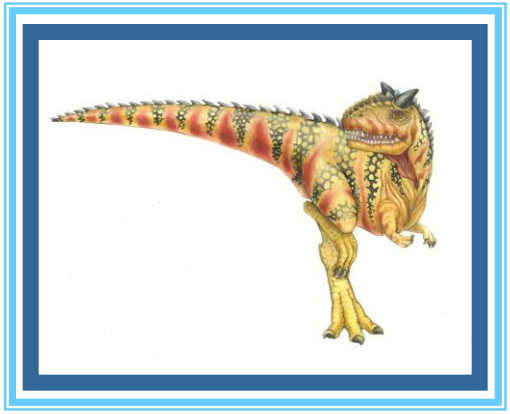
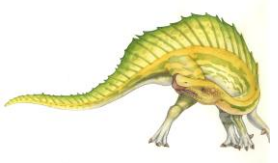
Chapter 3: Processes



Chapter 3: Processes

- Process Concept
- Process Models
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

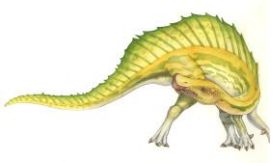


Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

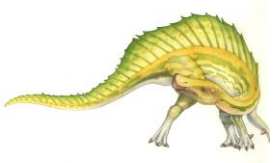
Requirement of an Operating System

- Interleave the execution of multiple processes to maximize processor utilization while providing reasonable response time
- Allocate resources to processes
- Support interprocess communication and user creation of processes



Manage execution of Application

- Resources made available to multiple applications
- Processor is switched among multiple application
- The processor and I/O devices can be used efficiently

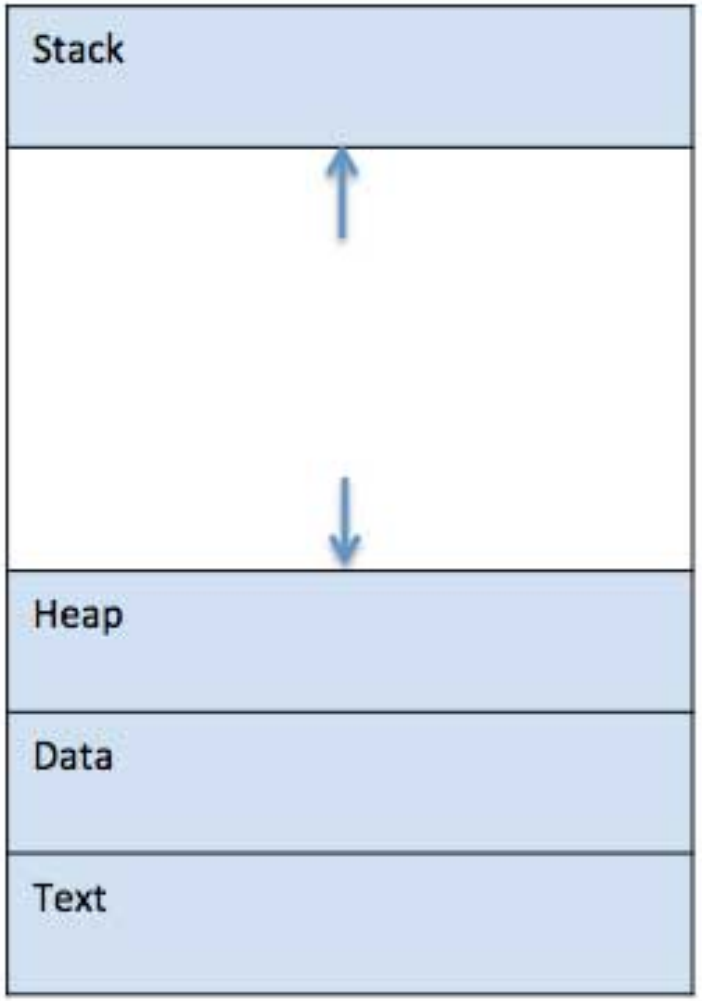


Process Concept

- An operating system executes a variety of programs:
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
    - The text segment of a process's memory contains the executable code or machine instructions of the program
    - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data (smaller than heap)
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
    - The data segment holds the initialized and uninitialized global and static variables used by the program
  - **Heap** containing memory dynamically allocated during run time
    - The heap allows the program to request memory as needed and release it when no longer required. (not fixed size)

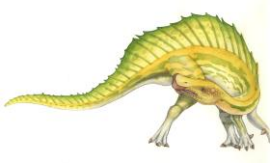
Process

- When a program is loaded into the memory and it becomes a process, it can be divided into four sections — **stack, heap, text and data**.



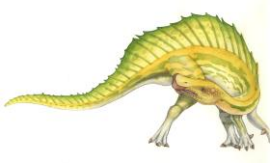
Process

- Stack
  - The process Stack contains the temporary data such as method/function parameters, return address and local variables.
- Heap
  - This is dynamically allocated memory to a process during its run time.
- Data
  - contains the global and static variables.
- Code/Text Segment
  - contains machine code of the compiled program



Process Concept (Cont.)

- 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
- Same user may invoke 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 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

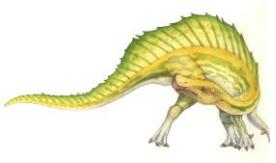
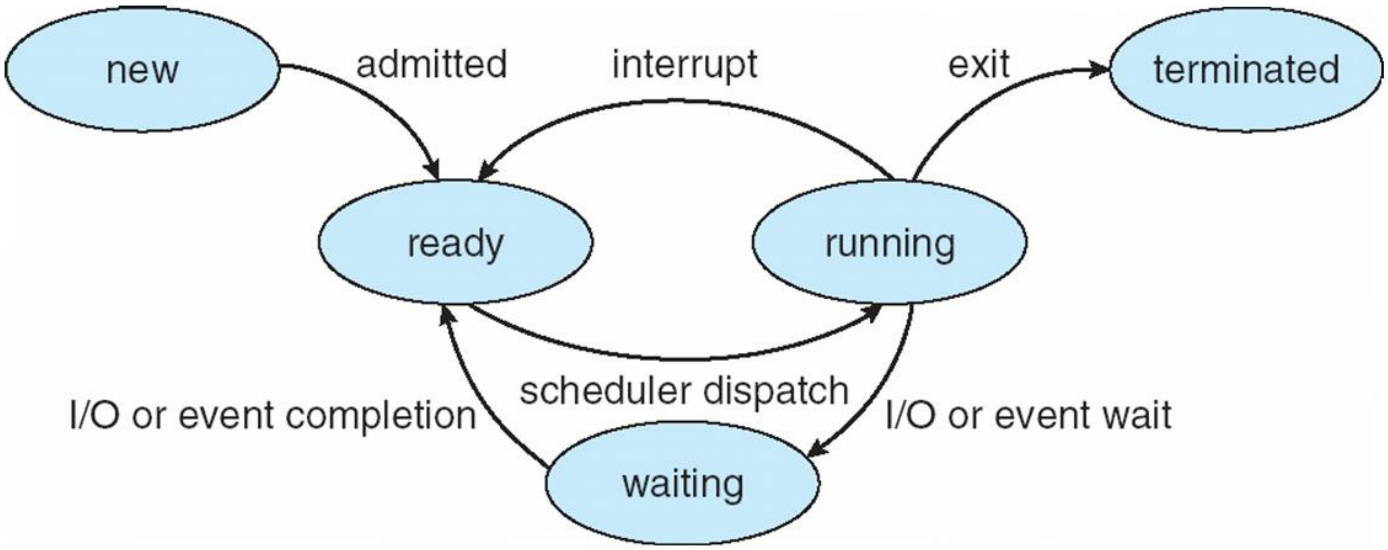


Diagram of Process State



Process control box

- A process in an operating system is represented by a data structure known as a process control block (PCB) or process descriptor
- Information associated with each process

(also called **task control block**)

- Contains the process elements
- Created and manage by the operating system
- Allows support for multiple processes

