Chapter 2. Processes

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. (2018). *Operating System Concepts* (10th ed.). Wiley.

Contents

1. Process Concept

- An operating system executes a variety of programs that run as **processes**.
- 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's registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

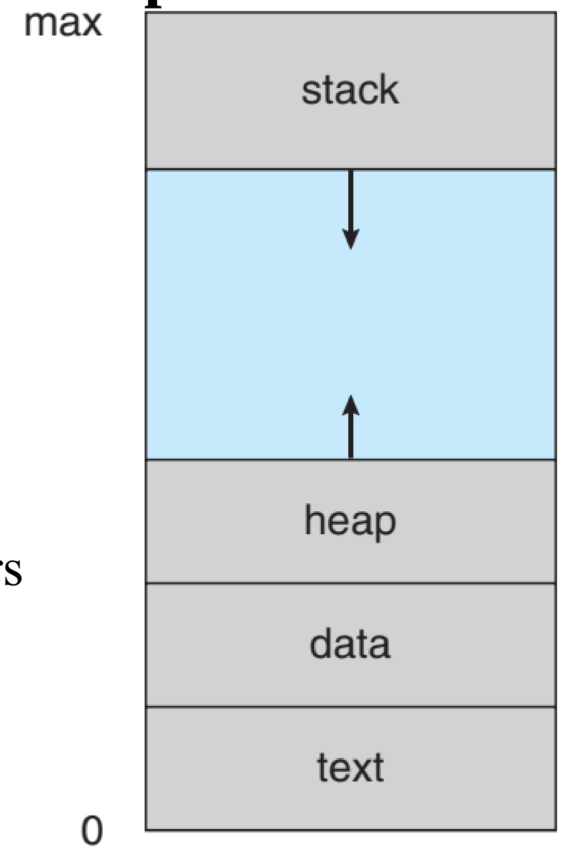
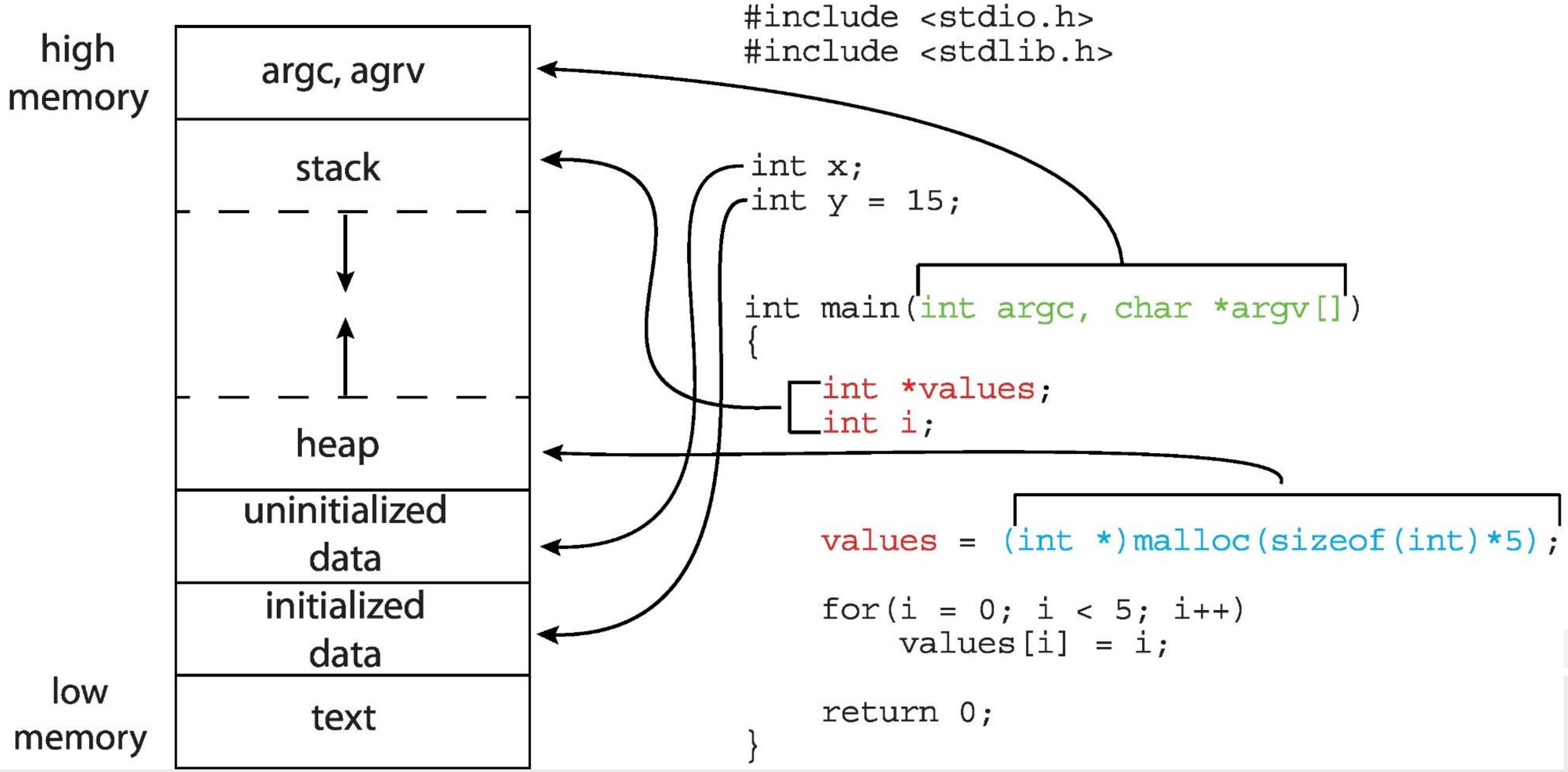


