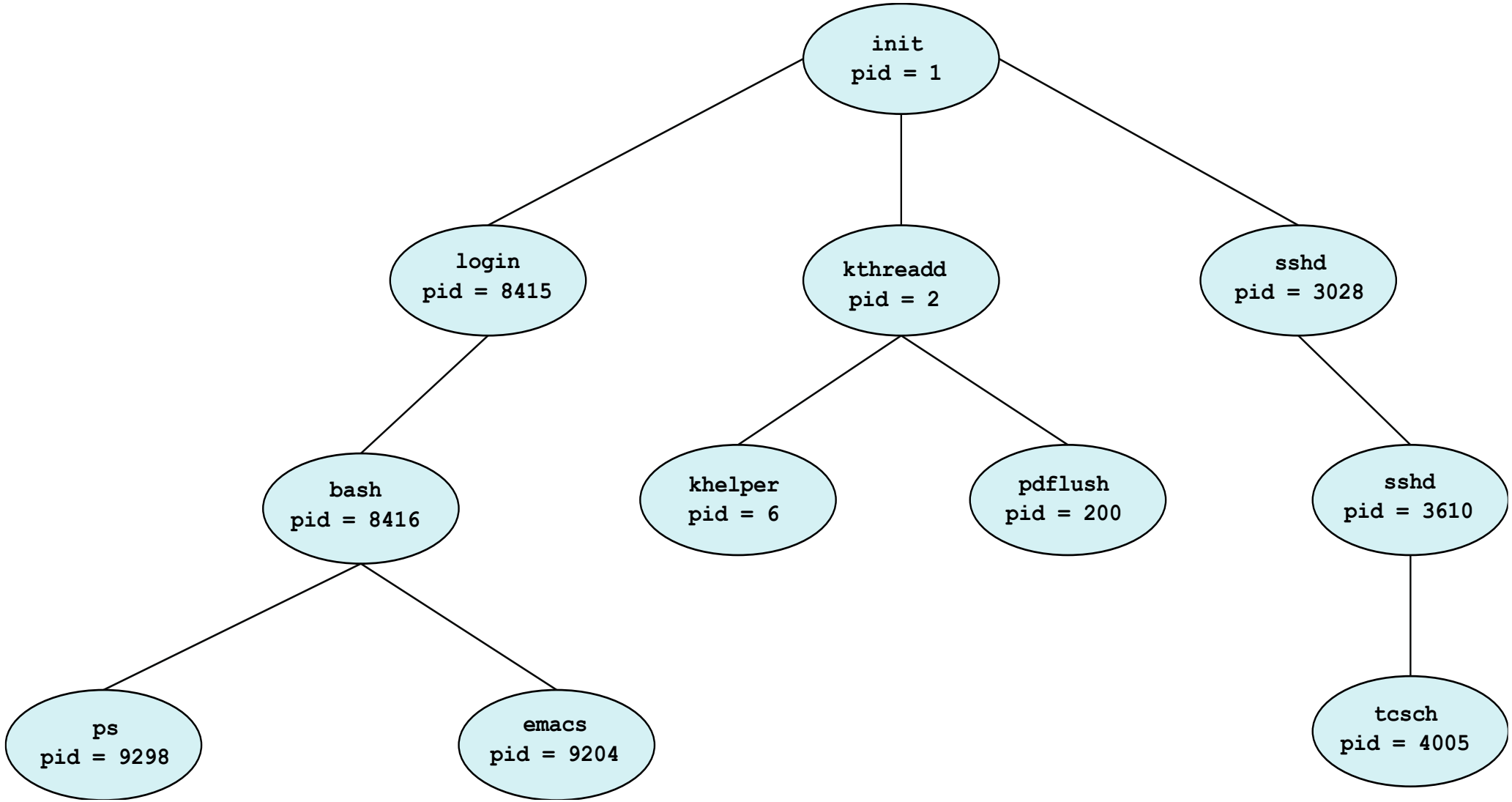
Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - Others

Process Creation.

