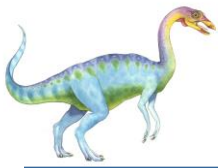


# Chapter 3: Processes

---





# Outline

---

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems





# Objectives

---

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.





# Process Concept

---

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- 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.)

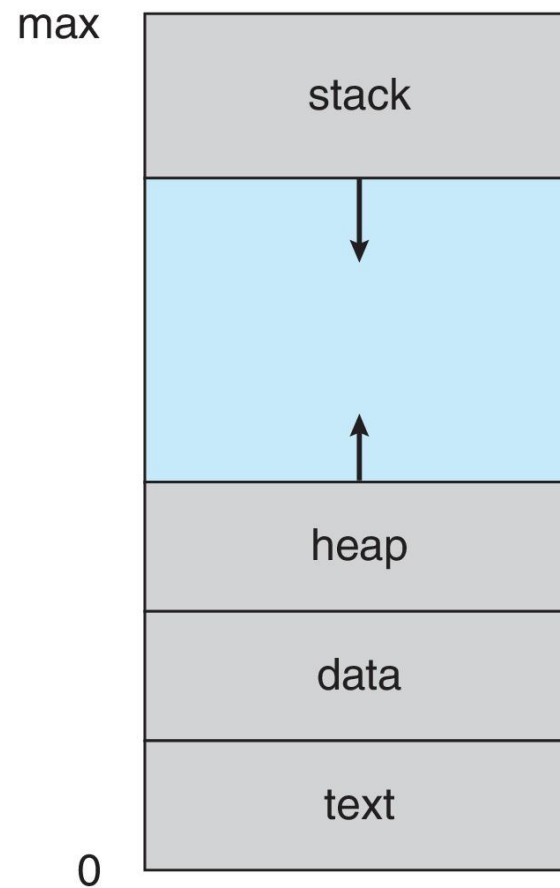
---

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is 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