Figure 3.1 Layout of a process in memory.

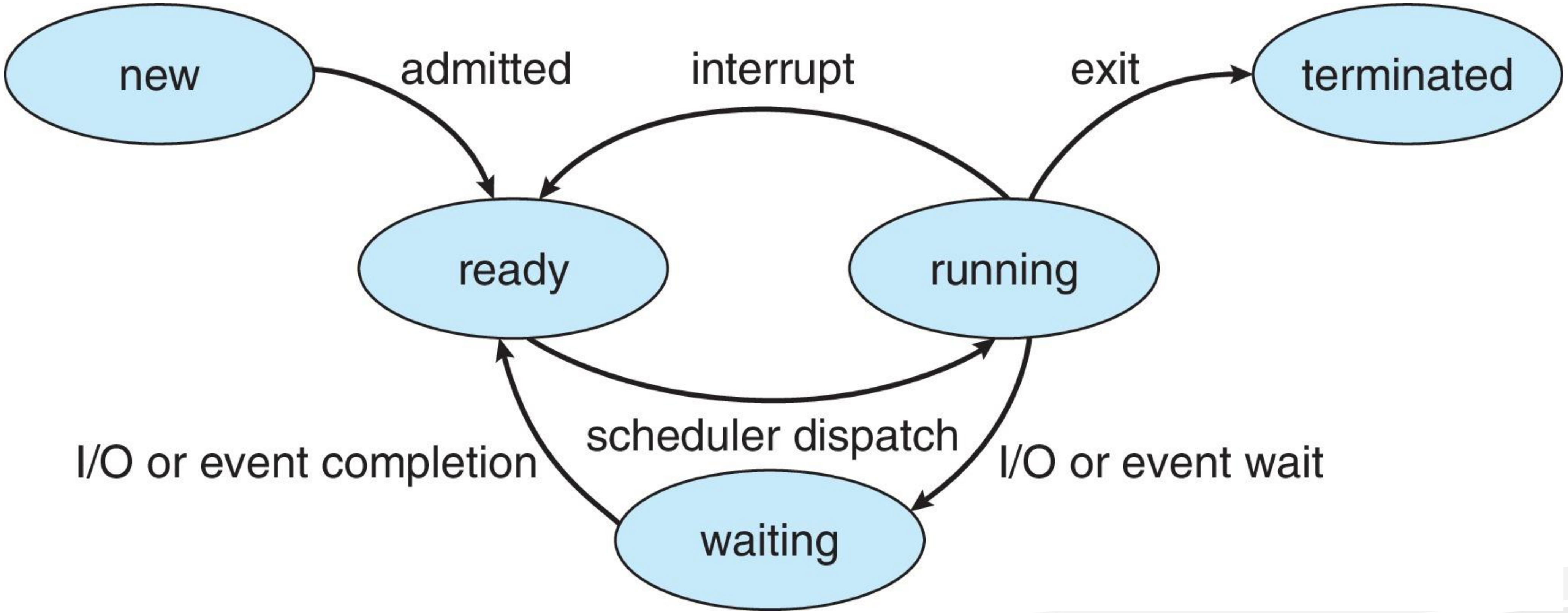
1. Process Concept - Memory Layout of a C program



