Operating Systems: CSE 3204

Chapter Two

Process Management

Lecture 2: Processes

ASTU
Department of CSE

Overview of the lecture

- Process
- Process Concept
- Process Scheduling
- Operations on Processes: Process Creation and Termination
- Cooperating Processes
- Interprocess Communication

An Analogy

- **Example**
  - Various workers (males, females, skilled, unskilled, semi-skilled) are working on some job.
  - **Question**: How to organize them effectively to improve the performance ?
  - Sitting worker is similar to program.
    - Does not use any tools and resources.
  - Working worker is similar to process
    - Uses tools, interacts with other,
    - If he/she is working with others, cooperation and synchronization is required. Otherwise accidents might occur.
    - If he/she is working alone, no cooperation is needed.
  - To manage working workers, the manager should have some information about the working worker.
    - What kind of tools he/she is using.
    - The nature of the job, and status of the job.
    - ….
  - **In computer system**
    - Operating System/OS is similar to manager
    - The workers are similar to the programs reside on the disk.
    - The working workers are processes
    - CPU is an expensive tool which should not be kept idle. It is OK if some workers wait for expensive tool.

Process

- A program written in any language like C, C++ or Java is a passive entity, while a Process is a program in execution.
- A process is an active entity. A program can be converted into a process if it is loaded from the disk into main memory and begin its execution on processor.
- Process need resources to complete its tasks, the main resources required in the lifetime of a process are memory, CPU, I/O devices etc.
- A process is composed of unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions

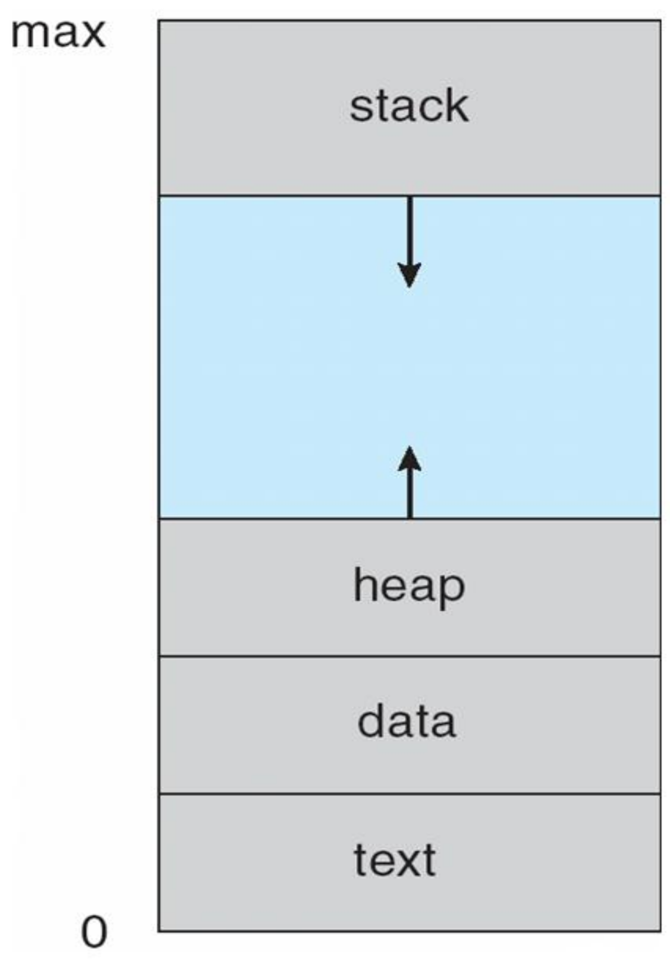
Process Concept

- **Early systems:** One program at a time was executed and a single program has a complete control.
- **Modern OSs:** allow multiple programs to be loaded in to memory and to be executed concurrently.
- This requires firm control over execution of programs.
- The notion of process emerged to control the execution of programs.
- **A process:** Unit of work → Program in execution
- **OS consists of a collection of processes**
  - OS processes executes system code.
  - User processes executes user code.
- By switching CPU between processes, the OS can make the computer more productive.

Process Concept

- **Process** – A process has Multiple parts other than source code
  - **Text section:** The source code text file, contain the logic of program which is executed on behalf of process
  - **Program counter**: Program counter is the Processor register(s) which stores the address of next executable instruction. Multiple PC are possible one for each flow of control during the run time in multiprogramming environment.
  - **Stack** is an area in RAM containing temporary data, Function parameters, return addresses, local variables belonging to a process
  - **Data section** containing global variables declared in the process
  - **Heap** containing memory dynamically allocated during run time of the process

Process in Memory

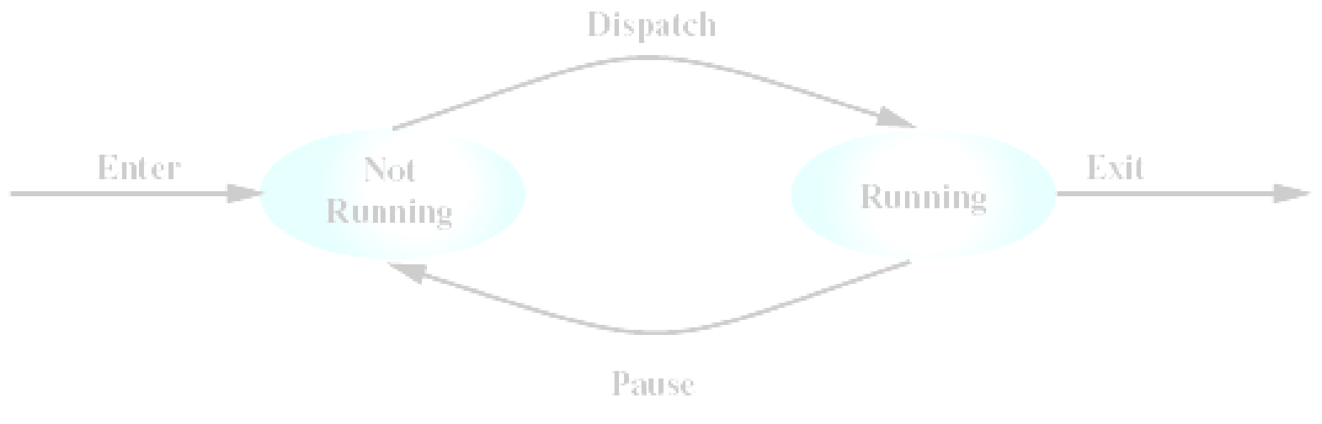


Process state: lifecycle of a process

- The lifetime of a process can be divided into states. As the process executes, it changes its states.
- At any particular instance of time a process can be in one of the states: newly created, waiting for processor, running, suspended and etc.
- OS manages the state transition and scheduling of the processes and facilitates context switching
- There are different models for process state transition for example two state model describe a process life in two states only, while a five state model describe the life of a process in five different states.
- Some Unix based operating systems follow seven state model of process