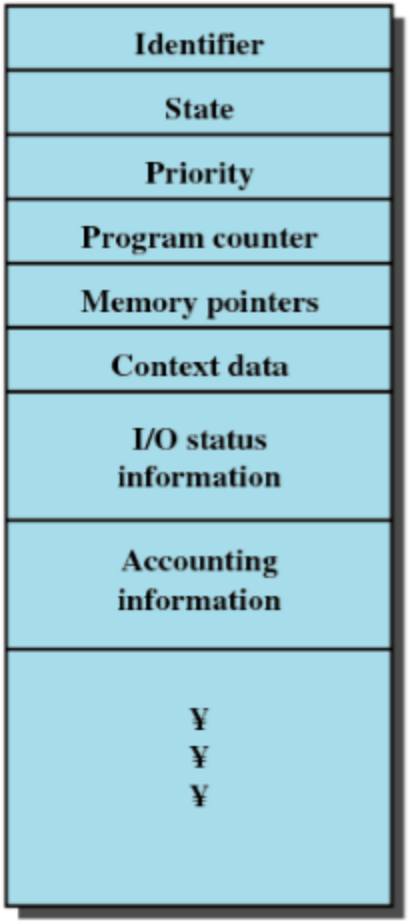
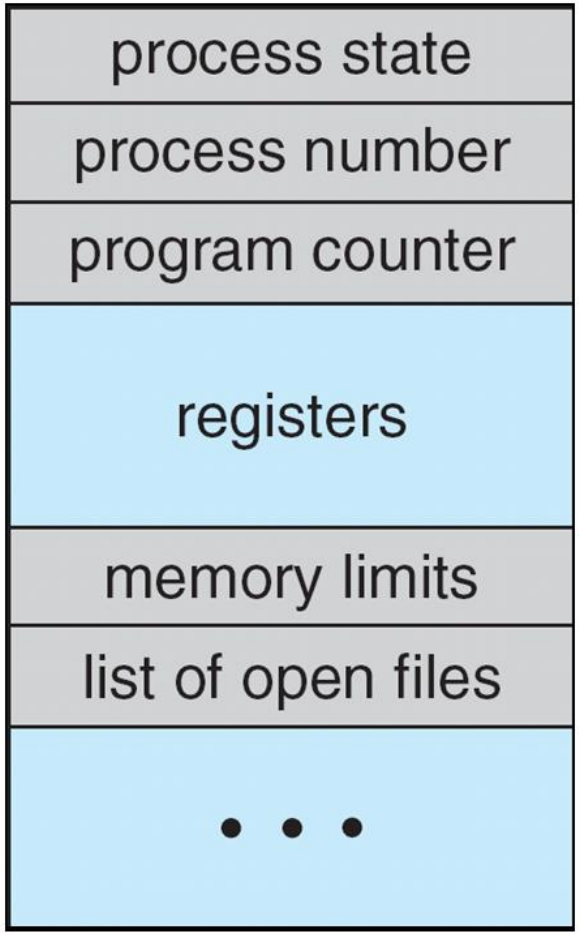


Figure 3.1 Simplified Process Control Block

Process Control Block (PCB)

- Process state – running, waiting, etc
- **Identifier (**A unique identifier associated with each process.)
- **Priority-**Priority level relative to other processes
- **Program counter** – location of instruction to next execute
  - this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward
- **Context data** (data in registers in processor, while process is executing)
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process(Pointers to the upper and lower bounds of the memory)
- 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



Traces of process

- characterize the behavior of an individual process by listing the sequence of instructions
  - Sequence of instruction that execute for a process,
  - Such a listing is referred to as a trace of the process.
- characterize behavior of the processor by showing how the traces of the various processes are interleaved.
- **Dispatcher** is a program that switches the processor from one process to another

Traces of process

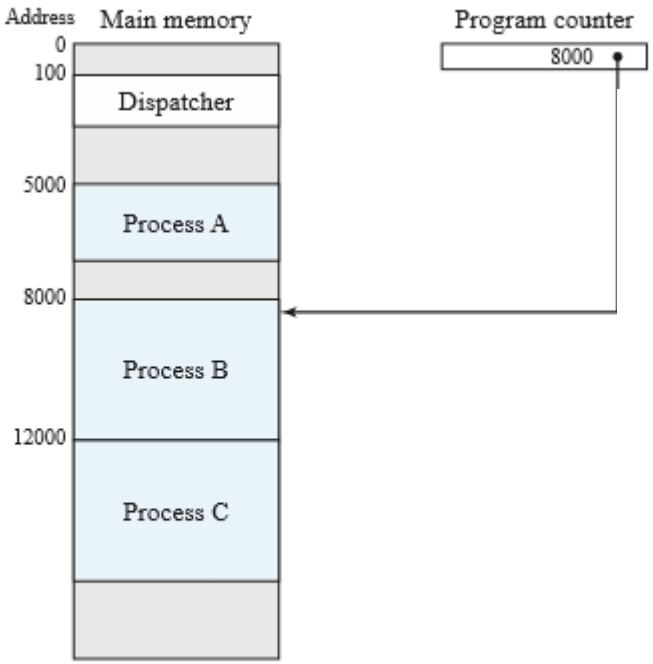


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

Traces of process

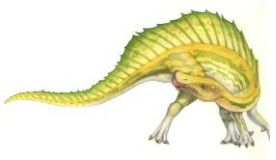
1	5000	27	12004
2	5001	28	12005
3	5002	----------Timeout	
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------Timeout		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----------Timeout	
16	8003	41	100
----------I/O Request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		----------Timeout	

100 = Starting address of dispatcher program

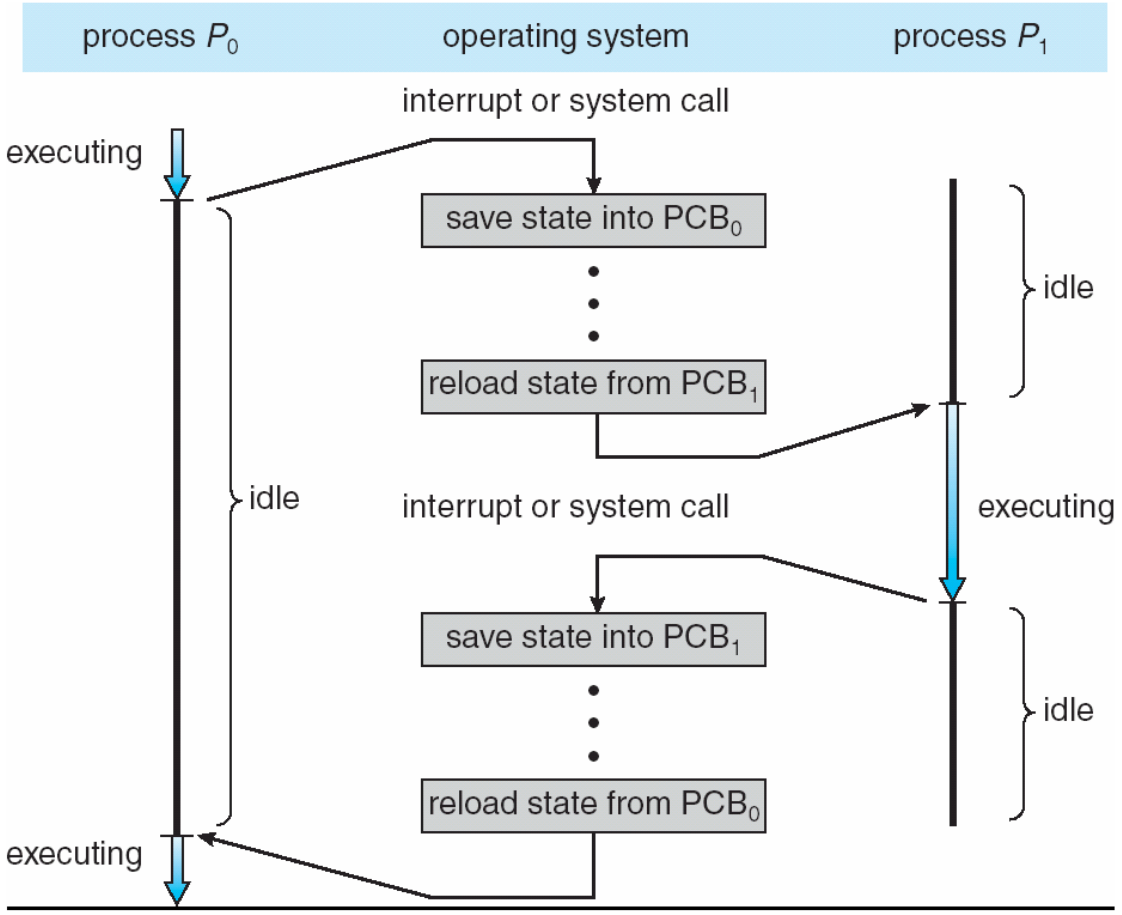
Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

- We assume that the OS only allows a process to continue execution for a maximum of six instruction cycles
- Then it is interrupted;
- This prevents any single process from monopolizing processor time

