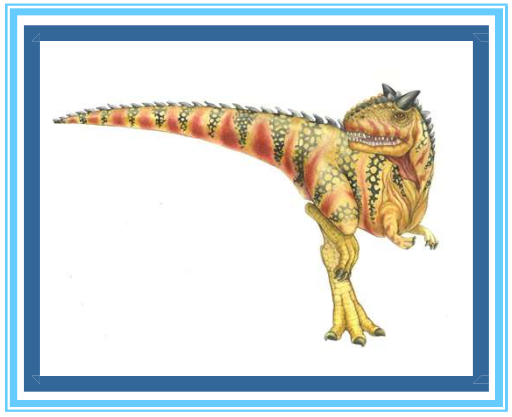
Chapter 3: Processes

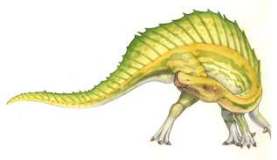


Outline

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

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.



Process

- Fundamental to the structure of operating systems

A *process* can be defined as:

A program in execution

An instance of a running program

The entity that can be assigned to, and executed on, a processor

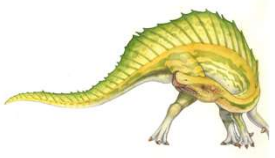
A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

Terminology

- Application = service = program
- Script
- Process
- Daemon
- Threads
- Job

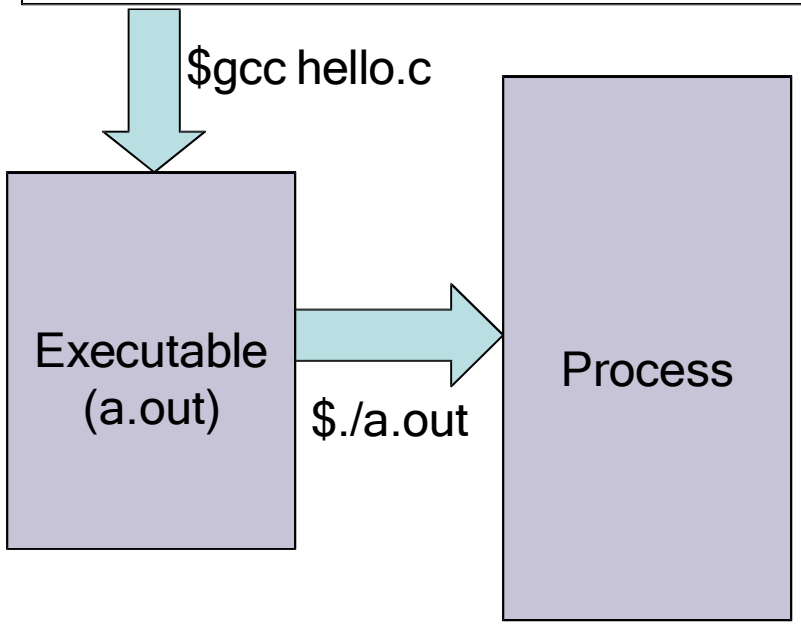
Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time



Executing Program (Process)

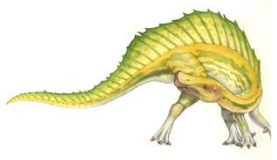
```c
#include <stdio.h>

int main(){
   char str[] = "Hello World\n";
   printf("%s", str);
}
```



- Process
  - A program in execution
  - Most important abstraction in an OS
  - Comprises of
    - Code
    - Data
    - Stack
    - Heap
    - State in the OS
    - Kernel stack

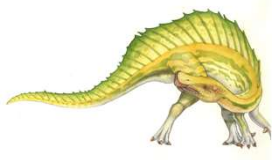
    In the user space of process

    In the kernel space
  - State contains: registers, list of open files, related processes, etc.



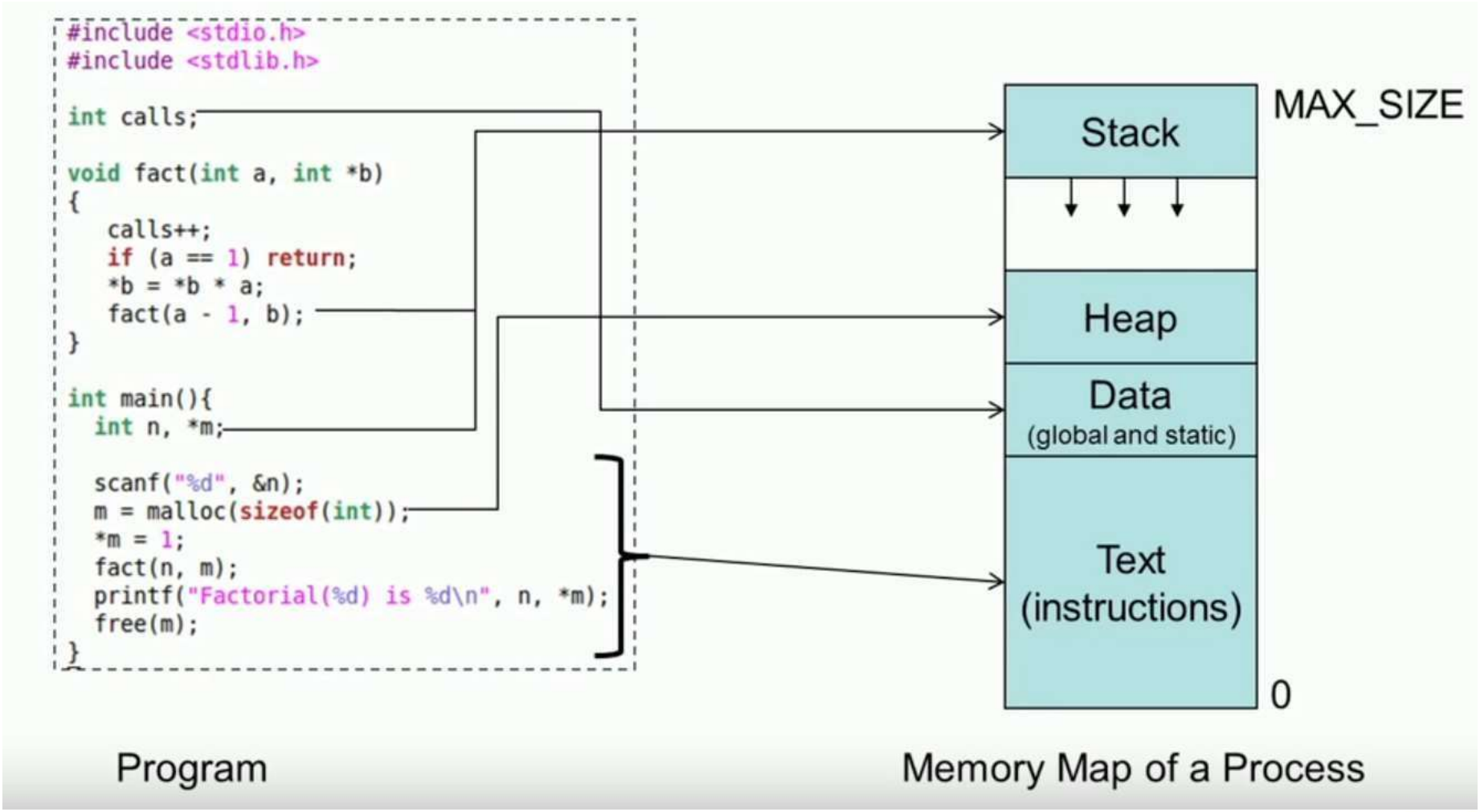
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program



Process Memory Map

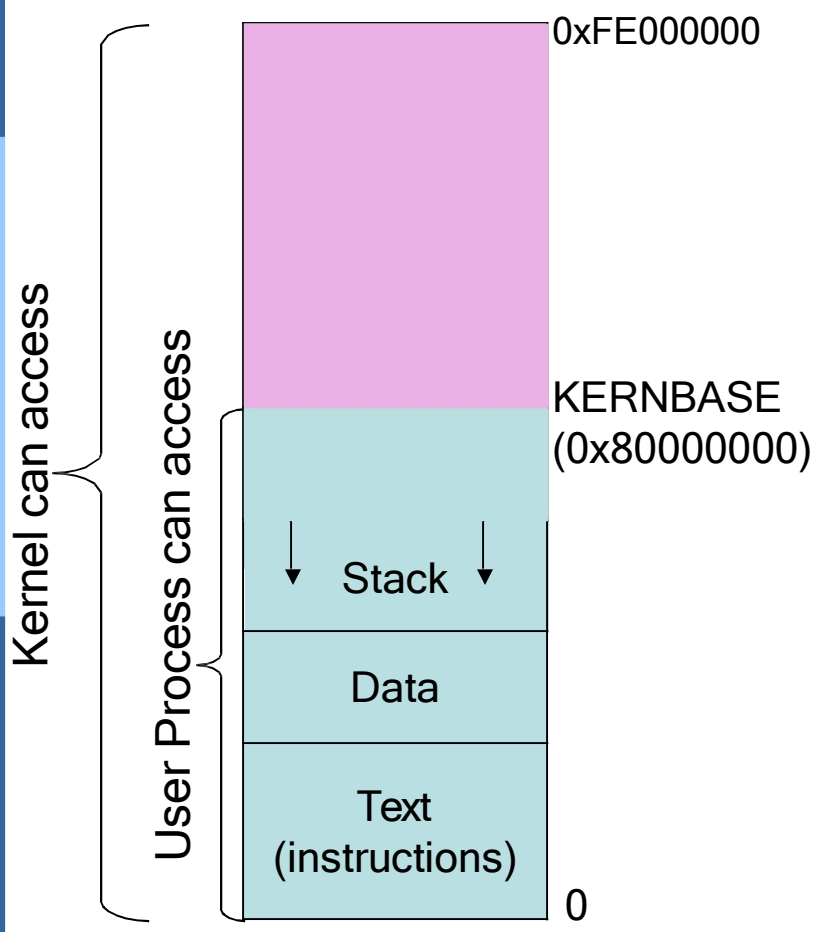
```c
#include <stdio.h>
#include <stdlib.h>

int calls;

void fact(int a, int *b)
{
   calls++;
   if (a == 1) return;
   *b = *b * a;
   fact(a - 1, b);
}

int main(){
  int n, *m;

  scanf("%d", &n);
  m = malloc(sizeof(int));
  *m = 1;
  fact(n, m);
  printf("Factorial(%d) is %d\n", n, *m);
  free(m);
}
```



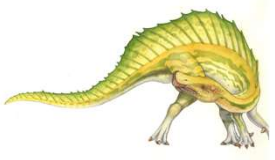
Program ≠ Process

Program	Process
code + static and global data	Dynamic instantiation of code + data + heap + stack + process state
One program can create several processes	A process is unique isolated entity

Process Address Map in xv6



- Entire kernel mapped into every process address space
  - This allows easy switching from user code to kernel code (ie. during system calls)
    - No change of page tables needed
  - Easy access of user data from kernel space



Process Management

- *Fundamental Task: Process Management*
- The Operating System must
  - Interleave the execution of multiple processes
  - Allocate resources to processes, and protect the resources of each process from other processes,
  - Enable processes to share and exchange information,
  - Enable synchronization among processes.