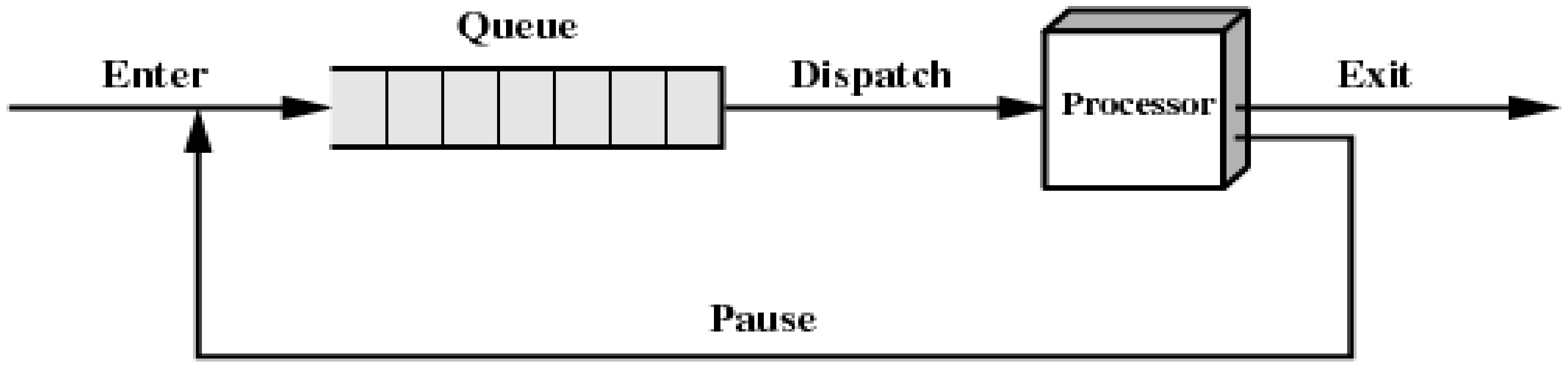
Two-State Process Model

- Process may be in one of two states
  - Running
  - Not-running



(a) State transition diagram

Not-Running Process in a Queue

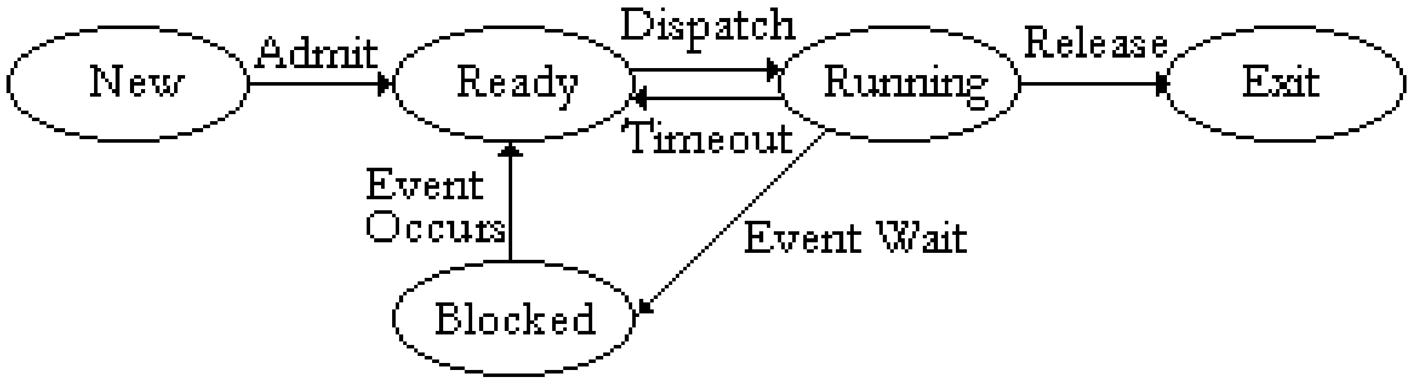


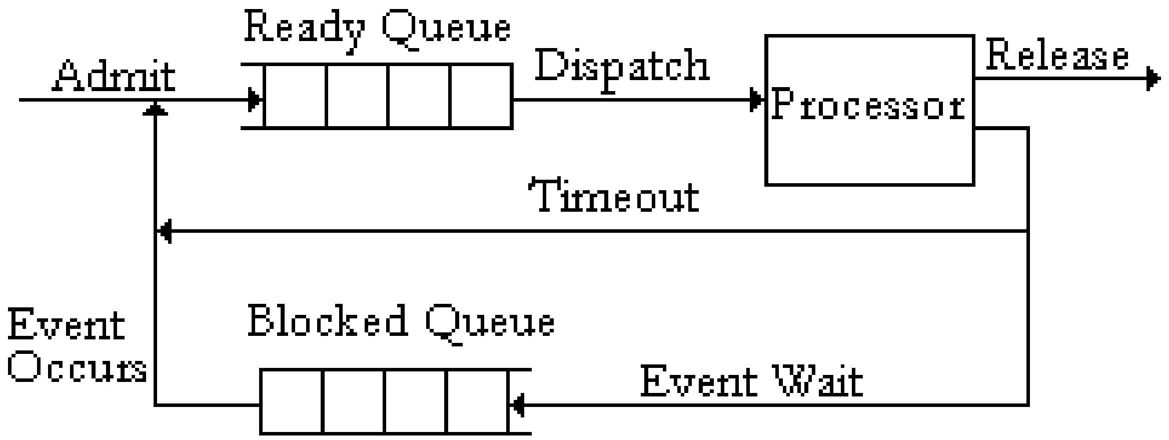
(b) Queuing diagram

Process States: Five-state Model

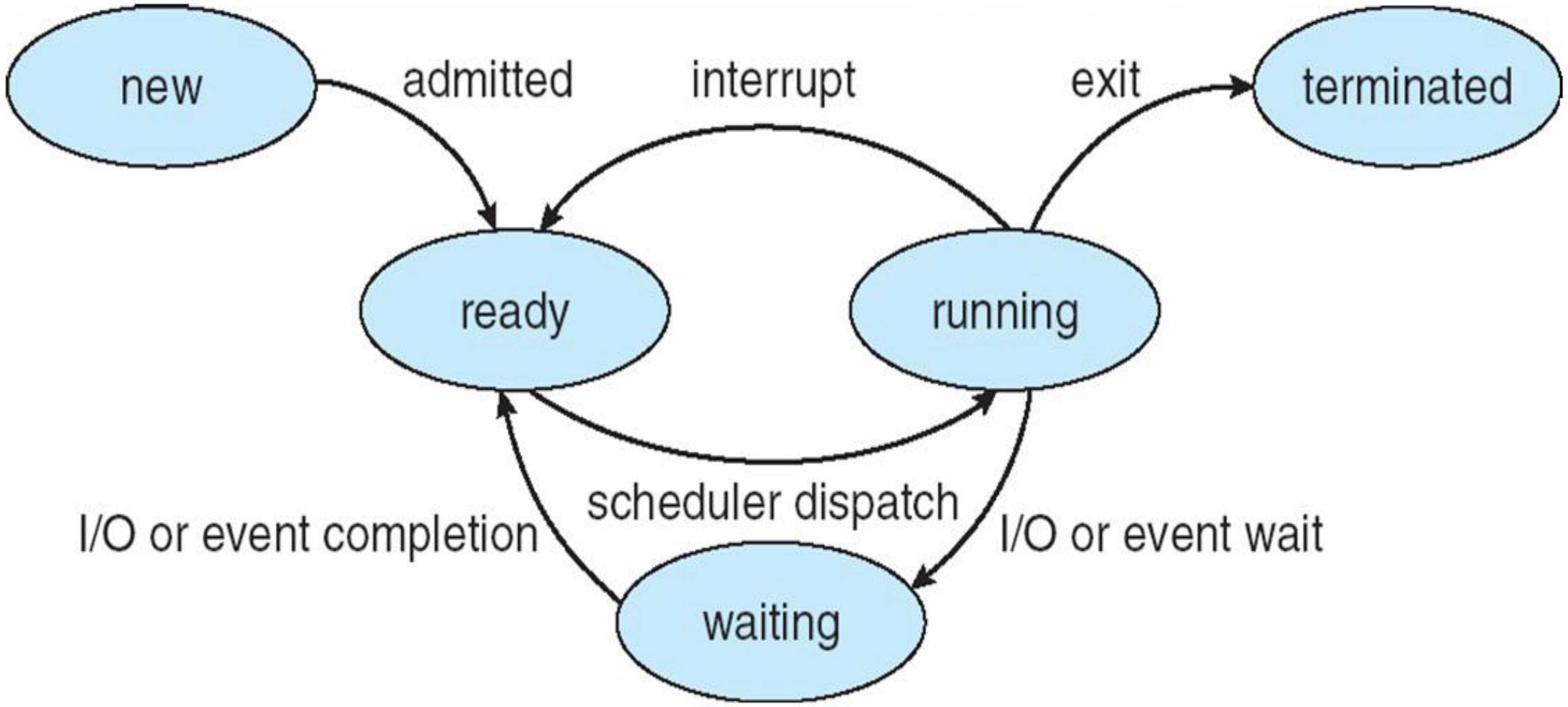
- As a process executes, it changes state
- Each process can be in one of state
  - ***New***: just created, not yet admitted to set of runnable processes
  - ***Ready***: ready to run
  - ***Running***: currently being run
  - ***Blocked/Waiting***: waiting for an event (I/O)
  - ***Exit/Terminate***: completed/error exit