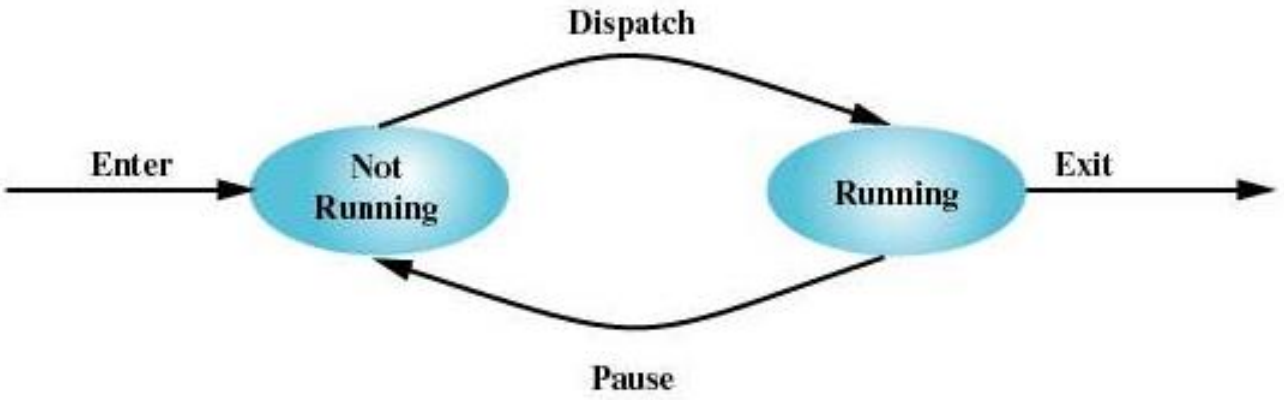


CPU Switch From Process to Process



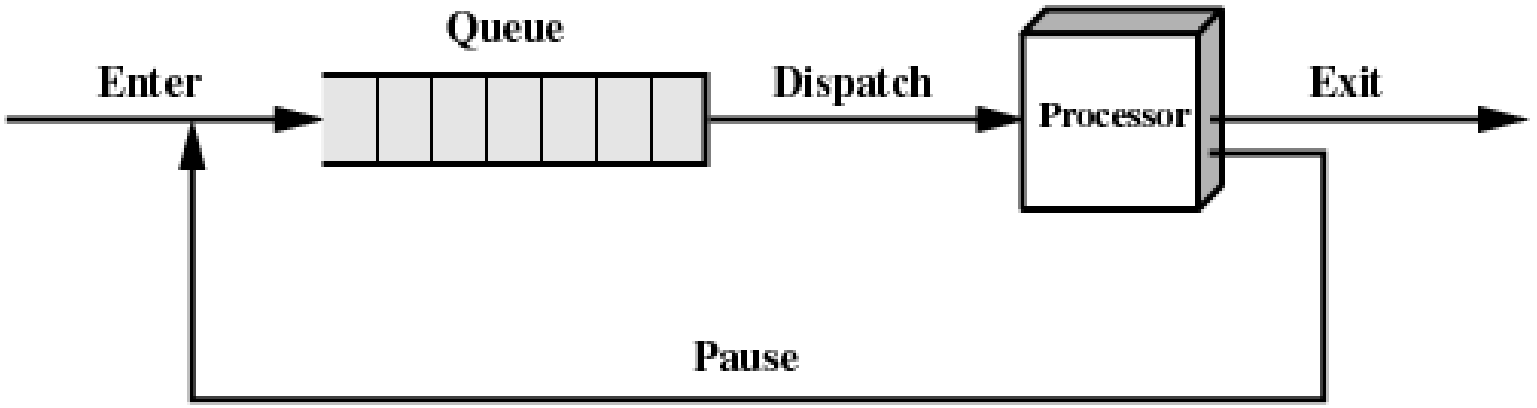
Two state model

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

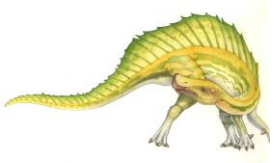


(a) State transition diagram

Not Running Process in Queue



(b) Queuing diagram



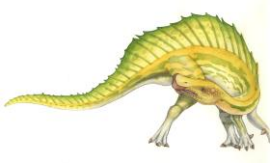
Five State Model

- If all processes were always ready to execute, then the queuing discipline would be effective.
- The queue is a first-in-first-out list and the processor operates in round-robin fashion on the available processes.
- Inadequate Implementation
  - some processes in the Not Running state are ready to execute
  - others are blocked, waiting for an I/O operation to complete.

Five State Process Model

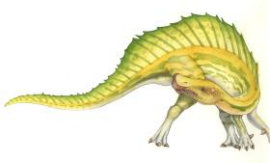
- A natural way to handle this situation is to split the Not Running state into two states:
- Not Running
  - Ready to execute
- Blocked
  - waiting for I/O

- the dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest



Five State Process Model

- 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
- The New and Exit states are useful constructs for process management.
- While a process is in the new state, information concerning the process that is needed by the OS is maintained in control tables in main memory.



Five State Process Model

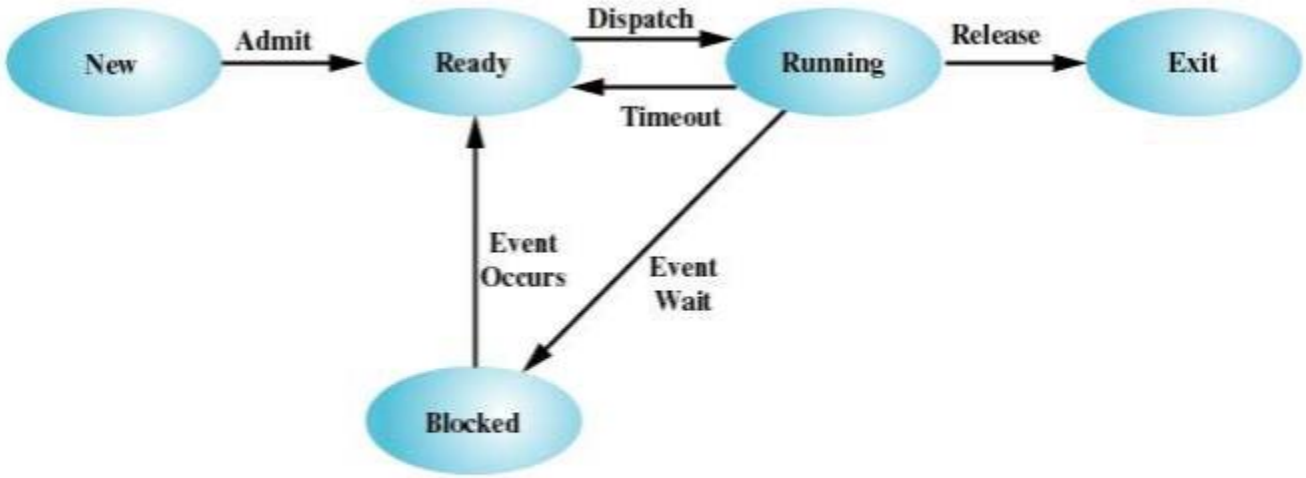
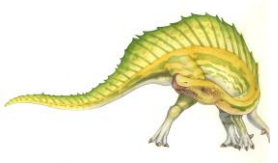