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

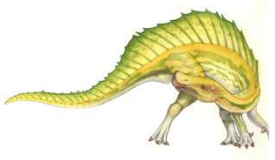
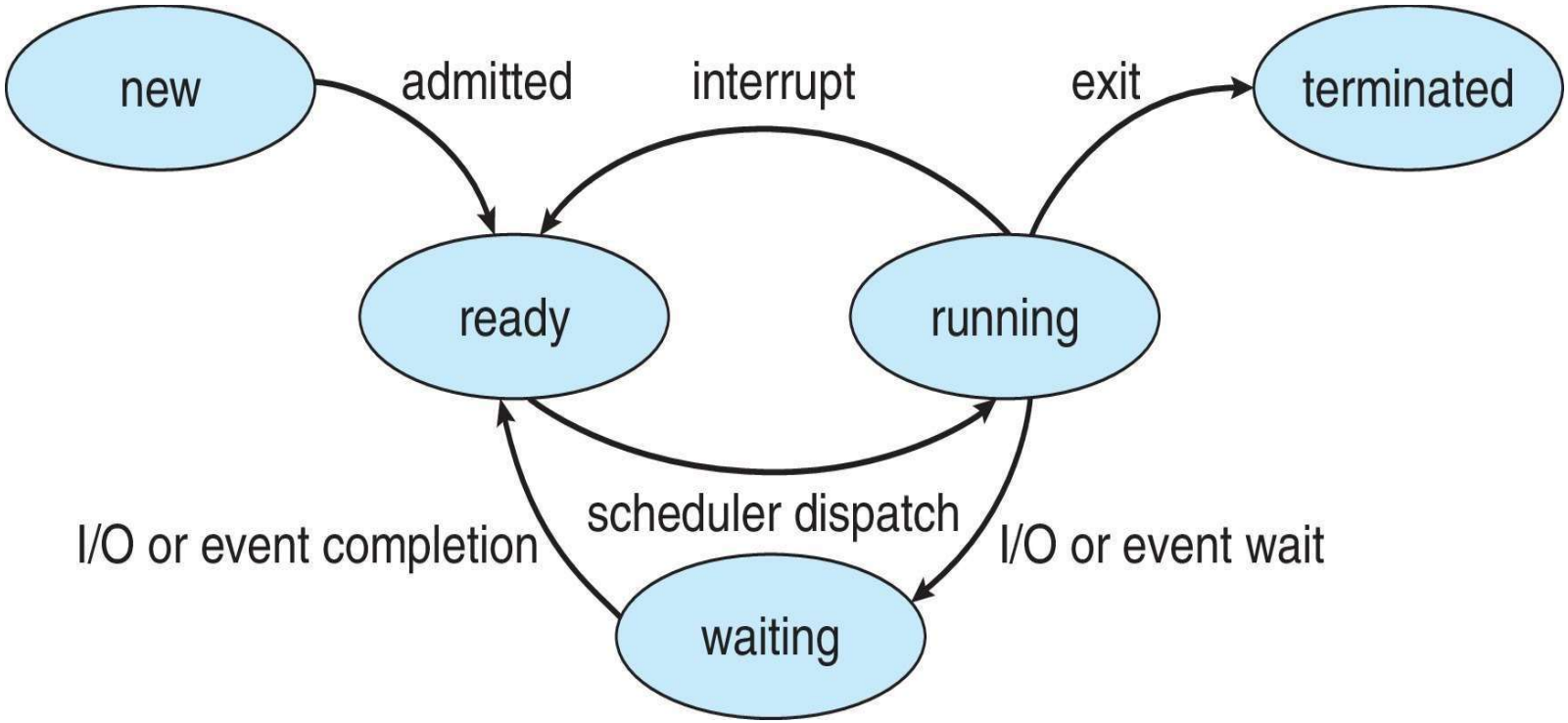
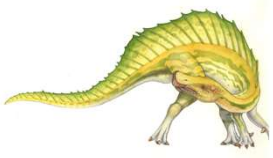
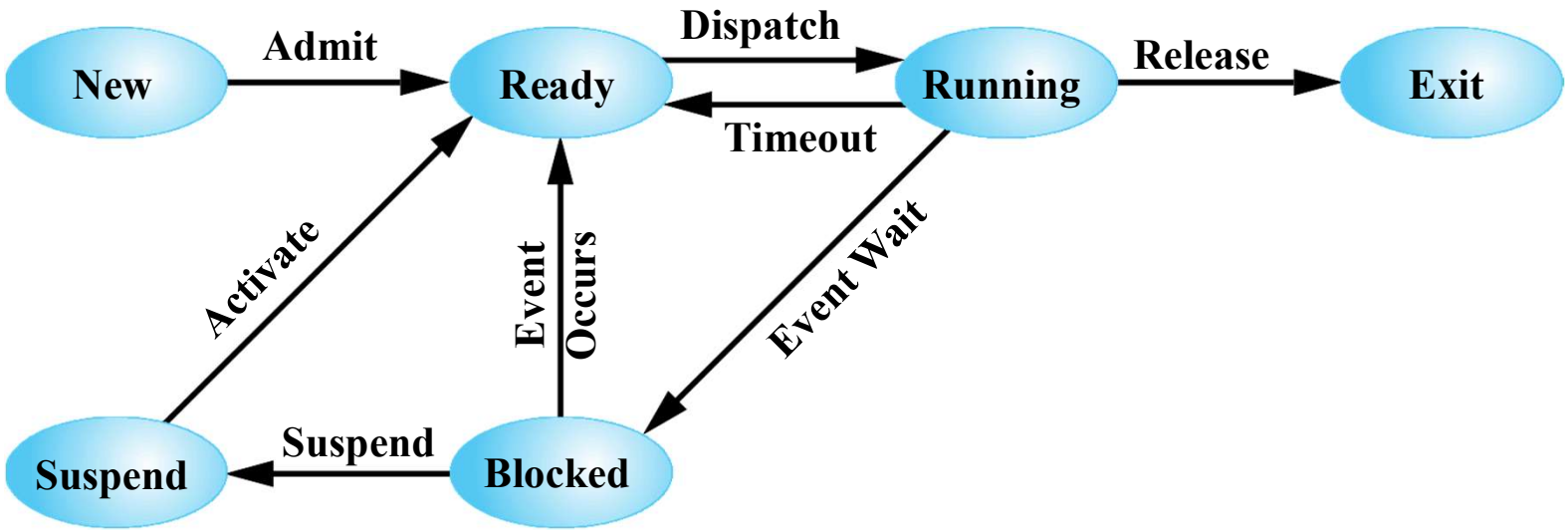


Diagram of Process State (five states)