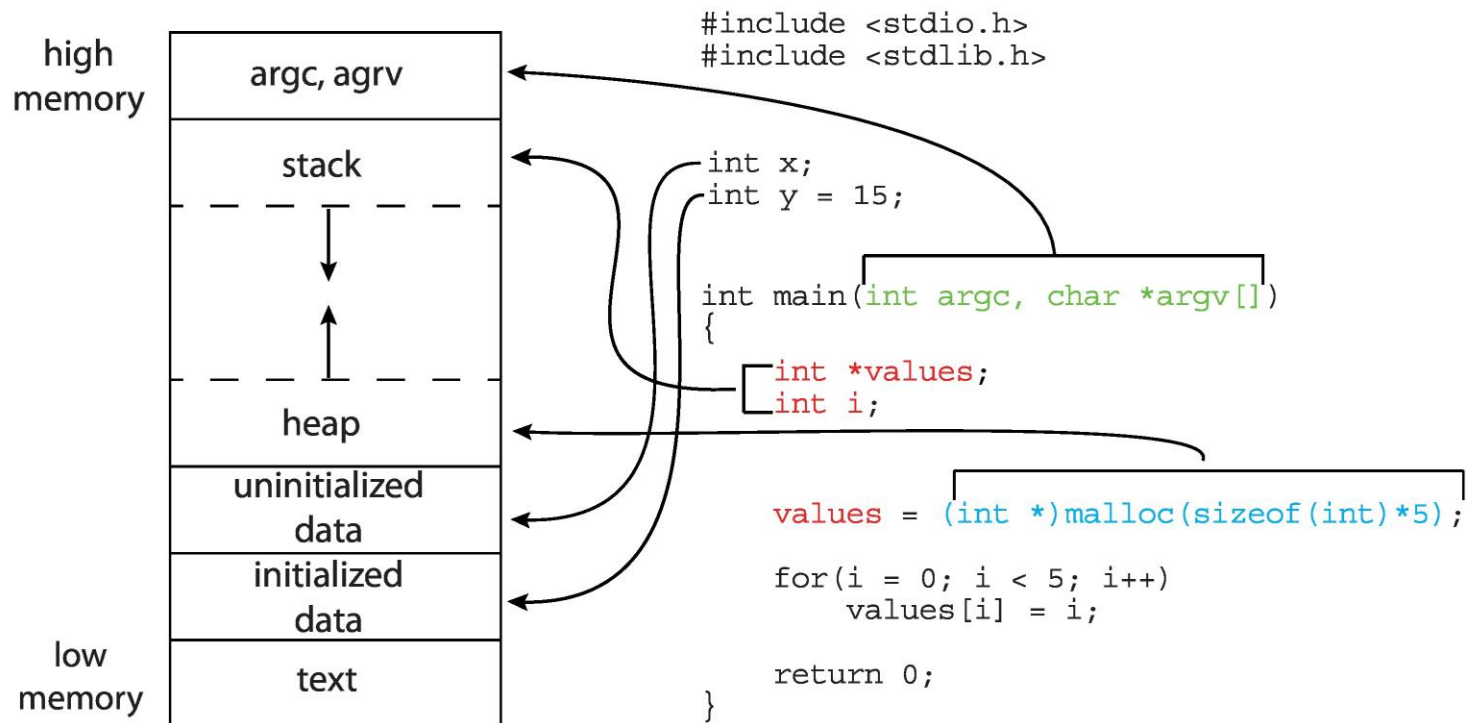


# Process in Memory





# Memory Layout of a C Program





# Process State

---

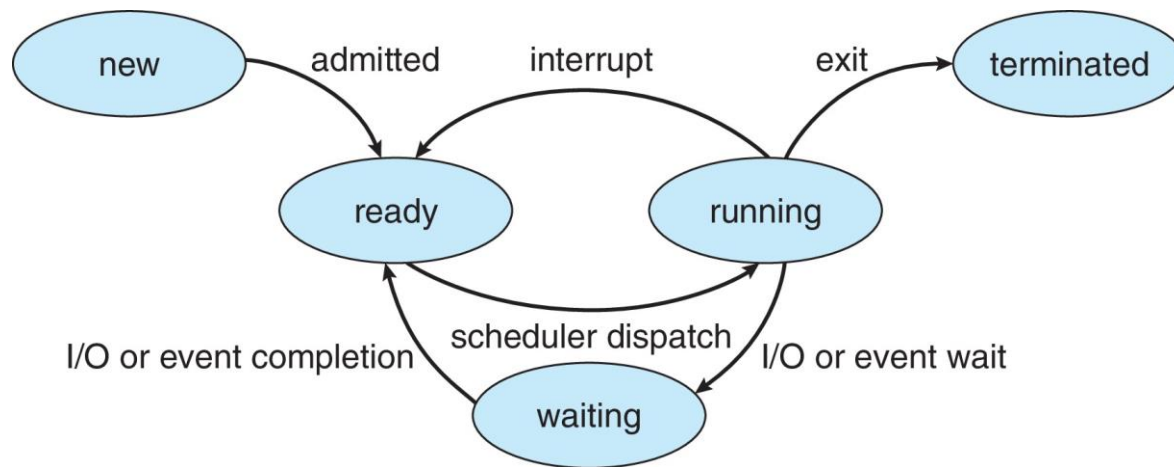
- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution







# Diagram of Process State





# Process Control Block (PCB)

Information associated with each process(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 state
process number
program counter
registers
memory limits
list of open files
...





# Process Scheduling

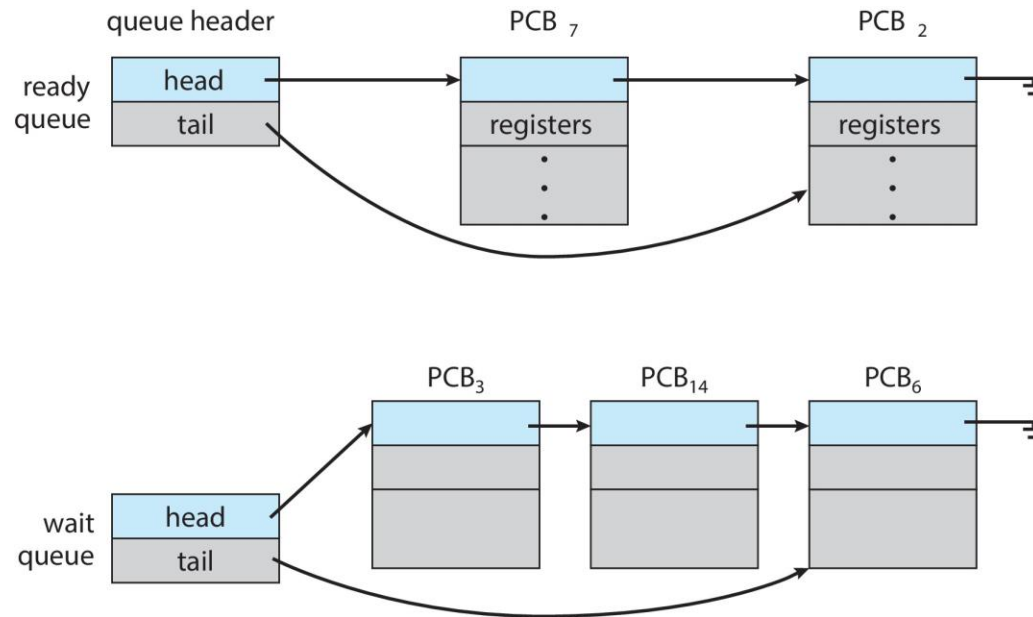
---

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues



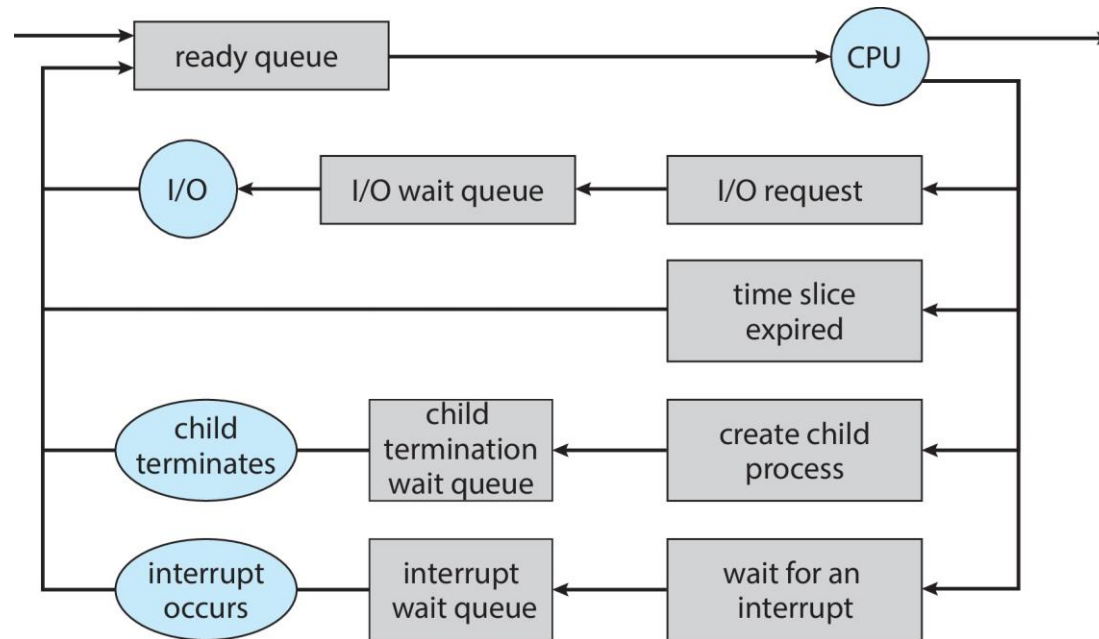


# Ready and Wait Queues





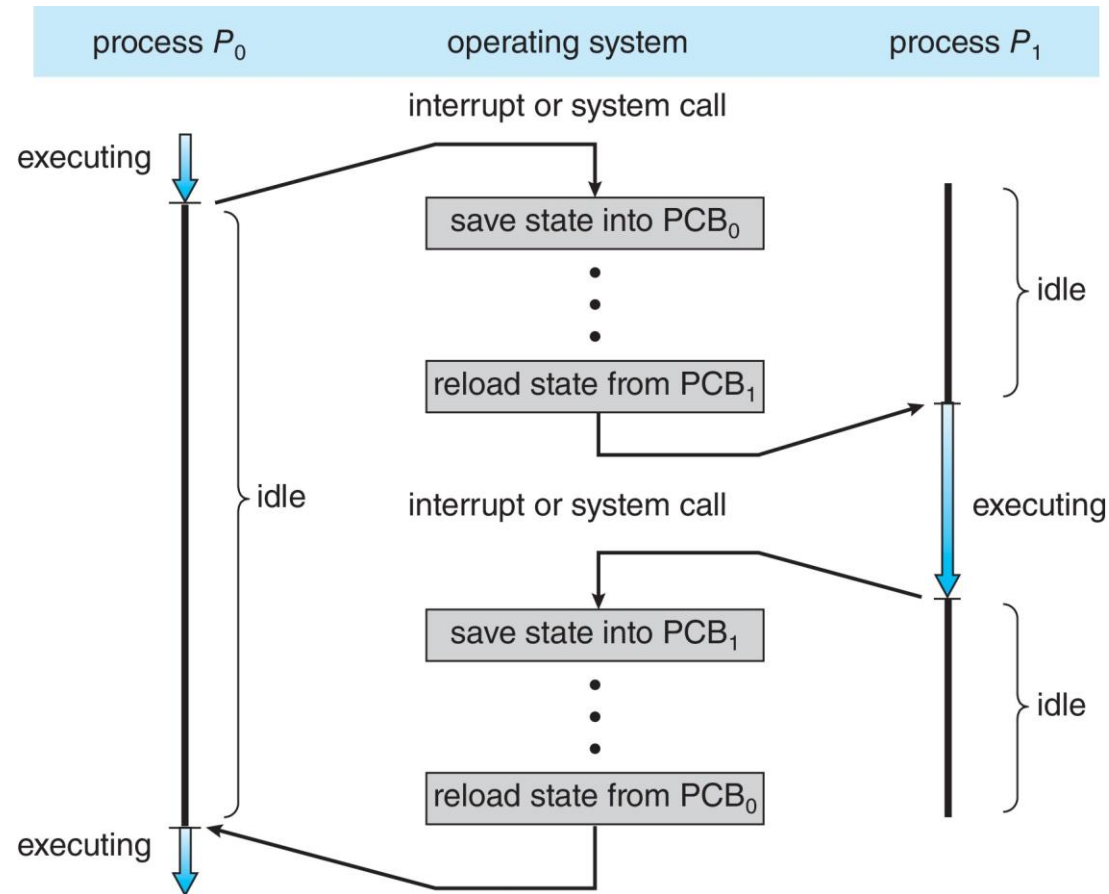
# Representation of Process Scheduling





# CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.





# 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 pure 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





# Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits **iOS** provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- **Android runs** foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use







# Operations on Processes

---

- System must provide mechanisms for:
  - Process creation
  - Process termination





# Process Creation

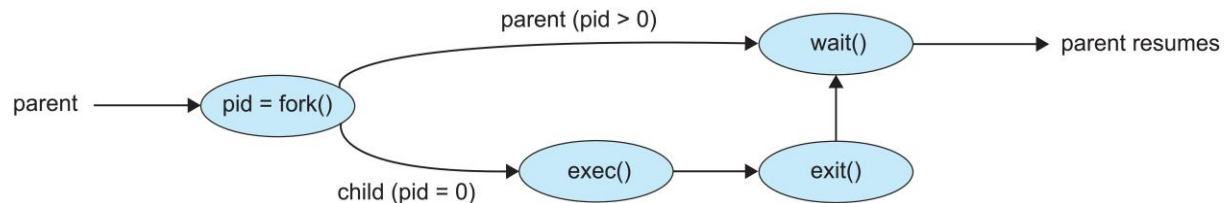
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of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





# 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
  - Parent process calls **wait()** waiting for the child to terminate