1. Process Concept - Process State

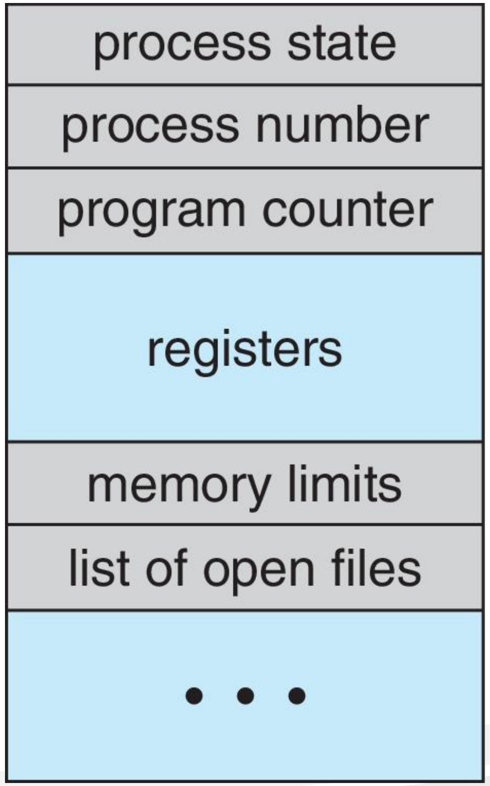
- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed (use CPU)
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The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

1. Process Concept – Diagram of Process State



1. Process Concept – Process Control Block (PCB)

- 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



1. Process Concept - Threads

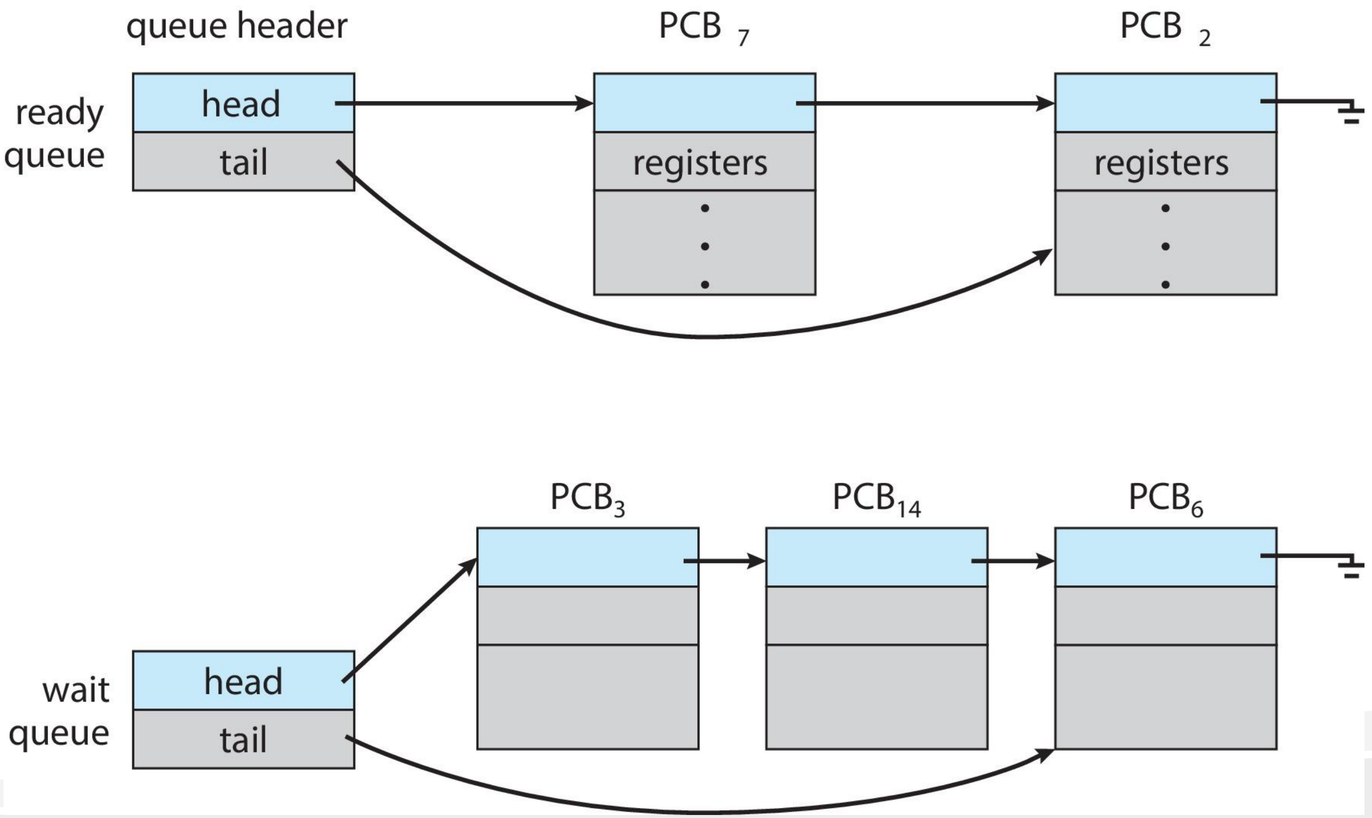
- So far, process has a single thread of execution.
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> threads
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4