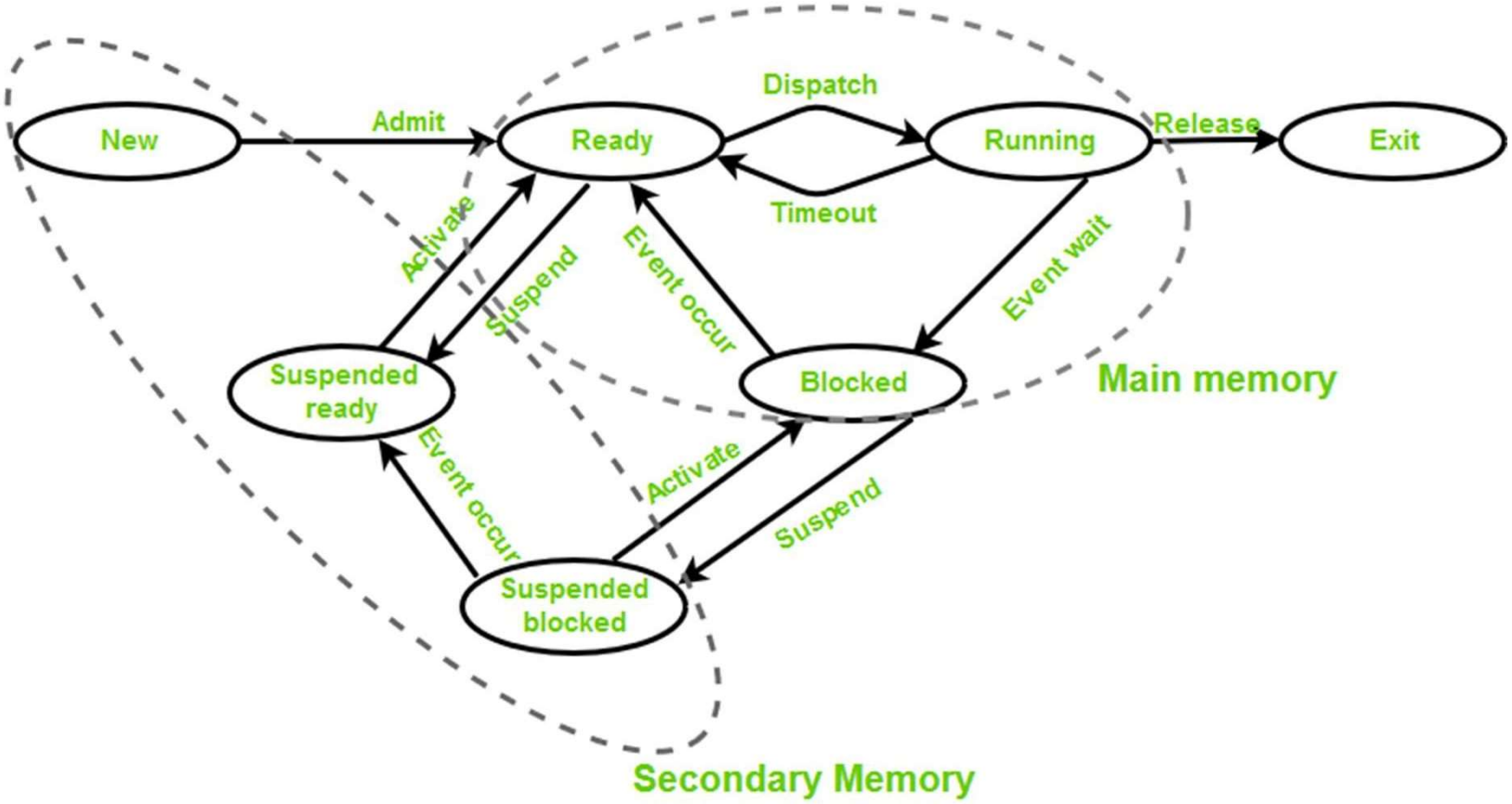




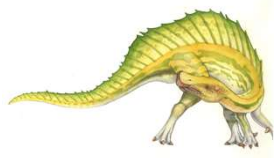
(a) With One Suspend State

Process State Transition Diagram with Suspend States

States of a Process in Operating Systems



https://www.geeksforgeeks.org/states-of-a-process-in-operating-systems/



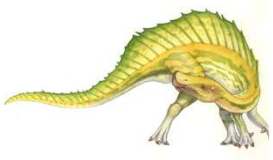
Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

Process Control Block (PCB)

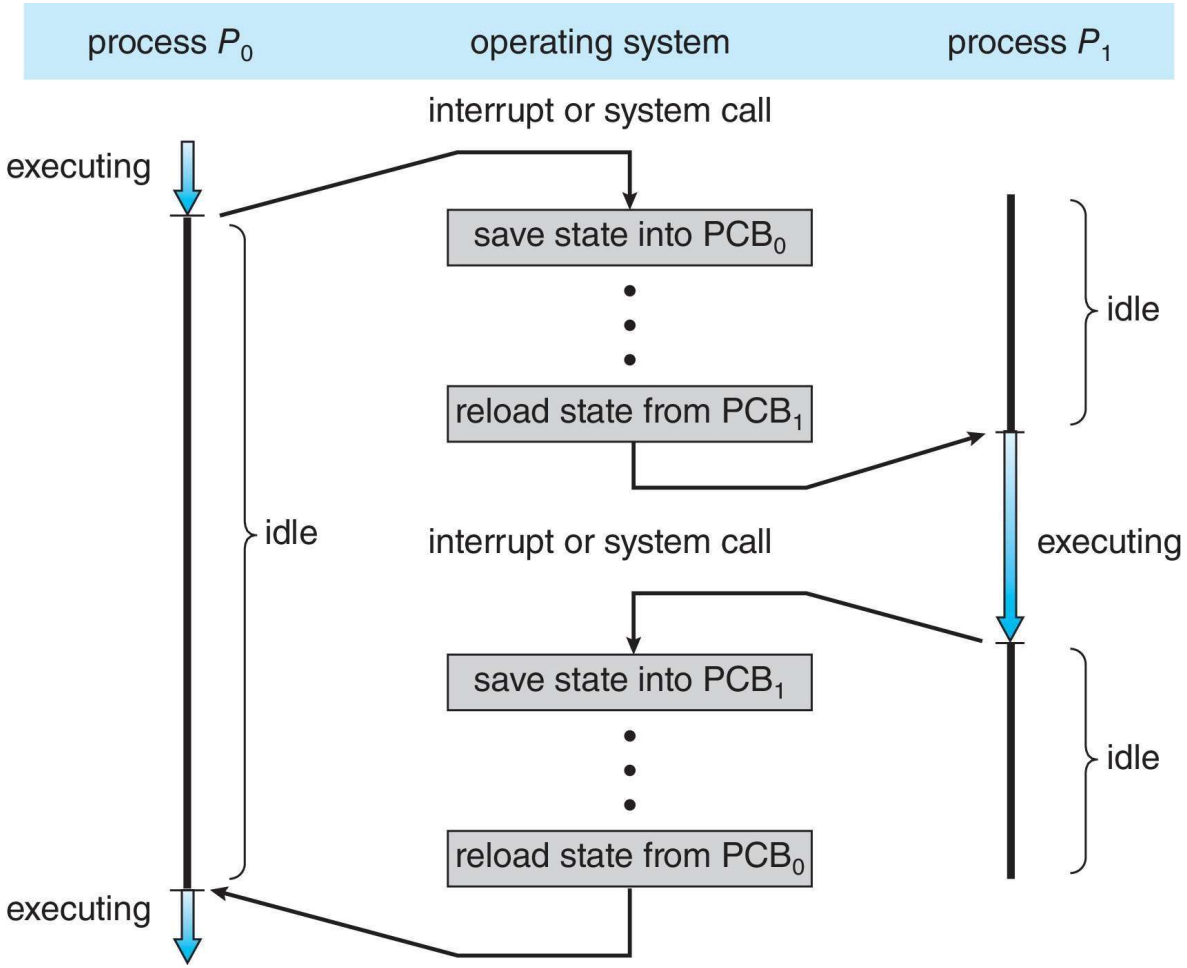
Information associated with each process(also called **task control block**)

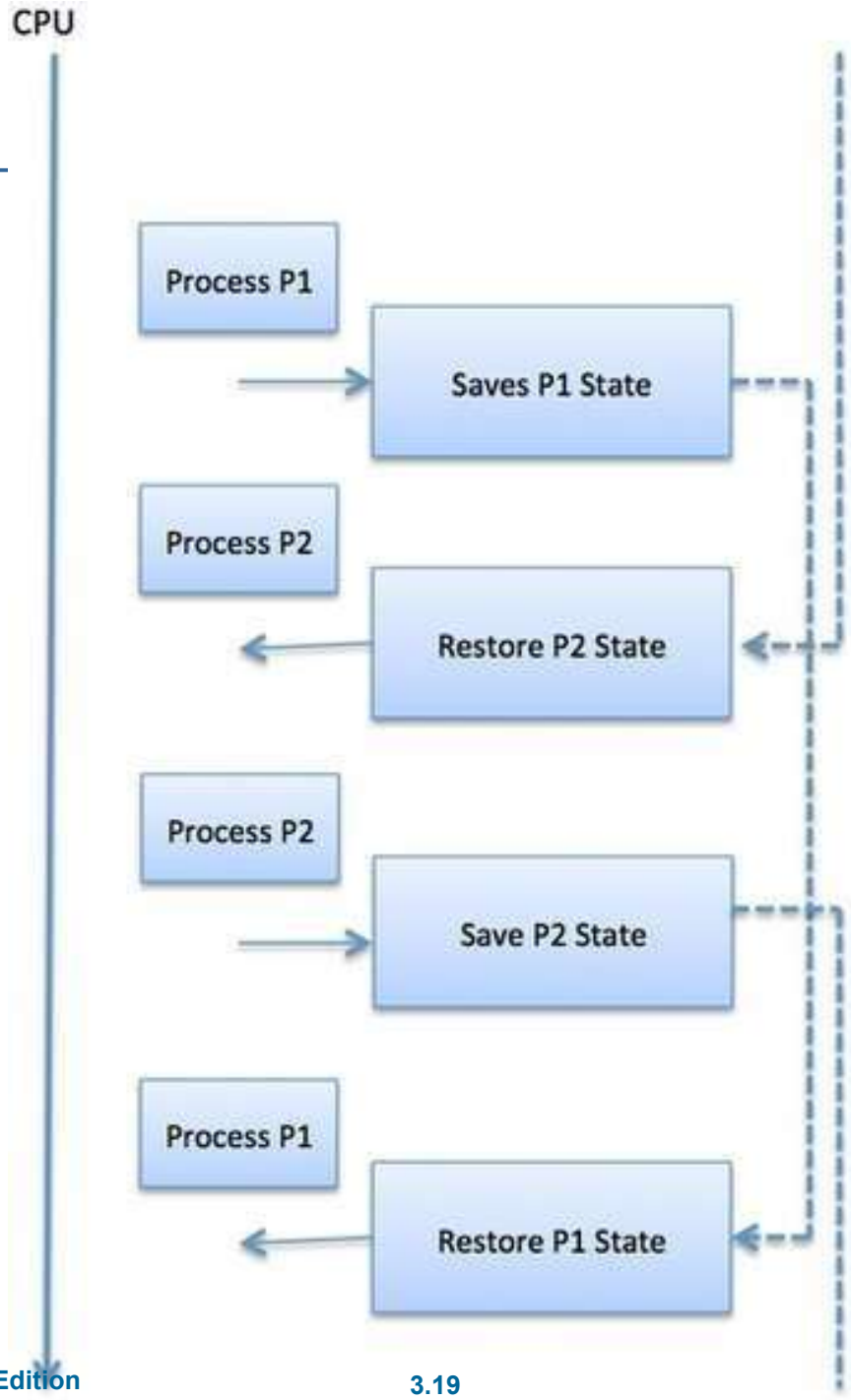
- Process identifier
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU **registers** – contents of all process-centric registers – no need for the variables as they are still in main memory
- CPU scheduling information- priorities, scheduling queue 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

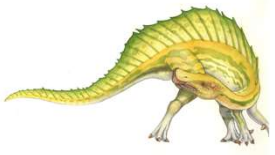


CPU Switch From Process to Process

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

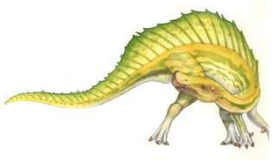



CPU
Process P1
Saves P1 State
Process P2
Restore P2 State
Process P2
Save P2 State
Process P1
Restore P1 State



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
  - We do not need to save the registers of a process once switching as there are for example two sets of registers.



