Chapter 3: Processes

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing 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.

- Describe and contrast interprocess communication using shared memory and message passing.

Process Concept

- Program is *passive* entity stored on disk (**executable file**); 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
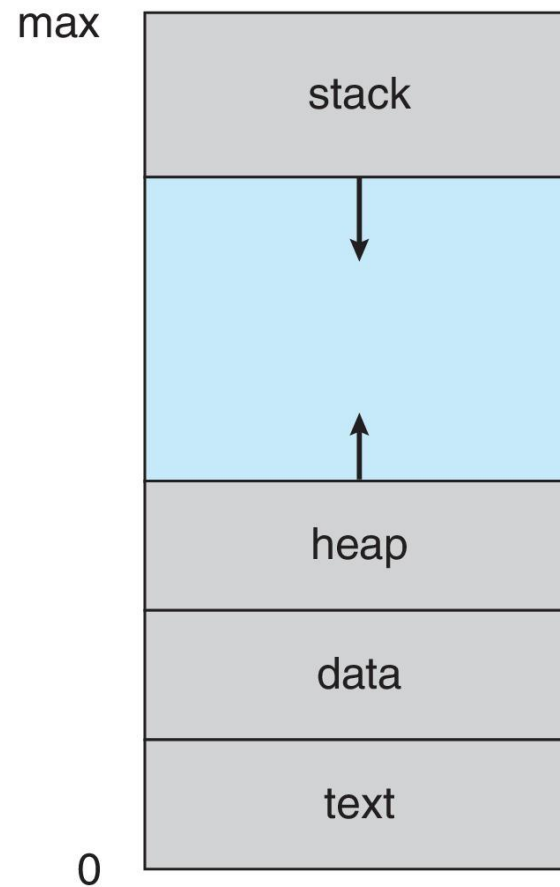
    - Consider multiple users executing the same program

Process Concept (Cont.)

- 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

- 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

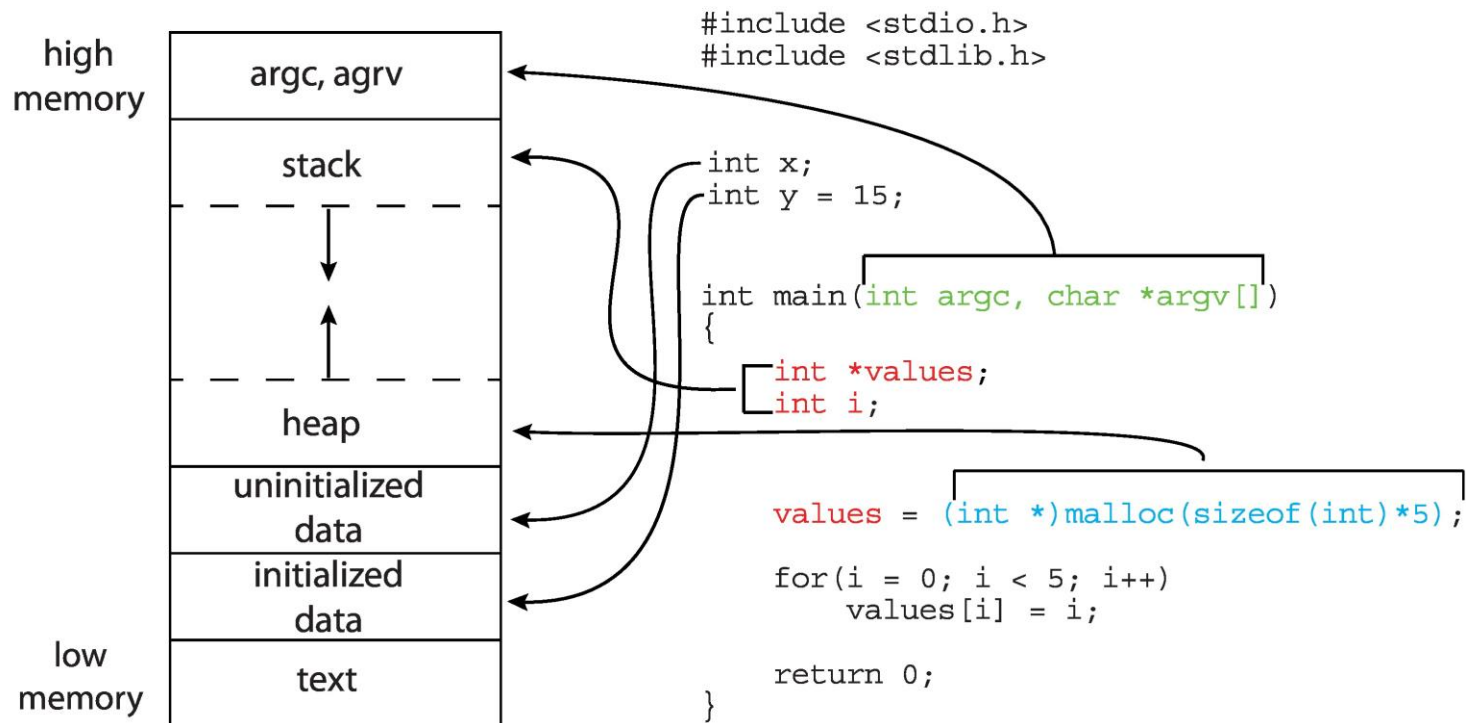   - **Heap** containing memory dynamically allocated during run time