Figure 3.6 Five-State Process Model



Process States

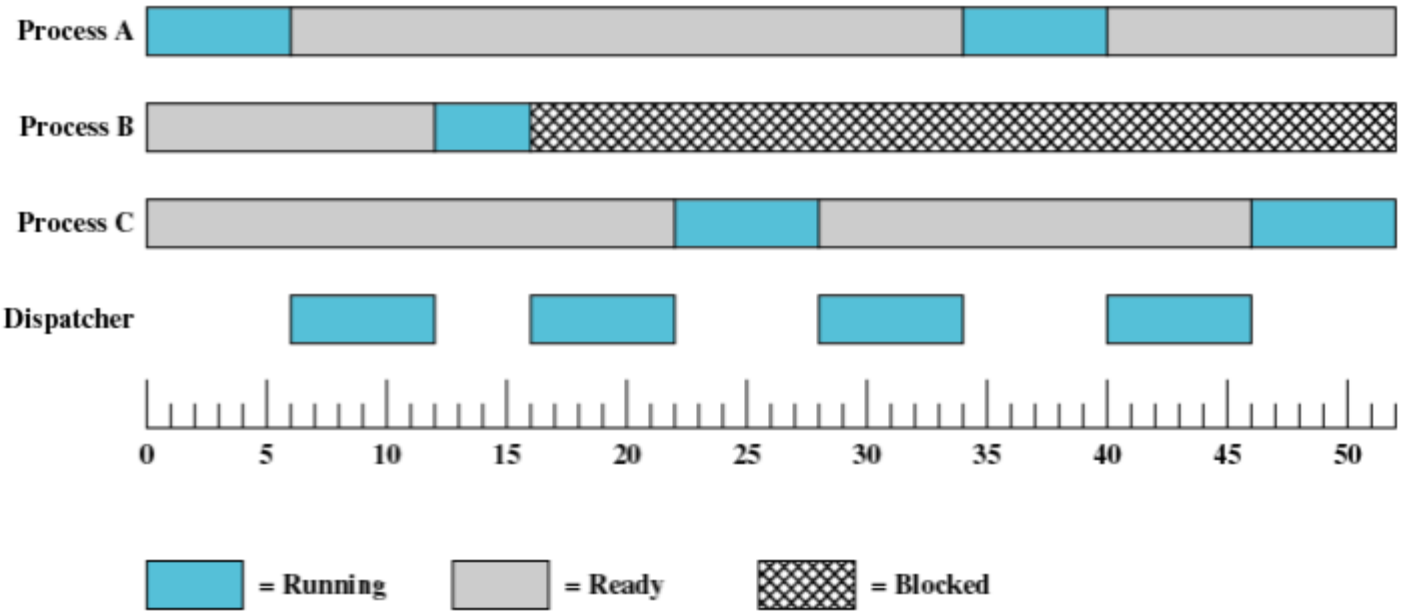
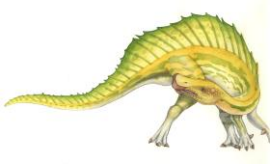
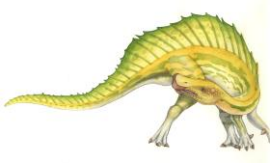


Figure 3.7 Process States for Trace of Figure 3.4



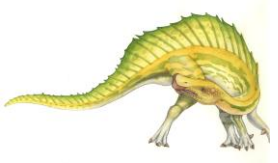
State Transitions

- **Null →New:** A new process is created to execute a program. This event occurs for any of the reasons listed in Table 3.1.
- **New → Ready:** The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process.
- **Ready → Running:** When it is time to select a process to run, the OS chooses one of the processes in the Ready state.
  - job of the scheduler
- **Running →Exit:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts. See Table 3.2.



State Transitions

- **Running → Ready:**
  - running process has reached the maximum allowable time for uninterrupted execution.
  - High Priority process is submitted in the ready queue.
  - a process may voluntarily release control of the processor
    - background process that performs some accounting or maintenance function periodically.
- **Running →Blocked:**
  - A process is put in the Blocked state if it requests something for which it must wait.
- **Blocked →Ready:** A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs.
- **Ready →Exit: I**n some systems, a parent may terminate a child process at any time. Also, if a parent terminates, all child processes associated with that parent may be terminated.



Process Creation

Table 3.1 Reasons for Process Creation

New batch job	The operating system is provided with a batch job control stream, usually on tape or disk. When the operating system is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The operating system can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Process Termination

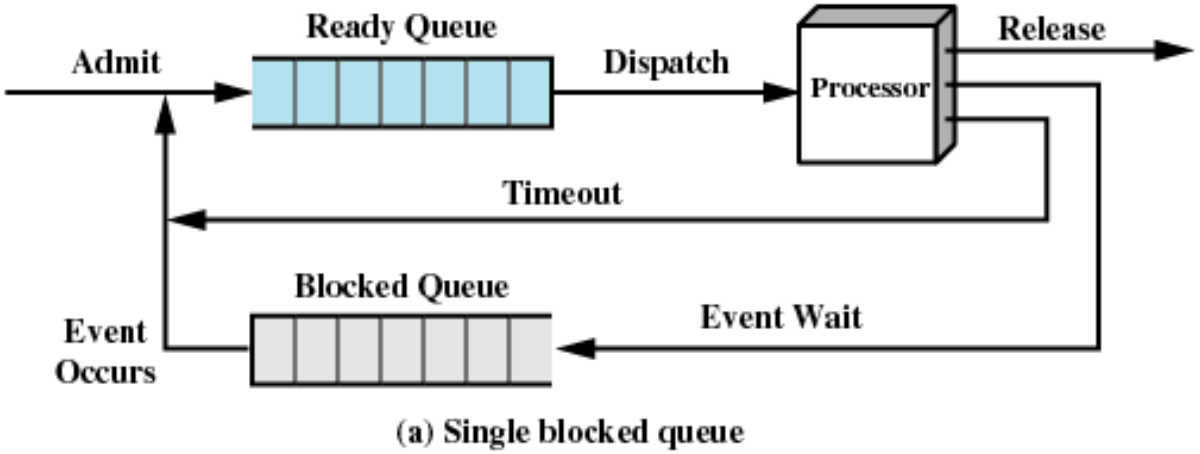
Table 3.2 Reasons for Process Termination

Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

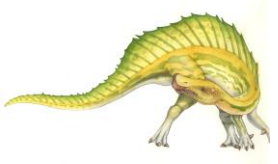
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **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