- The names may differ between OS.





State Transition Diagram of a Process



Five-state Model state transitions

- **Null → New :** a new process is created to execute the program,
  - New batch job, log on
  - Created by OS to provide the service
- **New → ready**: OS will move a process from prepared to ready state when it is prepared to take additional process.
- **Ready → Running**: when it is a time to select a new process to run, the OS selects one of the process in the ready state.
- **Running → terminated:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts.
- **Running → Ready**: The process has reached the maximum allowable time or interrupt.
- **Running → Waiting:** A process is put in the waiting state, if it requests something for which it must wait. **Example: System call request**.
- **Waiting → Ready**: A process in the waiting state is moved to the ready state, when the event for which it has been waiting occurs.
- **Ready → Terminated:** If a parent terminates, child process should be terminated
- **Waiting → Terminated:** If a parent terminates, child process should be terminated

Process Creation

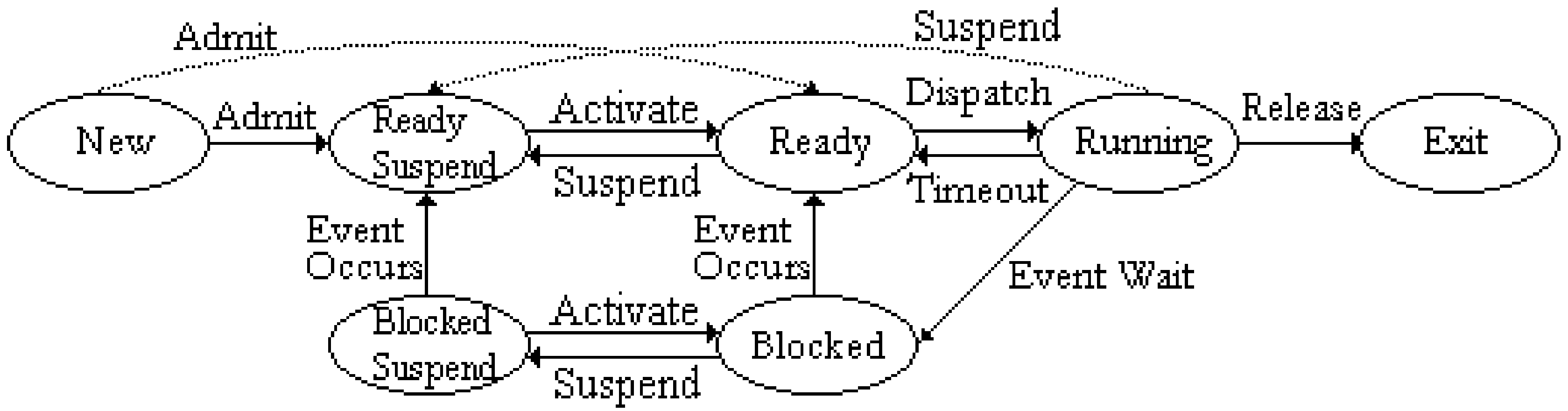
- **When a new process created**, OS perform the following steps:
  - Assign a unique process identifier
  - Allocate space for the process
  - Initialize process control block
  - Set up appropriate linkages
    - Ex: add new process to linked list used for scheduling queue
  - Create or expand other data structures
    - Ex: maintain an accounting file

Change of Process State

- **When a process changes its state**, the following steps are taken by OS:
  - Save context of processor including program counter and other registers
  - Update the process control block of the process that is currently in the Running state
  - Move process control block to appropriate queue – ready; blocked; ready/suspend
  - Select another process for execution
  - Update the process control block of the process selected
  - Update memory-management data structures
  - Restore context of the selected process

Suspending Processes

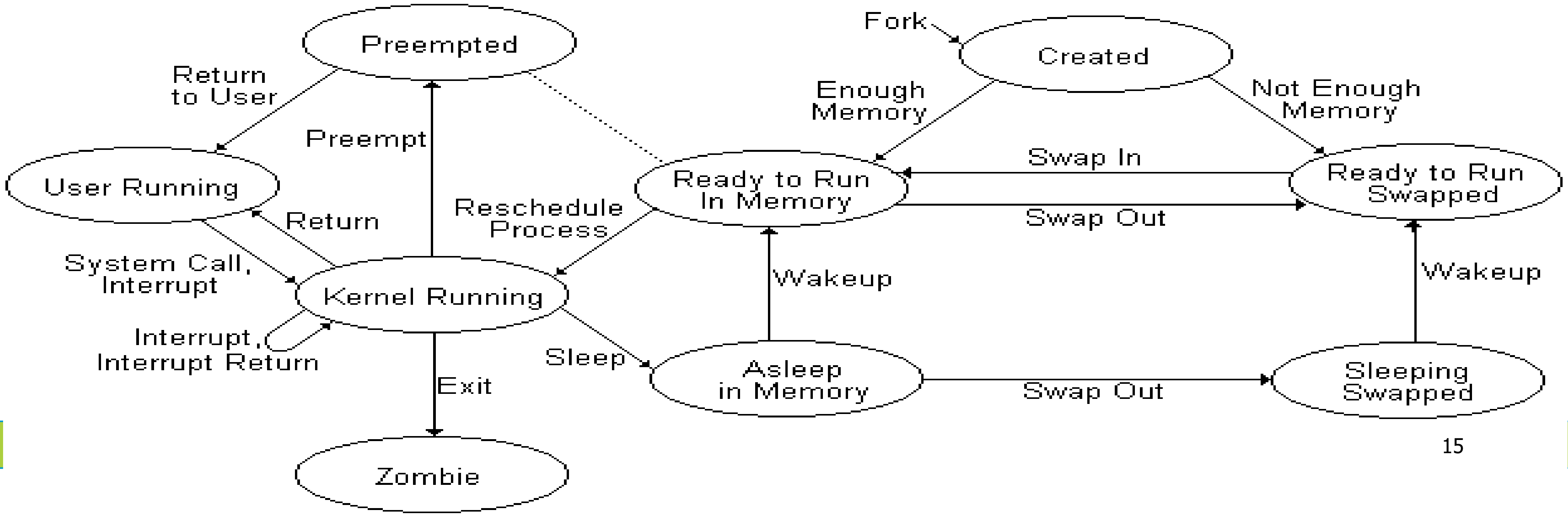
- **OS may suspend a process** by swapping part or all of it to disk, this will increase two more states in the diagram.
  - Suspending a process is most useful if we are waiting for an event that will not arrive soon (printer, keyboard)
  - If not done well, can slow system down by increasing disk I/O activity
- **State Transition Diagram with suspend state(Totally 7 states)**



- **Key States in above model:**
  - **Ready** – In memory, ready to execute
  - **Blocked** – In memory, waiting for an event
  - **Blocked Suspend** – On disk, waiting for an event
  - **Ready Suspend** – On disk, ready to execute

Unix SVR4 Processes

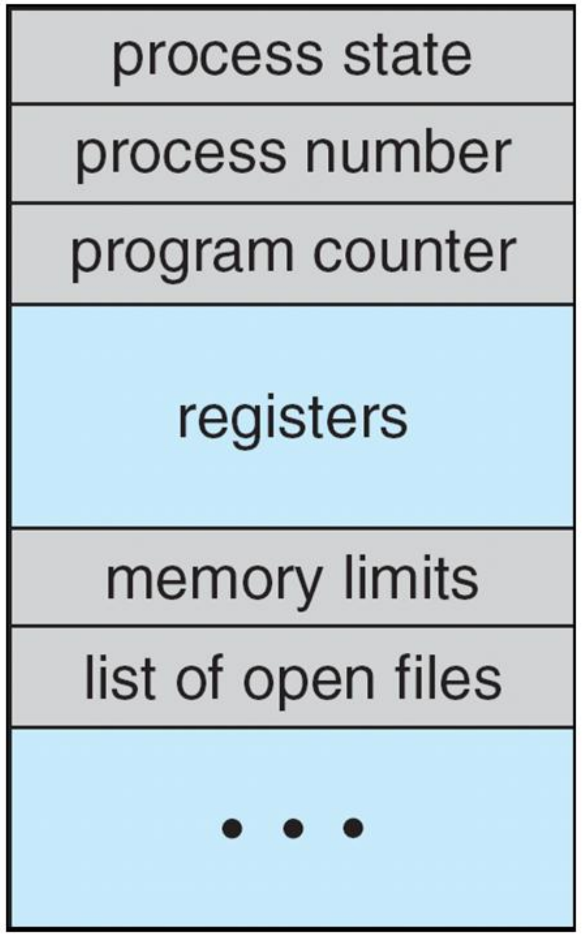
- **Uses 9 processes states**
- ***Preempted*** **and** ***Ready to run***, *in memory* are nearly identical
    - A process may be preempted for a higher-priority process at the end of a system call
- ***Zombie*** – Saves information to be passed to the parent of this process
- **Process 0** – *Swapper*, created at boot
- **Process 1** – *Init*, creates other processes



Process Control Block (PCB)

PCB is a data structure which stores information associated with each process (also called **task control block**). Every process have a PCB, while threads share the PCB of their parent process.

- **PCB contain following information:**
- **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 Control Block details

- **Processor State Information**
  - **Process Identification**
    - Numeric identifiers that may be stored with the process control block include
      - Identifier of this process
      - Identifier of the process that created this process (parent process)
      - User identifier
  - **Stack Pointers**
    - Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Process Control Block details

- **Processor State Information**
  - **Control and Status Registers**

    These are a variety of processor registers that are employed to control the operation of the processor.

    These include

    - ***Program counter**:* Contains the address of the next instruction to be fetched
    - ***Condition codes**:* Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
    - ***Status information**:* Includes interrupt enabled/disabled flags, execution mode

Process Control Block details

- **Process Control Information**
  - **Scheduling and State Information**

    This is information that is needed by the OS to perform its scheduling function.

    Typical items of information:

    •***Process state***: defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).

    •***Priority***: One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)

    •***Scheduling-related information***: This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.

    •***Event:*** Identity of event the process is awaiting before it can be resumed

Process Control Block details

- **Process Control Information**
  - **Data Structuring**
    - A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.
  - **Inter-process Communication**
    - Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.
  - **Process Privileges**
    - Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Need of Process Management

An OS should perform the function of process management in order to control the behavior of different processes running under the current context, We can consider process management as a module of operating system.

A process management module in operating systems is required in order to preform following duties:

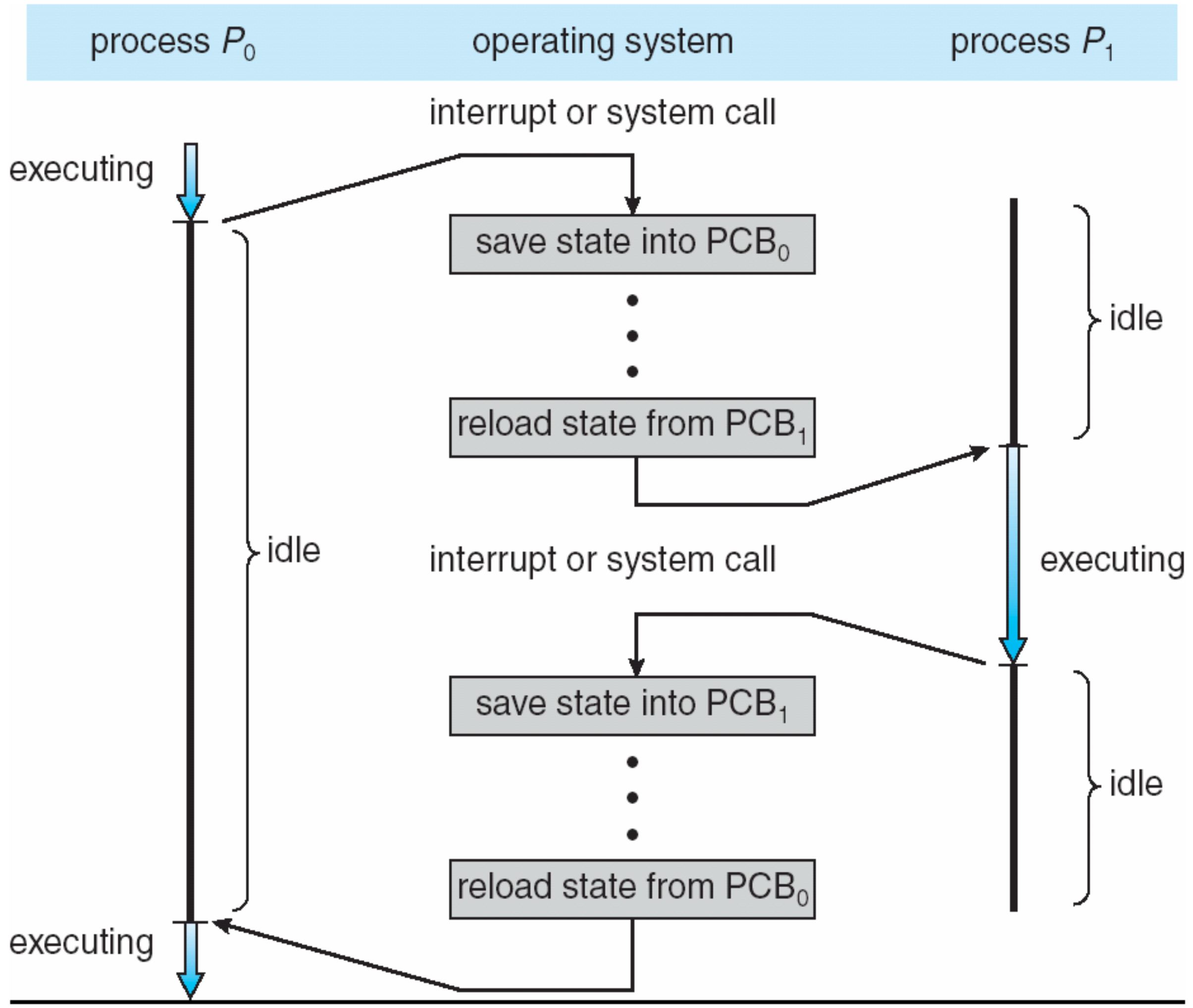
- To be able to create, schedule and terminate a process
- To be able to manage resource utilization across various processes
- To be able to respond to the various signals/messages destined for different processes running under current context and to facilitate their state transition
- To be able to perform context switch among various processes
- To be able to do memory and I/O management for each process etc.