Process in Memory

Memory Layout of a C Program


```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Memory layout diagram (high memory to low memory): argc, argv; stack (grows down); heap (grows up); uninitialized data; initialized data; text
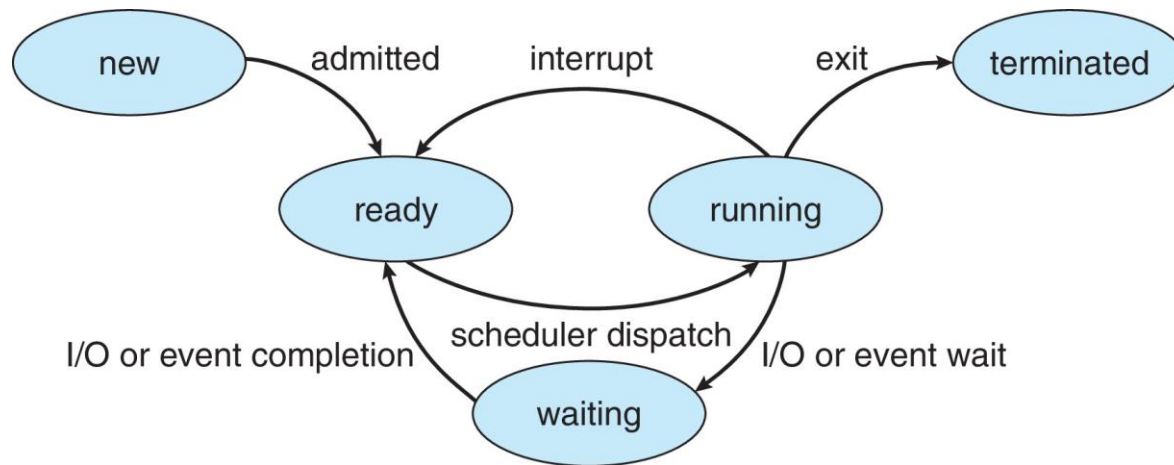
Process State

- 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
. . .

Threads

- So far, process has a single thread of execution

- Consider having multiple program counters per process

  - Multiple locations can execute at once

    ▸ Multiple threads of control -> **threads**

- Must then have storage for thread details, multiple program counters in PCB
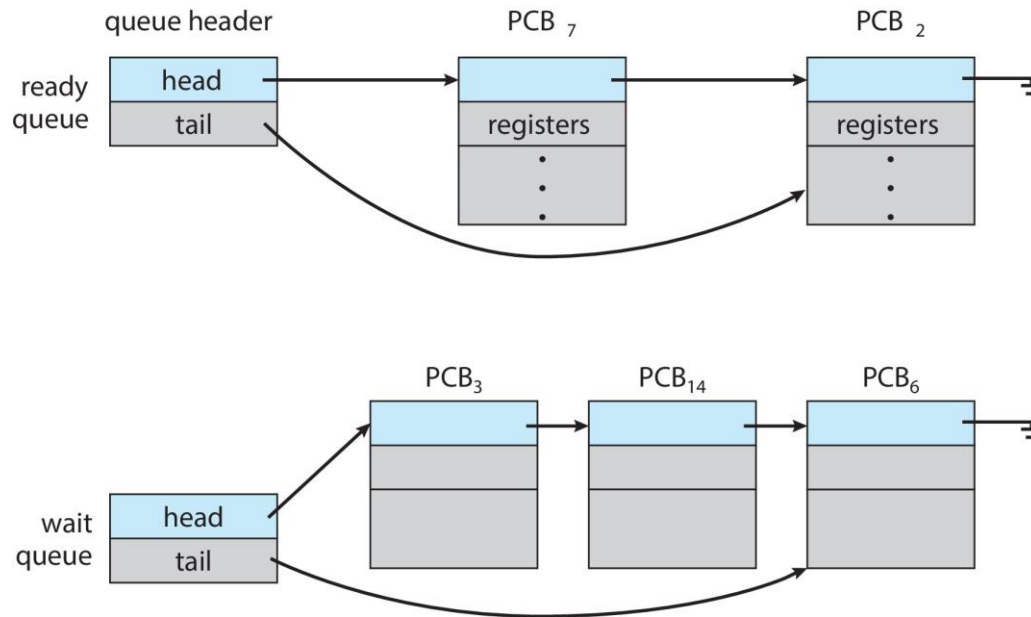
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU core

- **Process scheduler** selects among available processes for next execution on 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)