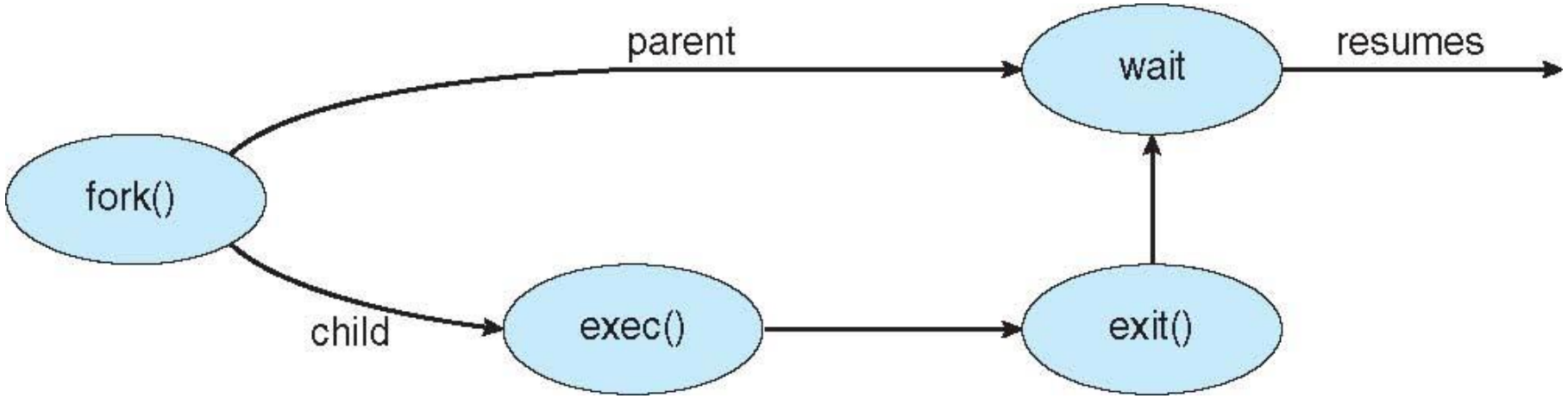
- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Parent and child share no resources
- Execution options
    - Parent and children execute concurrently
    - Parent waits until children terminate

A Tree of Processes in Linux



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



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

Process Termination.

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()**system call**.** The call returns status information and the pid of the terminated process

pid = wait(&status);

- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait** , process is an **orphan**

How to fix any computer

The Oatmeal http://theoatmeal.com

Windows

Step 1. Reboot

Did that fix it?

No? Proceed to step 2

Step 2.

Format hard drive.
Reinstall Windows.

Lose all your files. Quietly weep.

Apple

Step 1. Take it to an Apple store.

Did that fix it?

No? Proceed to step 2

Step 2. Buy a new Mac.

Overdraw your account. Quietly weep.

Linux

Step 1.

Learn to code in C++. Recompile the kernel. Build your own microprocessor out of spare silicon you had lying around. Recompile the kernel again. Switch distros. Recompile the kernel again but this time using a CPU powered by refracted light from Saturn. Grow a giant beard. Blame Sun Microsystems. Turn your bedroom into a server closet and spend ten years falling asleep to the sound of whirring fans. Switch distros again. Abandon all hygiene. Write a regular expression that would make other programmers cry blood. Learn to code in Java. Recompile the kernel again (but this time while wearing your lucky socks).

Did that fix it?

No? Proceed to step 2

Step 2.

Revert back to using Windows or a Mac.

Quietly weep.

Recap Questions (2)

1. Which system call does the parent process invoke to terminate a child process?
2. What do you call processes that spend more time doing I/O than computations in short CPU bursts?
3. What is the child process referred to as if the parent process terminates without invoking wait() system call?
4. The set of all processes in the system is stored in the ready queue (Yes or No)?
5. What does the PC or the program counter segment in the PCB hold?