Switching among processes

- In a multi-programming system different processes runs at the CPU in time interleaved fashion
- Since, processor is being shared among various processes, each process get chance of executing its code on CPU in certain order that depends upon scheduling algorithm
- Each process is given a number of time slices to execute on processor and when it completes the time allocated to it, it changes its state and goes into queue, while other process in the queue waiting for processor is given chance to execute its code on CPU.
- The interleaving is possible because of processor is being switched among processes

CPU Switch From Process to Process

- A **context switch** is the process of storing and restoring state (context) of a CPU. So that execution can be resumed from the same point at a later time.
- This enables multiple processes to share a single CPU.
- Switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and list etc.



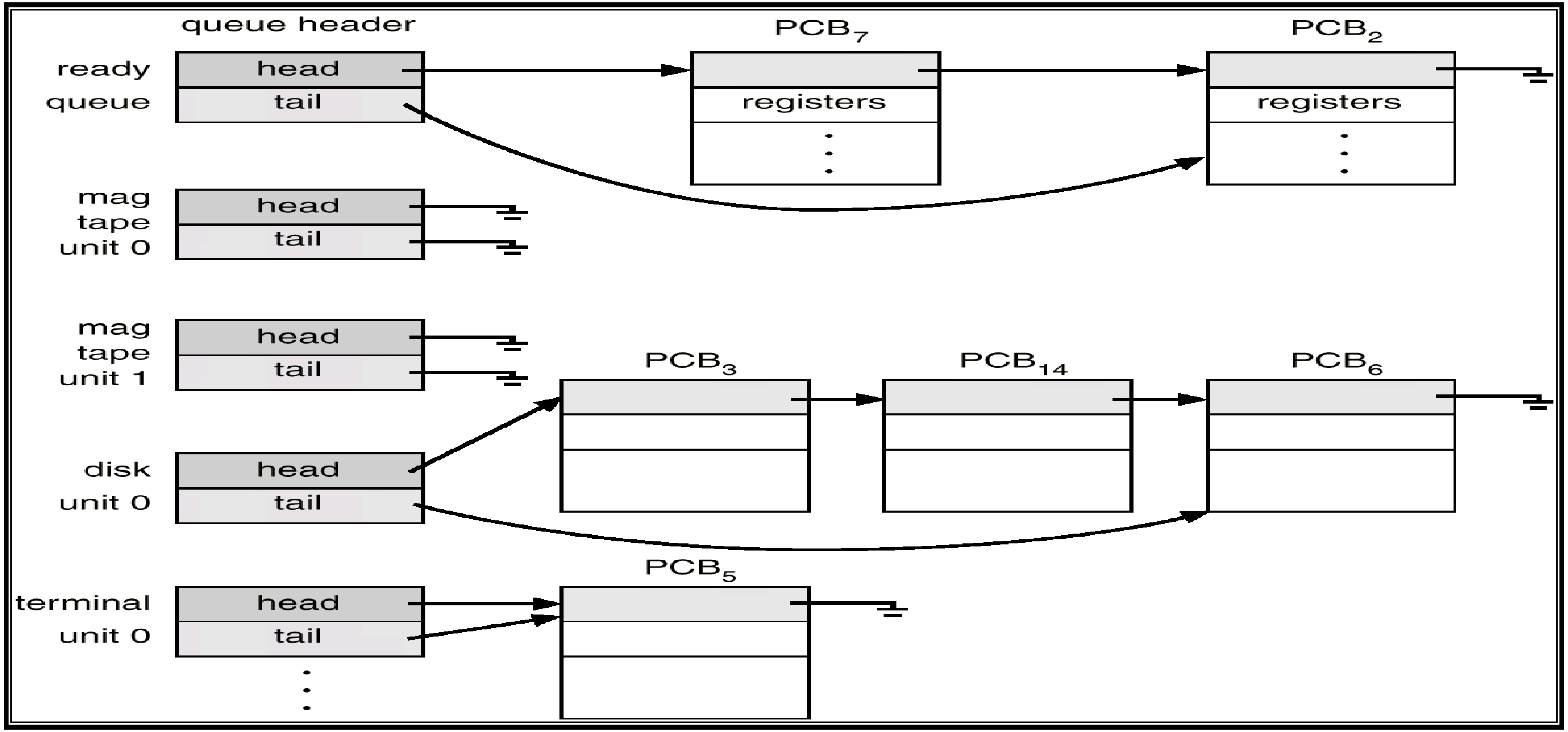
When to Switch a Process

- **Clock interrupt**
  - process has executed for the maximum allowable time slice
- **I/O interrupt**
- **Memory fault**
  - memory address is in virtual memory so it must be brought into main memory
- **Trap**
  - error or exception occurred
  - may cause process to be moved to Exit state
- **Supervisor call**
  - such as file open

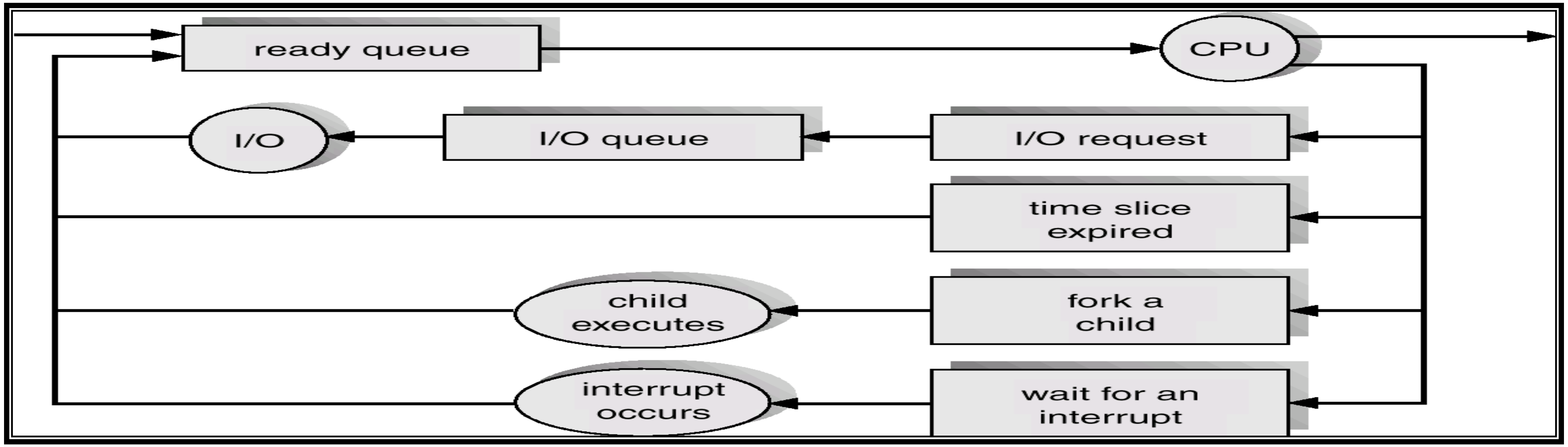
Process Scheduling Queues

- Scheduling is to decide which process to execute and when
- The objective of **multi-programming**
  - To have some process running at all times.
- **Timesharing**: Switch the CPU frequently that users can interact with the program while it is running.
- If there are many processes, the rest have to wait until **CPU is free**.
- **Scheduling queues:**
  - **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.
    - Each device has its own queue.
- Process migrates between the various queues during its life time.

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



- **A new process is initially put in the ready queue**
- Once a process is allocated CPU, the following events may occur
    - A process could issue an I/O request
    - A process could create a new process
    - The process could be removed forcibly from CPU, as a result of an interrupt.
- When process terminates, it is removed from all queues. PCB and its other resources are de-allocated.

Schedulers

- A process migrates between the various scheduling queues throughout its lifetime.
- The OS must select a process from the different process queues in some fashion. The selection process is carried out by a scheduler.
- In a batch system the processes are spooled to mass-storage device.
- **Long-term scheduler (or job scheduler)** – selects which processes should be brought into the ready queue.
  - It may take long time
- **Short-term scheduler (or CPU scheduler)** – selects which process should be executed next and allocates CPU.
  - It is executed at least once every 100 msec.
  - If 10 msec is used for selection, then 9 % of CPU is used (or wasted)
- The long-term scheduler executes less frequently.
- The long-term scheduler controls degree of multiprogramming.
- **Multiprogramming:** the number of processes active in the system.
- **If MPL is stable: average rate of process creation= average departure rate of processes.**

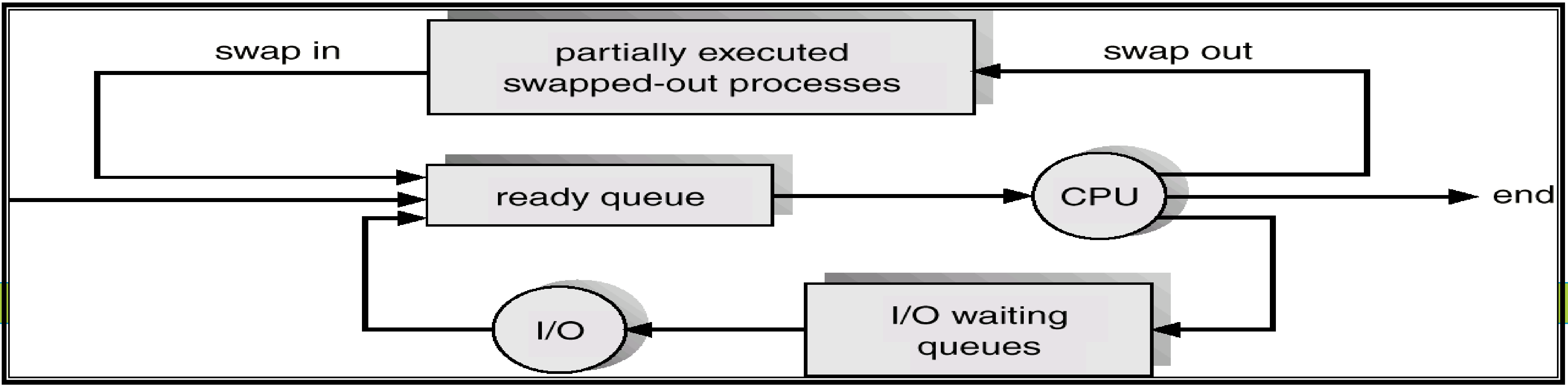
Schedulers (cont…)

- The long-term scheduler should make a careful selection.
- **Most processes are either I/O bound or CPU bound.**
    - **I/O bound process** spends more time doing I/O than it spends doing computation.
    - **CPU bound process** spends most of the time doing computation.
- The **LT scheduler** should select a good mix of I/O-bound and CPU-bound processes.
- **Example:**
    - **If all the processes are I/O bound,** the ready queue will be empty
    - **If all the processes are CPU bound,** the I/O queue will be empty, the devices will go unutilized and the system will be imbalanced.
- **Best performance:** best combination of CPU-bound and I/O-bound process.

Schedulers (cont…)

- Some OSs introduced a medium-term scheduler using swapping.
  - **Key idea**: it can be advantageous, to remove the processes from the memory and reduce the multiprogramming.
- **Swapping:** removal of process from main memory to disk to improve the performance. At some later time, the process can be reintroduced into main memory and its execution can be continued when it left off.
- Swapping improves the process mix (I/O and CPU), when enough main memory is unavailable.

