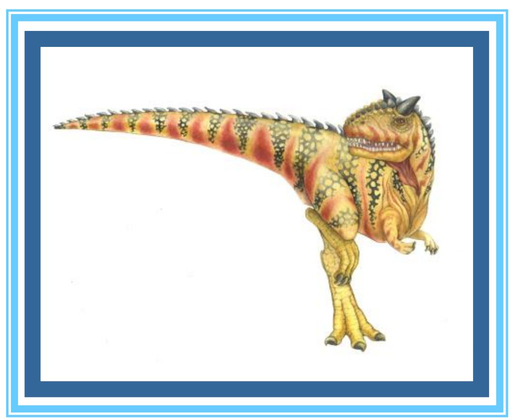
Chapter 2: Processes



Introduction

- A is a program in execution and process execution must progress in sequential fashion
- A process will need certain resources—such as CPU time, memory, files, and I/O devices — to accomplish its task.
- These resources are allocated to the process either when it is created or while it is executing.
- Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code.
- All these processes may execute concurrently.
- The operating system is responsible for several important aspects of process management:
  - creation and deletion of both user and system processes
  - scheduling of processes
  - provision of mechanisms for synchronization, communication, and deadlock handling for processes.

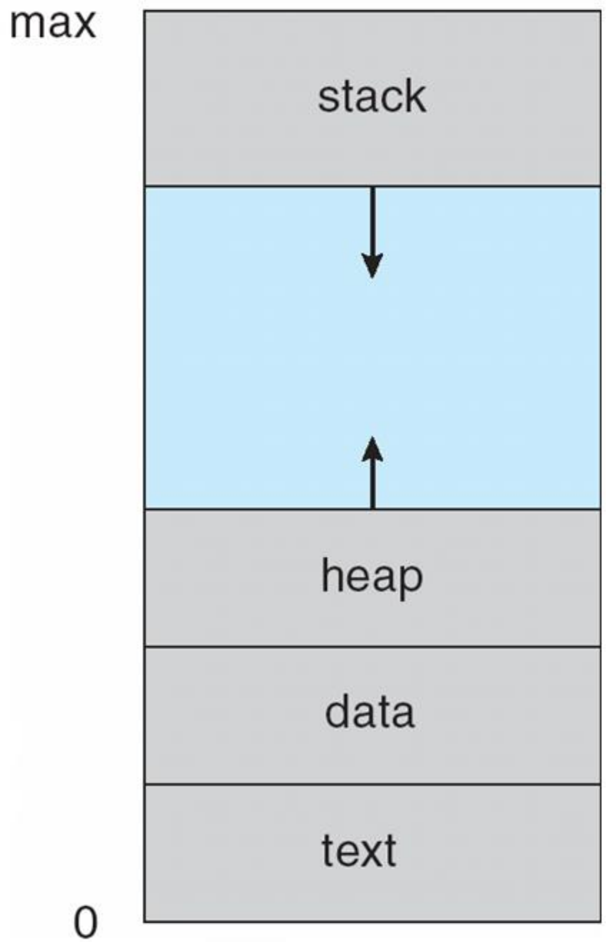
Process Concept

- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- Process contains 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 (Cont.)

- Program is ***passive*** : a file containing a list of instructions stored on disk (**executable file**), but a 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 (many copies of the web browser program).
  - Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