Interprocess Communication

- Processes executing concurrently within a system may be ***independent or cooperating***
- **Cooperating process** can affect or be affected by other processes, including sharing data
- **Reasons for cooperating processes:**
    - Information sharing
    - Computation speedup
    - Modularity
    - Convenience
- **Cooperating processes need interprocess communication (IPC)**
- Two models of IPC
    - **Shared memory**
    - **Message passing**

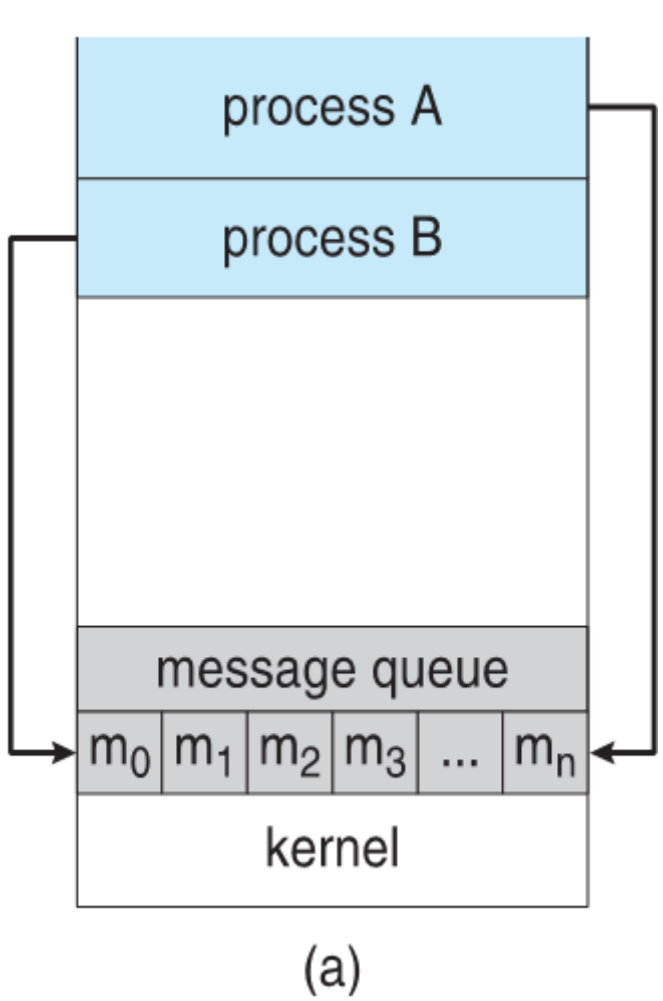
Communications Models

(b) Message passing

Smaller amounts of data, no conflicts.

Implemented using system calls.

Kernel intervention



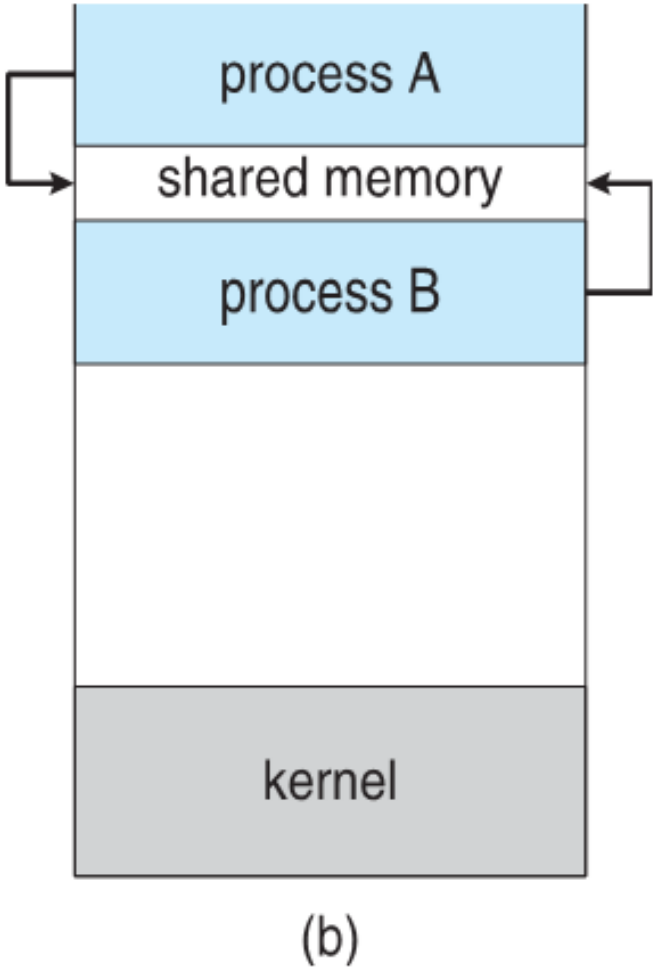
(a) shared memory

Faster.

system calls only to establish shared memory

No kernel assistance

Cache coherency



Cooperating 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
- Advantages of 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
- Two types of buffer
  - **unbounded-buffe**r places no practical limit on the size of the buffer (limit on consumer)
  - **bounded-buffer** assumes that there is a fixed buffer size (limit of producer & consumer)

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded Buffer - Producer

```
item next_produced;
while (true) {
     /* produce an item in next produced */
     while (((in + 1) % BUFFER_SIZE) == out)
          ; /* do nothing */
     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer - Consumer

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

IPC – Shared Memory

- An **area of memory shared** among the processes that wish to communicate
- The **communication** is **under the control** of the **users** processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

IPC- Message Passing (1/3)

- Mechanism for processes to communicate and to synchronize their actions
- **Message system:** processes communicate with each other ***without resorting to shared variables***
- IPC facility provides two operations:
  - **send(message)**
  - **receive(message)**
- The message size is either **fixed** or **variable**

IPC- Message Passing (2/3)

- If processes **P and Q wish to communicate,** they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- **Implementation issues:**
  - 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?

IPC- Message Passing (3/3)

- **Implementation of communication link**
- **Physical:**
  - Shared memory
  - Hardware bus
  - Network
- **Logical:**
  A. Direct or indirect
  B. Synchronous or asynchronous
  C. Automatic or explicit buffering

A. Direct Communication

- **Naming:** Processes must name each other explicitly:
  - **`send` (*P, message*):** send a message to process P
  - **`receive`(*Q, message*):** receives 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
- *Shows symmetry in addressing*

A. Direct Communication

- *Shows asymmetry in addressing*
- Only sender names the recipient
  - **`send` (*P, message*):** send a message to process P
  - **`receive`(*id, message*):** receives message from any process, variable id is set to the name of process with which communication has taken place

A. Indirect Communication (1/3)

- 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

A. Indirect Communication(2/3)

- **Who owns the mailbox?**
- **Operations**
  - create a new mailbox (port)
  - 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

A. Indirect Communication (3/3)

- **Mailbox sharing**
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive() from A
  - 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.

B. Synchronization (1/1)

- Message passing may be either **blocking or non-blocking**
- **Blocking** is considered **synchronous**
  - Blocking send -- the sender is blocked until the message is received
  - Blocking receive -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - Non-blocking send -- the sender sends the message and continue
  - Non-blocking receive -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
- If both send and receive are blocking, we have a **rendezvous**

B. Synchronization (1/2)

- Producer-consumer becomes trivial

```
message next_produced;
        while (true) {
            /* produce an item in next produced */
        send(next_produced);
        }
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

C. Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways:
  1. **Zero capacity:** no messages are queued on a link. 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


End of Chapter 3

Thank you