Process Execution

Consider three processes being executed

- All three processes are in main memory (plus dispatcher)
- **dispatcher** is a small program that switches the processor from one process to another

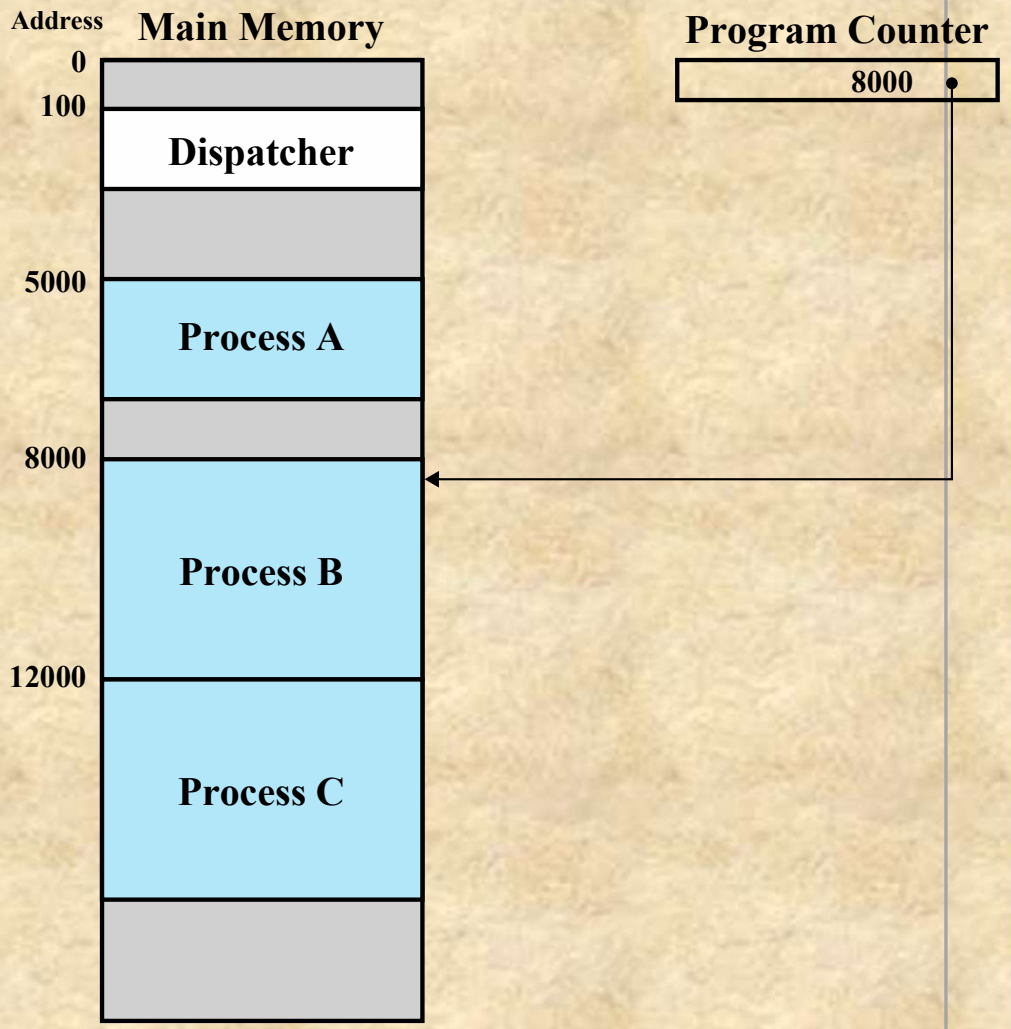


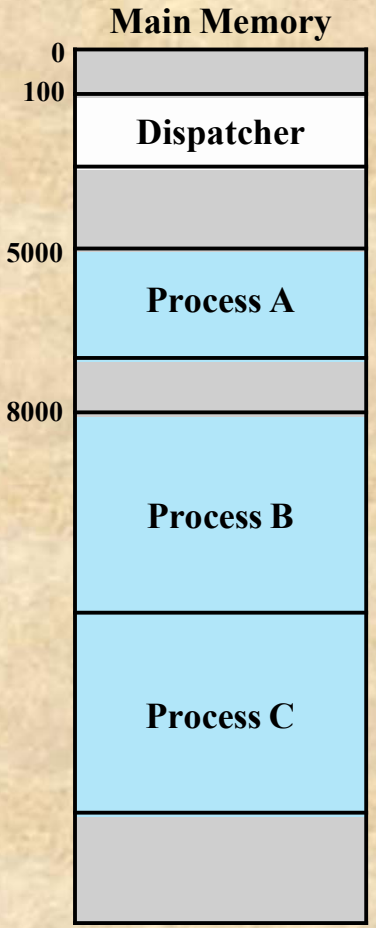
Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

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

(a) Trace of Process A **(b) Trace of Process B** **(c) Trace of Process C**

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



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

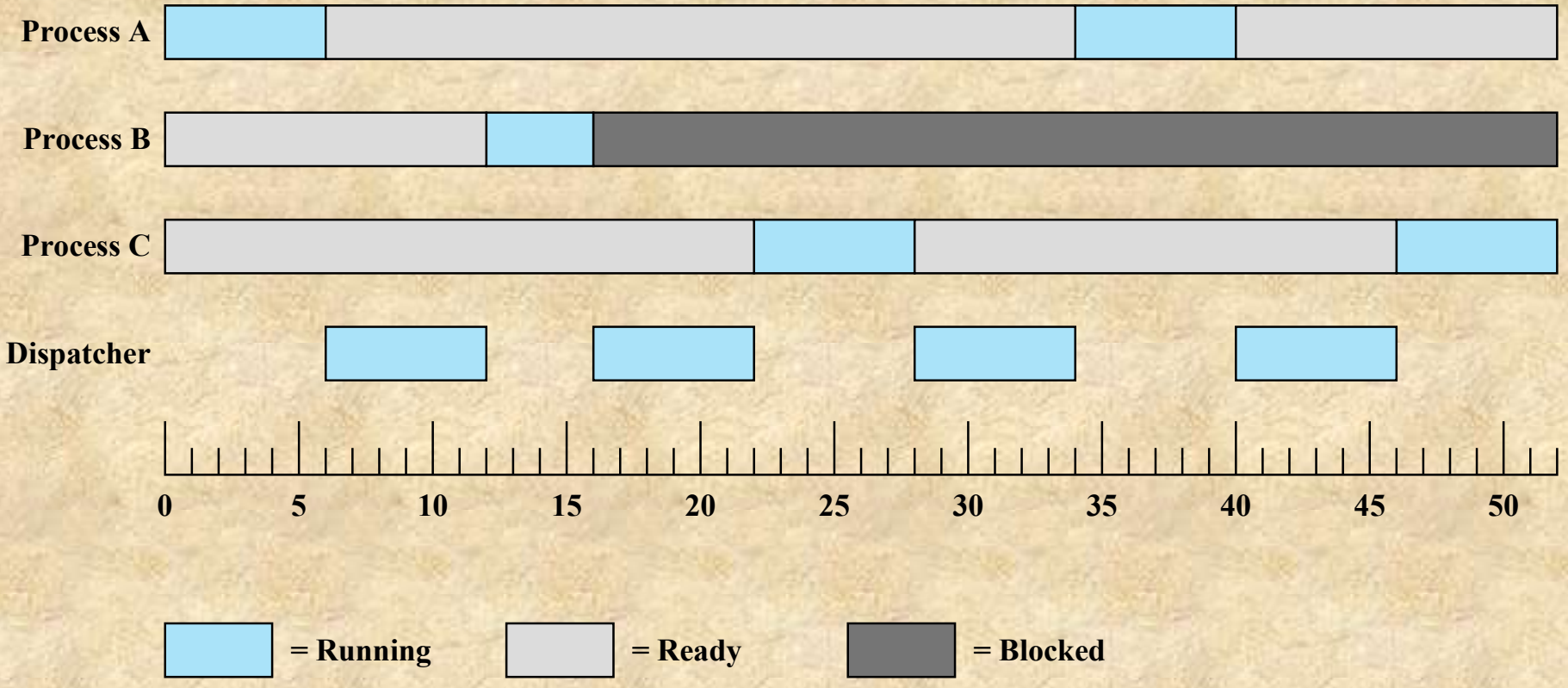
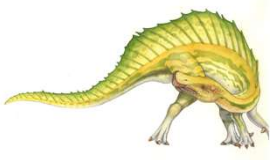


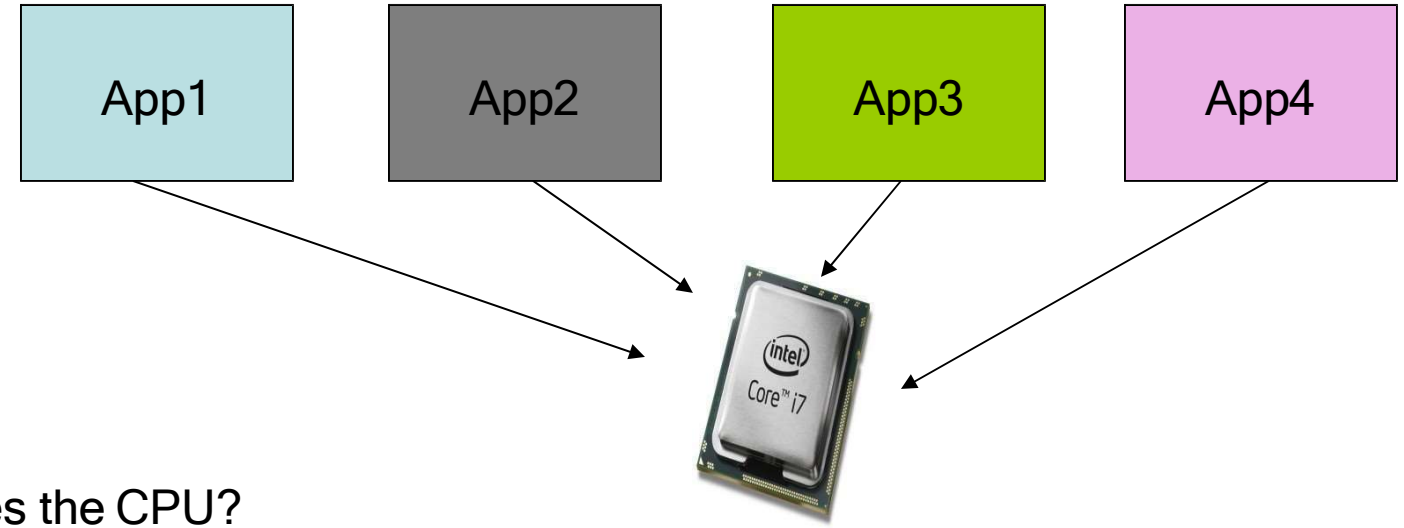
Figure 3.7 Process States for Trace of Figure 3.4

Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Main Goals:
  - Maximize CPU use (keep the CPU busy at all time)
  - To deliver “acceptable” response times for **all** programs
  - There is a **tradeoff** between these two goals

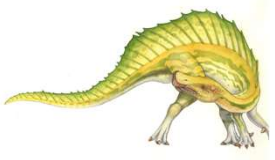


Sharing the CPU

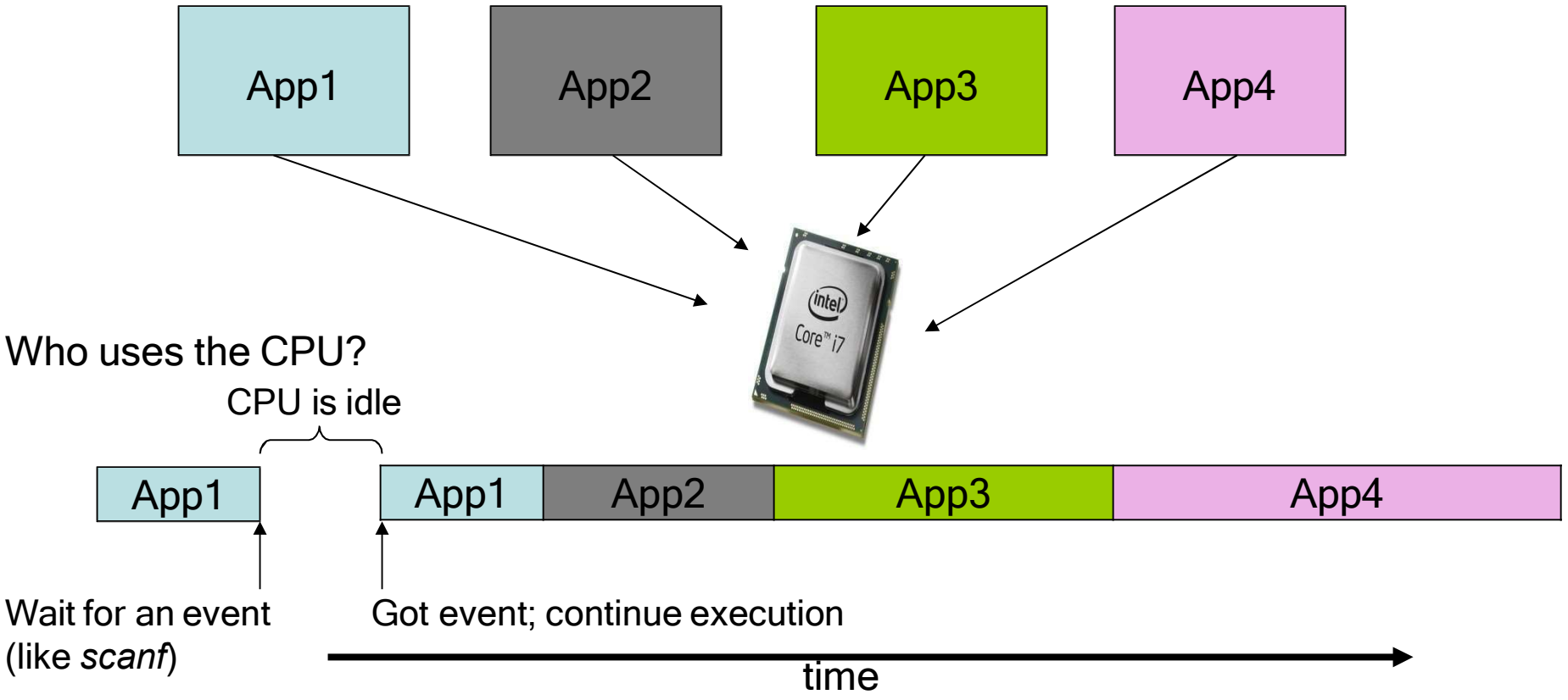


Who uses the CPU?

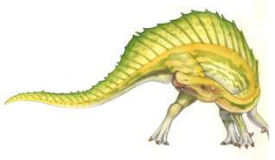
When one app completes the next starts



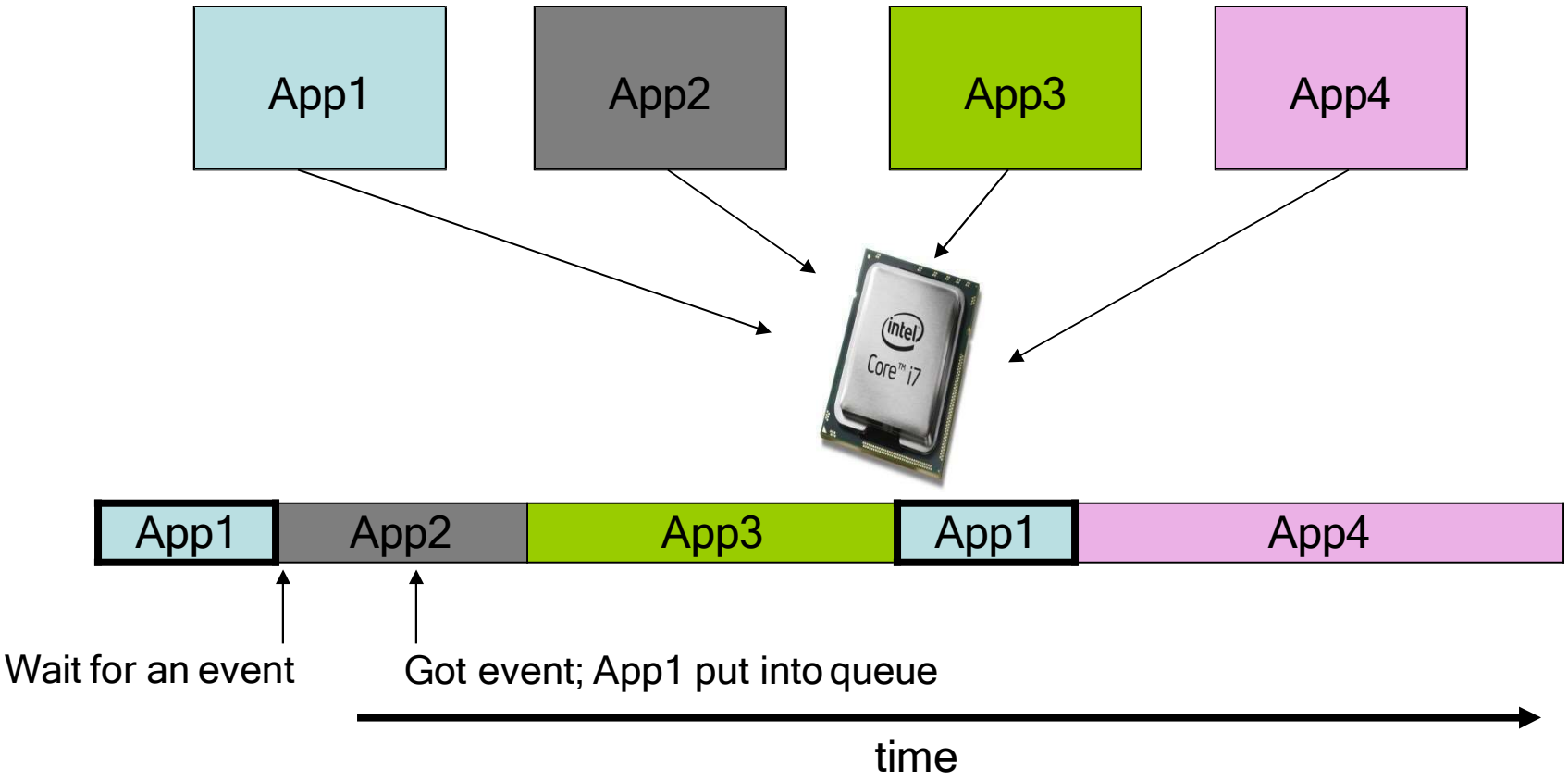
Idle CPU Cycles



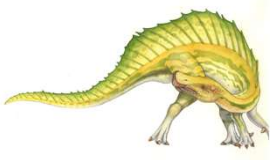
CPU is idle when executing app waits for an event. Reduced performance.