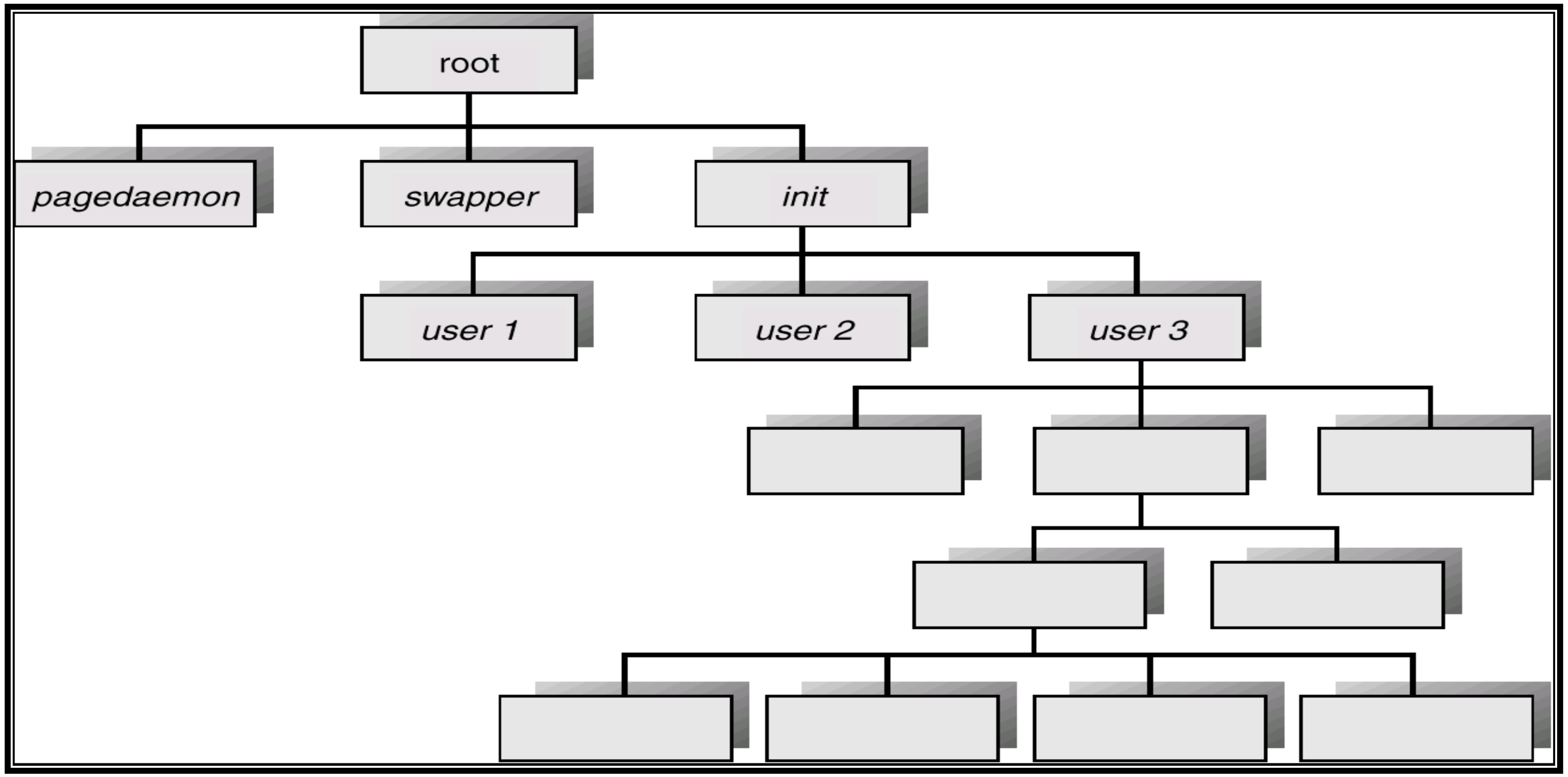
Addition of Medium Term Scheduling



Operations on Processes: process creation

- **A system call is used to create process.**
  - Assigns unique id, Memory Space and PCB in order to initialize it.
- The creating process is called **parent process**.
- **Parent process** creates children processes, which, in turn create other processes, forming a tree of processes.
  - In UNIX **pagedaemon, swapper,** and **init** are children **of root process**. **Users are children of init process.**
- A process needs certain resources to accomplish its task.
  - CPU time, memory, files, I/O devices.
- **When a process creates a new process:**
  - **Resource sharing possibilities.**
    - Parent and children share all resources.
    - Children share subset of parent's resources.
    - Parent and child share no resources.
  - **Execution possibilities**
    - Parent and children execute concurrently.
    - Parent waits until children terminate.

Processes Tree on a UNIX System



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.
    - The new process is a copy of the original process.
    - The **exec system call** is used after a fork by one of the two processes to replace the process memory space with a new program.
- **WINDOWS NT supports both models:**
    - Parent address space can be duplicated or
    - parent can specify the name of a program for the OS to load into the address space of the new process.

UNIX: fork() system call

- fork() is used to create processes. It takes no arguments and returns a process ID.
- fork() creates a new process which becomes the child process of the caller.
- After a new process is created, both processes will execute the next instruction following the fork() system call.
- The checking the return value, we have to distinguish the parent from the child.
- **fork()**
  - If returns a negative value, the creation is unsuccessful.
  - Returns 0 to the newly created child process.
  - Returns positive value to the parent.
- Process ID is of type pit_t defined in sys/types.h
- getpid() can be used to retrieve the process ID.
- The new process consists of a copy of address space of the original process.

UNIX: fork() system call

- If the fork() is executed successfully, Unix will
  - Make two identical copies of address spaces; one for the parent and one for the child.
  - Both processes start their execution at the next statement after the fork().

Parent

```
main()
{
            fork();
            pid=…; →
}
```

Child

```
main()
{
      fork();
      pid=… →
}
```

UNIX: fork() system call

Parent

```
void main(void)   {
        pid=fork();
        if (pid==0)
                        ChildProcess();
     else
                        ParentProcess();
   }
}
Void ChildProcess(void)
     {
        }
Void ParentProcess(void)
     {
}
```

PID=3456

Child

```
void main(void)   {
        pid=fork();
        if (pid==0)
                        ChildProcess();
     else
                        ParentProcess();
   }
}
Void ChildProcess(void){
        }
Void ParentProcess(void)
     {
}
```

PID=0

Process Termination

- Processes terminate in one of ways:
    - Normal Termination occurs by a return from **main** or when requested by an explicit call to **exit** or **_exit**.
    - Abnormal Termination occurs as the default action of a signal or when requested by **abort**.
- Process executes last statement and asks the operating system to decide it (**exit**).
    - Output data from child to parent (via **wait**).
    - Process' resources are deallocated by operating system.
- Parent may terminate the execution of children processes (**abort**).
    - Child has exceeded allocated resources.
    - Task assigned to child is no longer required.
    - Parent is exiting.
        - Operating system does not allow child to continue if its parent terminates.
        - Cascading termination.

Cooperating Processes

- Processes executing concurrently in the OS may be either **independent processes or cooperating processes.**
  - A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system.
  - Any process that does not share data with any other process is independent.
  - A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system.
  - Clearly, any process that shares data with other processes is a cooperating process.
- **Advantages of process cooperation**
  - **Information sharing**
  - **Computation speed-up**
    - Break into several subtasks and run in parallel
  - **Modularity**
    - Constructing the system in modular fashion.
  - **Convenience**
    - User will have many tasks to work in parallel → editing, compiling, printing

Inter-process Communication (IPC)

- When processes communicate with each other it is called "Inter-process communication" (IPC). Processes frequently need to communicate, for instance in a shell pipeline, the output of the first process need to pass to the second one, and so on to the other process.
- IPC facility provides a mechanism to allow processes to communicate and synchronize their actions.
- **There are two fundamental models for implementation of IPC:**

  1. Shared memory

  2. Message passing
- The Shared-memory method requires communication processes to share some variables.
- The responsibility for providing communication rests with the programmer.
- **Example: producer-consumer problem**.

IPC(Cont…..)