2. Process Scheduling

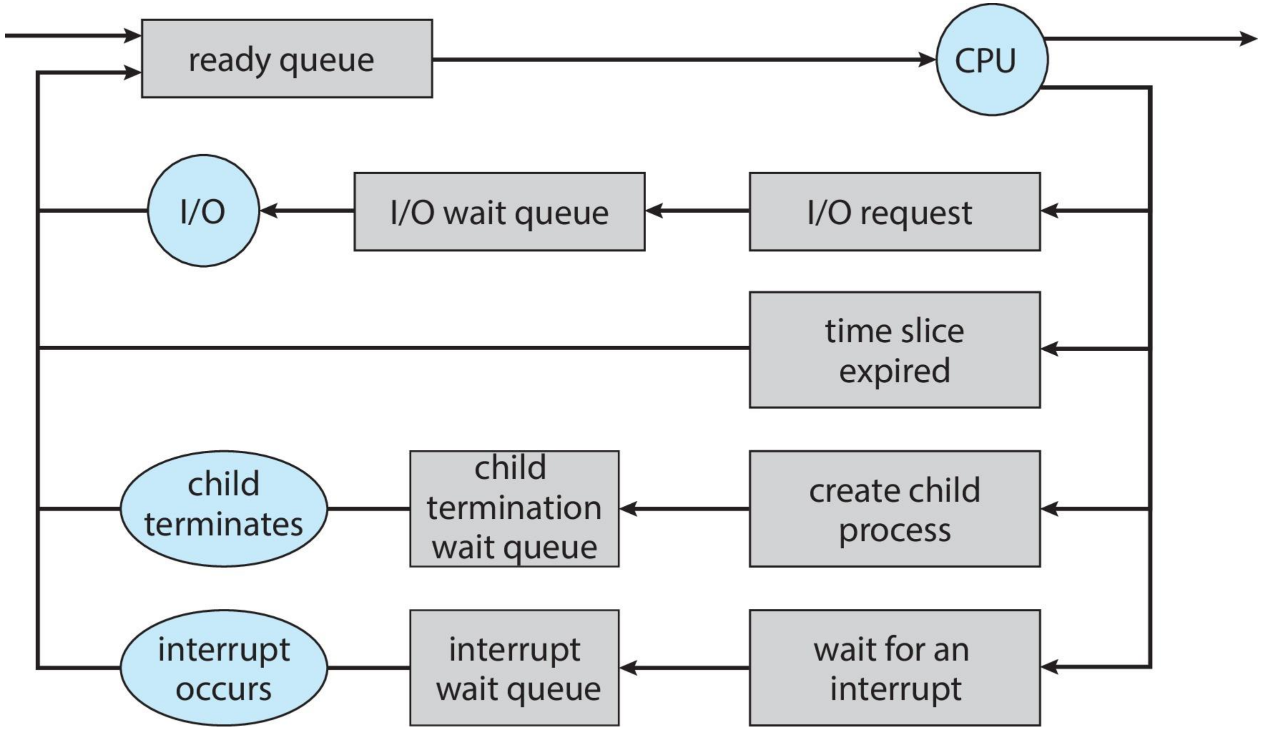
- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
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