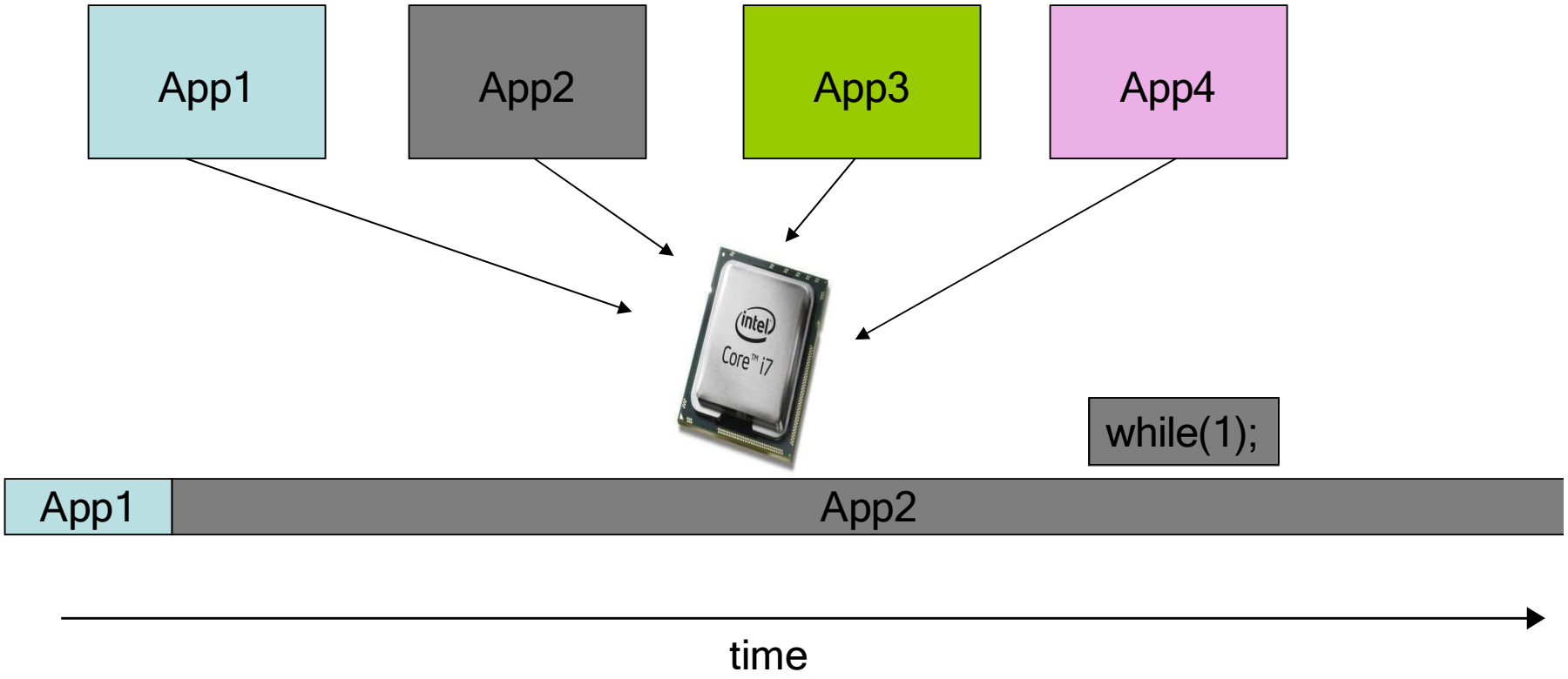
When OS supports Multiprogramming



When CPU idle, switch to another app

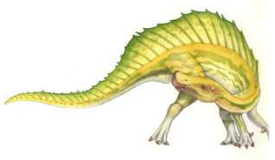


Multiprogramming could cause starvation



One app can hang the entire system

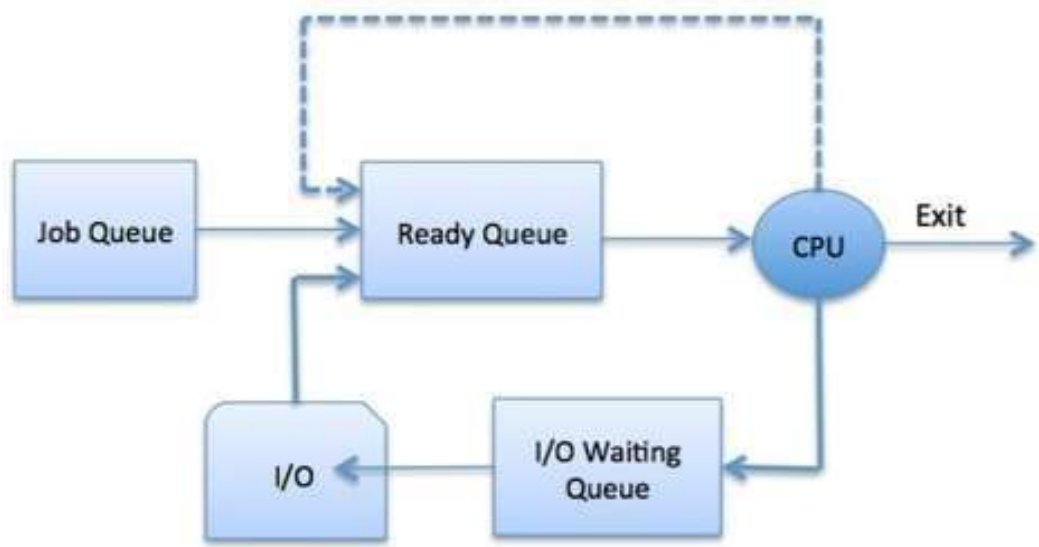
To deliver "acceptable" response times for **all** programs!!!

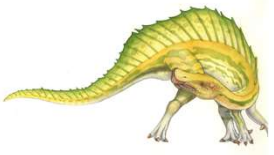


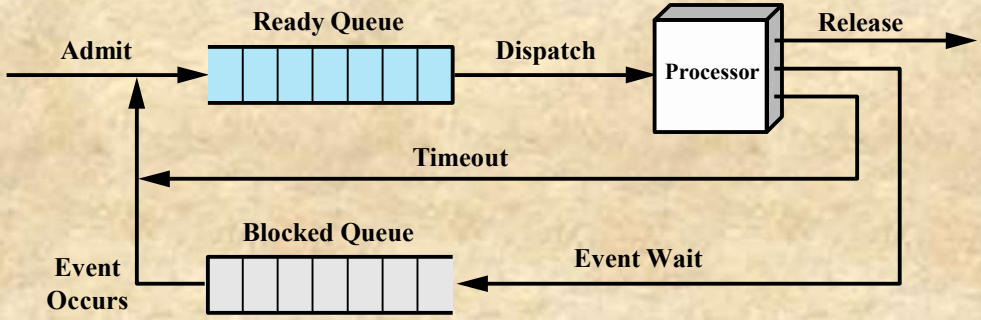
Process Scheduling Queues

The Operating System maintains the following important process scheduling queues:

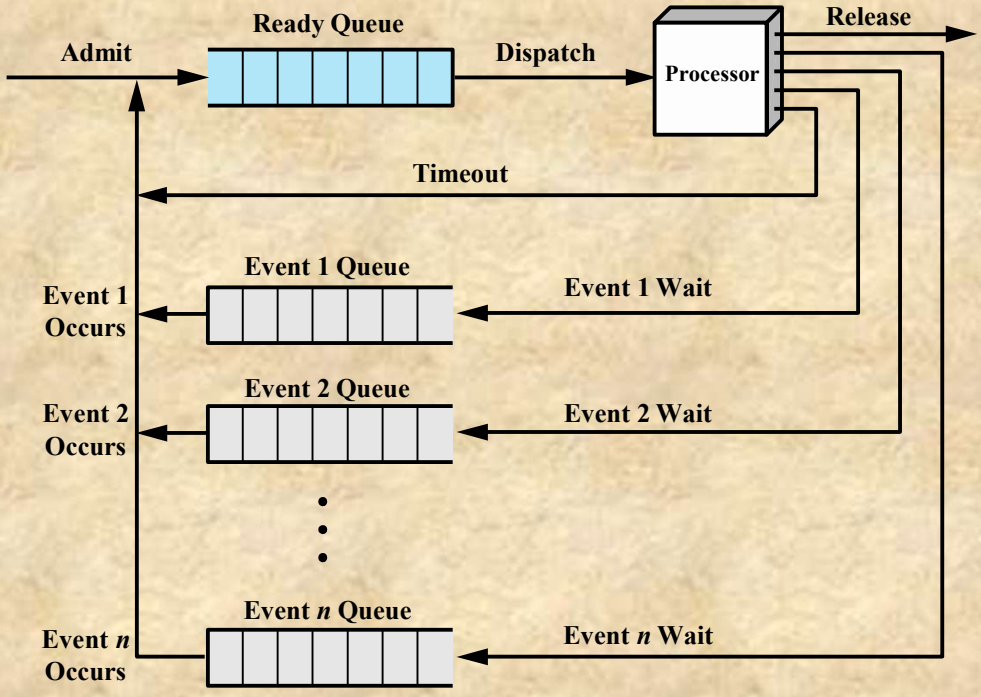
- **Job queue** − This queue keeps all the processes in the system (mostly in mainframe).
  - PCs usually do not have this queue.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute by CPU. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.
  - There is generally a separate device queue for each device
- **Processes** migrate among the various queues