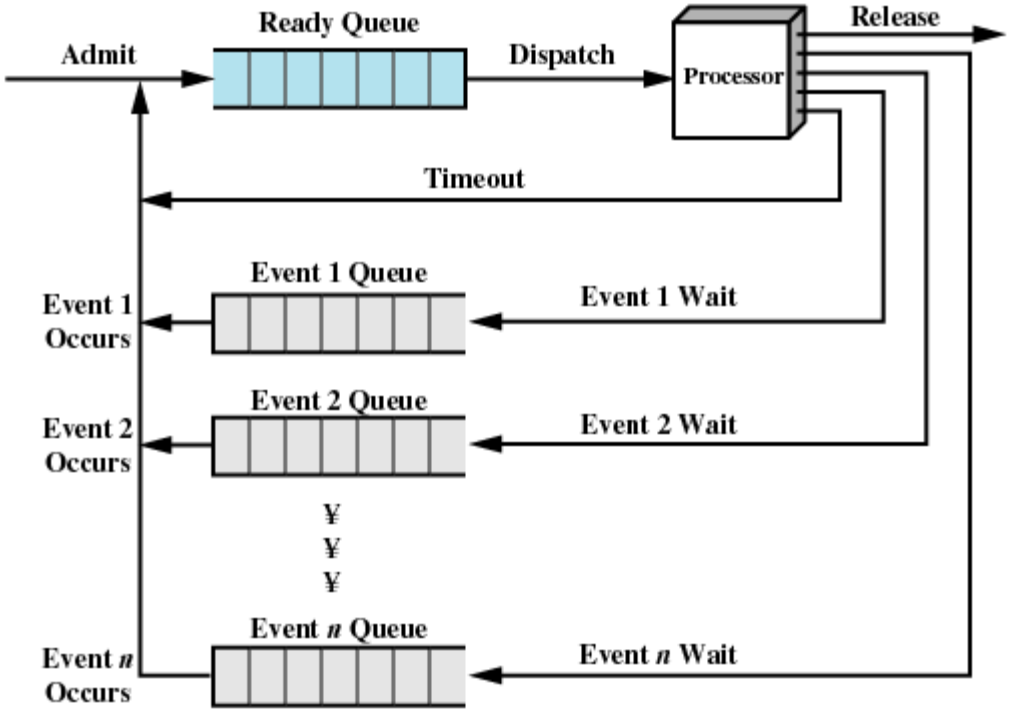
Using two Queue's



(a) Single blocked queue



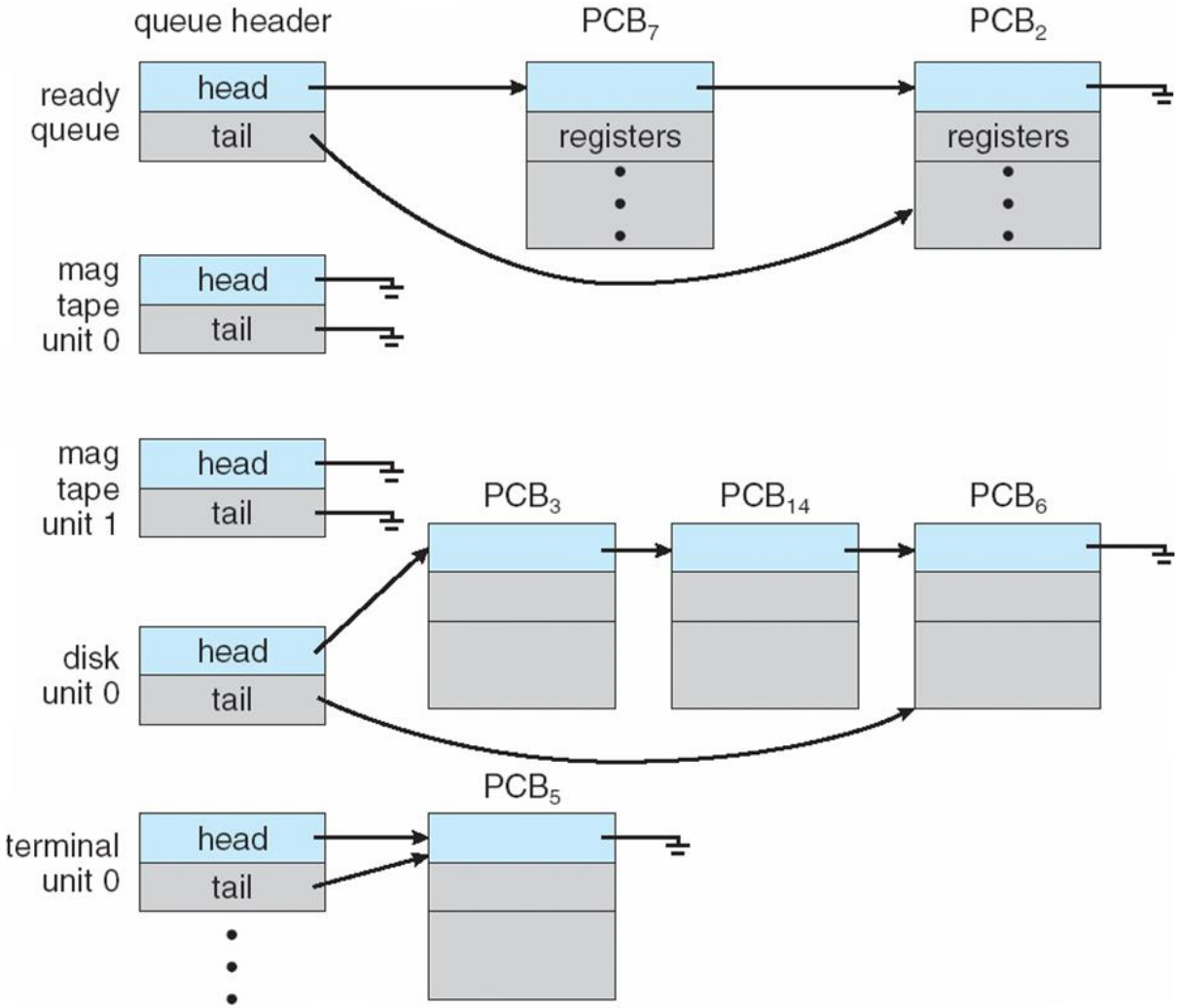
Multiple Blocked Queue's

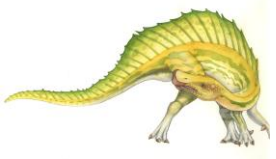


(b) Multiple blocked queues

Figure 3.8 Queuing Model for Figure 3.6

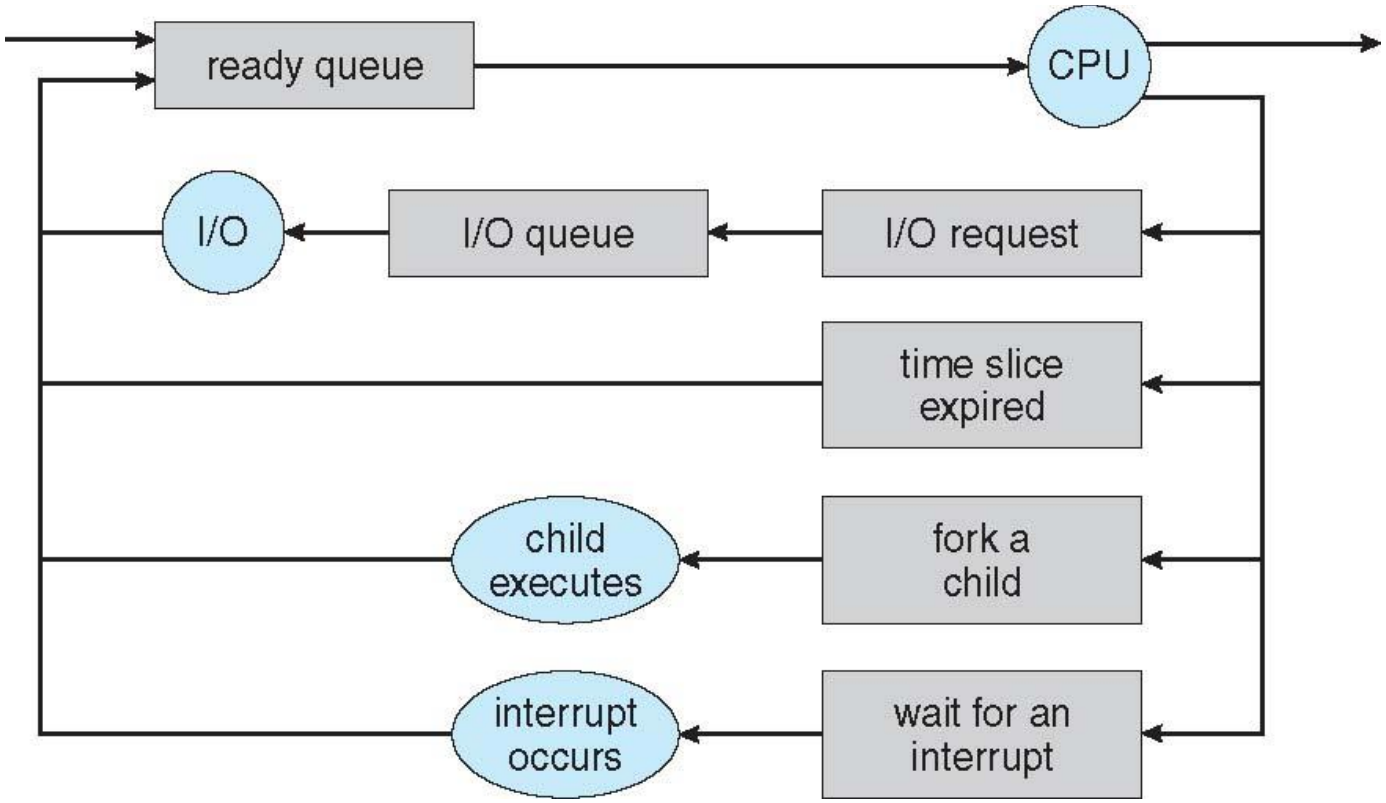
Ready Queue And Various I/O Device Queues

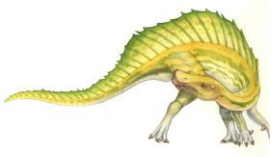




Representation of Process Scheduling

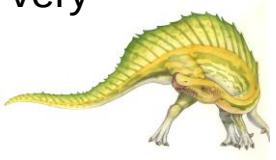
- **Queueing diagram** represents queues, resources, flows





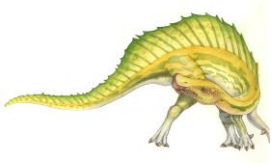
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
  - Often, the short-term scheduler executes at least once every 100 milliseconds
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- 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***