(1) Shared memory:- a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

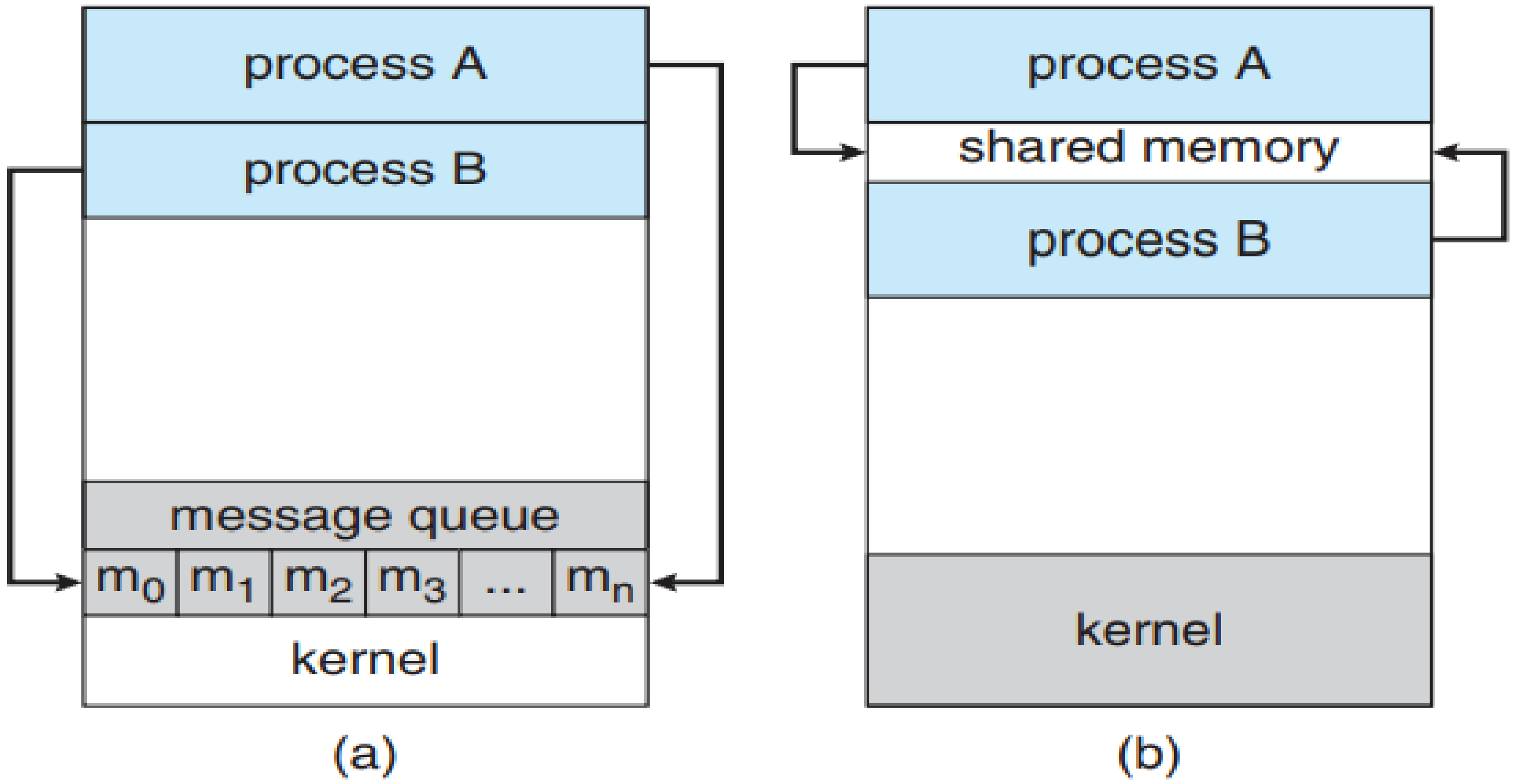
- It allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.
- It is faster than message passing, as message-passing systems are typically implemented using **system calls** and thus require the more time consuming task of kernel intervention.
- **In contrast**, 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.

(2) Message passing:- communication takes place by means of messages exchanged between the cooperating processes.

- it is useful for exchanging **smaller amounts of data**, because no conflicts need be avoided.
- It is also easier to implement than is shared memory for **intercomputer** communication.
- Both of the models just discussed are common in operating systems, and many systems implement both.

Communication Models

Processes



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

Producer-Consumer Problem

- **producer-consumer problem (bounded-buffer problem)** is a classical example of a multi- process synchronization problem.
- The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
- The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using Semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

Producer-Consumer Problem: Shared memory

- 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.
    - Producer can produce any number of items.
    - Consumer may have to wait
  - ***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;
```

in points to next free position.
out points to first full position.

…….
Bounded-Buffer – Producer Process

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

Bounded-Buffer – Consumer Process

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

IPC: Message Passing

- **Message system** – processes communicate with each other without resorting to shared variables.
- If P and Q want to communicate, a communication link exists between them.
- OS provides this facility.
- **IPC facility provides two operations:**
  - **send(*message*)** – message size fixed or variable
  - **receive(*message*)**
- **If *P* and *Q* wish to communicate, they need to:**
  - establish a *communication link* between them
  - exchange messages via send/receive
- **Implementation of communication link**
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- 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?
  - (How much is the buffer space ?)
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Message passing systems

1. Direct or Indirect communication
2. Synchronous or asynchronous communication
3. 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.
- This exhibits both symmetry and asymmetry in addressing
  - **Symmetry:** Both the sender and the receiver processes must name the other to communicate.
  - **Asymmetry:** Only sender names the recipient, the recipient is not required to name the sender.
    - The send and receive primitives are as follows.
      - Send (P, message)– send a message to process P.
      - Receive(id, message)– receive a message from any process.
- **Disadvantages: Changing a name of the process creates problems.**

Indirect Communication

- The messages are sent and received from **mailboxes** (also referred to as ports).
- A **mailbox is an object**
  - Process can place messages
  - Process can remove messages.
- Two processes can communicate only if they have a **shared mailbox.**
- **Operations**
  - create a new mailbox
  - 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(cont…)

- **Mailbox sharing**
  - *P_1, P_2,* and *P_3* share mailbox A.
  - *P_1*, sends; *P_2* and *P_3* receive.
  - Who gets a 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.
- **Properties of a link:**
  - A link is established if they have a shared mailbox
  - A link may be associated with more than two boxes.
  - Between a pair of processes they may be number of links
  - A link may be either unidirectional or bi-directional.
- **OS provides a facility**
  - To create a mailbox
  - Send and receive messages through mailbox
  - To destroy a mail box.
- The process that creates mailbox is a owner of that mailbox
- The ownership and send and receive privileges can be passed to other processes through system calls.

Indirect Communication(cont…)

- 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.
- **Example:**
- **Producer process:**

```
repeat
 ….
Produce an item in nextp
 …
send(consumer,nextp);
until false;
```

- Consumer process
  - **repeat** …. receive(producer, nextc);…. Consume the item in nextc … **until false;**

Synchronous or asynchronous

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.
  - **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - **Non-blocking send:** The sending process sends the message and resumes operation.
  - **Blocking receive:** The receiver blocks until a message is available.
  - **Non-blocking receive:** The receiver receives either a valid message or a null.

Automatic and explicit buffering

- A link has some capacity that determines the number of messages that can reside in it temporarily.
- Queue of messages is attached to the link; implemented in one of three ways.
  1. Zero capacity – 0 messages
     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.
- In non-zero capacity cases a process does not know whether a message has arrived after the send operation.
- The sender must communicate explicitly with receiver to find out whether the later received the message.
- **Example:** Suppose P sends a message to Q and executes only after the message has arrived.
- Process P:
  - **send (Q. message)** : send message to process Q
  - **receive(Q,message) :** Receive message from process Q
- Process Q
  - Receive(P,message)
  - Send(P,”ack”)

Exception conditions

- **When a failure occurs error recovery (exception handling) must take place.**
- **Process termination**
  - A sender or receiver process may terminate before a message is processed. (may be blocked forever)
  - A system will terminate the other process or notify it.
- Lost messages
  - Messages may be lost over a network
  - Timeouts; restarts.
- Scrambled messages
  - Message may be scrambled on the way due to noise
  - The OS will retransmit the message
  - Error-checking codes (parity check) are used.

Reading Assignment:

- ➢ Client-Server Communication
  - ✓ Sockets
  - ✓ Remote Procedure Calls
  - ✓ Remote Method Invocation (Java)