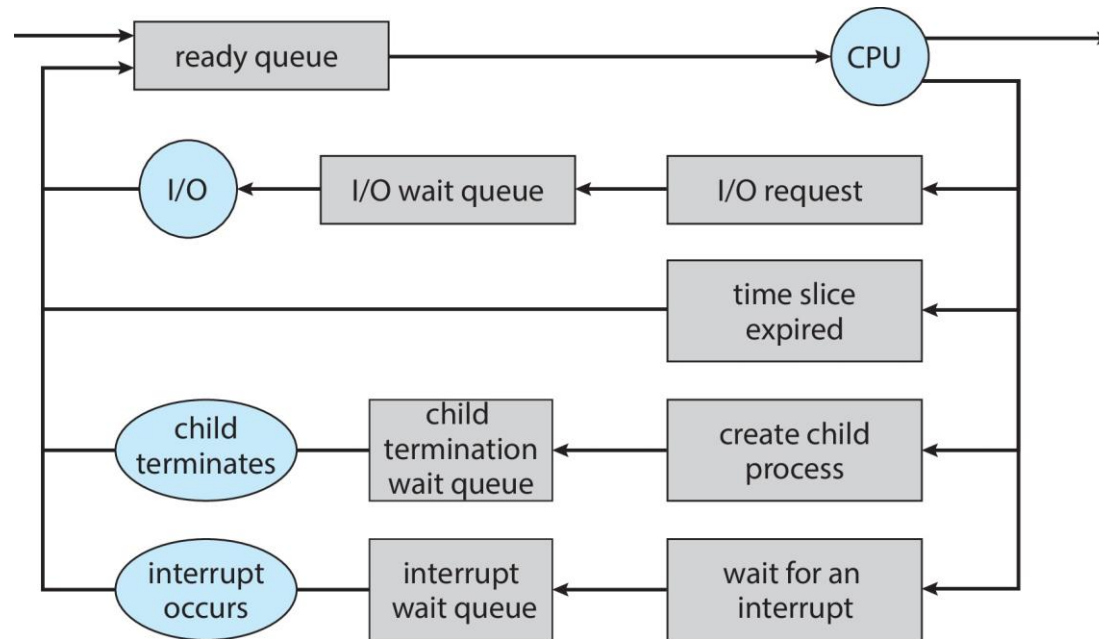
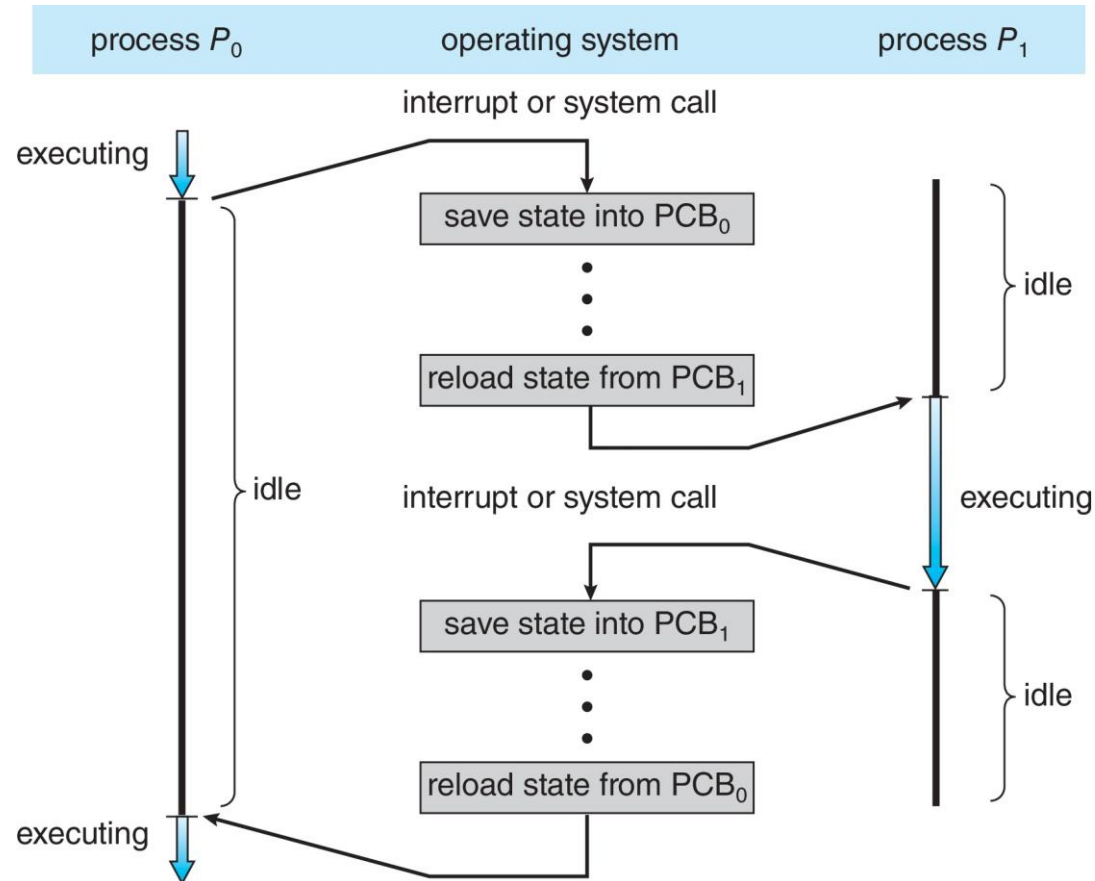 - Processes migrate among the various queues

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 overhead; the system does no useful work while switching

  - The more complex the OS and the PCB ➜ the longer the context switch

- Time depends 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
    - 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

Operations on Processes

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

Process Creation

- **Parent** process creates **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()` for the child to terminate
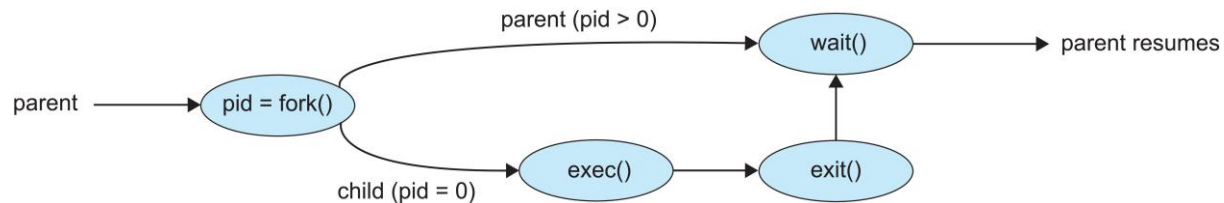
Process Termination

- 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

- 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.
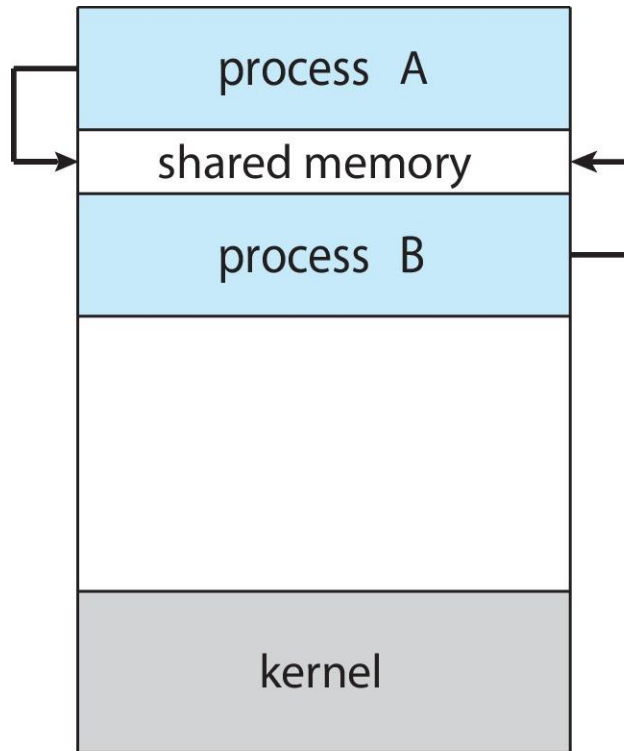
Interprocess Communication

- 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

- Cooperating processes need **interprocess communication** (**IPC**)

- Two models of IPC
  - **Shared memory**
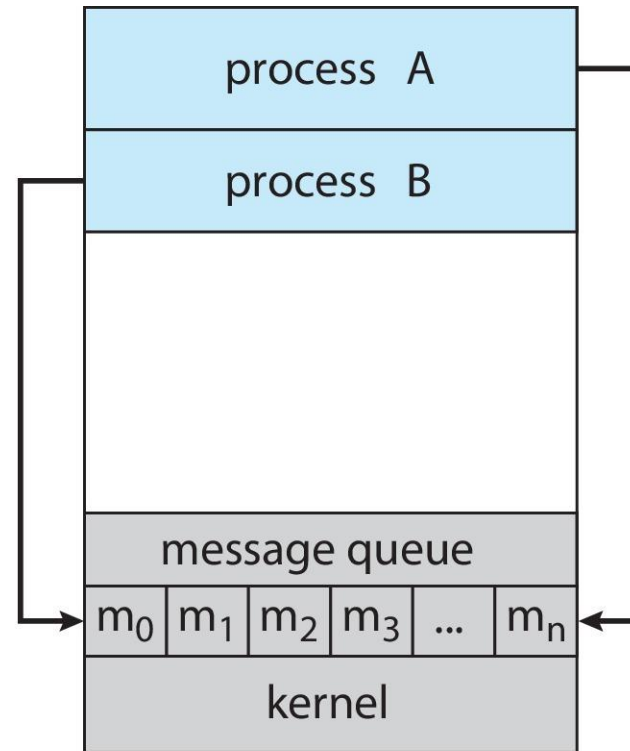 - **Message passing**

Communications Models

(a) Shared memory.

(b) Message passing.

(a)   (b)

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 details in Chapters 6 & 7.

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

    - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

☐ Shared data

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


item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```
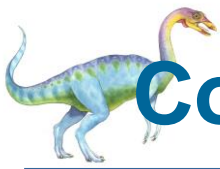
Producer Process – Shared Memory

```
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;
}
```

Consumer Process – Shared Memory

```
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 */
}
```

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?

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

    - 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

Indirect Communication

- 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

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** -- 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**

```
message next_produced;

while (true) {
       /* produce an item in next_produced */

       send(next_produced);
}
```

Consumer– Message Passing

```
message next_consumed;

while (true) {
        receive(next_consumed)

        /* consume the item in next_consumed */
  }
```

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