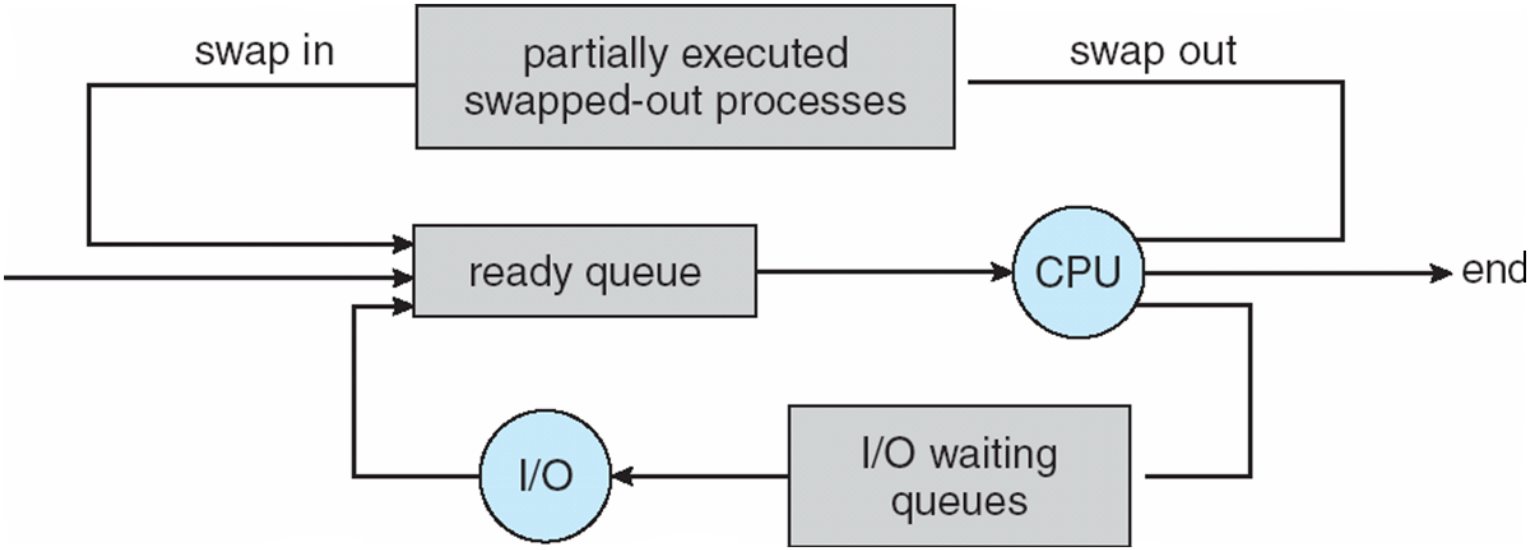
Schedulers

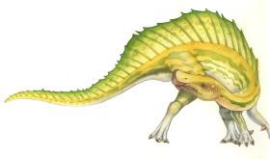
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***
- If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- What if all processes are CPU bound?
- The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.



Addition of Medium Term Scheduling

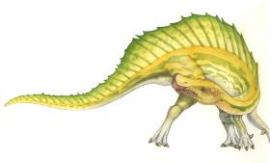
- **Medium-term scheduler** can be added
- degree of multiple programming needs to decrease
- Necessary to improve the process mix.
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**





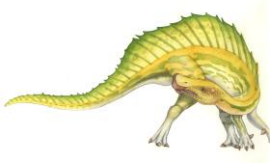
Context Switch

- 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
  - 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



Operations on Processes

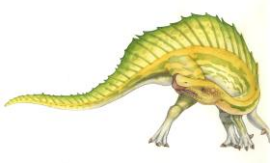
- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next



Suspended Processes

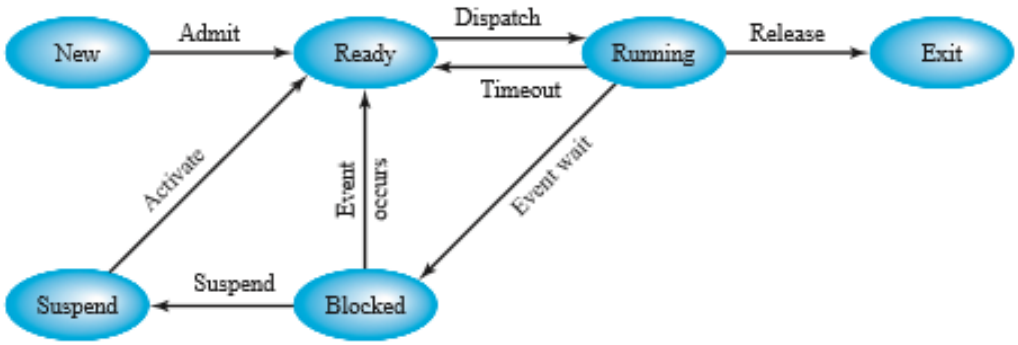
- Processor is faster than I/O so all processes could be waiting for I/O
- Multiprogramming solve the problem to some extent.
  - processor could be idle most of the time.
- A solution is swapping, which involves moving part or all of a process from main memory to disk.
- When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out onto disk into a suspend queue.

- This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended.
- Two Choices:
  - The OS then brings in another process from the suspend queue
  - it honors a new-process request.
- Execution then continues with the newly arrived process.

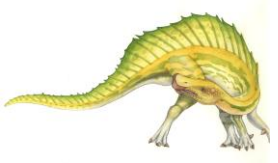


Suspended Processes

- A new state must be added to our process behavior model:
- the Suspend state.
  - When all of the processes in main memory are in the Blocked state
  - the OS can suspend one process by putting it in the Suspend state and transferring it to disk.
  - The space that is freed in main memory can then be used to bring in another process.



(a) With one suspend state

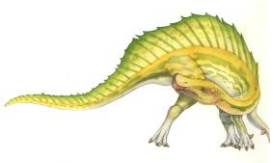


Suspended Processes

- When OS has performed Swapping out Operation, It has two choices to bring new process in memory.
  - It can admit a newly created process
  - or it can bring in a previously suspended process.
- Preference should be to bring in a previously suspended process, to provide it with service rather than increasing the total load on the system.

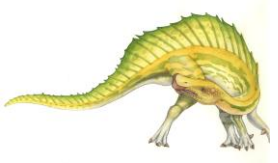
Problem:

- All the processes were in blocked state at the time of suspension.
- It is not good to bring a blocked process back into main memory
  - because it is still not ready for execution
- Recognize, .When an event occurs for blocked process, the process is not blocked
  - is potentially available for execution.

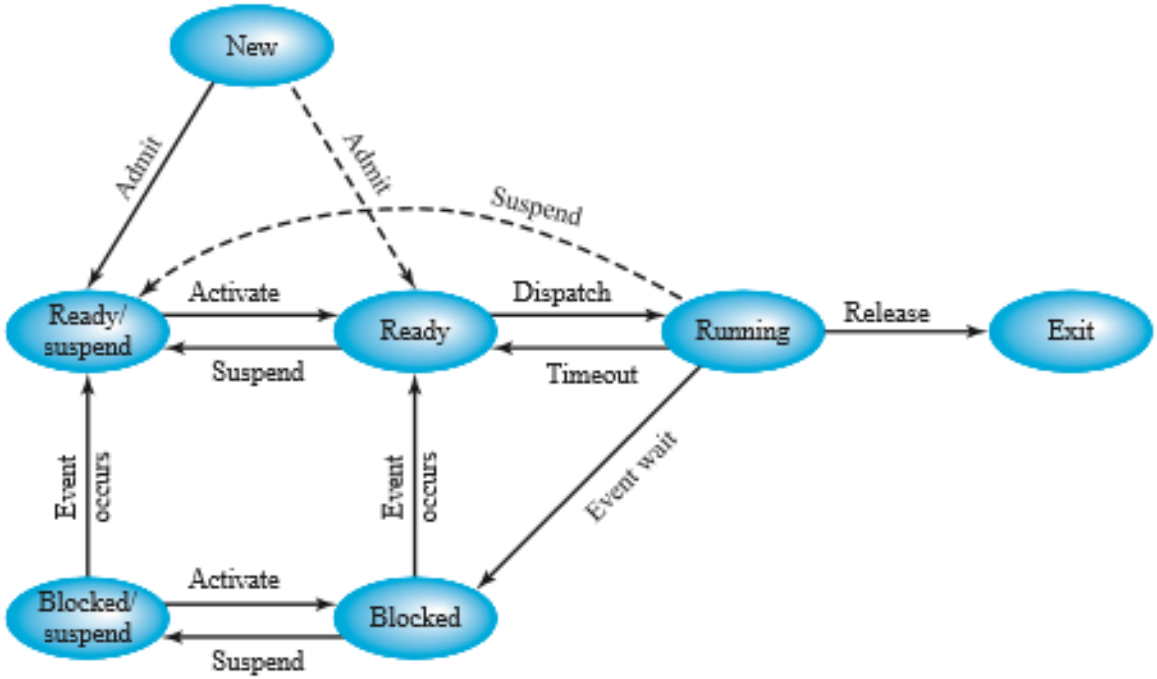


Suspended Processes

- Two independent concepts here:
- **Whether** a process is waiting on an event (blocked or not) and **whether** a process has been swapped out of main memory (suspended or not).
- To accommodate this 2 " 2 combination, we need four states:
  - **Ready:** The process is in main memory and available for execution.
  - **Blocked:** The process is in main memory and awaiting an event.
  - **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
  - **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.