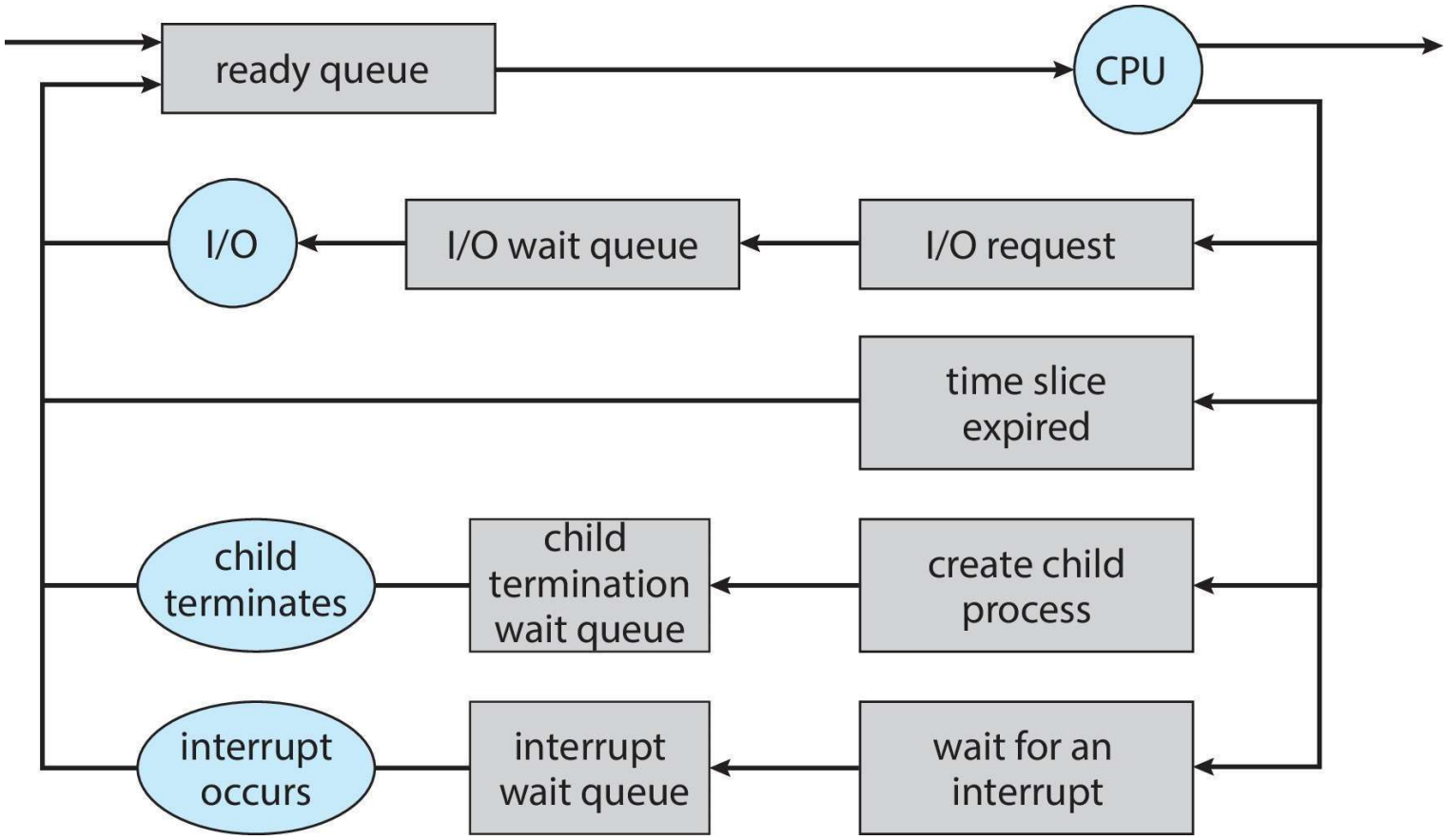


(a) Single blocked queue



(b) Multiple blocked queues

Representation of Process Scheduling



Types of Schedulers

1. Long term – performance – Makes a decision about how many processes should be made to stay in the ready state, this decides the degree of multiprogramming. Once a decision is taken it lasts for a long time hence called long term scheduler. It is called job scheduler as well.

2. Short term – Context switching time – Short term scheduler will decide which process to be executed next and then it will call dispatcher. A dispatcher is a software that moves process from ready to run and vice versa. In other words, it is context switching. It is called CPU scheduler as well.

3. Medium term – Swapping time – Suspension decision is taken by medium term scheduler. Medium term scheduler is used for swapping that is moving the process from main memory to secondary and vice versa. This process is called swapping, and the process is said to be swapped out or rolled out.

Multiprogramming

CPU and IO Bound Processes:

If the process is intensive in terms of CPU operations then it is called CPU bound process. Similarly, If the process is intensive in terms of I/O operations then it is called IO bound process.

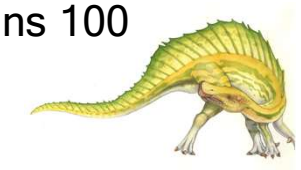
Multiprogramming – We have many processes ready to run. There are two types of multiprogramming:

1. Pre-emption – Process is forcefully removed from CPU. Pre-emption is also called as time sharing or multitasking.

2. Non pre-emption – Processes are not removed until they complete the execution.

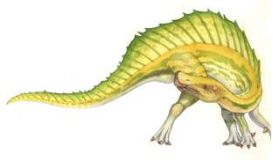
Degree of multiprogramming:

The number of processes that can reside in the ready state at maximum decides the degree of multiprogramming, e.g., if the degree of programming = 100, this means 100 processes can reside in the ready state at maximum.



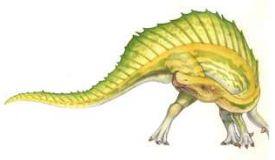
Operations on Processes

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



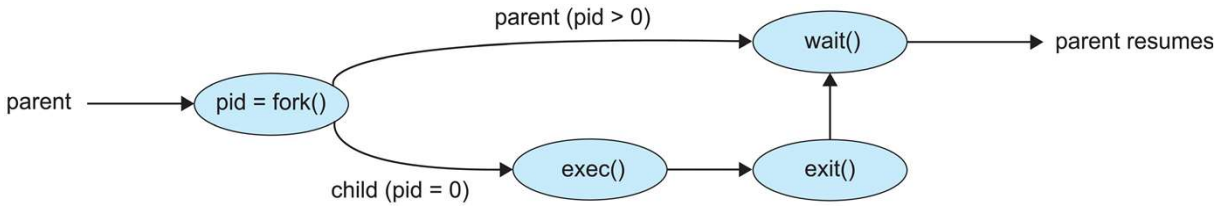
Process Creation

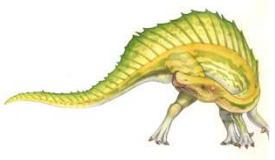
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset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



Process Creation (Cont.)

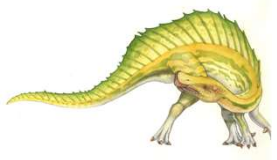
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate





Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting, and the operating systems does not allow a child to continue if its parent terminates

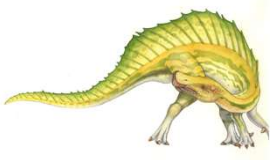


Process Termination

- Some operating systems do not allow child to 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

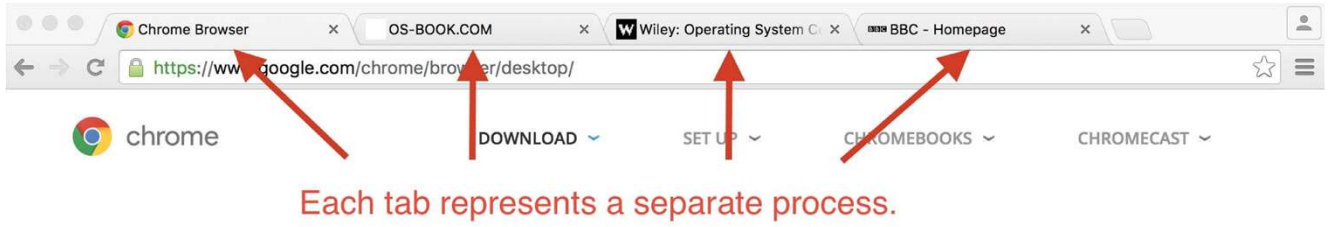
```
pid = wait(&status);
```

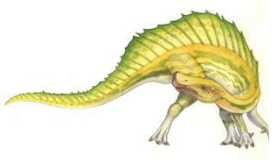
- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait()`**, process is an **orphan**



Multiprocess Architecture – Chrome Browser

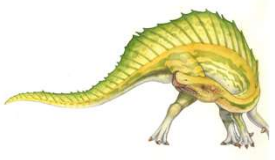
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders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





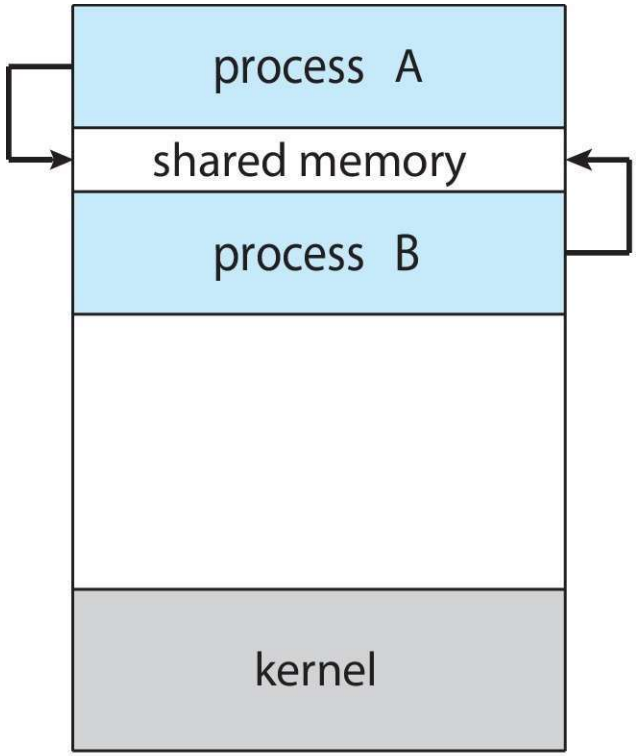
Interprocess Communication

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



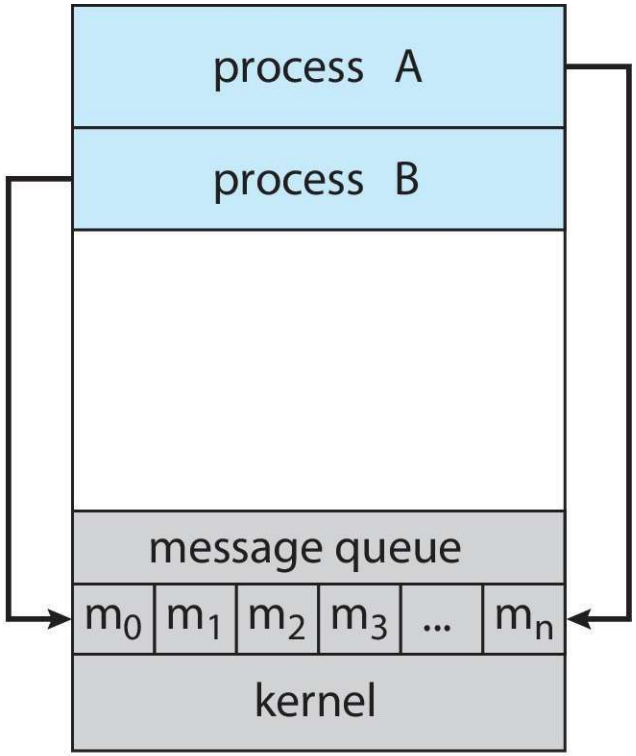
Communications Models

(a) Shared memory.