# 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 exist 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);
```

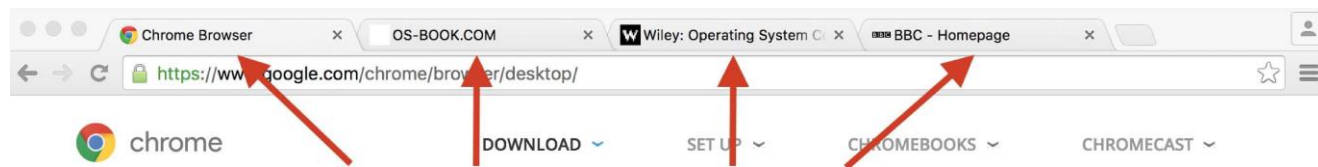
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait()**, process is an **orphan**





# Multiprocess Architecture – Chrome Browser

- Many web browsers run as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - ▶ Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



Each tab represents a separate process.





# Interprocess Communication

---

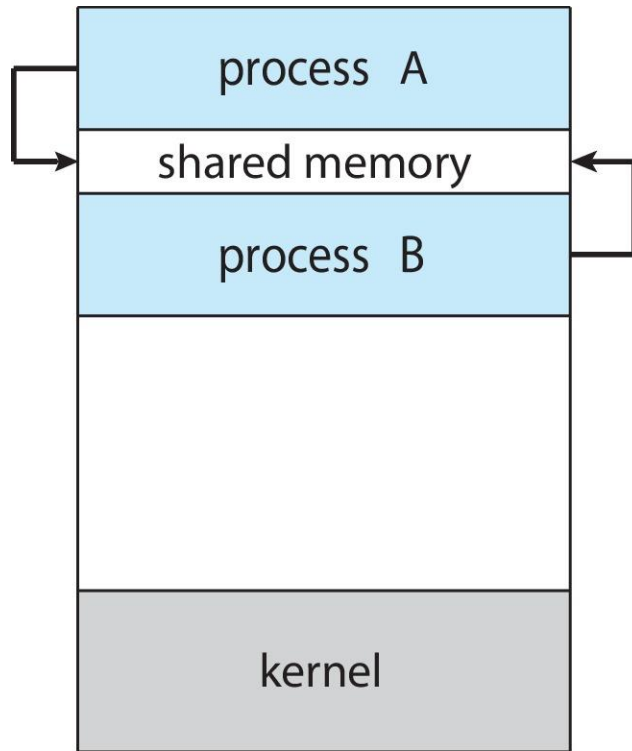
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need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





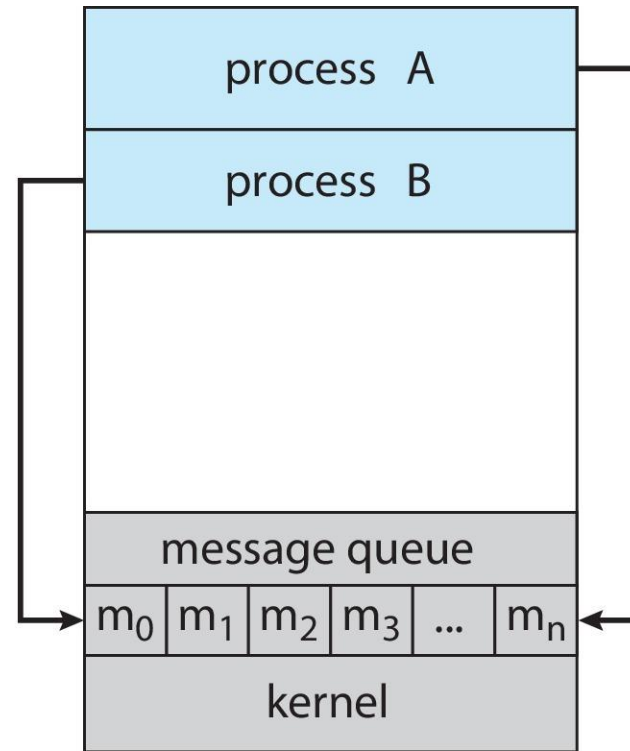
# Communications Models

(a) Shared memory.



(a)

(b) Message passing.



(b)







# Interprocess Communication

---

**Message passing** is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory

Shared memory can be faster than message passing, since **message-passing** systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

**In shared-memory systems**, system calls are required only to establish shared memory regions.

Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required





# Interprocess Communication

---

**Interprocess communication using shared memory** requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this ***shared-memory segment*** must attach it to their address space.

Recall that, normally, the operating system tries to prevent one process from accessing another process's memory.

**Shared memory requires** that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.





# Producer-Consumer Problem

---

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - ▶ Producer never waits
    - ▶ Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - ▶ Producer must wait if all buffers are full
    - ▶ Consumer waits if there is no buffer to consume





# 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.
- Synchronization is discussed in great details in Chapters 6 & 7.





# What about Filling all the Buffers?

---

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **counter** that keeps track of the number of full buffers.
- Initially, **counter** is set to 0.
- The integer **counter** is incremented by the producer after it produces a new buffer.
- The integer **counter** is and is decremented by the consumer after it consumes a buffer.

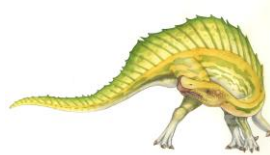




# IPC – Message Passing

---

- 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

- [Redacted]
  - [Redacted]
  - [Redacted]
  - [Redacted]
  - [Redacted]
  - [Redacted]
  - [Redacted]
  - [Redacted]





# Implementation of Communication Link

---

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







# 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





# 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 (Cont.)

---

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete 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 (Cont.)

- 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.

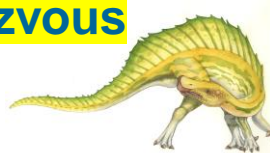




# Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** -- 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**
- **Ends at**





# Synchronization

---

- The solution to the producer–consumer problem

becomes trivial when we use blocking `send()` and `receive()` statements. The producer merely invokes the blocking `send()` call and waits until the message is delivered to either the receiver or the mailbox.

Likewise, when the consumer invokes `receive()`, it blocks until a message is available.





# 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

***We will end at that slide***



# End of Chapter 3

---