2. Process Scheduling - Ready and Wait Queues

- Ready and Wait Queues

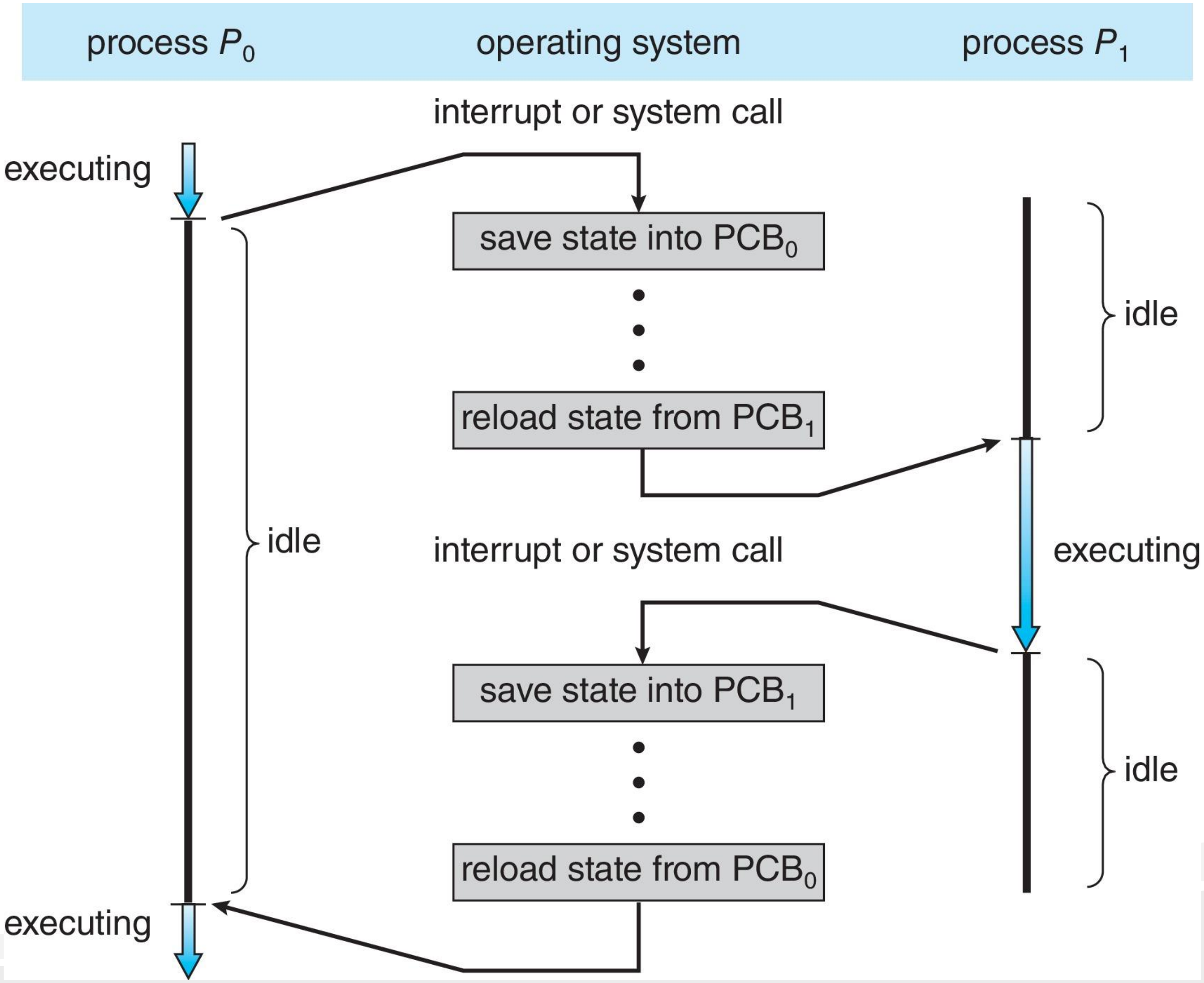


2. Process Scheduling – Representation of Process Scheduling



2. Process Scheduling – CPU Switch from Process to Process

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



2. Process Scheduling – 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

3. Operations on Processes

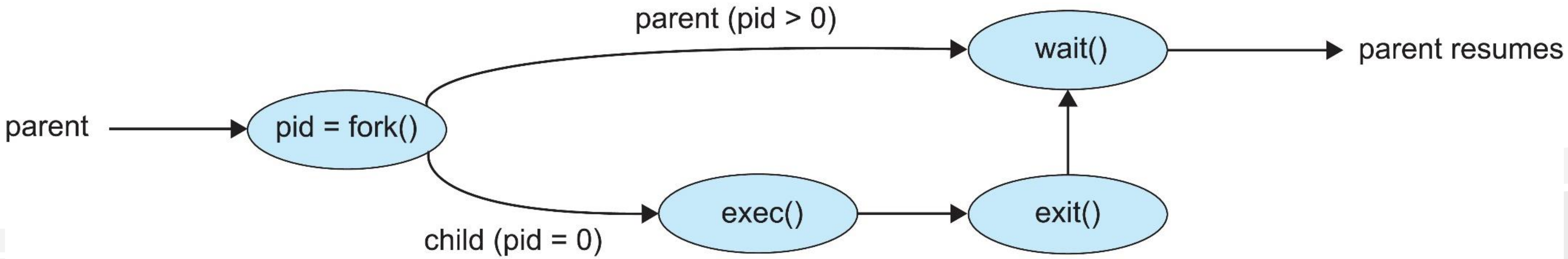
- System must provide mechanisms for:
  - Process creation
  - Process termination

3. Operations on Processes – 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

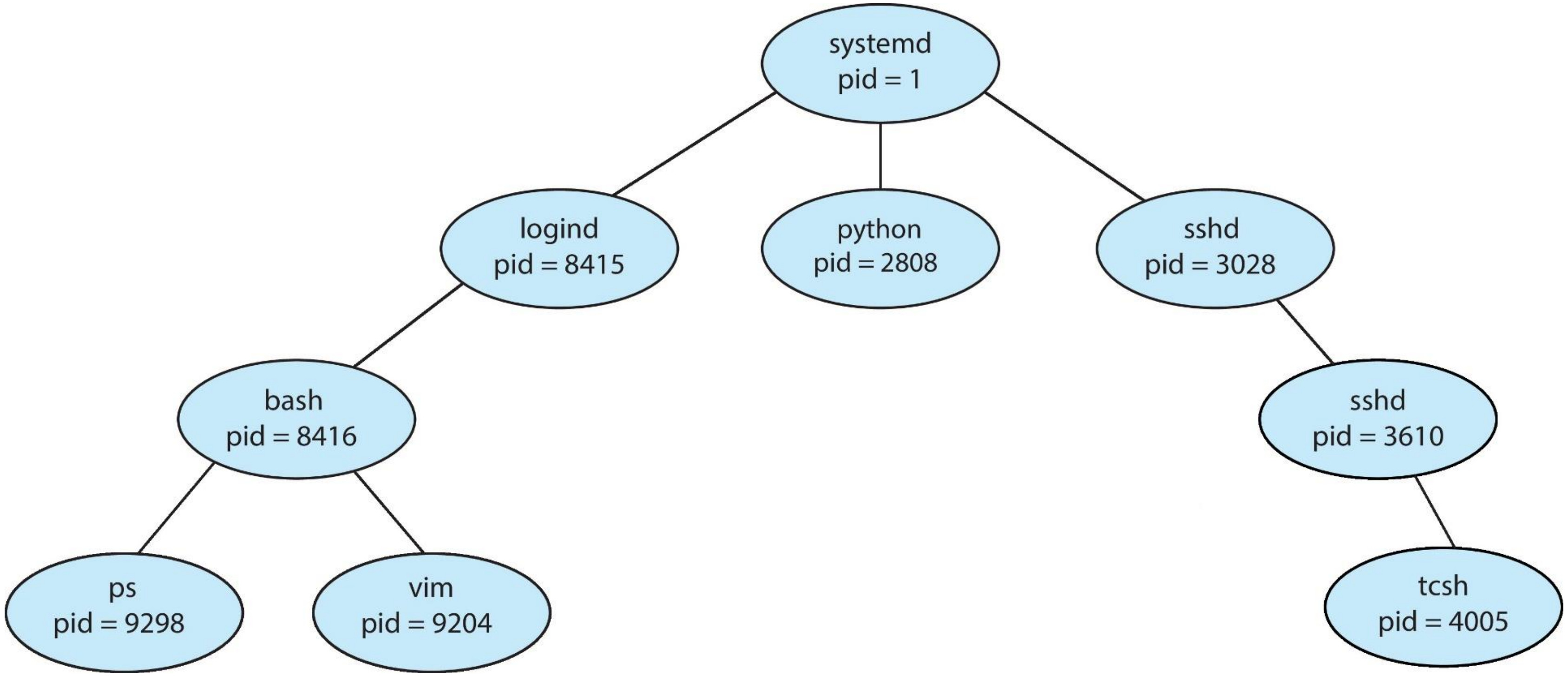
3. Operations on Processes – Process Creation

- 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



3. Operations on Processes

- A tree of processes in Linux



3. Operations on Processes

- C program forking separate process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

3. Operations on Processes

- Creating a separate process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

3. Operations on Processes – 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

3. Operations on Processes – 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);
- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls **wait()**, because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called **wait()**, is known as a **zombie** process.
- If parent terminated without invoking **wait()**, process is an **orphan.**

4. 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

4. Interprocess Communication – Communication Modes

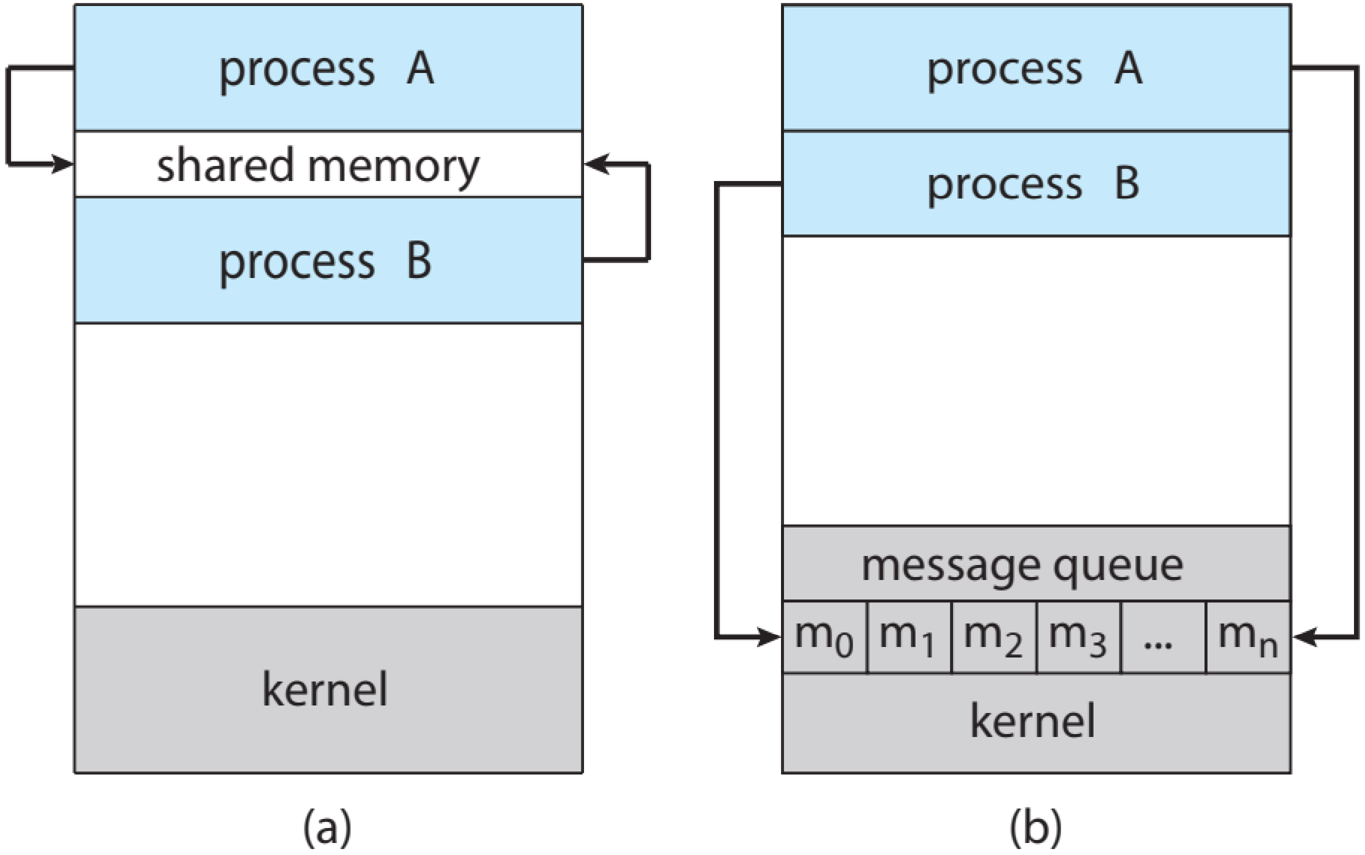


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

4. Interprocess Communication – Producer-Consumer Problem

- 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 item to consume (the buffer is empty)
    - **bounded-buffer** assumes that there is a fixed buffer size
      - Producer must wait if all buffers are full
      - Consumer waits if there is no item to consume (the buffer is empty)

4. Interprocess Communication – IPC in Shared-memory Systems

- IPC in 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 following chapters.

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

The shared buffer is implemented as a circular array with two logical pointers: **in** and **out**. The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer.