(a)

(b) Message passing.



(b)

Communications Models

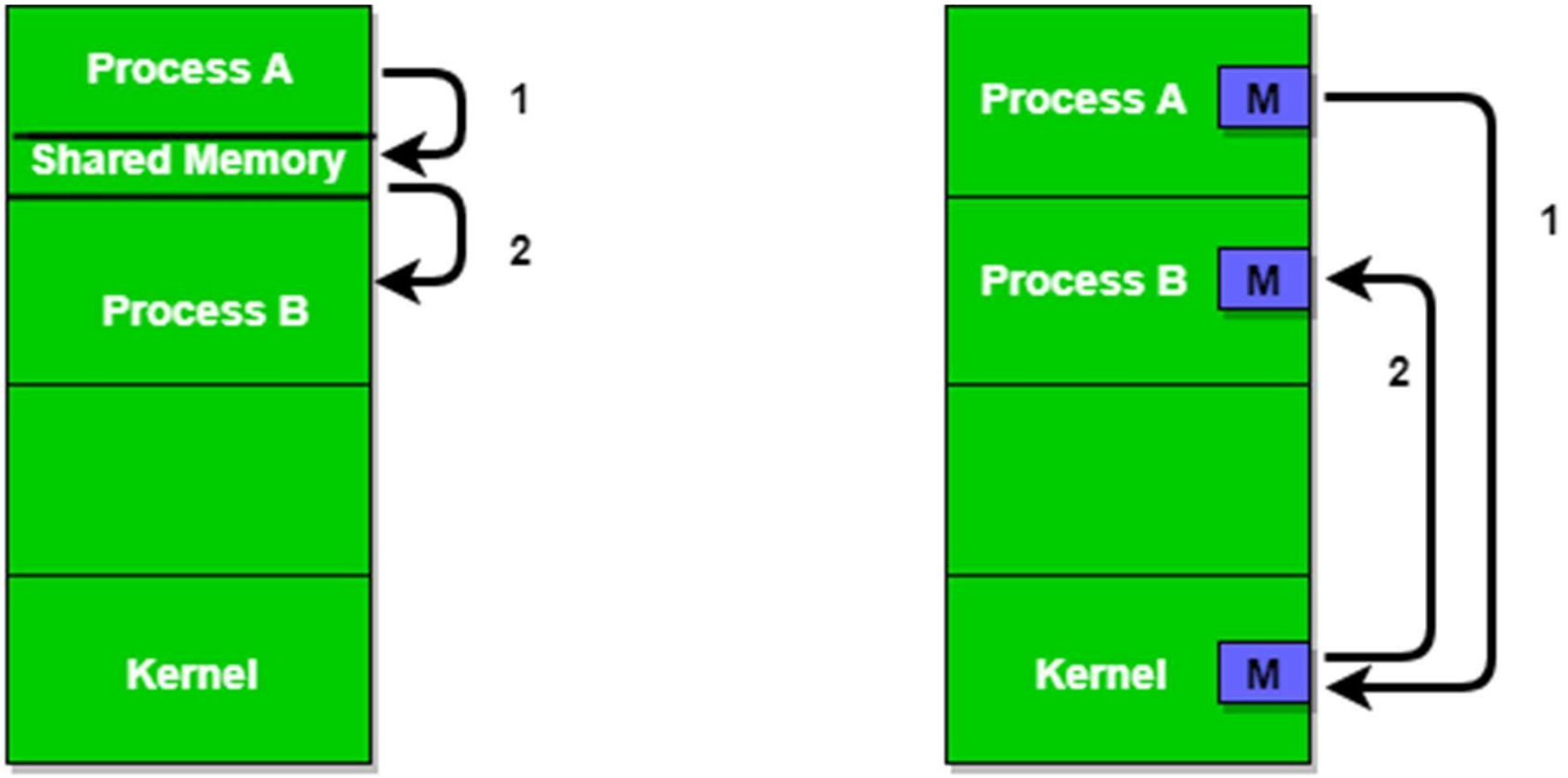
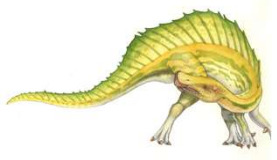


Figure 1 - Shared Memory and Message Passing

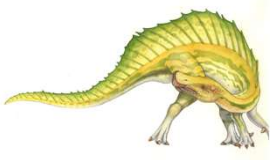
Producer-Consumer Problem

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



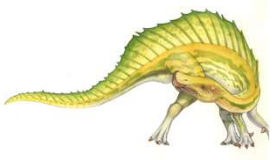
IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Typically, a shared memory region resides in the address space of the process creating the shared memory segment.
- Other processes that wish to communicate using this shared memory segment must attach it to their address space.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Example: Producer-Consumer
- Synchronization is discussed in great details in Chapters 6 & 7.



IPC – Message Passing

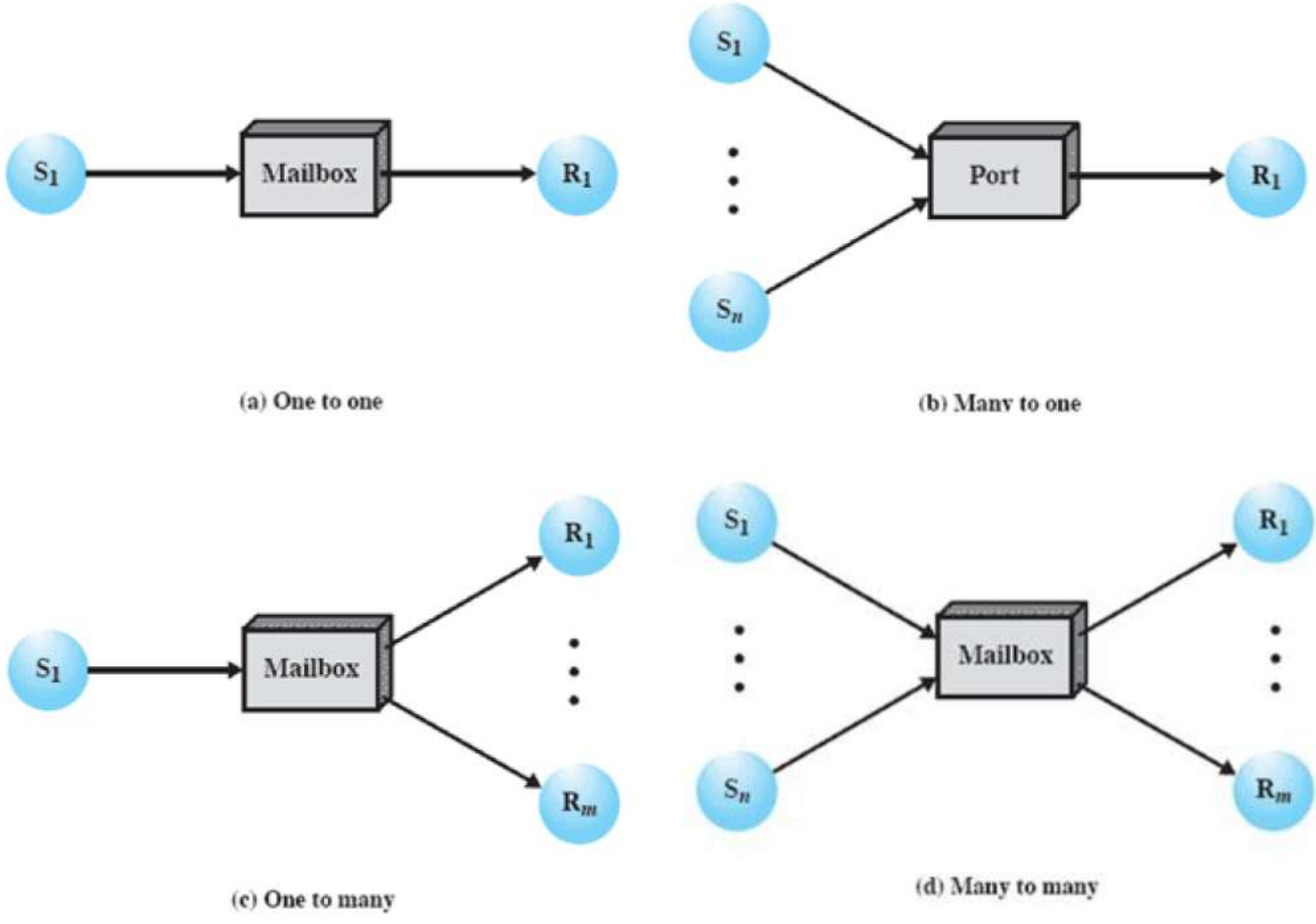
- Mechanism for processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a **distributed environment**.
- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable



Types of Communication

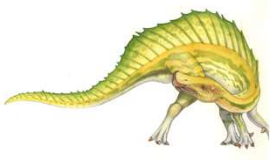
- Direct Communication:
  - Processes must name each other explicitly:
    - **`send`** (*P, message*) – send a message to process P
    - **`receive`**(*Q, message*) – receive a message from process Q
- Indirect Communication:
  - Messages are directed and received from mailboxes (also referred to as ports (in Unix, Linux))
    - Each mailbox has a unique id
    - Processes can communicate only if they share a mailbox
  - 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



Indirect Communication (Cont.)

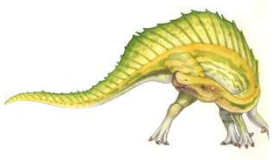
- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



Synchronization

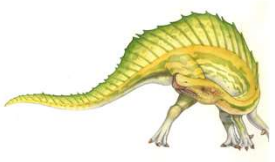
Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**



Buffering

- Queue of messages attached to the link; implemented in one of three ways:
  1. Zero capacity – no (0) messages are queued. Sender must **wait** for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages Sender must wait if the queue is full
  3. Unbounded capacity – infinite length Sender never waits



End of Chapter 3