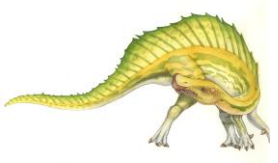


Process State Transition with Suspended States



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States



Process State Transition with Suspended States

- **Blocked → Blocked/Suspend:** If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked. This transition can be made even if there are ready processes available, if the OS determines that the currently running process or a ready process that it would like to dispatch requires more main memory to maintain adequate performance.
- **Blocked/Suspend → Ready/Suspend:** A process in the Blocked/Suspend state is moved to the Ready/Suspend state when the event for which it has been waiting occurs. Note that this requires that the state information concerning suspended processes must be accessible to the OS.
- **Ready/Suspend → Ready:** When there are no ready processes in main memory, the OS will need to bring one in to continue execution. In addition, it might be the case that a process in the Ready/Suspend state has higher priority than any of the processes in the Ready state. In that case, the OS designer may dictate that it is more important to get at the higher-priority process than to minimize swapping.

Process State Transition with Suspended States

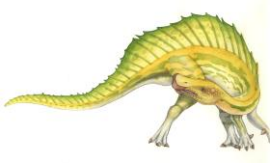
- **Ready → Ready/Suspend:** Normally, the OS would prefer to suspend a blocked process rather than a ready one, because the ready process can now be executed, whereas the blocked process is taking up main memory space and cannot be executed. However, it may be necessary to suspend a ready process if that is the only way to free up a sufficiently large block of main memory. Also, the OS may choose to suspend a lower–priority ready process rather than a higher–priority blocked process if it believes that the blocked process will be ready soon.

- **Blocked/Suspend → Blocked:** Inclusion of this transition may seem to be poor design. After all, if a process is not ready to execute and is not already in main memory, what is the point of bringing it in? But consider the following scenario: A process terminates, freeing up some main memory. There is a process in the (Blocked/Suspend) queue with a higher priority than any of the processes in the (Ready/Suspend) queue and the OS has reason to believe that the blocking event for that process will occur soon. Under these circumstances, it would seem reasonable to bring a blocked process into main memory in preference to a ready process.

Process State Transition with Suspended States

- **Running → Ready/Suspend:** Normally, a running process is moved to the Ready state when its time allocation expires. If, however, the OS is preempting the process because a higher-priority process on the Blocked/Suspend queue has just become unblocked, the OS could move the running process directly to the (Ready/Suspend) queue and free some main memory.

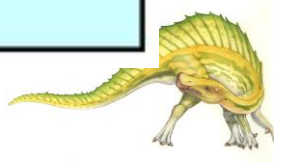
  Any State → Exit: Typically, a process terminates while it is running, either because it has completed or because of some fatal fault condition. However, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated. If this is allowed, then a process in any state can be moved to the Exit state.



Uses of Suspension

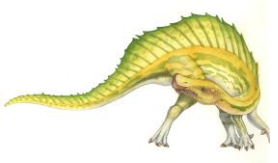
Table 3.3 Reasons for Process Suspension

Swapping	The operating system needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The operating system may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendents.

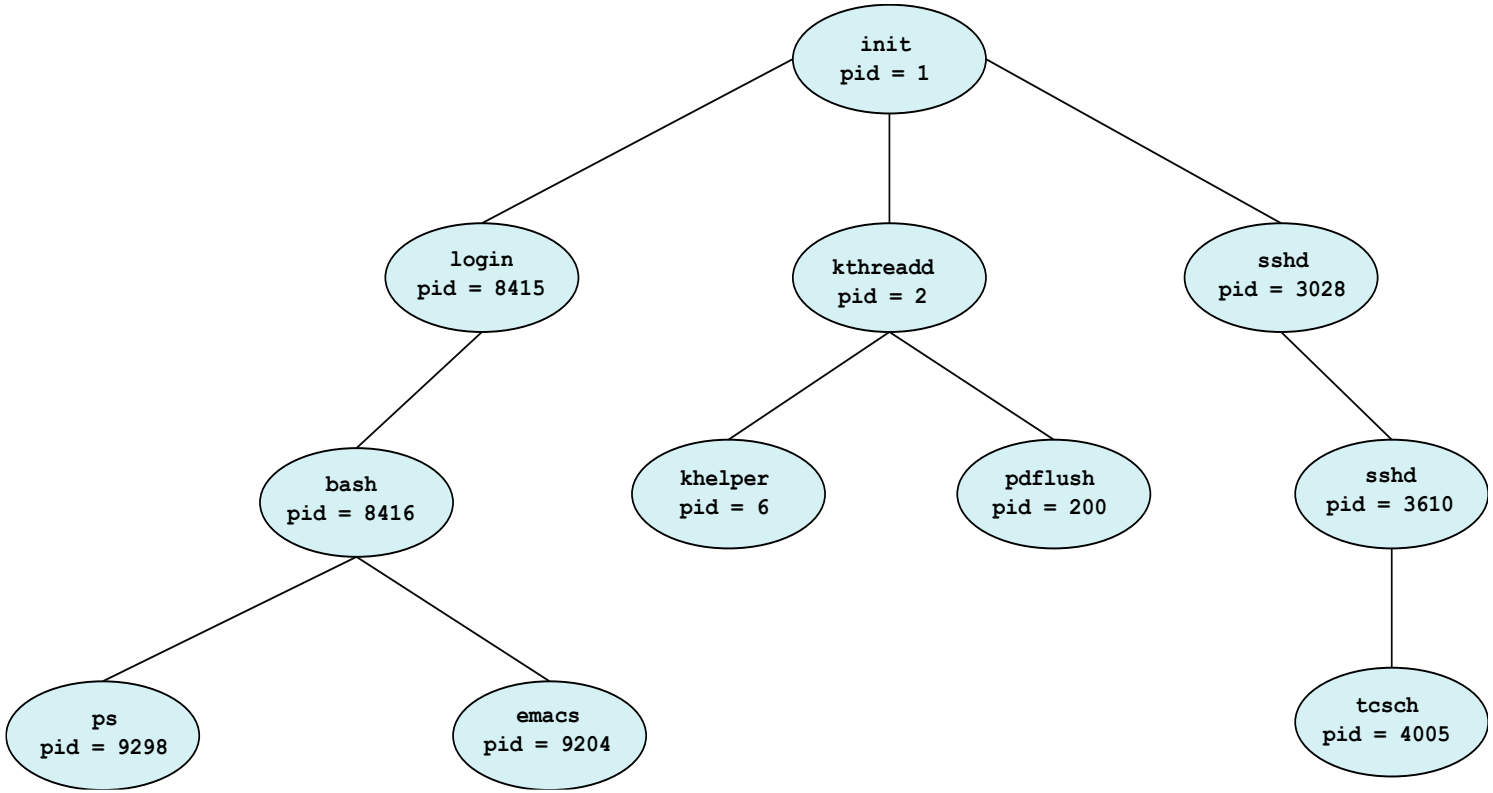


Process Creation

- **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

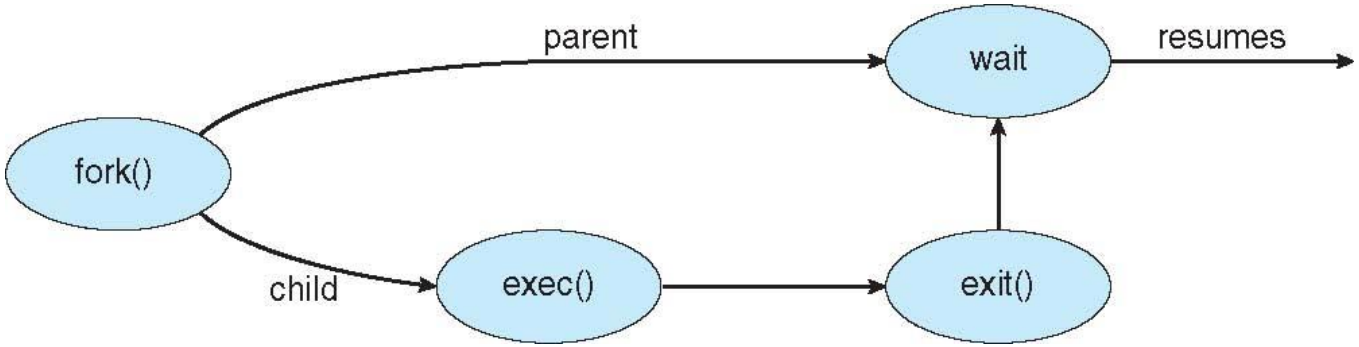


A Tree of Processes in Linux



Process Creation (Cont.)