Process in Memory

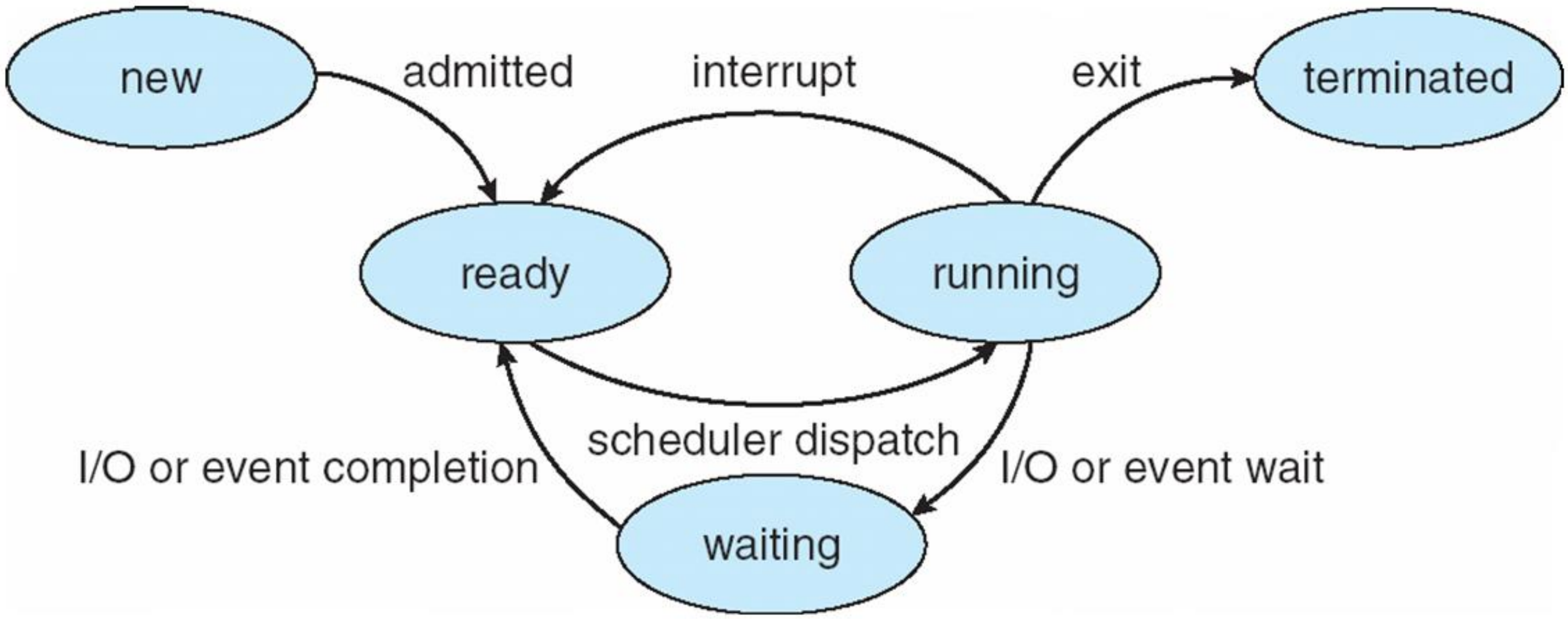


Process State

- 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 (I/O)
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

  - Only **one** process can be running on any processor at any instant.
  - **Many** processes may be ready and waiting.

Diagram of Process State

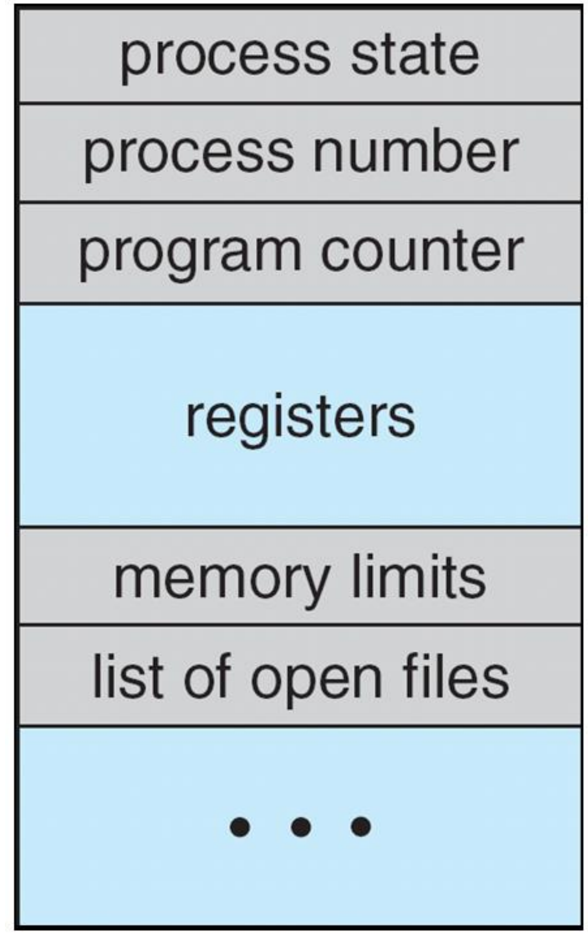


Process Control Block (PCB)

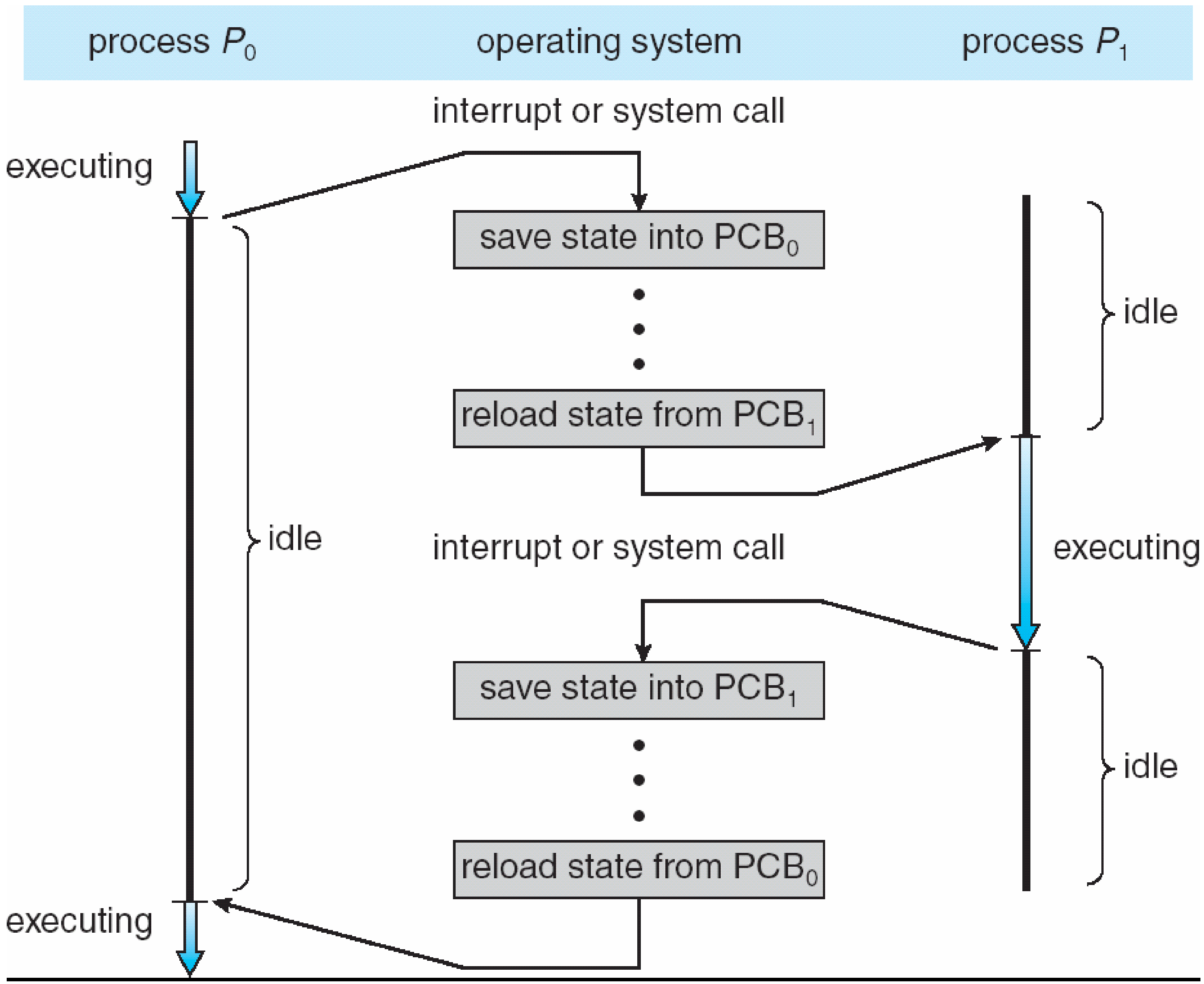
Information associated with each process

(also called **task control block**)

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



CPU Switch From Process to Process



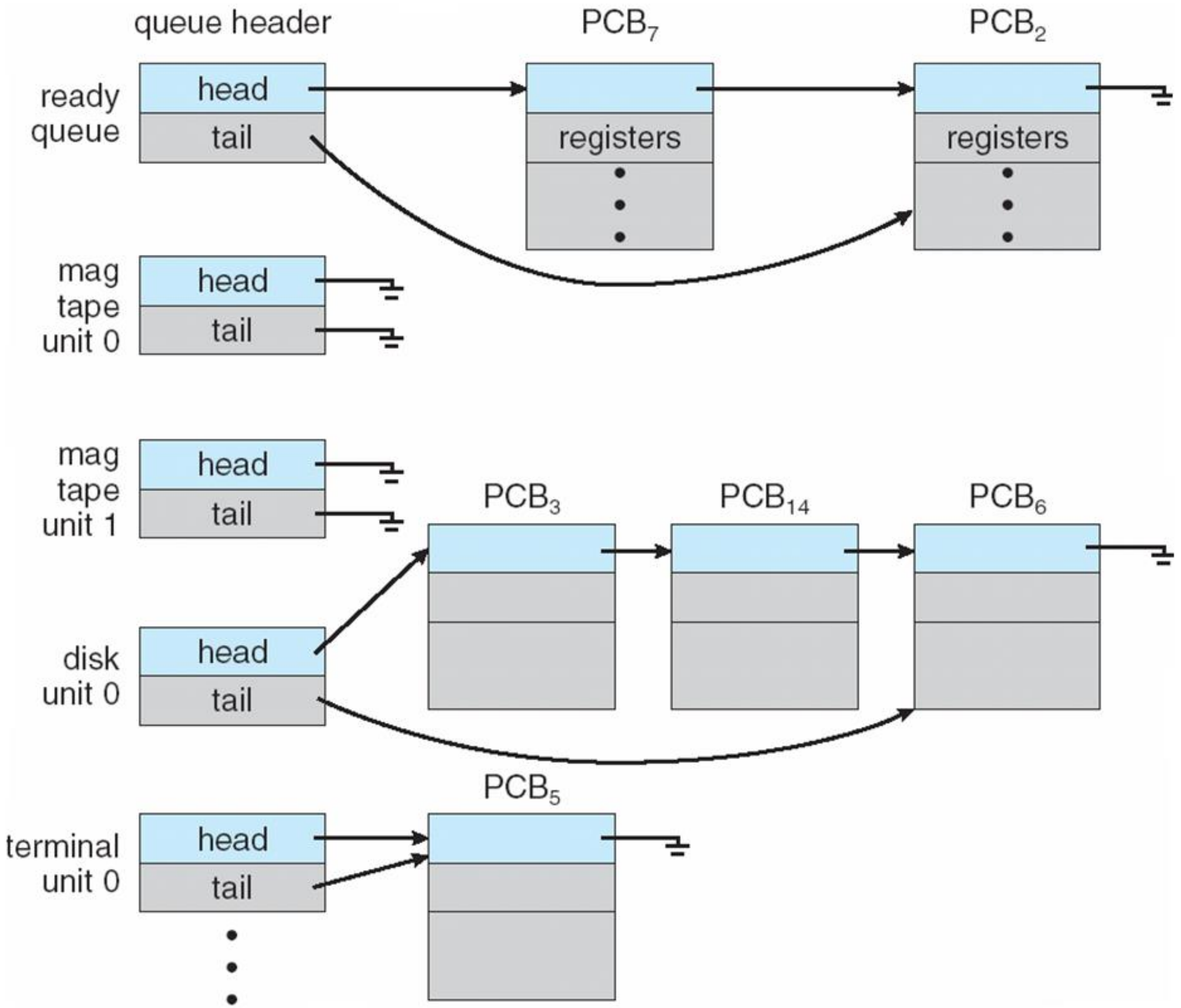
Threads

- So far, process has a single thread of execution
- This single thread of control allows the process to perform only one task at a time.
- For example, The user cannot simultaneously type in characters and run the spell checker within the same process.
- Most modern operating systems (multicore systems) allow a process to have multiple threads of execution run in parallel and perform more than one task at a time.
- PCB is expanded to include information for each thread.

Process Scheduling

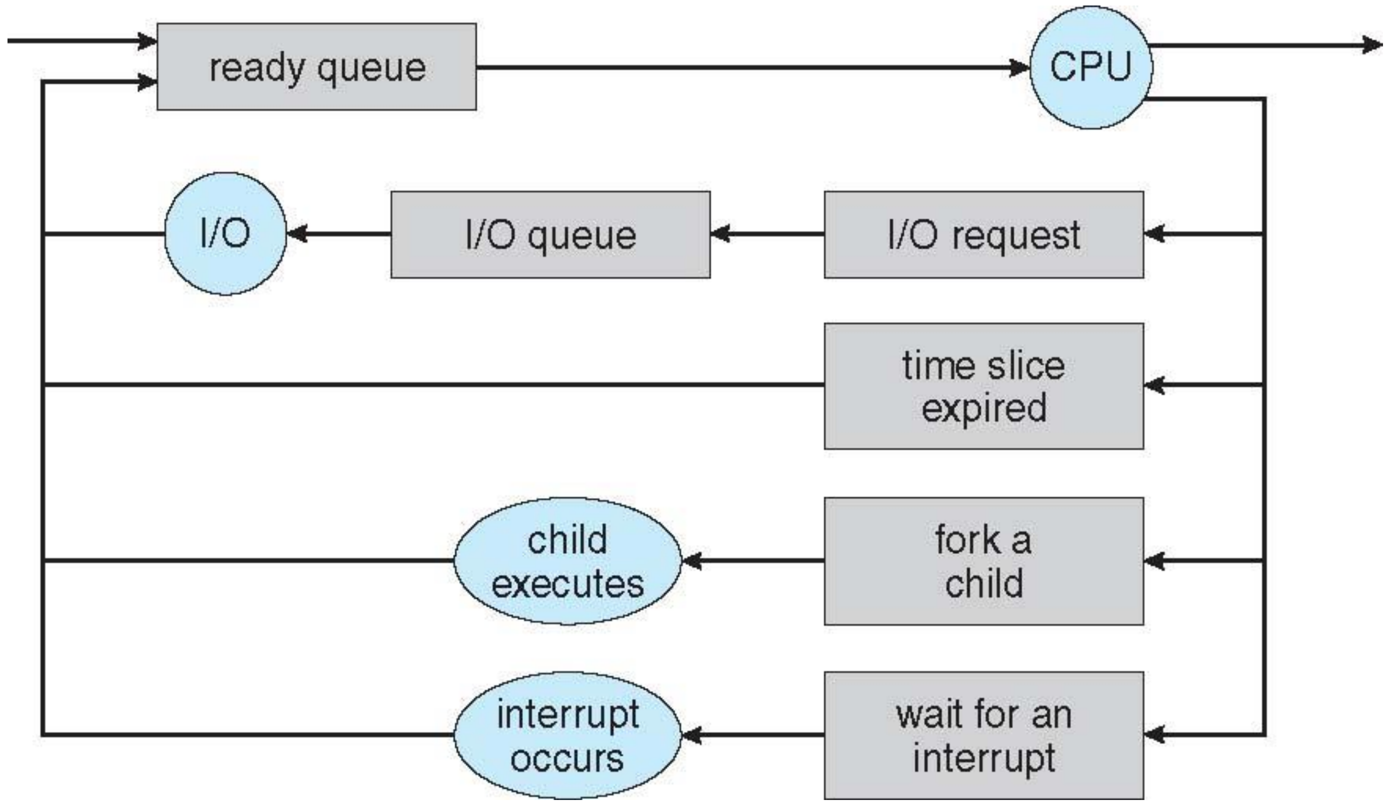
- The objective of multiprogramming is to maximize CPU utilization.
- Switch processes onto CPU so frequently that users can interact with each program while it is running
- one running process, the rest will have to wait until the CPU is free
- **Process scheduler** selects among available processes 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



Representation of Process Scheduling

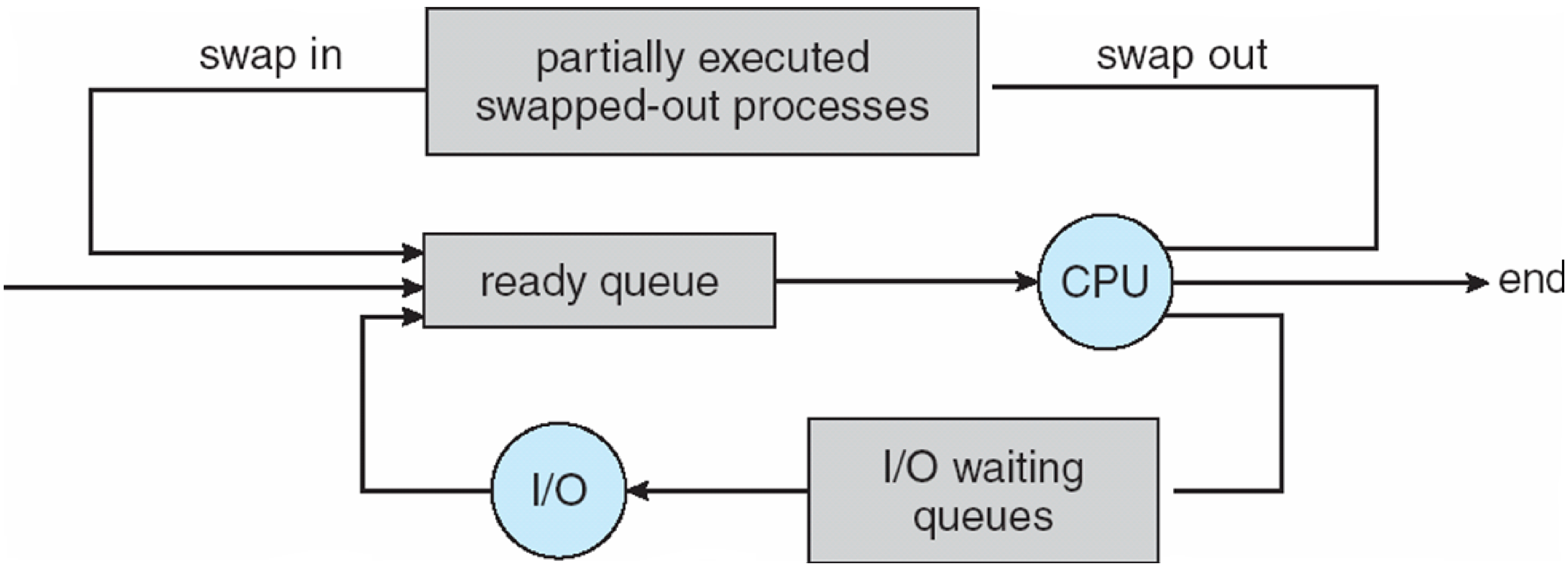
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution.
- Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new child process and wait for the child's termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects (from among the processes that are ready to execute) which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes (from the disk) should be brought into the ready queue and loads them into memory for execution
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory)
- 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
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler:** Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Context Switch

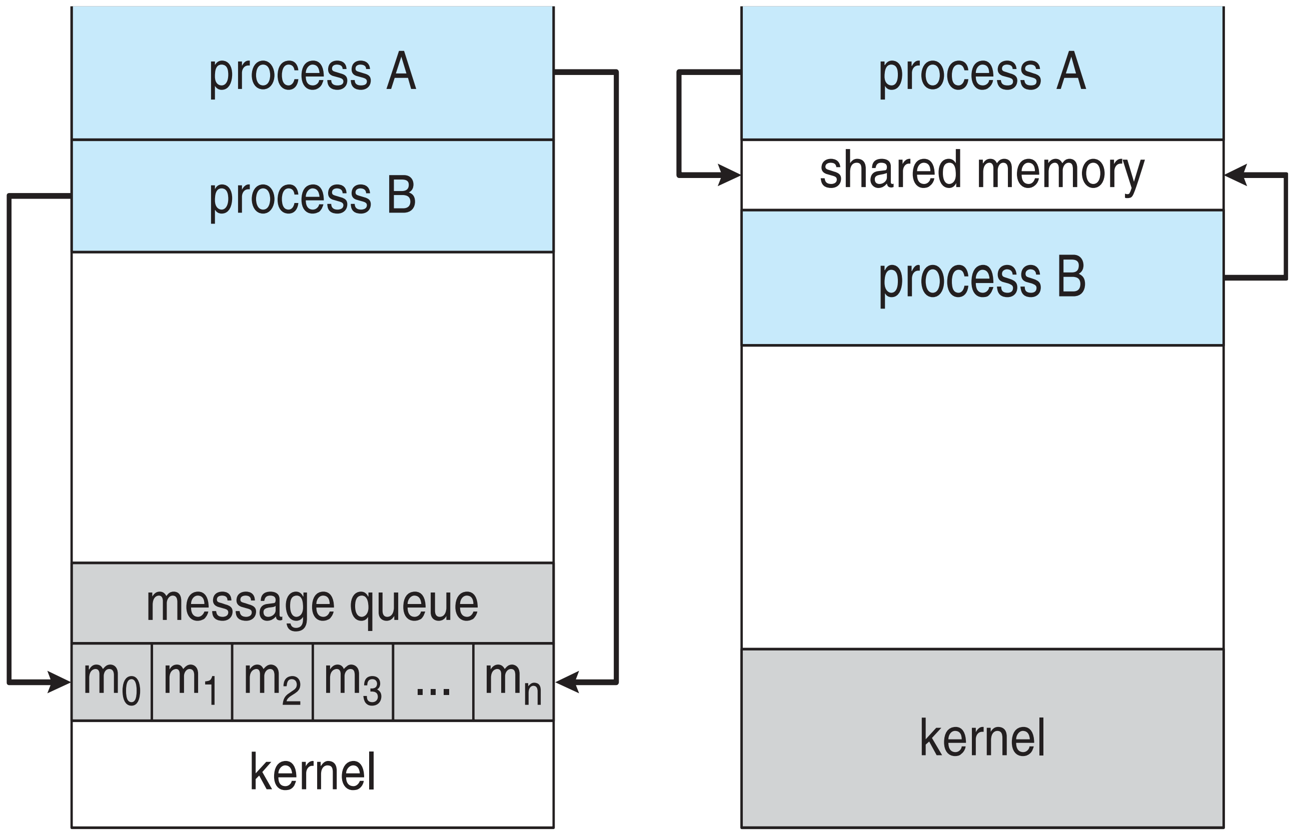
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support

Interprocess Communication

- Processes 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

(a) Message passing. (b) shared memory.



(a) (b)

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

Interprocess Communication – 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.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

- 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

Message Passing (Cont.)

- 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?

Message Passing (Cont.)

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