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

Bounded-Buffer – Shared-Memory Solution

- Producer process

```
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

```
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*/
}
```

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 decremented by the consumer after it consumes a buffer.

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

- The shared buffer is implemented as a circular array with two logical pointers: **in** and **out**. The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer.

- The **counter** keeps track of the number of items in the buffer.

Bounded-Buffer – Shared-Memory Solution

- Producer process

```
item next_produced;

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

- Consumer process

```
item next_consumed;


while(true) {
    while(counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /*consume the item in next consumed*/
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition

- **Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?**

4. Interprocess Communication – IPC in Message Passing

- IPC facility provides two operations:
  - send(message)
  - receive(message)
- The message size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive

4. Interprocess Communication – IPC in Message Passing

- 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?

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

- 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

- 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.

Indirect Communication

- 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

Producer-Consumer: Message Passing

- Producer

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

- Consumer

```
message next_consumed;
while (true) {
     receive(next_consumed)
     /* consume the item in next_consumed */
}
```

5. Communication in Client–Server Systems

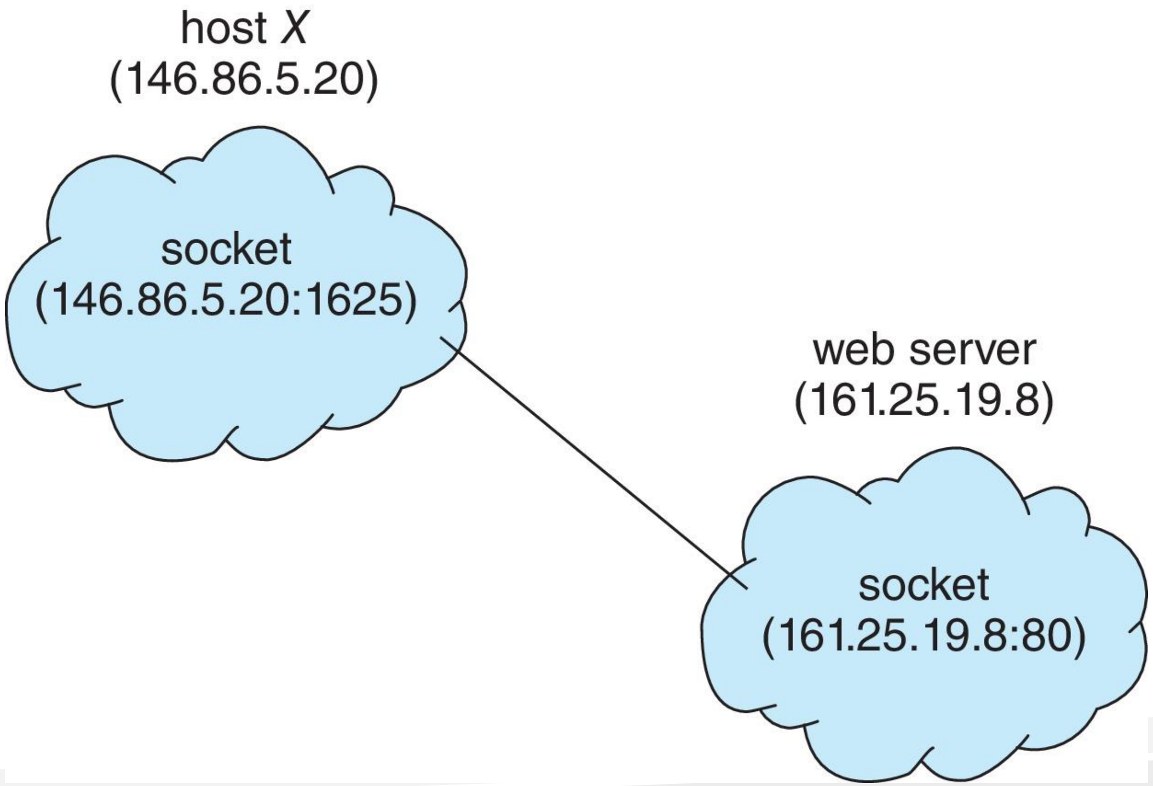
- Sockets
- Remote Procedure Calls

5. Communication in Client–Server Systems

- Sockets
  - A **socket** is defined as an endpoint for communication
  - Concatenation of **IP address** and **port** – a number included at start of message packet to differentiate network services on a host
  - The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
  - Communication consists between a pair of sockets
  - All ports below **1024** are well known, used for standard services
  - Special IP address **127.0.0.1** (loopback) to refer to system on which process is running

5. Communication in Client–Server Systems

- Socket Communication



5. Communication in Client–Server Systems

- Three types of sockets
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients

5. Communication in Client–Server Systems

- Sockets in Java
  - Date server in Java

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

5. Communication in Client–Server Systems

- Sockets in Java
  - Date client in Java

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```