- Address space
  - Child duplicate of parent(same program & data)
  - Child has a new 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



Process Creation (Cont.)

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's `pid` to the parent process

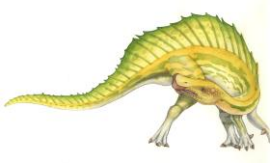
C Program Forking Separate Process

```
#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;
}
```

- Calls `wait(NULL)`, blocking until **any** child terminates.
- Once the child's `ls` finishes, `wait` returns and the parent prints

C Program Forking Separate Process(example 1)

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



Process Termination

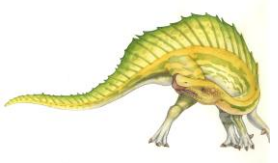
- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using 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 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

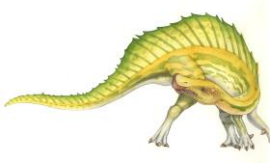
```
pid = wait(&status);
```

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



Process Termination

- **Zombie**
  - A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
  - The parent process has not yet read the child's exit status using the wait() system call or a similar mechanism.
  - This situation commonly occurs when a parent process fails to call wait() :
    - parent being busy
    - not programmed to handle such events
    - or having terminated abruptly.
  - A child process always first becomes a zombie before being removed from the process table.
  - The parent process reads the exit status of the child process which reaps off the child process entry from the process table.
  - Having too many zombie processes can potentially exhaust system resources.

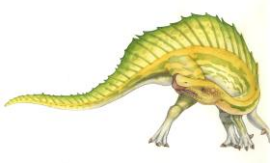


Process Termination

- **Orphan**
  - A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.
  - This can occur when a parent process unexpectedly terminates or exits before the child process has finished.
  - Having a large number of orphan processes left unchecked over time can potentially occupy process table entries, which may affect the system's ability to create new processes.

Process Termination

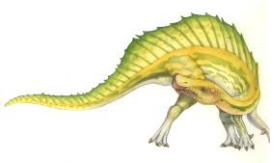
- **Key Differences:**
- **Cause of Existence:** Zombie processes exist due to the parent process not collecting the termination status of its child, whereas orphan processes occur when the parent process terminates before the child.
- **Resource Utilization:** Zombie processes consume minimal resources (mostly a process table entry) but don't execute any code. Orphan processes, when adopted by the init process, continue execution until completion.
- **Handling:** Zombie processes are typically handled by the parent process collecting the exit status, while orphan processes are adopted and managed by the init process to ensure proper termination.



Zombie process example

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   // Fork returns process id
   // in parent process
   pid_t child_pid = fork();

   // Parent process
   if (child_pid > 0)
      sleep(50);

   // Child process
   else
      exit(0);

   return 0;
}
```

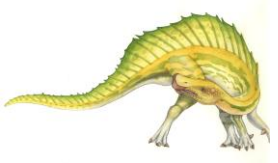
The child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn't call wait() and the child process's entry still exists in the process table.



Orphan process example

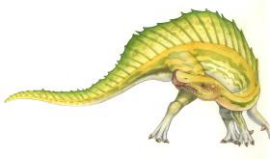
```
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    // Create a child process
    int pid = fork();

    if (pid > 0)
        printf("in parent process");

    // Note that pid is 0 in child process
    // and negative if fork() fails
    else if (pid == 0)
    {
        sleep(30);
        printf("in child process");
    }

    return 0;
}
```

Parent finishes execution and exits while the child process is still executing and is called an orphan process now.
However, the orphan process is soon adopted by init process, once its parent process dies.



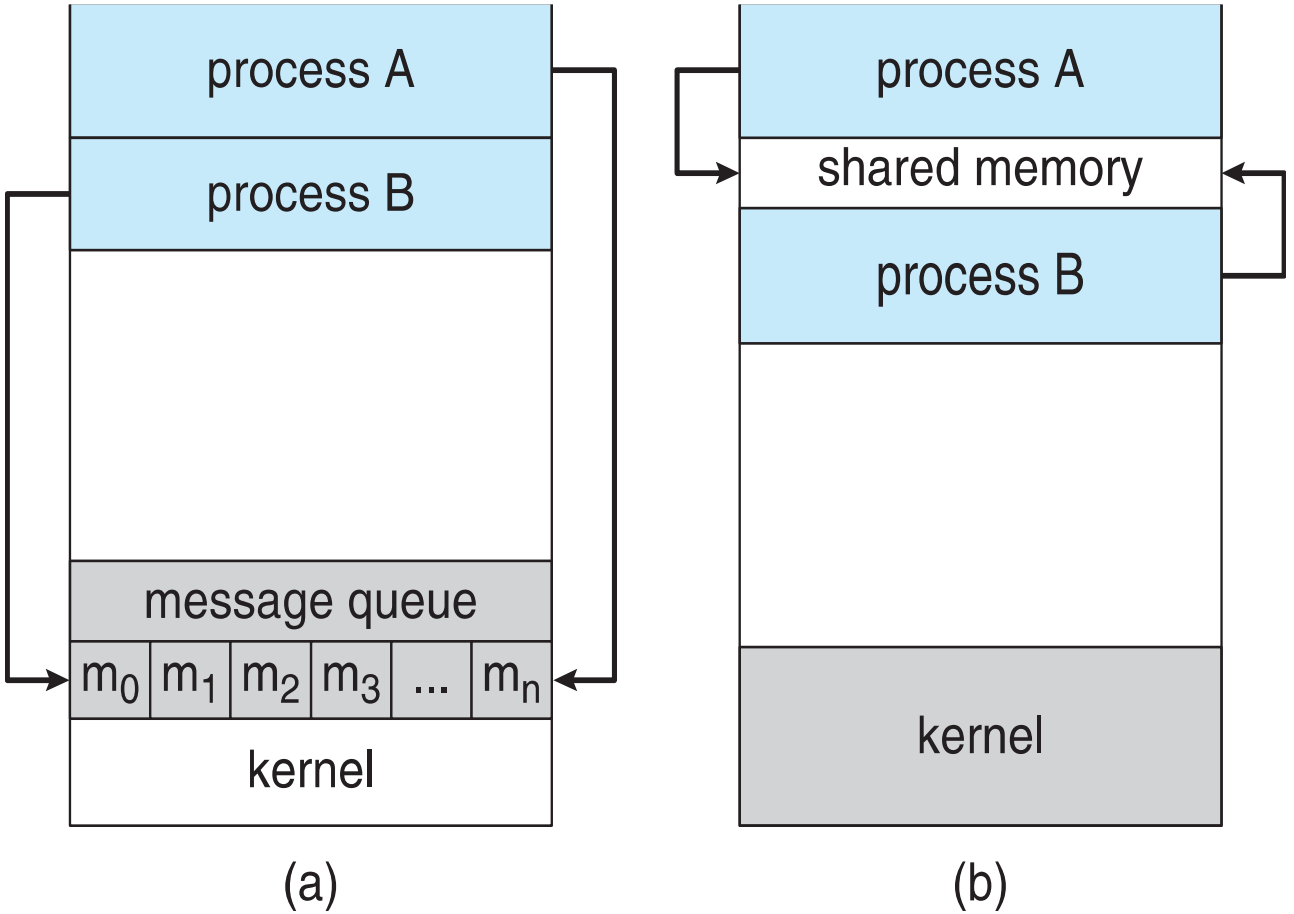
Interprocess Communication

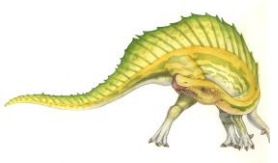
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes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



Communications Models

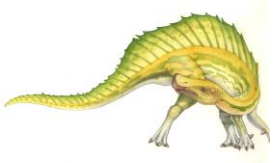
(a) Message passing. (b) shared memory.





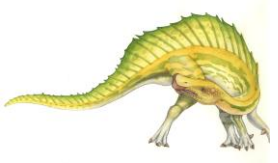
Producer-Consumer Problem

- 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



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

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

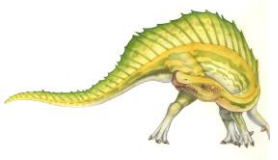
- Messages are directed and received through a mailbox, message queue, or some form of communication buffer. (also referred to as ports).
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes 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
  - 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

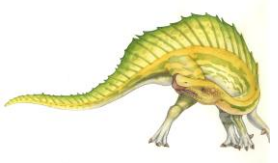


Indirect Communication

- 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



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
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits