

---

# **UNIT II**

## **Process Management**

---

---

# CONTENTS

- Process Concept, Process States, Process Description
  - Processes and Threads
  - Concurrency: Principles of Concurrency
  - Mutual Exclusion: Programming Language construct: Semaphores
  - Classical Problems of Synchronization: Readers-Writers problem, Producer Consumer problem, Dining Philosopher problem.
-

# Roadmap

## Process Concept, Process States, Process Description

- How are processes represented and controlled by the OS.
- **Process states** which characterize the behaviour of processes.
- **Data structures** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

# Requirements of an Operating System

- *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.
- The OS Manages Execution of Applications
  - Resources are made available to multiple applications
  - The processor is switched among multiple application
  - The processor and I/O devices can be used efficiently

---

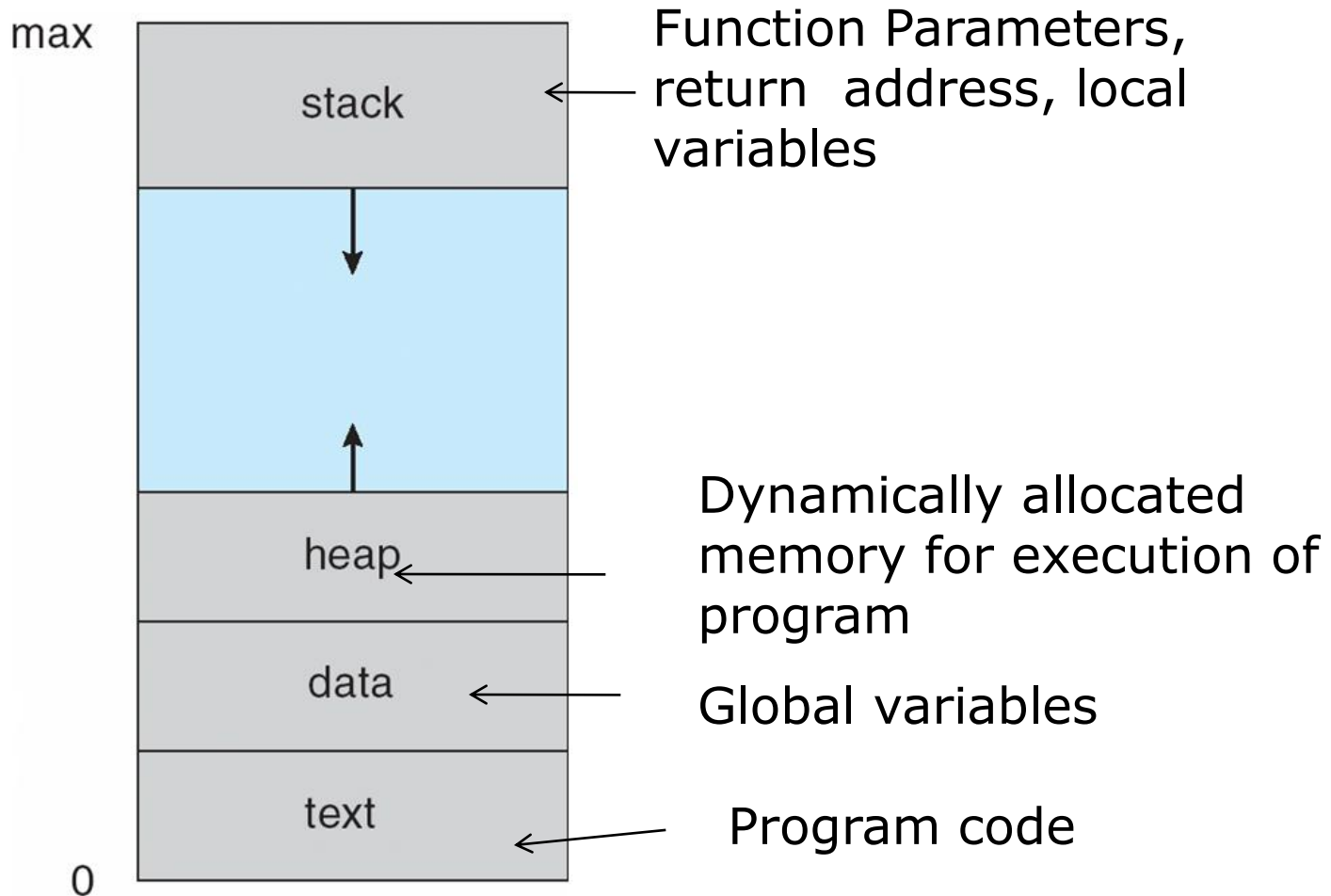
## What is a Process?

- Recall from Unit 1 that *a process is a program in execution.*
  - *A process in execution needs resources like processing resource, memory and IO resource.*
  - Imagine a program written in C – *my\_prog.c.*
  - After compilation we get an executable.
  - If we now give a command like *./a.out* it becomes a *process.*
-

# Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- process is comprised of:
  1. Program code (possibly shared)
  2. A set of data
  3. A number of attributes describing the state of the process

# Process in Memory



## Process

- *A program in execution*
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions



## Process Elements

- While the process is running it has a number of elements including
  - Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers
  - Context data
  - I/O status information
  - Accounting information

---

## Process Control Block

- Contains the process elements
  - Created and manage by the operating system
  - The PCB is constructed at process creation.
  - PCB includes a pointer to be used in lists (queues) of PCBs.
  - Allows support for multiple processes
  - The PCB is used to save information about a process when switched out of CPU.
-

# Process Control Block

<b>Identifier</b>
<b>State</b>
<b>Priority</b>
<b>Program counter</b>
<b>Memory pointers</b>
<b>Context data</b>
<b>I/O status information</b>
<b>Accounting information</b>
<b>Y Y Y</b>

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
• • •	

## Process Control Block

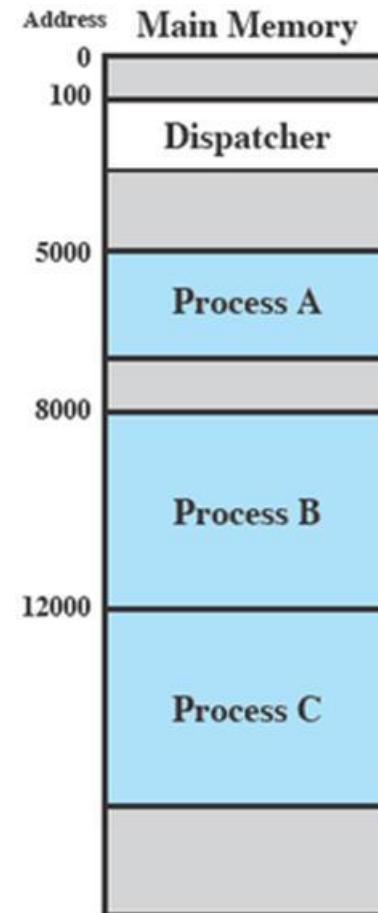
Information associated with each process is stored in PCB.

- Process state : new, running, etc.
- Program counter : address of next instruction to be executed
- CPU registers :E.g. accumulator, stack pointers, general purpose registers.
- CPU scheduling information :Process priority, elapsed time, other scheduling parameters etc.
- Memory-management information :E.g. base and limit registers, page table, segment table
- Accounting information: E.g. amount of CPU & real time used, time limits, account number, etc.
- I/O status information: I/O devices allocated, list of open files, etc.

## Trace of the Process

- The behavior of an individual process is shown by listing the sequence of instructions that are executed
- This list is called a **Trace**
- **Dispatcher** is a small program which switches the processor from one process to another
- It acts as a mediator between the CPU and processes, coordinating the execution and resource allocation to ensure smooth operation of the system.

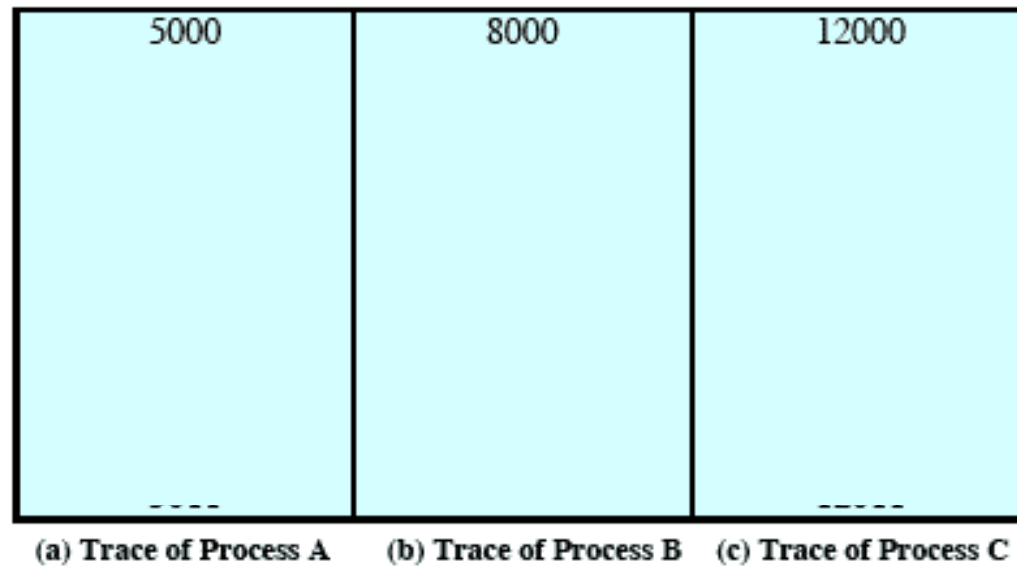
## Process Execution



Consider three processes being executed  
All are in memory (plus the dispatcher)

Trace from the *processes* point of view:

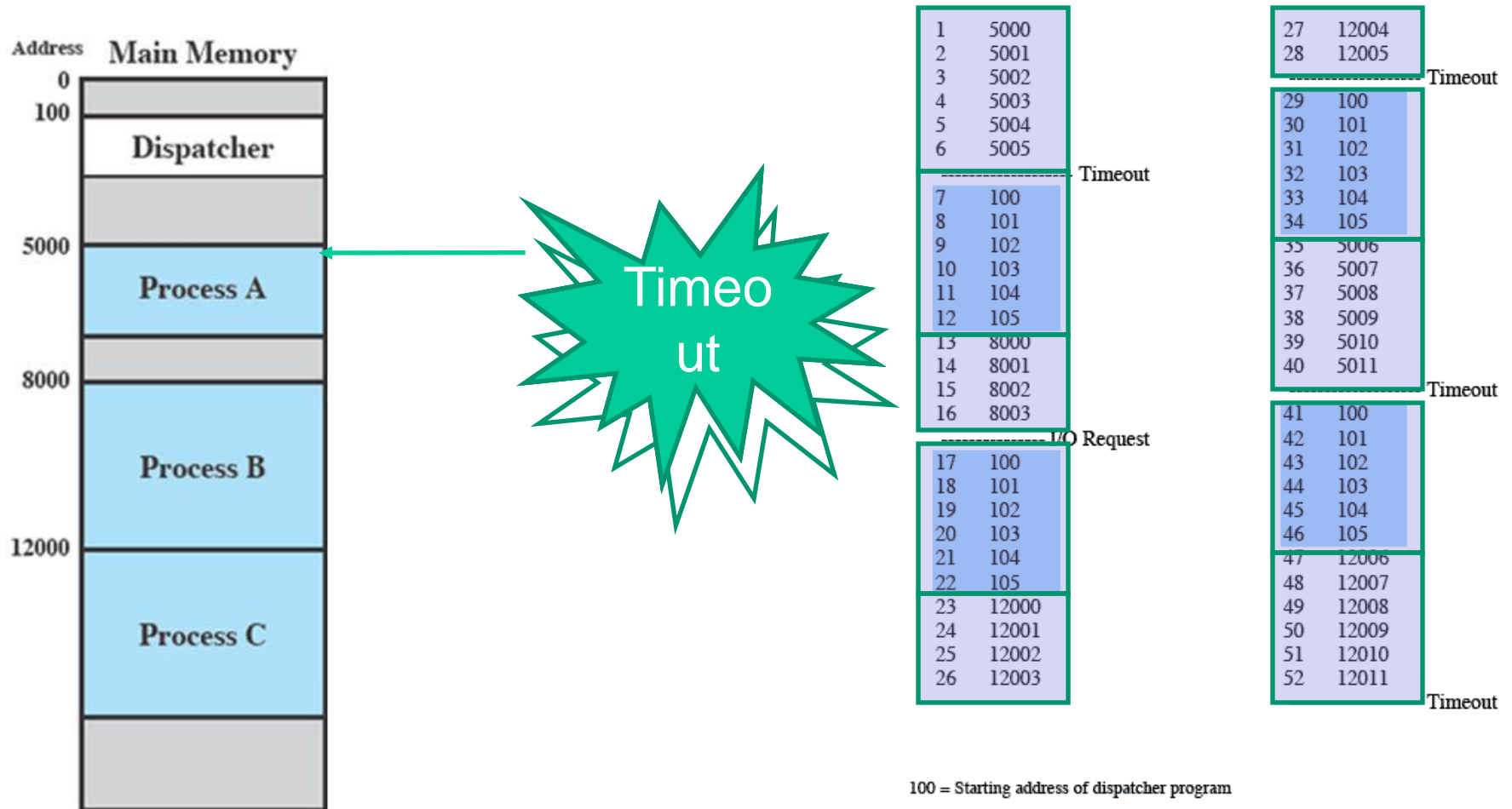
Each process runs to completion



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

# Trace from Processors point of view



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

## Trace from Processors point of view

- The shaded areas represent code executed by the dispatcher.
- The same sequence of instructions is executed by the dispatcher in each instance because the same functionality of the dispatcher is being executed.
- We assume that the OS only allows a process to continue execution for a maximum of six instruction cycles, after which it is interrupted; this prevents any single process from monopolizing processor time.
- The first six instructions of process A are executed, followed by a time-out and the execution of some code in the dispatcher, which executes six instructions before turning control to process B2.
- After four instructions are executed, process B requests an I/O action for which it must wait. Therefore, the processor stops executing process B and moves on, via the dispatcher, to process C.
- After a time-out, the processor moves back to process A. When this process times out, process B is still waiting for the I/O operation to complete, so the dispatcher moves on to process C again



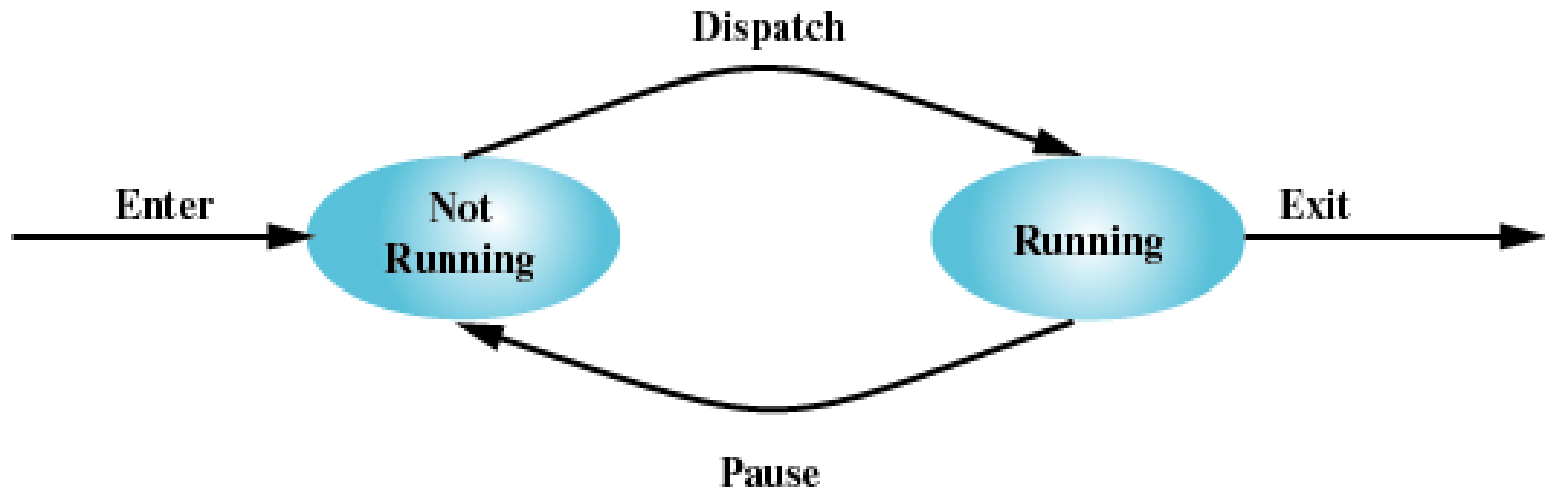
# Roadmap

## Process Concept, Process States, Process Description

- How are processes represented and controlled by the OS.
- **Process states** which characterize the behaviour of processes.
- **Data structures** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

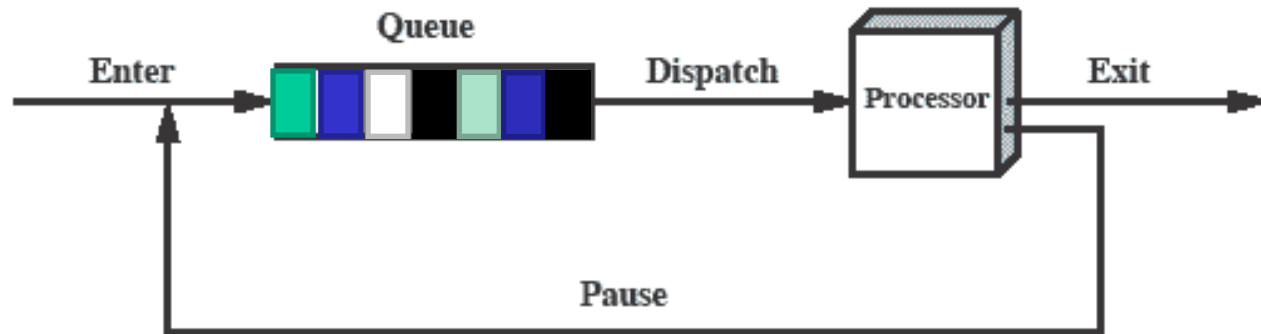
## Two-State Process Model

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



(a) State transition diagram

# Queuing Diagram



(b) Queuing diagram

Etc ... processes moved by the dispatcher of the OS to the CPU then back to the queue until the task is completed

# 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 Creation

- The OS builds a data structure to manage the process
- Traditionally, the OS created all processes
  - But it can be useful to let a running process create another
- This action is called ***process spawning***
  - ***Parent Process*** is the original, creating process
  - ***Child Process*** is the new process

# Process Termination

- There must be some way that a process can indicate completion.
- This indication may be:
  - A HALT instruction generating an interrupt alert to the OS.
  - A user action (e.g. log off, quitting an application)
  - A fault or error
  - Parent process terminating

# Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.

# 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.



## Problems with two state model?

- Are all processes that are “not running” always ready to run?
- Split “Not Running” into two states!
  - Ready
  - Blocked

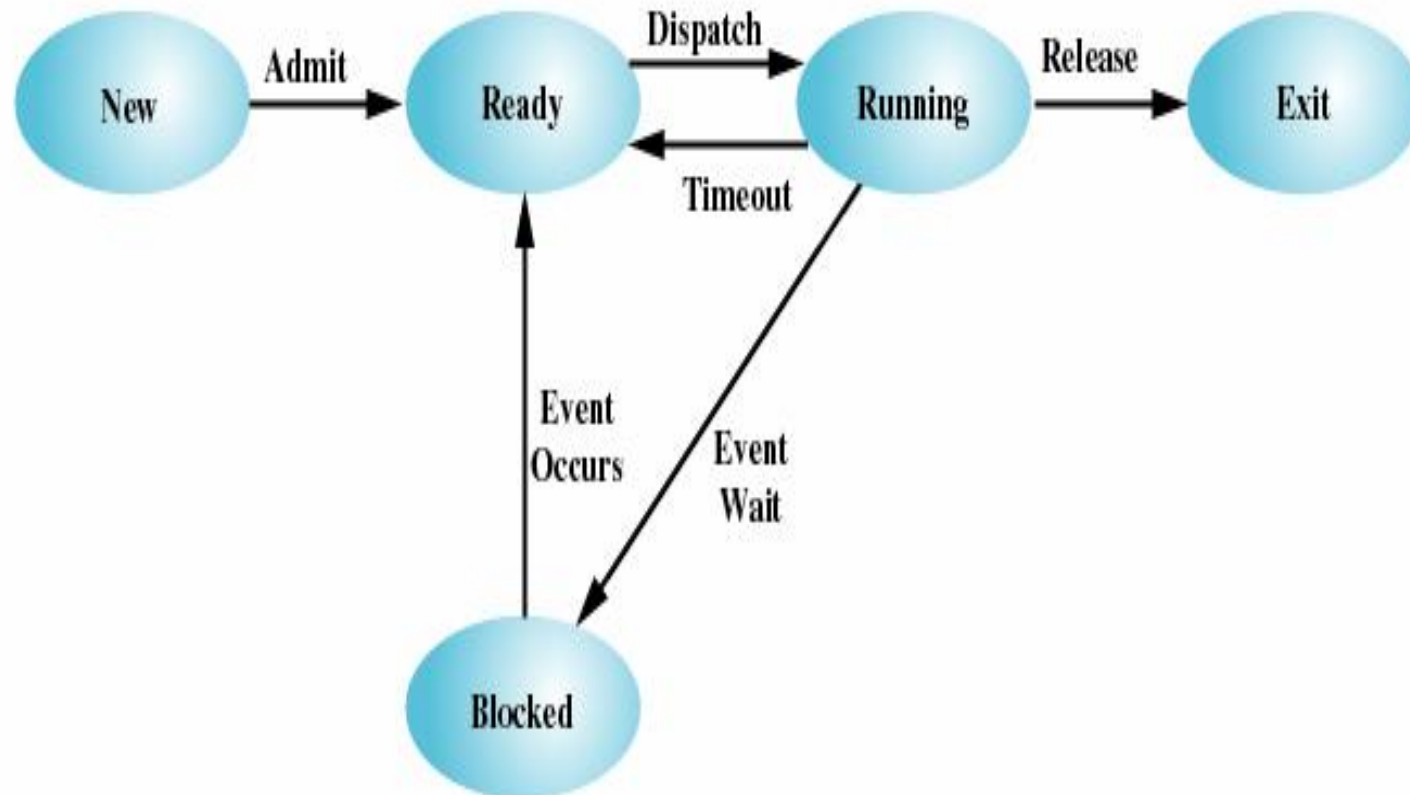
## Process states

- Ready
  - ready to execute
- Blocked
  - waiting for I/O
- Dispatcher cannot just select the process that has been in the queue the longest because it may be blocked

## A Five-State Model

- In all the previous examples, we said
- A process is in “*RUN*” state if it is engaging the processor,
- A process is in “*WAIT*” state if it is waiting for IO to be completed
- In our *simplistic model* we may think of 5 states:
  - Running
  - Ready
  - Blocked
  - New
  - Exit

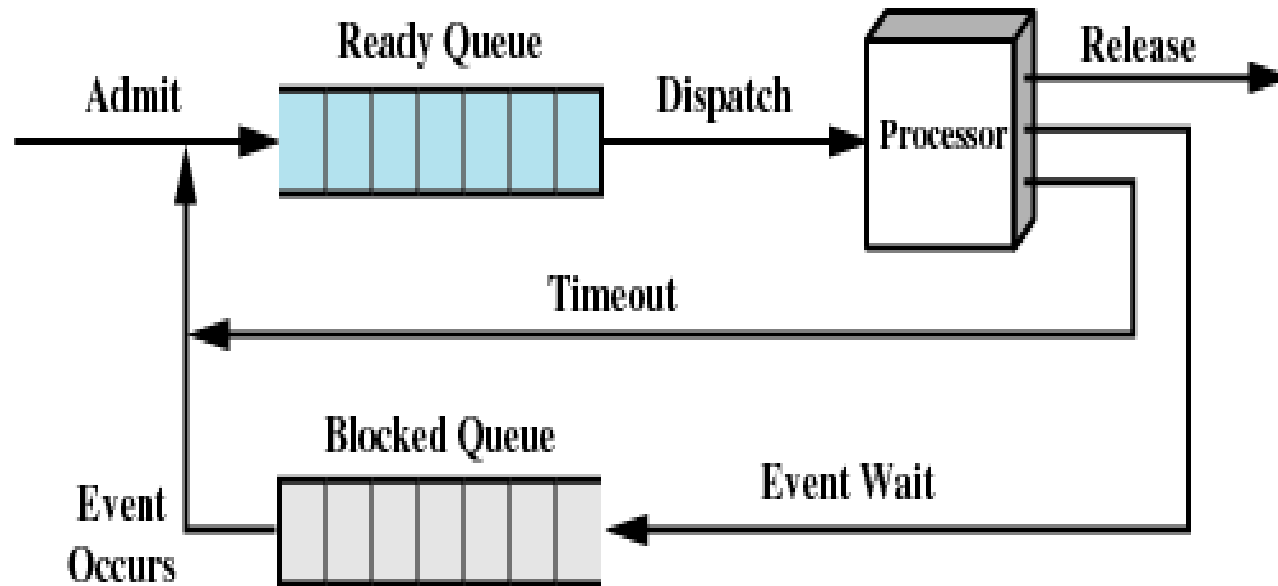
# Five-State Process Model



## Possible transitions of Process State:

- **Null → New** : New process is created to execute program
- **New → Ready** : Submit the new process to ready queue
- **Ready → Running** : Send the Ready Process to Execute
- **Running → Exit** : After completion of Execution, exit the process
- **Running → Ready** : In time sharing system, After time expired of process, put that process again into Ready queue
- **Running → Blocked** : if process requesting for something (I/O), put that in  
blocked queue until it get the resource
- **Blocked → Ready**: After completing request put that process in ready queue
- **Ready → Exit** : Parent process can terminate Child Process at any time
- ~~**Blocked → Exit** : If process requesting inaccessible resource.~~

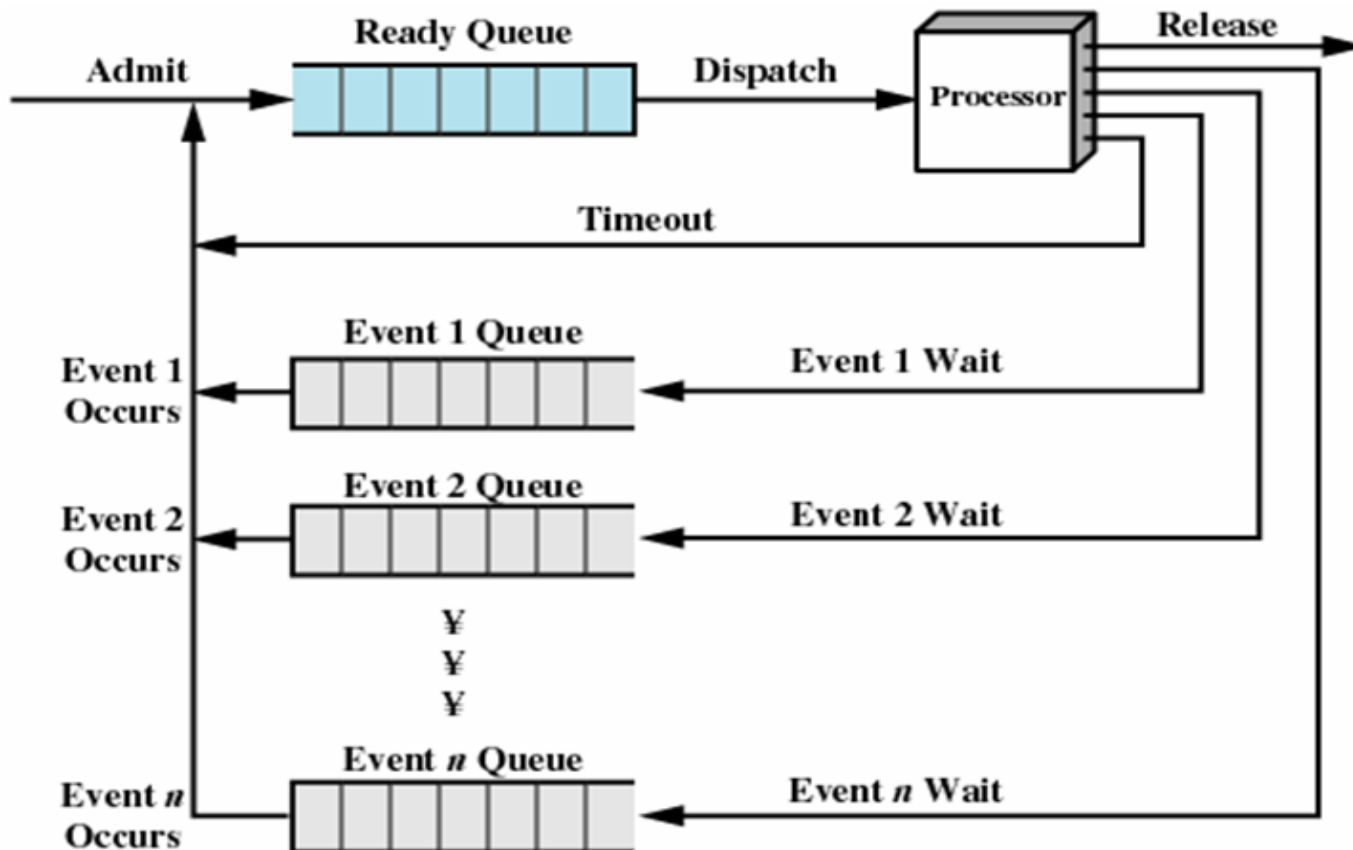
## Using Two Queues



(a) Single blocked queue

This model would require an additional queue for the blocked processes. **But** when an event occurs the dispatcher would have to cycle through the entire queue to see which process is waiting for the event. This can cause huge overhead when there may be 100's or 1000's of processes

## Multiple Blocked Queues



More efficient to have a separate 'blocked' queue for each type of event.

## A problem still exists!

- Even with multi programming processor would be idle most of the time!
  - It is common for all processes in memory to be waiting for I/O soon
- Solution?
  - Increase main memory to accommodate more processes
  - Two problems:
    - Cost
    - More memory need not necessarily mean more processes!



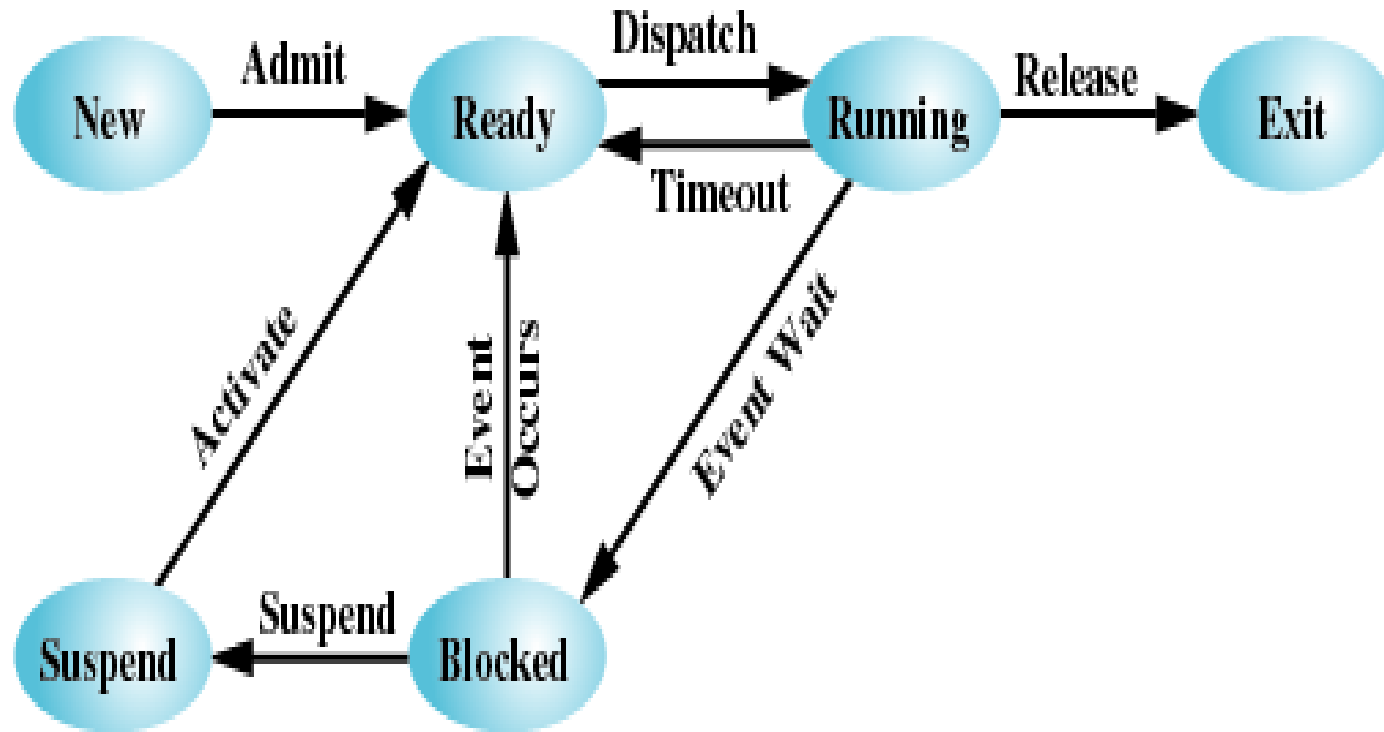
## Swapping

- Moving **part or all** of a process from main memory to disk
  - “Part” in case of virtual memory
- OS swaps one of the blocked processes out onto disk into a suspended queue
  - Queue of existing processes temporarily kicked out
  - Even with Virtual Memory this is required at times for performance reasons
- OS then brings in another process from suspended queue or honors a new process request
- But isn't this also an I/O operation? Wont it make the problem worse?

## Suspended Processes

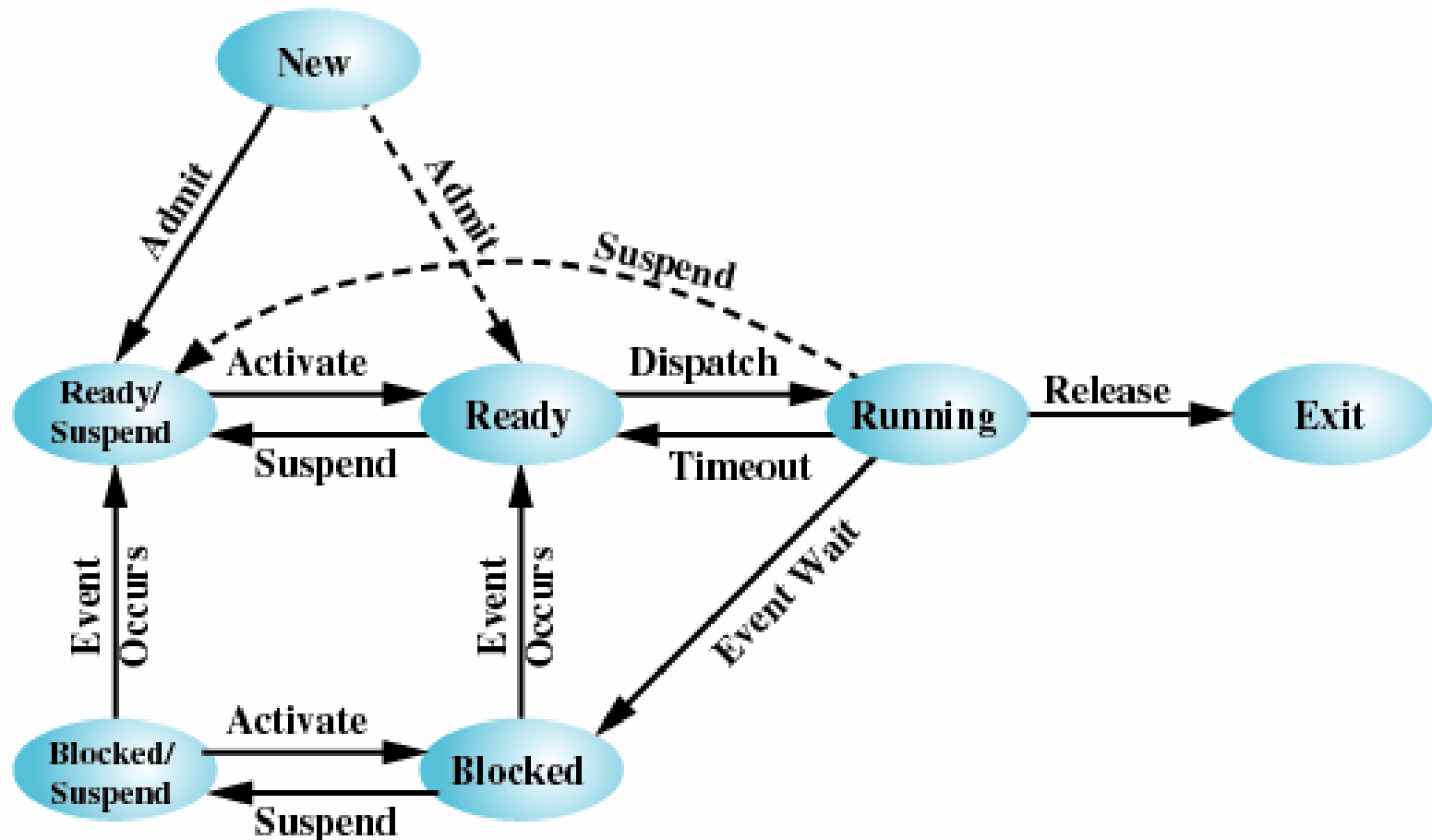
- Processor is faster than I/O so all processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Blocked state becomes **suspend** state when swapped to disk
- Two new states
  - Blocked/Suspend
  - Ready/Suspend

## One Suspend State



(a) With One Suspend State

## Two Suspend States



(b) With Two Suspend States

## Process State Transitions:

- **Ready:** The process is in main memory and available for execution
- **Blocked:** Process is in main memory and waiting for an event
- **Blocked / Suspended:** The process is in secondary memory and waiting for an event
- **Ready / Suspended:** Process is in secondary memory but is available for execution as soon as it loaded it into main memory

## Process State Transitions:

- ***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.
- ***Ready/Suspend → Ready:*** When there are no ready processes in main memory, or if a suspended process has a higher priority, the OS will need to bring one in to continue execution.
- ***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.

## Process State Transitions:

- ***New → Ready/Suspend and New → Ready:*** When a new process is created, it can either be added to the Ready queue or the Ready/Suspend queue.

In either case, the OS must create a process control block and allocate an address space to the process. There would often be insufficient room in main memory for a new process; hence the use of the (New → Ready/Suspend) transition.

- ***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 Transitions:

- ***Running → Ready/Suspend:*** Normally, a running process is moved to the Ready state when its time allocation expires. If, however, the OS is pre-empting 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.



## Characteristics of Suspended process

- The process is not immediately available for execution
- The process may or may not be waiting on an event.
- The process was placed in a suspended state by an agent: either itself, a parent process or OS, for preventing its execution.
- The process may not be removed from this state until the agent explicitly orders the removal.

## Reasons for Process Suspension

Reason	Comment
Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS Reason	OS suspects process of causing a problem.
Interactive User Request	e.g. 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.
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 descendants.

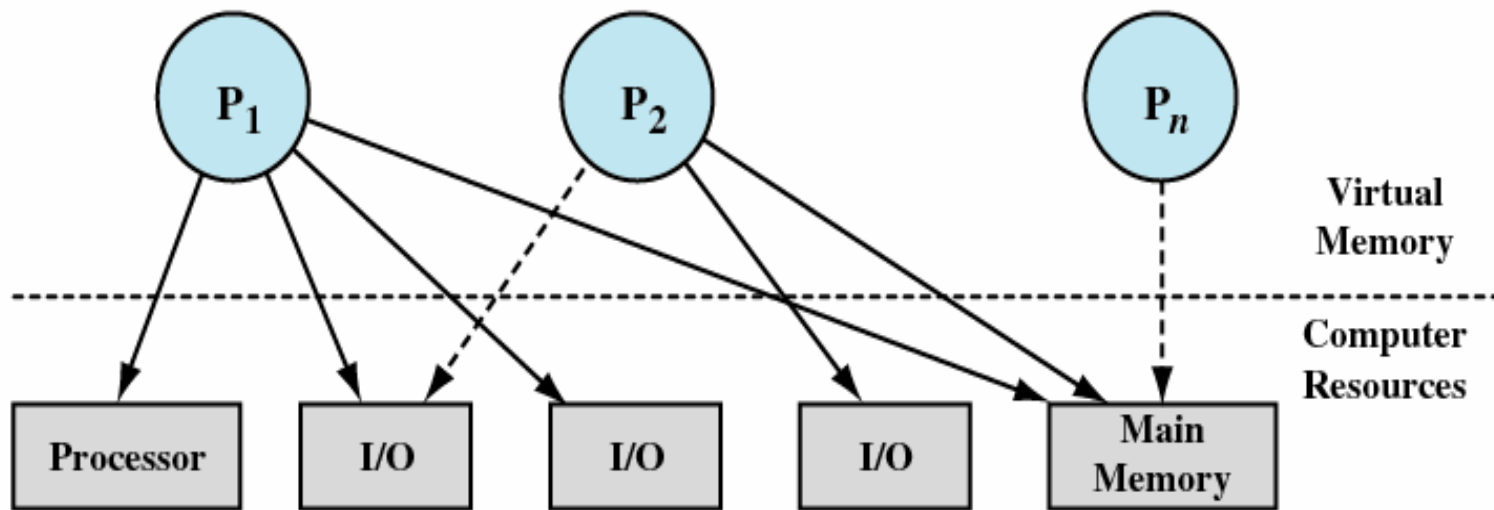
# Roadmap

## Process Concept, Process States, Process Description

- How are processes represented and controlled by the OS.
- *Process states* which characterize the behaviour of processes.
- *Data structures used to manage processes.*
- Ways in which the OS uses these data structures to control process execution.

## Process Description: Processes and Resources

- Operating systems are considered as a manager of the underneath various hardware resources.
- operating system is an entity that manages the use of system resources by processes

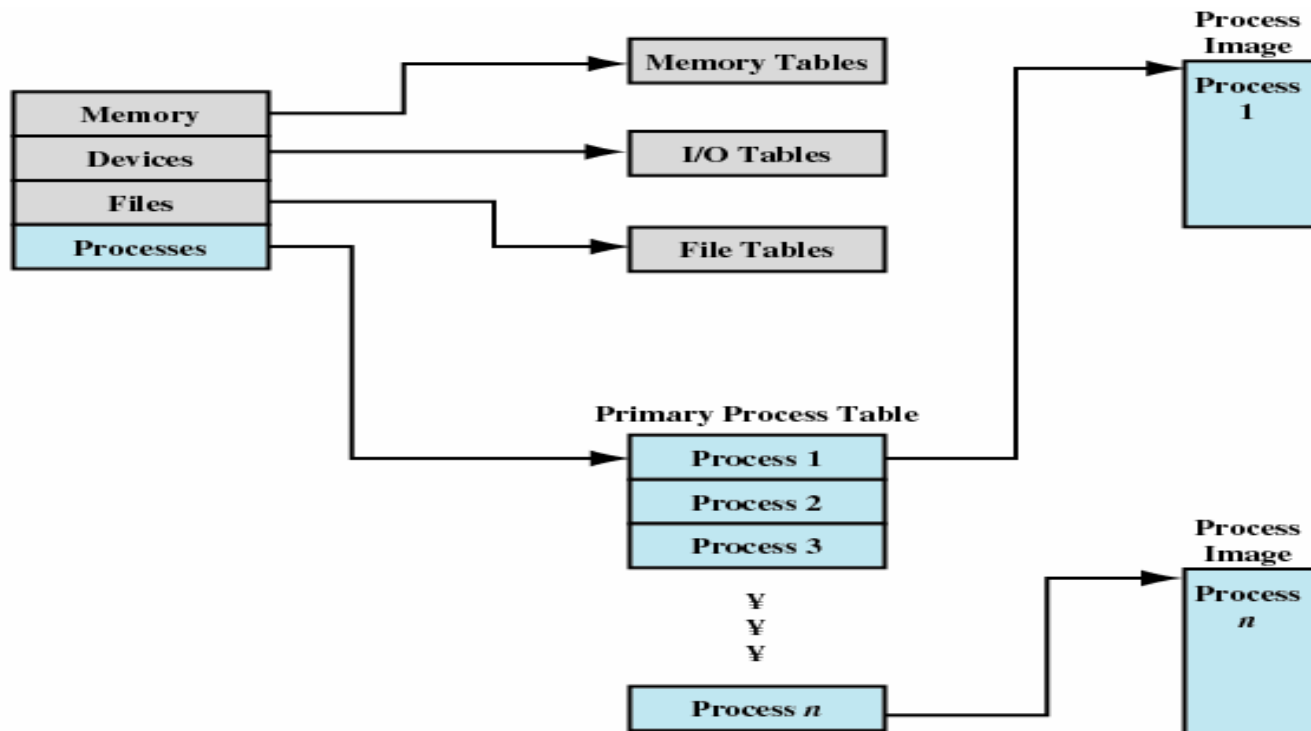


## Process Description: Processes and Resources

- There are a number of processes ( $P_1, \dots, P_n$ ) that have been created and exist in virtual memory.
- Each process, during the course of its execution, needs access to certain system resources, including the processor, I/O devices, and main memory.
- In the figure, process  $P_1$  is running; at least part of the process is in main memory, and it has control of two I/O devices.
- Process  $P_2$  is also in main memory but is blocked waiting for an I/O device allocated to  $P_1$ .
- Process  $P_n$  has been swapped out and is therefore suspended.

# Operating system control tables

- OS has to manage processes and resources, it must have information about the current status of each process and resource.
- Tables are constructed for each entity, the operating system manages.



## Control structures

**1. Memory tables:** are used to keep track of both main memory and secondary memory.

- Part of main memory is reserved for use by the operating system;

The memory tables generally include the following information:

- the allocation of main memory to processes
- the allocation of secondary memory to processes
- protection attributes of blocks of main or virtual memory

**2. I/O tables:** are used by the operating system to manage I/O devices. They should record:

- the availability of each particular device
- the status of I/O operations relating to each device
- the location in main memory being used as the source or destination of the I/O transfer.

# Control structures

**3. File tables:** provides information about

- Existence of files
- Location on secondary memory
- Current Status
- Attributes
- Sometimes this information is maintained by a file management system

**4. Process table :**

- To manage processes the OS needs to know details of the processes
  - Current state
  - Process ID
  - Location in memory
  - etc
- Process control block
  - **Process image** is the collection of program,Data, stack, and attributes



# Process Control Structure/Process Image

## ■ Elements of Process Image:

- User Data: Modifiable part of user space;
    - Includes program data, user stack area and programs that may be modified
  - User Program: the program to be executed
  - System Stack: each process have 1/more system stacks;
    - Used to store parameters and calling address for system calls
  - Process Control Block: data needed by OS to control the process
- 
- We can group the process control block information into three general categories:
    - Process identification
    - Processor state information
    - Process control information

# Elements of a PCB

- **Process Identifier:**

- ☐ Identifier of this process
- ☐ Identifier of the process that created this process (parent process)
- ☐ User identifier

- **Processor state information:**

- User Visible register
- Control and stack register
- Stack pointer
- Program status word (PSW) - contains status information

- **Process control information:**

- Scheduling and state Information
- Data Structuring
- Interprocess communication
- Memory Management
- Resource Ownership and Utilization

# Roadmap

## Process Concept, Process States, Process Description

- How are processes represented and controlled by the OS.
- **Process states** which characterize the behaviour of processes.
- **Data structures** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

# Modes of Execution

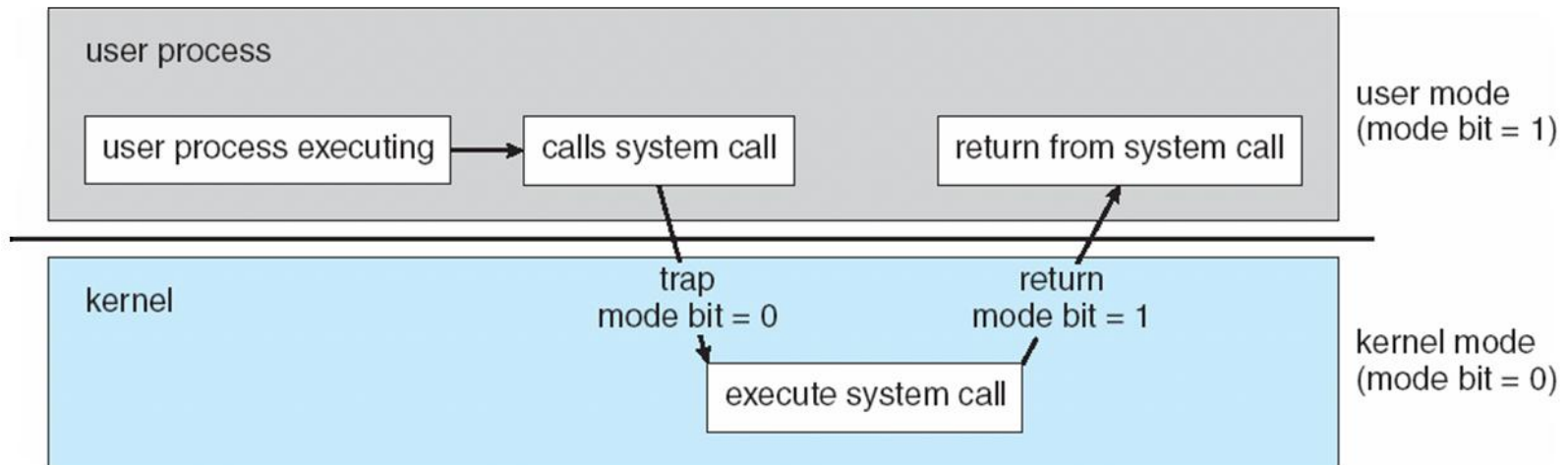
- Most processors support at least two modes of execution
- User mode
  - Less-privileged mode
  - User programs typically execute in this mode
- System mode:also known as
  - Control Mode/ Kernel Mode / Protected Mode
  - More-privileged mode
  - Kernel of the operating system

# Process Control

- **Mode of Execution:** **Dual-mode** operation allows OS to protect itself and other system components
- How does the processor know in which mode it is to be executing? And how does it change
  - **Mode bit** provided by hardware (there is a bit in PSW that indicates the mode of execution)
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

# Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - ❑ Set interrupt after specific period
  - ❑ Operating system decrements counter
  - ❑ When counter becomes zero generate an interrupt
  - ❑ Set up before scheduling process to regain control or terminate program that exceeds allotted time



## Process Creation

■ Once the OS decides to create a new process it:

1. Assign a unique process identifier
2. Allocate space for the process
3. Initialize process control block
4. Set up appropriate linkages

Ex: add new process to linked list used for scheduling queue

5. Create / expand other data structures

Ex: maintain an accounting file

# Switching Processes

- Several design issues are raised regarding process switching
  - What events trigger a process switch?
  - We must distinguish between mode switching and process switching.
  - What must the OS do to the various data structures under its control to achieve a process switch?



# When to switch processes

- A process switch may occur any time that the OS has gained control from the currently running process. Possible events giving OS control are:

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

**Table : Mechanisms for Interrupting the Execution of a Process**

## When to switch processes

- Two kinds of system interrupts,
  - one is simply called an **interrupt**,
  - and the other called a **trap**.
- “interrupts” are due to some sort of event that is external to and independent of the currently running process, such as the completion of an I/O operation.
- With an ordinary interrupt, control is first transferred to an interrupt handler, which does some basic housekeeping and then branches to an OS routine that is concerned with the particular type of interrupt that has occurred.
- With traps, the OS determines if the error or exception condition is fatal.
  - If so, then the currently running process is moved to the Exit state and a process switch occurs.
  - If not, then the action of the OS will depend on the nature of the error and the design of the OS.
    - It may attempt some recovery procedure or simply notify the user.
    - It may do a process switch or resume the currently running process.

# TRAP VERSUS INTERRUPT

## TRAP

A signal raised from a user program that indicates the operating system to perform on some functionality immediately

Generated by an instruction in the user program

Invokes OS functionality - it transfers the control to the trap handler

Synchronous and can arrive after the execution of any instruction

Also called a software interrupt

## INTERRUPT

A signal to the processor emitted by hardware indicating an event that needs immediate attention

Generated by hardware devices

Triggers the processor to execute the corresponding interrupt handler routine

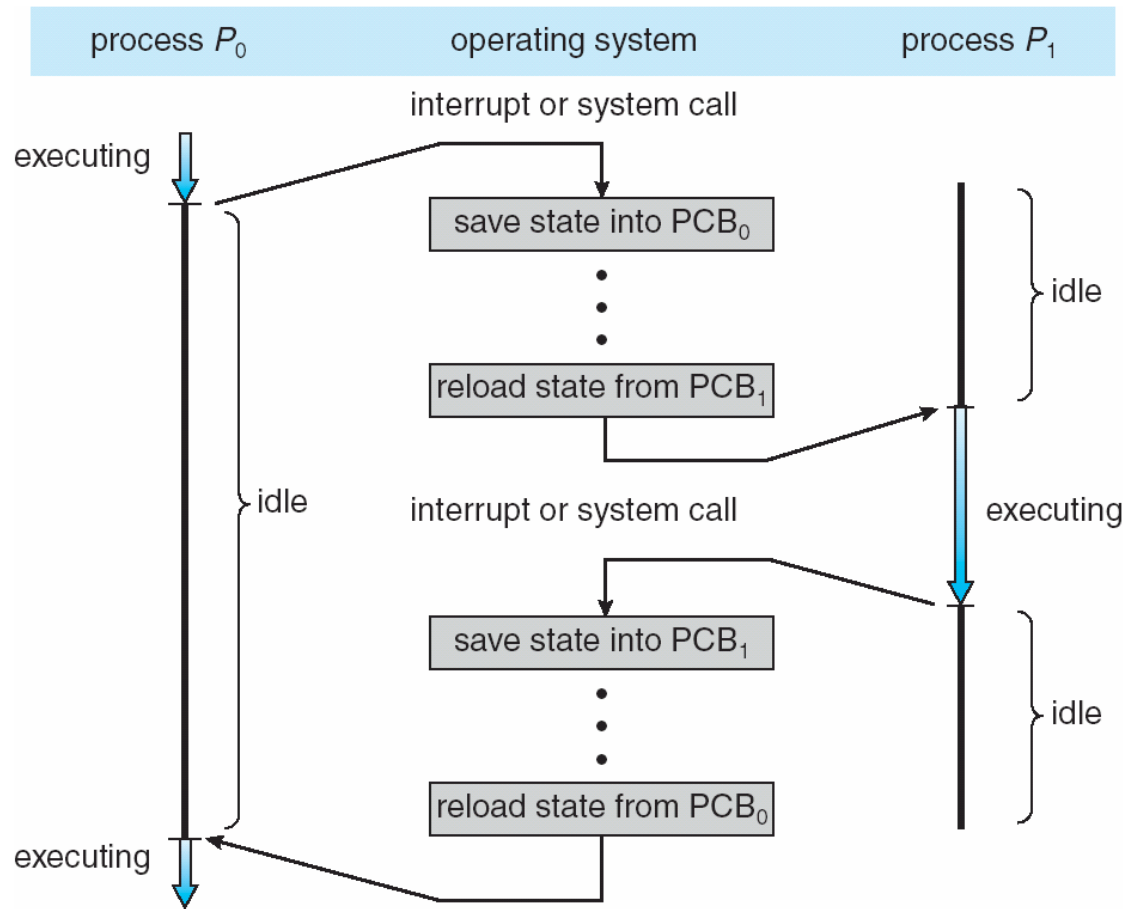
Asynchronous and can occur at the execution of any instruction

Also called a hardware interrupt

## When to switch processes

- the OS may be activated by a **supervisor call** from the program being executed.
  - For example, a user process is running and an instruction is executed that requests an I/O operation, such as a file open.
- This call results in a transfer to a routine that is part of the operating system code.
- The use of a system call may place the user process in the Blocked state

# CPU Switch From Process to Process



## Process Switching (Cont)

### ■ Mode of switching:

- Save the context of the current program being executed
- Set the program counter to the starting address of an interrupt-handler program
- Switch from user mode to kernel mode so that interrupt processing code may include privileged instructions

## Change process state:

- If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment.

### ■ Steps involved in process switching:

1. Save the context of the processor including program counter and other register.
2. Update the PCB of the process that is currently in the Running state
3. Move PCB to appropriate queue – ready; blocked; ready/suspend
4. Select another process for execution
5. Update PCB of selected process
6. Update memory management data structure
7. Restore context of the processor

---

## Threads, SMP

*Operating Systems: Internals and Design Principles,*  
William Stallings

---



# Roadmap

- Threads: Resource ownership and execution

# Processes and Threads

- Processes have two characteristics:
  - **Resource ownership** - process includes a virtual address space to hold the process image
  - **Scheduling/execution** - follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system
- The unit of dispatching is referred to as a **thread** or lightweight process
- The unit of resource ownership is referred to as a process or **task**

---

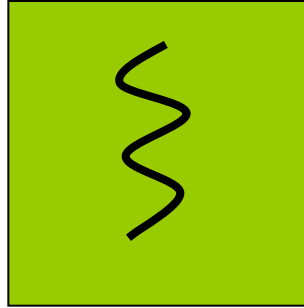
# Processes and Threads

- Unit of dispatching is referred to as a thread (Light weight process)
  - Resource ownership is referred to as a process or task (Heavy weight process)
  - A process has
    - a virtual address space which holds the process image
    - global variables, files, child processes, signals and signal handlers
  - A traditional process has single thread of control.
  - If a process has multiple thread of control, it can do more than one task at a time.
-

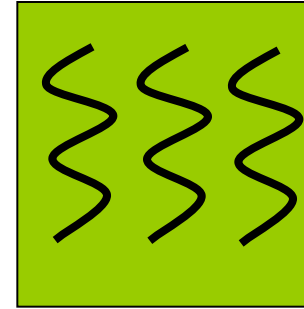
DOS

## Threads and Process

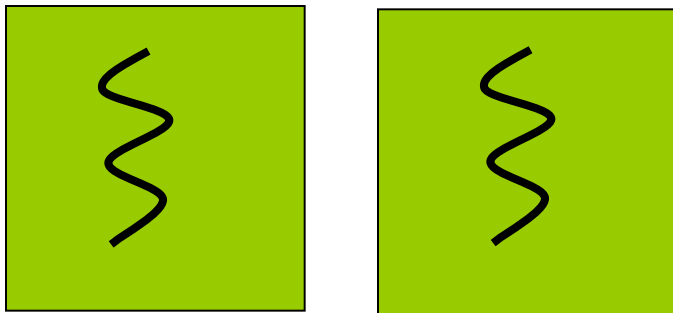
JAVA



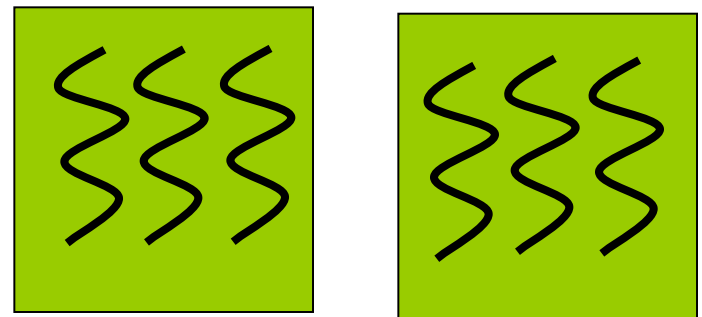
One process one thread



One process multiple thread



Multiple processes one thread per process



Multiple processes multiple thread per process

UNIX

WINDOWS

# Multithreading

- The ability of an OS to support multiple, concurrent paths of execution within a single process.

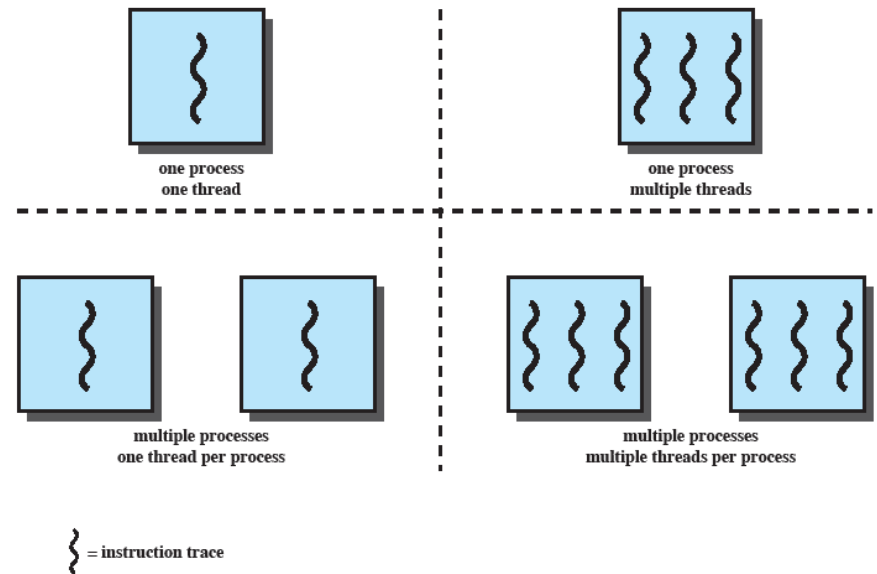


Figure 4.1 Threads and Processes [ANDE97]

# Single Thread Approaches

- MS-DOS supports a single user process and a single thread.
- Some UNIX, support multiple user processes but only support one thread per process

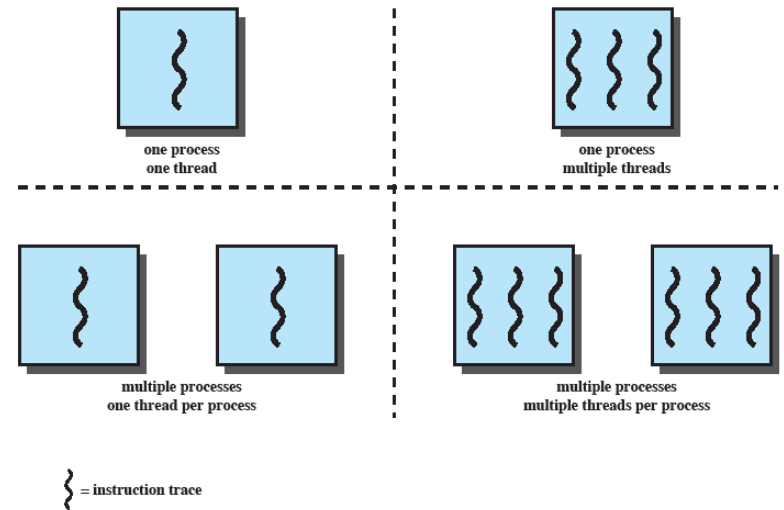


Figure 4.1 Threads and Processes [ANDE97]

# Multithreading

- Java run-time environment is a single process with multiple threads
- Multiple processes **and** threads are found in Windows, Solaris, and many modern versions of UNIX

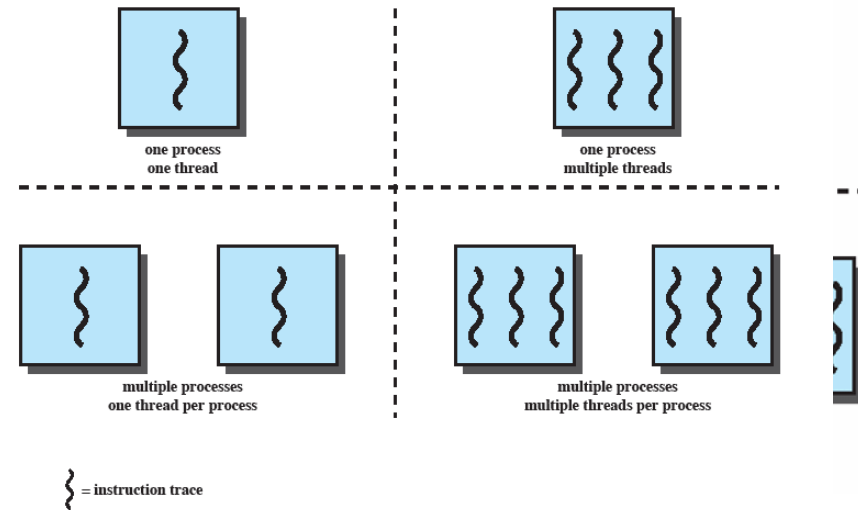
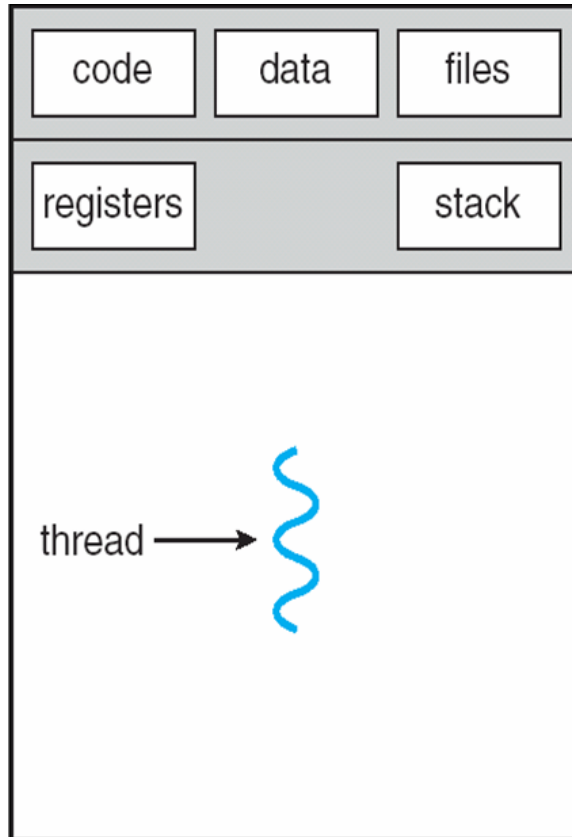
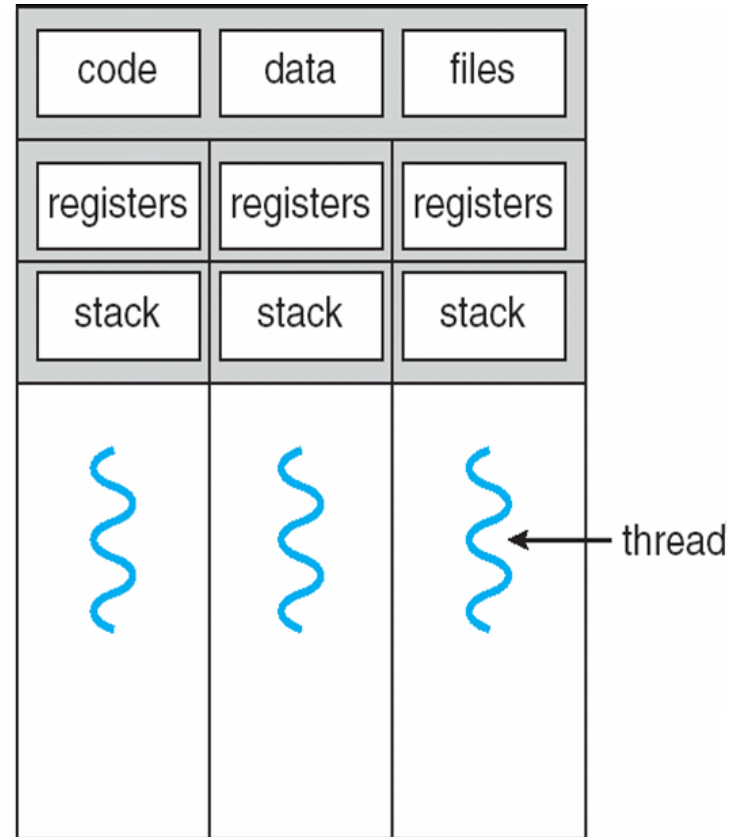


Figure 4.1 Threads and Processes [ANDE97]

# Single and Multithreaded Processes



single-threaded process



multithreaded process



## Thread attributes:

A thread has

- Execution state
- Save the thread context when not running
- Execution stack
- Per-thread static storage for local variable
- Access to the memory and resources of its process, shared with all other threads in that process.
- *One way to view a thread is as an independent program counter operating within a process.*

## Benefits of threads

1. Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.
2. It takes less time to create new thread in an existing process than to create brand new process.
3. It takes less time to terminate a thread than a process.
4. It takes less time to switch between two thread with in the same process.
5. Threads enhance efficiency in communication between different executing programs.
6. Utilization of multiple processor architecture, where each thread may be running in parallel on a different processor.

---

## Disadvantages of Threads

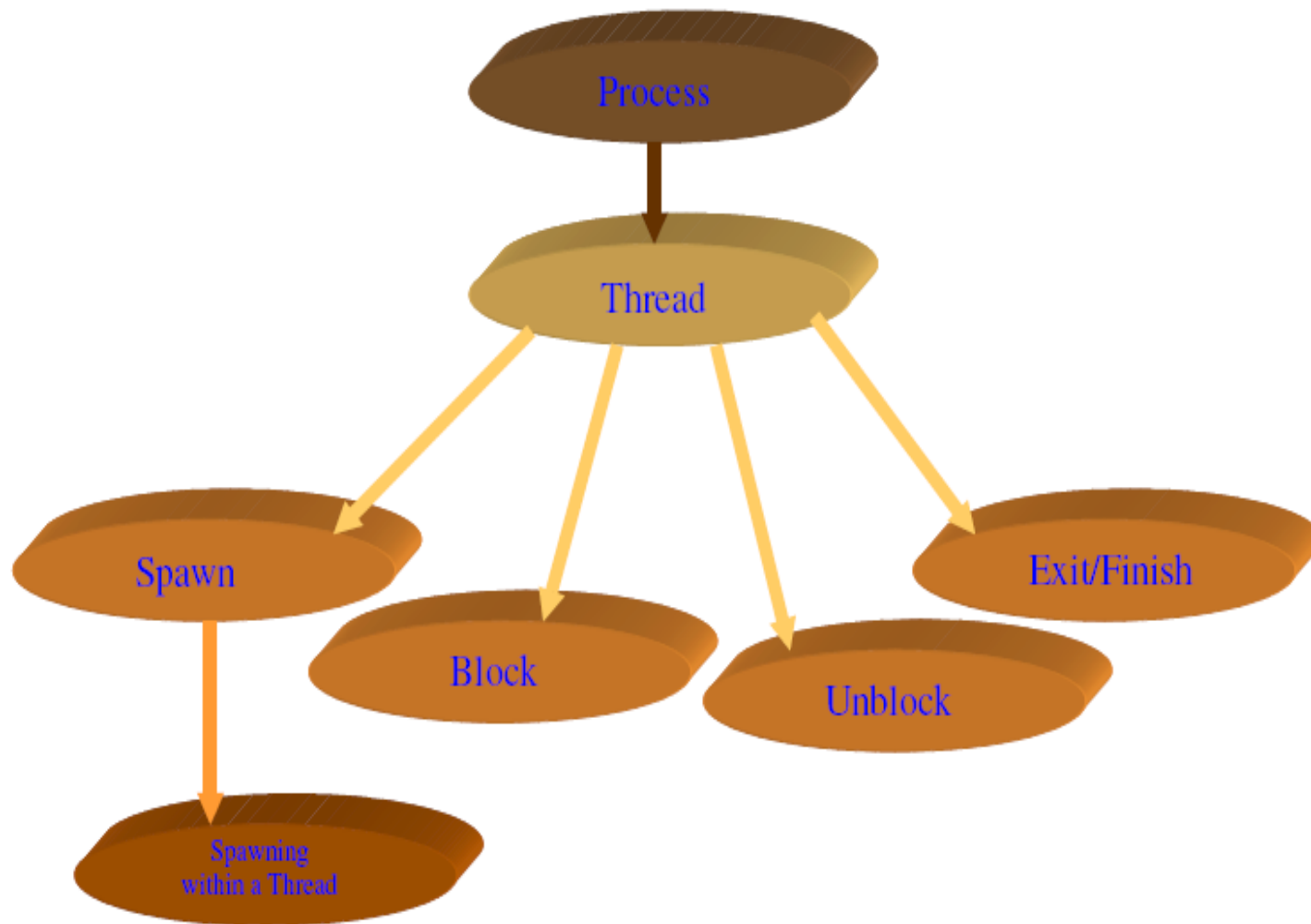
- Suspending a process involves suspending all threads of the process since all threads share the same address space.
  - Termination of a process, terminates all threads within the process.
-

# Thread Execution States

## ■ Thread State/ States associated with a change in thread state:

1. Spawn (another thread)
2. Block
  - Issue: will blocking a thread block other, or *all*, threads
3. Unblock
4. Finish (thread)
  - Deallocate register context and stacks

## Thread state tree



---

## Benefits

1. Responsiveness: ex: Web browser
  2. Resource Sharing: shares code /data section of program
  3. Economy: due to memory & resource sharing  
In Solaris, for example, creating process is 30 times slower than creating threads and context switching is 5 times slower.
1. Scalability: Utilization of multiprocessor architecture
  2. Efficient communication.
-

---

## Categories of Thread Implementation

- User Level Thread (ULT)
  - Kernel level Thread (KLT) also called:
    - kernel-supported threads
    - lightweight processes.
-

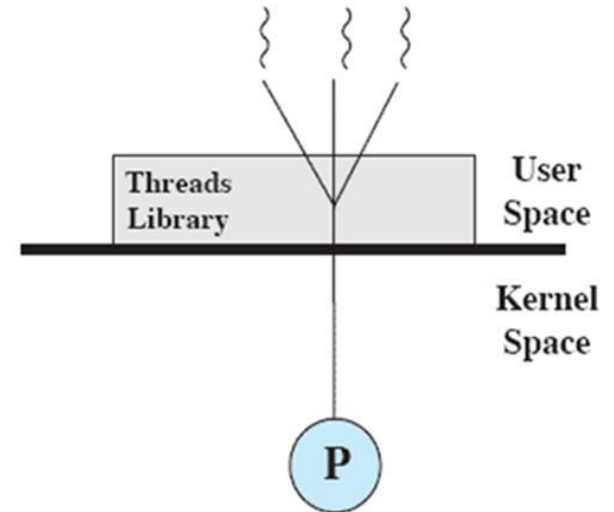
# Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - ❑ Library entirely in user space
  - ❑ Kernel-level library supported by the OS
    - Three primary thread libraries:
      1. Posix Pthreads: (IEEE) standard:
        - Specifies interface
        - Implementation (using user/kernel level threads) is up to the developer(s)
        - More common in UNIX systems
      2. Win32 thread library:
        - Kernel-level library, windows systems
      3. Java threads:
        - Supported by the JVM
        - Implementation is up to the developers –e.g. can use Pthreads API or Win32 API, etc



# User Threads

- Thread management done by user-level threads library
- The kernel is not aware of the existence of threads
- The library provides support for thread creation, scheduling and management without support from kernel.
- Generally fast to create and manage as no kernel intervention.
- Drawback:
  - ULT performing blocking system call will cause entire process to block.(solution: Jacketing)
  - Multithreaded application cannot take advantage of multiprocessing



(a) Pure user-level

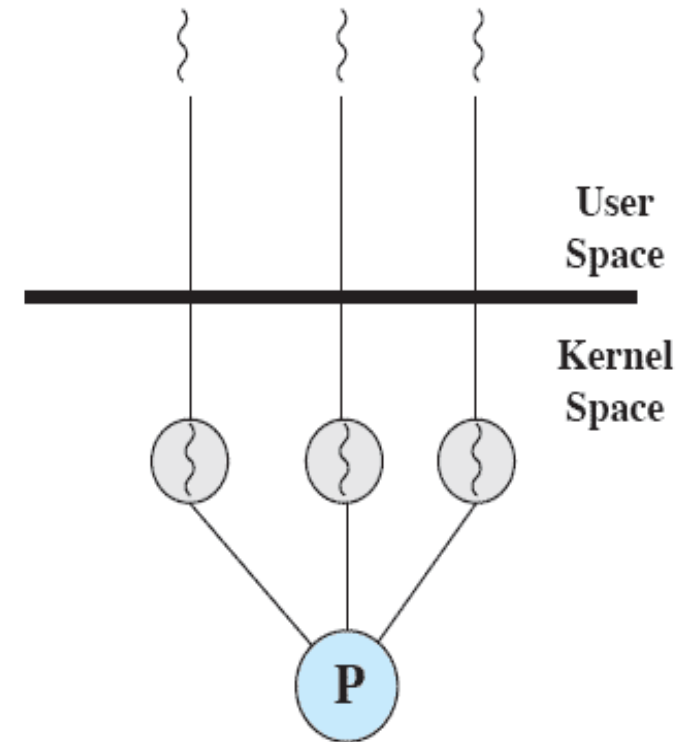
# Kernel Threads

- Supported by the Kernel (Operating system)
- Kernel performs thread creation, scheduling and management.
- Advantages:
  1. The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
  2. If one thread in a process is blocked, the kernel can schedule another thread of the same process.
  3. Kernel routines themselves can be multithreaded
  4. Disadvantages:
- Generally slower to create and manage than ULT
- Needs mode switch
- Examples

- Windows XP/2000 ,Solaris, Linux

- Tru64 UNIX

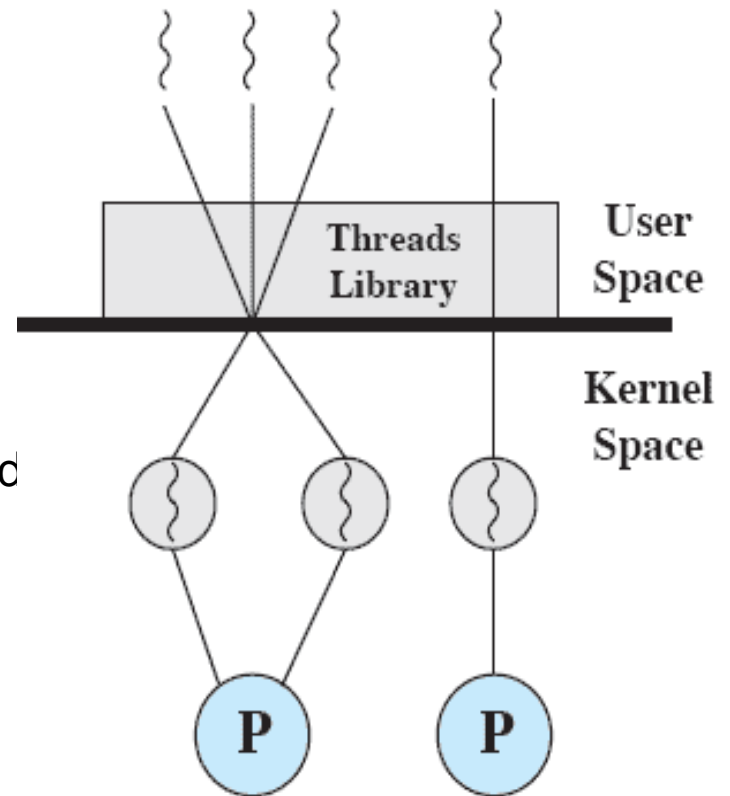
- Mac OS X



(b) Pure kernel-level

## Combined Approaches

- Example is Solaris
- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space.
- If properly designed, this approach should combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages



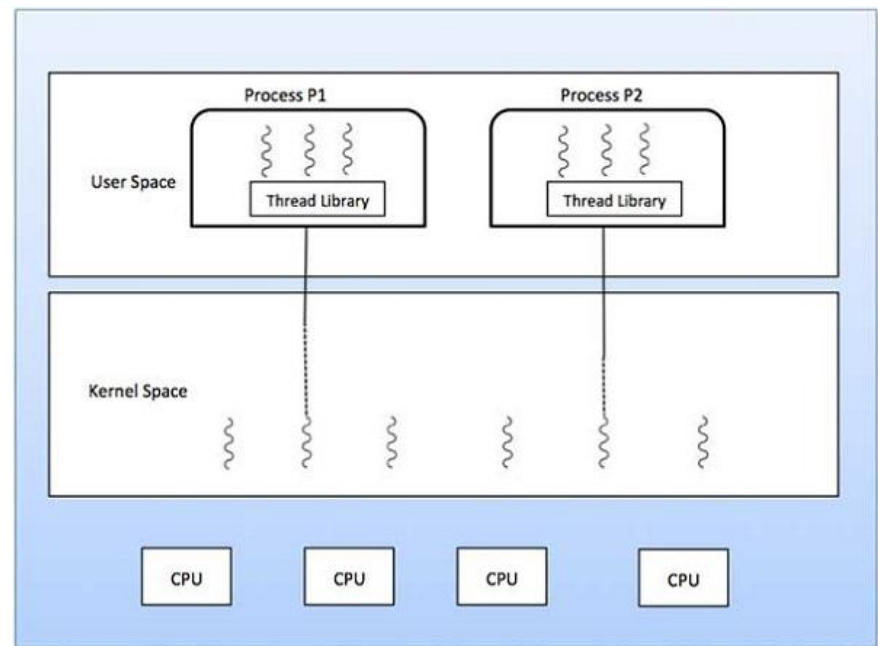
(c) Combined

# Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

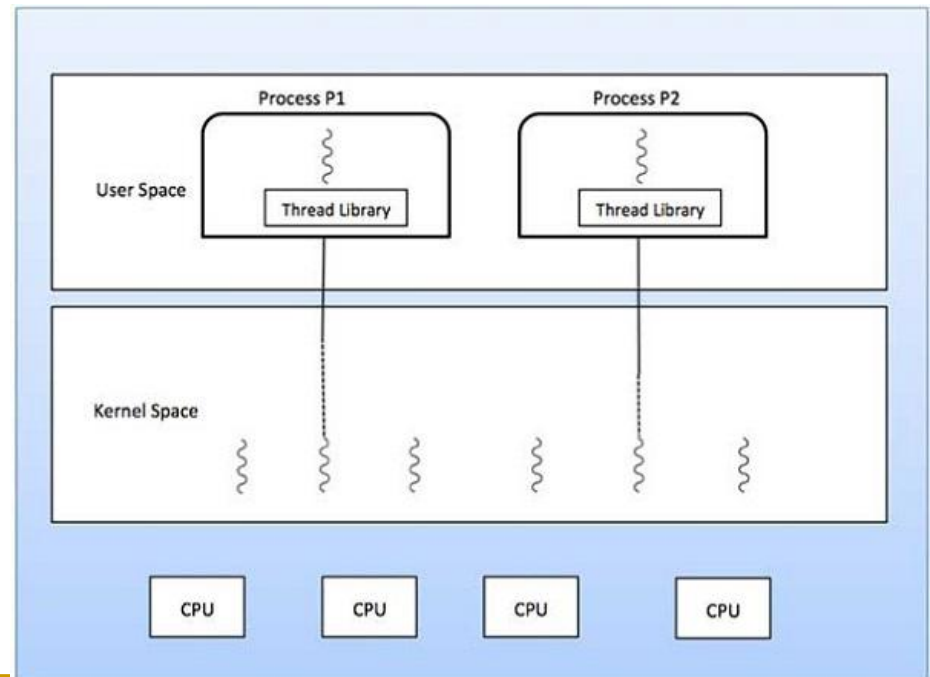
# Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by thread library in user space.
- Entire process will block if a thread makes a blocking system call.
- Examples:
  - ❑ Solaris Green Threads
  - ❑ Windows NT



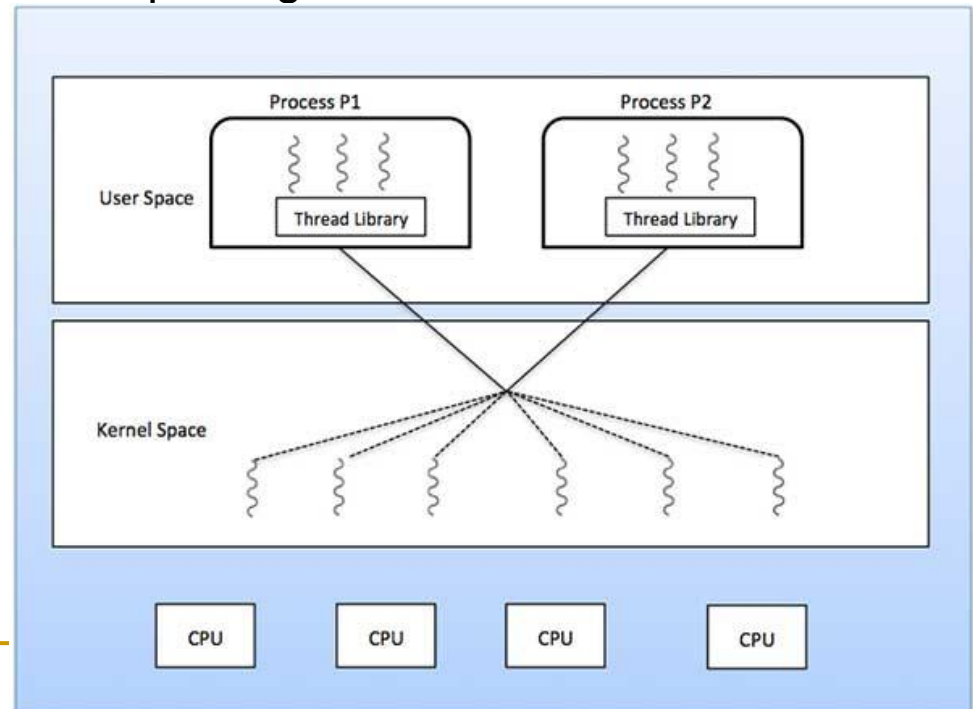
# One-to-One

- Each user-level thread maps to kernel thread
- Provides more concurrency than many-to-one
- Drawback: Creating user level thread requires creation of corresponding kernel level thread
- Examples
  - ❑ Windows 98/XP/2000
  - ❑ Traditional Unix
  - ❑ Solaris 9 and later



# Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- The number of kernel threads may be application /system dependent.
- This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package
- Ex: TRIX



# Relationship Between Thread and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX



---

## Advantages of ULT over KLT:

- Thread switching does not require kernel mode privileges.
- Scheduling can be application specific.
- ULT can run over any OS.

## Disadvantages of ULT over KLT:

- When a ULT executes a blocking system calls, not only is that thread blocked but all of the thread within the process are blocked
  - In pure ULT strategy, multithreaded application can not take advantage of multiprocessing.
-

# Process Vs Thread

## PROCESS

- Process is heavy weight
- Do not share their address space
- Can execute independent of each other and the synchronization between processes is taken care by kernel only
- Context switching is slow.
- The interaction between two processes is achieved only through the standard inter process communication
- Processes should be used for programs that need coarser parallelism.
- Need more resources

## THREAD

- It is light weight
- Threads executing under same process share the address space.
- Thread synchronization has to be taken care by the process under which the threads are executing
- Context switching is fast
- Threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc
- Threads should be used for programs that need fine-grained parallelism

## Process Vs Thread

### PID

- A process ID is unique across the system
- A process ID is an integer value
- A process ID can be printed very easily.

### TID

- A thread ID is unique only in context of a single process.
- Thread ID is not necessarily an integer value. It could well be a structure
- A thread ID is not easy to print.

## ULT Vs KLT

### ULT

- Created, controlled and destroyed using user space thread libraries
- Not known to kernel and hence kernel is nowhere involved in their processing
- These threads follow co-operative multitasking, the scheduler cannot preempt the thread.
- User-level thread is generic and can run on any operating system.

### KTL

- Created, controlled and destroyed by the kernel
- For every thread that exists in user space there is a corresponding kernel thread so are managed by kernel
- Since these threads are managed by kernel so they follow preemptive multitasking
- Kernel-level thread is specific to the operating system.

## ULT Vs KLT

### UTL

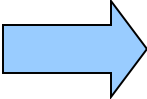
- The switching between two threads does not involve much overhead and is generally very fast
- These threads follow co-operative multitasking so if one thread gets blocked the whole process gets blocked.

### KTL

- The context switch is not very fast as compared to user space threads.
- The major advantage of kernel threads is that even if one of the thread gets blocked the whole process is not blocked

---

# CONTENTS

- Process Concept, Process States, Process Description
  - Processes and Threads
  - Symmetric Multiprocessing, Microkernel
  -  Concurrency: Principles of Concurrency
    - Mutual Exclusion: Programming Language construct: Semaphores, Monitors
    - Classical Problems of Synchronization: Readers-Writers problem, Producer Consumer problem, Dining Philosopher problem.
-

---

Principles of concurrency, Mutual exclusion

*Operating Systems :*  
*Internals and Design Principles*  
William Stallings

---

---

# Concurrency

- **Multi-programming:**

- Management of multiple processes within a uniprocessor system, every system has this support, whether big, small or complex.

- **Multi-processing:**

- Management of multiple processes within a multi-processor system, servers and works stations. (Shared Memory)

- **Distributed Processing:**

- Management of multiple processes executing on number of distributed computer systems, for example clusters. (Do not Share Memory)



**One thing is common among all  
these categories?**

- Concurrency: Execution of multiple processes no matter on single or more processing elements.
-



## concurrency and parallelism

- **Concurrency:** Concurrency is about dealing with multiple tasks at once, but not necessarily simultaneously. It involves managing the execution of multiple tasks in a way that they make progress independently, even if not all tasks are running at the same time.
- **Example:** a web server that handles multiple client requests. It can manage multiple connections by switching between requests, ensuring that no single request monopolizes the server's resources
- **Parallelism:** Parallelism is about performing multiple tasks simultaneously by using multiple processing units (e.g., multiple CPU cores or processors) to execute tasks at the same time
- **Example:** scientific simulation that divides its calculations into smaller chunks, which are then processed simultaneously by multiple CPU cores. Each core works on a different chunk of the computation in parallel.
- A system can be both concurrent and parallel. For example, a modern multi-core processor can handle multiple threads concurrently, and within each thread, parallel computation can occur across multiple cores.

---

# Principles of Concurrency

## ■ **Interleaving:**

In a single processor case, multiple processes are interleaved in time to provide the illusion of simultaneous execution of these processes.

- Although, it is not really parallel processing but there are benefits using such technique, apart from having overheads involved in switching of these processes.
-

# Key Terms

**Table 5.1 Some Key Terms Related to Concurrency**

<b>atomic operation</b>	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
<b>critical section</b>	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
<b>deadlock</b>	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
<b>livelock</b>	A situation in which two or more processes continuously change their states in response to changes in the other <u>process(es)</u> without doing any useful work.
<b>mutual exclusion</b>	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
<b>race condition</b>	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
<b>starvation</b>	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

- 
- critical region: Imagine a bank account system where multiple users can deposit or withdraw money from the same account simultaneously.
  - Example: Using the Family Bathroom
  - Livelock: Two Vehicles at an Intersection
  - race condition: Shared Online Shopping Cart
-

# Atomic operation

- **scenario:**
- Consider a family with a shared piggy bank where multiple members can add money. Each person can contribute to the piggy bank, but the total amount must be accurately recorded.
- **Problem:**
- Suppose two family members, Alice and Bob, want to add money to the piggy bank at the same time. If their additions aren't managed correctly, the total amount of money in the piggy bank might be inaccurately recorded.
- **Atomic Operation:**
- **Add Money:** Each time a person adds money to the piggy bank, they should update the total amount in a way that ensures the operation is completed without interference from others.
- **Atomicity in Action:**
  - **Step 1:** Alice wants to add \$10. She first checks the current total, which is \$50.
  - **Step 2:** At the same time, Bob wants to add \$20. He also checks the total, which is still \$50.
  - **Step 3:** Alice updates the total to \$60 ( $\$50 + \$10$ ) and adds her money to the piggy bank.
  - **Step 4:** Bob also updates the total to \$70 ( $\$50 + \$20$ ) and adds his money.
- If the operations are not atomic, it's possible both Alice and Bob might see the same initial total of \$50 and both update it, resulting in an incorrect total of \$70 instead of the correct \$80 ( $\$50 + \$10 + \$20$ ).

- **Scenario:**
- Imagine two vehicles approaching a four-way intersection where there are no traffic signals or signs, and they both need to turn left.
- **Problem:**
- Both vehicles want to turn left and they are facing each other at the intersection. To avoid colliding, each vehicle decides to yield and wait for the other to go first. However, as each vehicle waits, they keep inching forward and adjusting their positions to try to pass each other.
- **Livelock Situation:**
- **Vehicle A** starts to move forward slightly, trying to clear the intersection.
- **Vehicle B** sees Vehicle A moving and also starts to move forward to make room.
- **Vehicle A** notices that Vehicle B is moving, so it stops to wait again.
- **Vehicle B** also stops, waiting for Vehicle A to clear the way.
- Both vehicles are actively moving and adjusting their positions in response to each other, but neither vehicle is actually able to turn left and clear the intersection. They are stuck in a loop of continually adjusting their positions without making real progress.
- **Example: Two People Trying to Pass Through a Narrow Doorway**
- **Scenario:**
- Imagine two people trying to pass through a narrow doorway from opposite directions.
- **Problem:**
- **Person A** is trying to exit a room and **Person B** is trying to enter the room through the same doorway.
- To avoid colliding, both individuals try to make space for the other to pass.
- **Livelock Situation:**
- **Person A** steps aside slightly to let **Person B** enter, but **Person B** also moves to make space.
- **Person B** moves forward to pass through, but **Person A** shifts to accommodate.
- Each person keeps adjusting their position to allow the other to pass, but neither of them makes progress through the doorway.

# Race condition

- **Scenario:**

Imagine two people, Alice and Bob, are both shopping online on the same e-commerce site and they both have the same item in their shopping cart. They both try to purchase the last available item at the same time.

- **Problem:**

**Alice** and **Bob** both view the product and see that there is only one item left in stock.

They both add the item to their cart and proceed to checkout.

- **Race Condition:**

- **Alice** starts the checkout process and, at the same time, **Bob** starts his checkout process.
- The e-commerce site doesn't properly synchronize the inventory, so both checkouts might proceed simultaneously without one transaction updating the stock before the other.
- Both transactions could be processed, leading to a situation where both Alice and Bob think they have purchased the last item.

# Interleaving and Overlapping Processes

- Earlier we saw that processes may be interleaved on uniprocessors

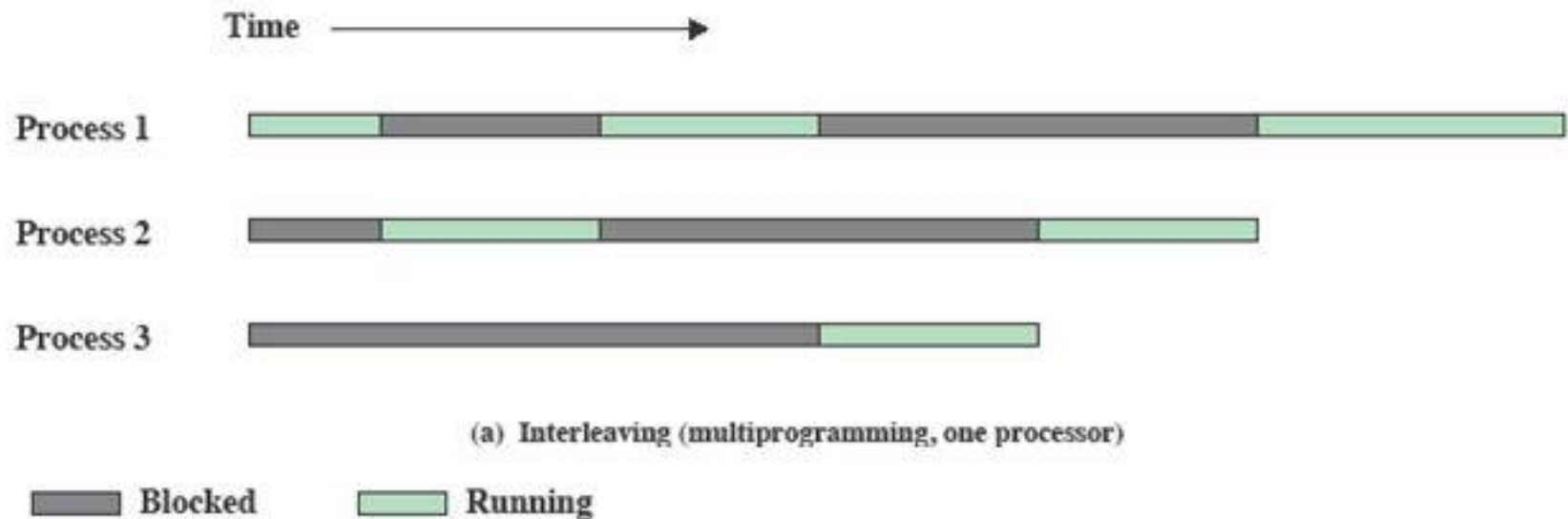


Figure 2.12 Multiprogramming and Multiprocessing



# Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors

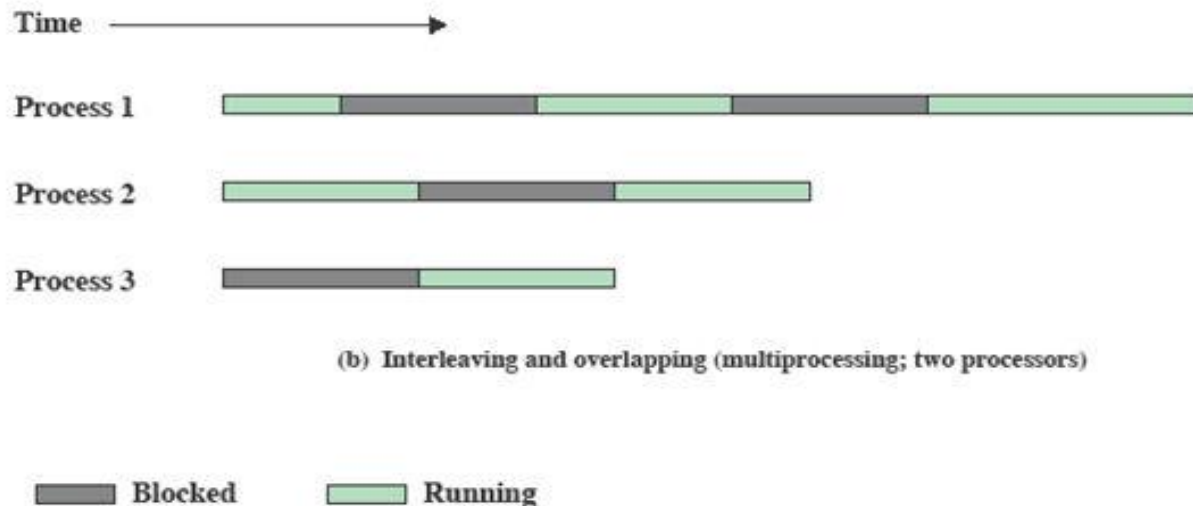


Figure 2.12 Multiprogramming and Multiprocessing

---

# Difficulties of Concurrency

- Sharing of global resources:
    - If two processes both make use of the same global variable and **both perform reads and writes** on that variable, then **the order** in which the various reads and writes are executed is critical.
  - Optimally managing the allocation of resources:
    - It is difficult for the OS to manage the allocation of resources optimally.
    - e.g. A process may request use of, and be granted control of, a particular I/O channel and then be suspended before using that channel.
      - It may be undesirable for the OS simply to lock the channel and prevent its use by other processes;
      - indeed this may lead to a deadlock condition,
  - Difficult to locate programming errors as results are not deterministic and reproducible.
-

---

# Principles of Concurrency

- Suppose we have two or more applications reading input from the keyboard and put the result on the screen.

***function\_test*** ( )

```
{  
Chin=getchar();  
Chout=chin;  
Putchar(chout)  
}
```

A program that will provide a character

- input is obtained from a keyboard one keystroke at a time.
- Each input character is stored in variable chin.
- It is then transferred to variable chout
- and finally sent to the display.

Now consider that we have a single-processor multiprogramming system supporting a single user.

- The user can jump from one application to another, and each application uses the same keyboard for input and the same screen for output.
-

# A Simple Example: On a uniprocessor, multiprogramming

**P1**

Process P1 has called function\_test and have just read (X) the input and being interrupted, so chin = X;

**P2**

Process P2 has called function\_test and allowed to run till end with input "Y",

**P1**

P1 resumed, it will display what ?



the character input to P1 is lost

# Principles of Concurrency

- So the first "chout" is lost, what can we do to prevent such problem (which is called mutual exclusion),

The above example is for uniprocessor system, but similar problem can occur on a Multi-processor platform while sharing of resources.

e.g. accessing the same location (variable) by two or more processes running on different machines.

## ***Process p1***

```
...  
Chin=getchar();  
...  
Chout=chin;  
Putchar(chout)  
..  
..
```

## ***Process P2***

```
...  
..  
Chin=getchar();  
Chout=chin;  
....  
.....  
Putchar( chout)
```

## Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then:
- P1 & P2 run on separate processors
- P1 enters `function_test()` first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes `function_test()`

# Race Condition

- When two/more processes are reading/writing some shared data and the final result depends on who runs precisely when.
- Consider P1 & P2 processes sharing same variable 'a' .
- P3 & P4 have b & c with values  $b=1$  &  $c=2$
- P3 executes the statement  $b=b+c$
- P4 executes the statement  $c=b+c$
- Two processes update different variables & final value depend on the order execution of two processes P3 & P4.
- If P3 executes before P4 then  $b=3, c=5$
- If P4 executes before P3 then  $b=4, c=3$

# Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - the “loser” of the race is the process that updates last and will determine the final value of the variable





# Operating system concerns

- Design and management issues raised by the existence of concurrency:
  - O/S must keep track of all the processes (normally it is done through PCB's)
  - O/S must allocate and deallocate the resources for the processes. Which includes processor time, memory, files and i/o devices.
  - Protection of resources against unintended interference from other processes.
  - The results of a process must be independent of the speed at which the execution is taking place relative to the speed of other concurrent processes.

# Process Interaction

1. Competition among processes: Mutual exclusion, deadlock, starvation
2. Cooperation among processes by sharing: shared variables ,files ,databases
  - Access to data item-read/write
  - Writing must be mutually exclusive
  - Need to maintain data integrity
3. Cooperation among processes by communication: send ,receive message primitives-synchronization
  - No sharing of data/resources b/n processes

# Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - for example: I/O devices, memory, processor time, clock

In the case of competing processes three control problems must be faced:

- ☐ the need for mutual exclusion
- ☐ deadlock
- ☐ starvation



# Process Interaction

**Table 5.2** Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none"><li>• Results of one process independent of the action of others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Mutual exclusion</li><li>• Deadlock (renewable resource)</li><li>• Starvation</li></ul>
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none"><li>• Results of one process may depend on information obtained from others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Mutual exclusion</li><li>• Deadlock (renewable resource)</li><li>• Starvation</li><li>• Data coherence</li></ul>
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none"><li>• Results of one process may depend on information obtained from others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Deadlock (consumable resource)</li><li>• Starvation</li></ul>

Pipes, MQ, shared memory, Signales, semaphorese, sockets

---

multiprogramming of multiple independent processes.

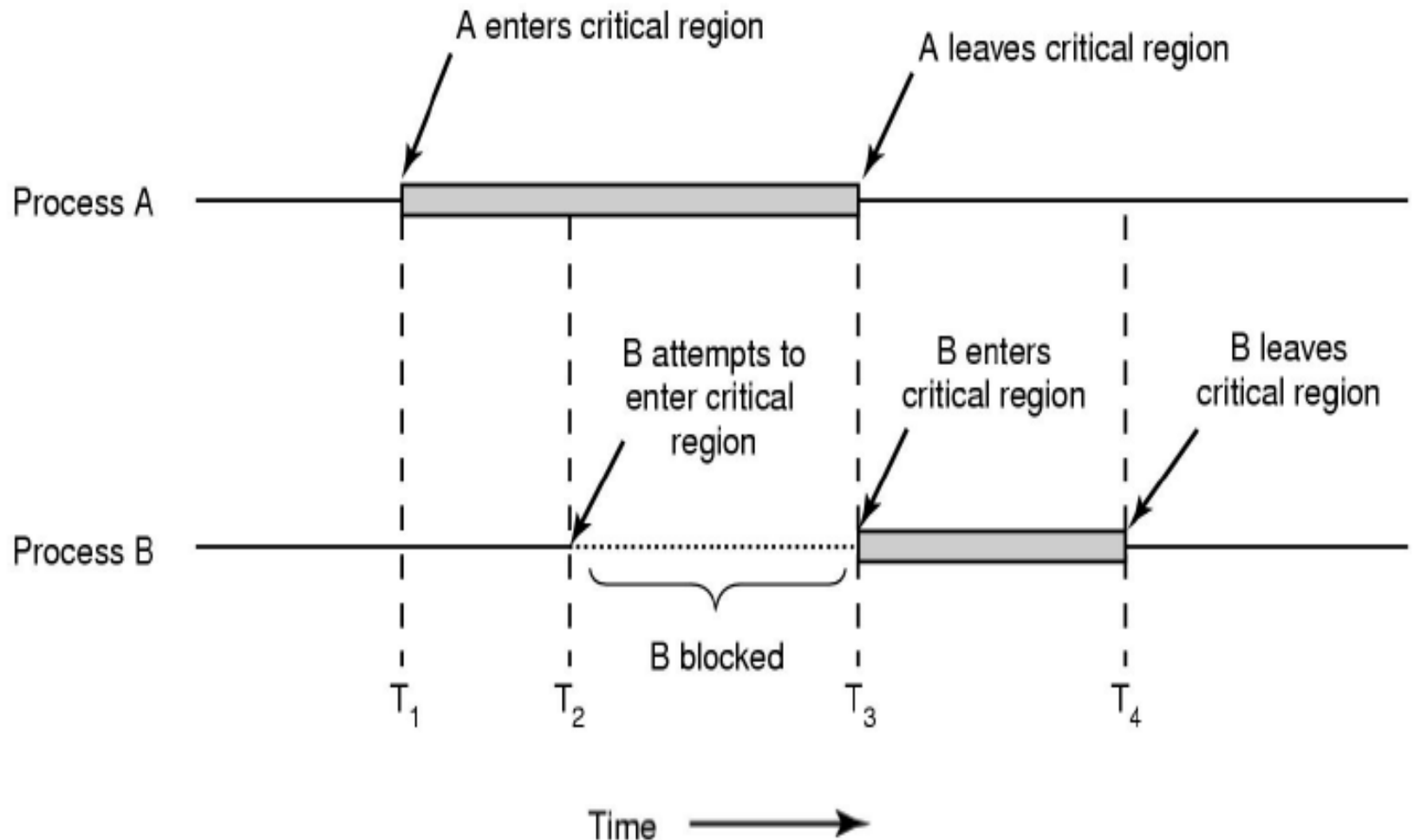
# Conditions for Mutual Exclusion

## **Necessary conditions to provide mutual exclusion**

- 1) No two processes simultaneously in critical region
- 2) No assumptions made about speeds or numbers of CPUs
- 3) No process running outside its critical region may block another process
- 4) No process must wait forever to enter its critical region
- 5) A process remains inside its critical section for a finite time only
- 6) No deadlock or starvation



# How to achieve mutual exclusion



## CONTENTS

- Process Concept, Process States, Process Description
- Processes and Threads
- Symmetric Multiprocessing, Microkernel
- Concurrency: Principles of Concurrency
- Mutual Exclusion: S/W approaches, H/W Support
- ▶ Programming Language construct: Semaphores, Monitors
- Classical Problems of Synchronization: Readers-Writers problem, Producer Consumer problem, Dining Philosopher problem.

<b>Semaphore</b>	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a <b>counting semaphore</b> or a <b>general semaphore</b>
<b>Binary Semaphore</b>	A semaphore that takes on only the values 0 and 1.
<b>Mutex</b>	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
<b>Condition Variable</b>	A data type that is used to block a process or thread until a particular condition is true.
<b>Monitor</b>	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are <i>critical sections</i> . A monitor may have a queue of processes that are waiting to access it.
<b>Event Flags</b>	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
<b>Mailboxes/Messages</b>	A means for two processes to exchange information and that may be used for synchronization.
<b>Spinlocks</b>	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

## Common Concurrency Mechanisms



# Motivation for Semaphores

- Locks only provide mutual exclusion
  - Ensure only one thread is in critical section at a time
- May want more: Place ordering on scheduling of threads
  - Example: Producer/Consumer
    - Producer: Creates a resource (data)
    - Consumer: Uses a resource (data)
  - Example
    - `ps | grep "gcc" | wc`
  - Don't want producers and consumers to operate in lock step
    - Place a fixed-size buffer between producers and consumers
    - Synchronize accesses to buffer
    - Producer waits if buffer full; consumer waits if buffer empty

---

# Semaphore

- Semaphores: Introduced by Dijkstra in 1960s
  - Principle: Two or more processes can cooperate by means of simple signal.
  - Semaphores have two purposes
    - Mutex: Ensure threads don't access critical section at same time
    - Scheduling constraints: Ensure threads execute in specific order
-

# Semaphore Operations

**Initialization:** Initialize a value atomically

**P (or Down or Wait) definition :**

- Atomic operation
- Wait for semaphore to become positive and then decrement

```
P(s){  
while (s <= 0)  
;  
s--;  
}
```

**V (or Up or Signal) definition :**

- Atomic operation
- Increment semaphore by 1

```
V(s){  
s++;  
}
```

Example: **Printer Access in an Office**

# Semaphore as General Synchronization Tool

- **Counting/general** semaphore – integer value can range over an unrestricted domain
- Used to manage access to a finite number of identical resources. Example: Managing access to a pool of printers where multiple threads can use available printers concurrently.
- **Binary** semaphore – integer value can be either 0 or 1; can be simpler to implement
  - Also known as **mutex locks**
  - Used for mutual exclusion where only one process or thread can access the critical section at a time. Example: Ensuring only one thread can access a critical section.

## Semaphore as General Synchronization Tool

- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);
    - Critical Section
    - signal (S);

# Semaphore Primitives

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives

# Binary Semaphore Primitives

```
struct binary_semaphore {  
    enum {zero, one} value;  
    queueType queue;  
};  
void semWaitB(binary_semaphore s)  
{  
    if (s.value == one)  
        s.value = zero;  
    else {  
        /* place this process in s.queue */;  
        /* block this process */;  
    }  
}  
void semSignalB(semaphore s)  
{  
    if (s.queue is empty())  
        s.value = one;  
    else {  
        /* remove a process P from s.queue */;  
        /* place process P on ready list */;  
    }  
}
```

**Figure 5.4 A Definition of Binary Semaphore Primitives**

# Binary Semaphore

- A more restrictive semaphore which may only have the value of 0 or 1
- A similar concept related to the binary semaphore is the **mutex**.
  - A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
  - In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.
  - Example: Mutual Exclusion in a Critical Section



---

# Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore
    - In what order are processes removed from the queue?
  - **Strong Semaphores** uses FIFO-No starvation
  - Strong semaphore guarantee freedom from starvation
  - Example: Producer-Consumer Problem
  - **Weak Semaphores** don't specify the order of removal from the queue
  - Weak semaphore do not guarantee freedom from starvation
  - Example: Simple Resource Allocation Without Fairness
-

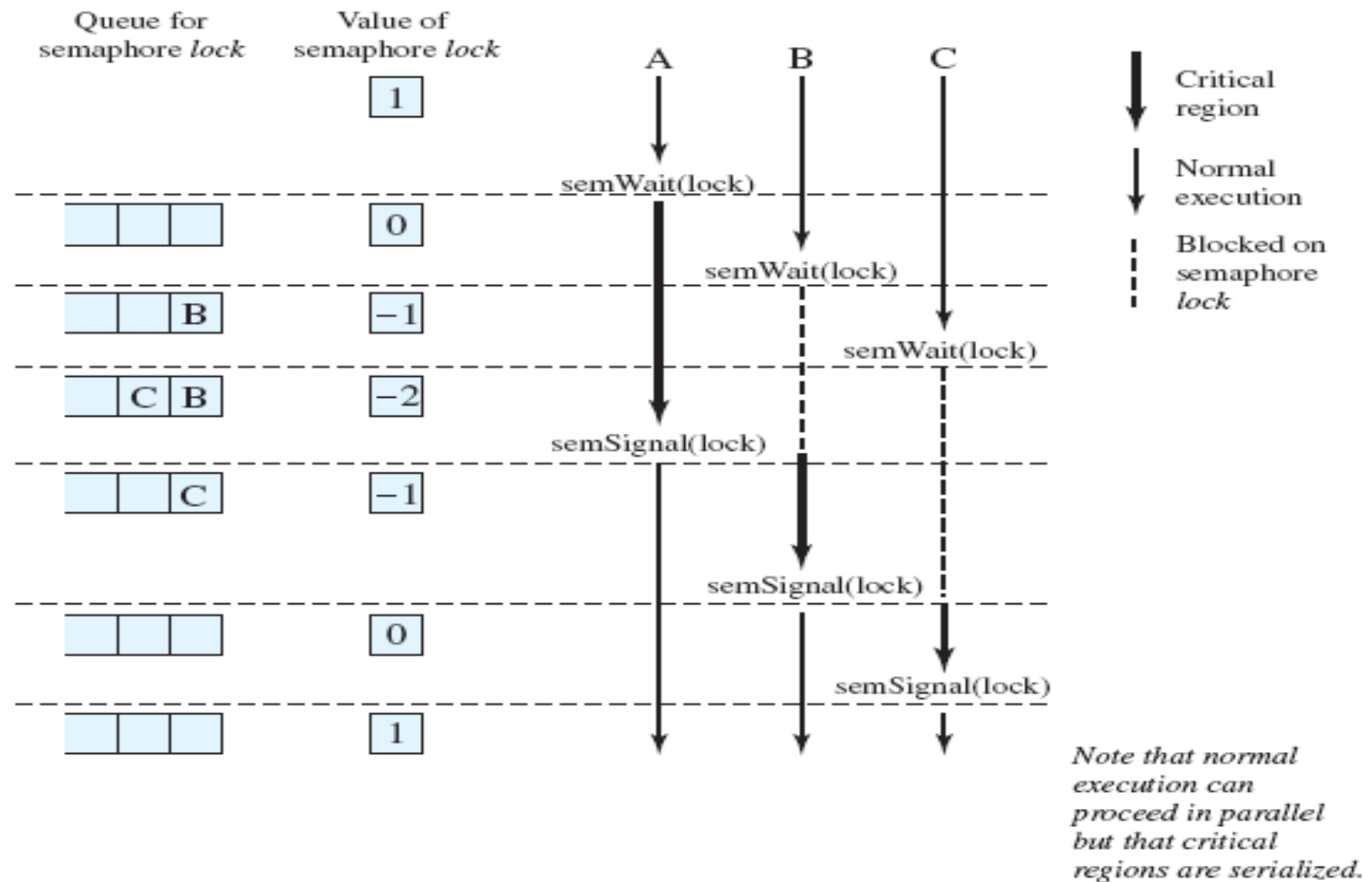
# Mutual Exclusion using Semaphore

- The following pseudo ops can be used for the mutual exclusion problem
- There could be  $n$  processes
- Each process executes wait before entering to its critical section
- Positive value means it can enter into critical section
- If the value becomes negative, process is suspended
- Semaphore  $s = 1$ ;  $n = \text{"number of processes"}$ ;

```
Process ()
{
{
wait (s)
critical section
signal (s)
}
remaining execution
}
```

As the semaphore is initialized to 1, the first process executing wait will enter into its critical section and will set the value of  $s$  to 0. Any other process may attempt to enter, resulting in decrementing  $s$  to a negative value and suspension of that process. Ultimately, the process initially entered into its critical section will depart with incrementing the value of  $s$ , resulting one of the suspended process to be released. Next a possible sequence of events for 3 processes are shown to achieve mutual exclusion.

# Mutual Exclusion using Semaphore



**Figure 5.7** Processes Accessing Shared Data Protected by a Semaphore

# consequences of the semaphore definition

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

---

## Limitations of the semaphore

- While semaphores are useful for managing concurrency and resource access
  - they have limitations related to complexity, deadlocks, priority inversion, fairness, performance overhead, debugging difficulties, resource leakage, and system design constraints.
-

# Mutex(MUTual EXclusion locks)

- These are useful to manage mutual exclusion to some shared resource
- Easy & efficient to implement
- It can be in one of two states: locked /unlocked
- 0 means unlocked & other values means locked.
- Two operation: mutex\_locked & mutex\_unlocked
- When thread needs access to CR it calls mutex\_lock ()
- After working with CR it calls mutex\_unlock()
- Example of lavatory

```
Process ()  
{  
{  
  mutex_lock()  
  critical section  
  mutex_unlock ()  
}  
remaining execution  
}
```

# Producer/Consumer Problem

## General Statement:

one or more producers are generating data and placing these in a buffer

a single consumer is taking items out of the buffer one at a time

only one producer or consumer may access the buffer at any one time

## The Problem:

ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

---

## Producer/Consumer Problem

- **Analogy: Factory and Warehouse**
  - **Factory (Producer):** Produces products (items) and puts them into a warehouse (buffer).
  - **Warehouse (Buffer):** Has a limited capacity for storing products.
  - **Store (Consumer):** Takes products from the warehouse and sells them to customers.
-



# Semaphores: Producer-Consumer problem (Bounded buffer )

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

/\* number of slots in the buffer \*/  
/\* semaphores are a special kind of int \*/  
/\* controls access to critical region \*/  
/\* counts empty buffer slots \*/  
/\* counts full buffer slots \*/

/\* TRUE is the constant 1 \*/  
/\* generate something to put in buffer \*/  
/\* decrement empty count \*/  
/\* enter critical region \*/  
/\* put new item in buffer \*/  
/\* leave critical region \*/  
/\* increment count of full slots \*/

# Semaphores: Producer-Consumer problem

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item( );
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/\* infinite loop \*/  
/\* decrement full count \*/  
/\* enter critical region \*/  
/\* take item from buffer \*/  
/\* leave critical region \*/  
/\* increment count of empty slots \*/  
/\* do something with the item \*/

---

# Readers/Writers Problem

- A data area is shared among many processes
    - some processes only read the data area, (readers) and some only write to the data area (writers)
  - Conditions to satisfy:
    1. Multiple readers may read the file at once.
    2. Only one writer at a time may write
    3. If a writer is writing to the file, no reader may read it.
-

## Readers/Writers Problem

- **Analogy: Library Scenario**
- **Readers:** People reading books in a library. Multiple people can read the same book at the same time.
- **Writers:** Librarians who update the library catalog. Only one librarian can work on updating the catalog at a time, and while they do, no one else (including readers) can access the catalog.

# Semaphore: Reader- writer problem

```
typedef int semaphore;  
semaphore mutex = 1;  
semaphore db = 1;  
int rc = 0;
```

```
/* use your imagination */  
/* controls access to 'rc' */  
/* controls access to the database */  
/* # of processes reading or wanting to */
```

```
void reader(void)  
{
```

```
    while (TRUE) {  
        down(&mutex);  
        rc = rc + 1;  
        if (rc == 1) down(&db);  
        up(&mutex);  
        read_data_base();  
        down(&mutex);  
        rc = rc - 1;  
        if (rc == 0) up(&db);  
        up(&mutex);  
        use_data_read();  
    }
```

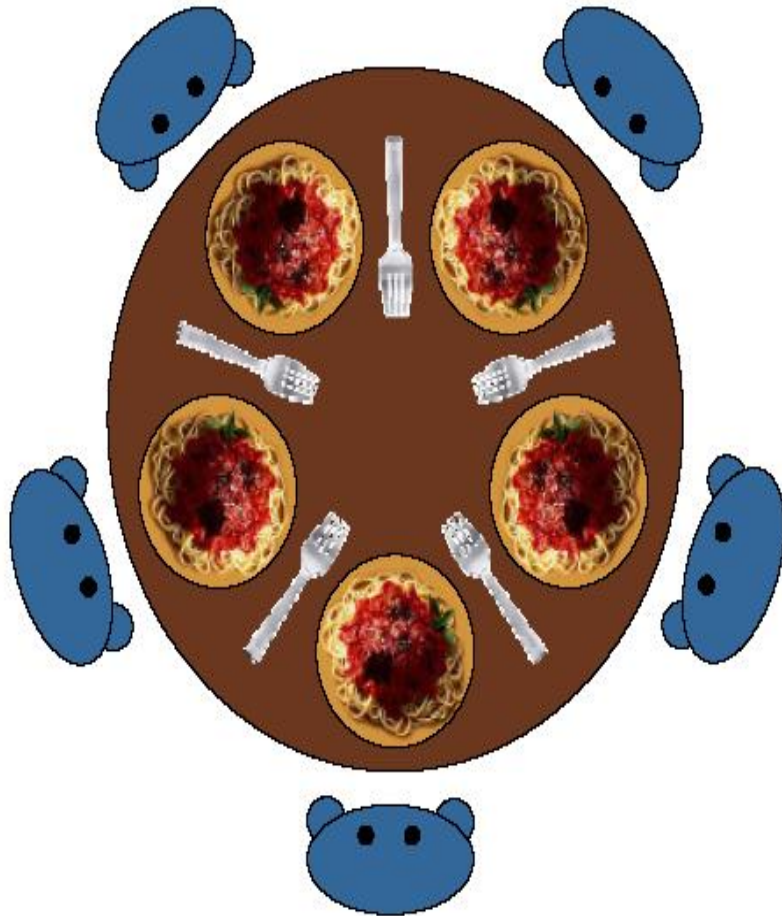
```
/* repeat forever */  
/* get exclusive access to 'rc' */  
/* one reader more now */  
/* if this is the first reader ... */  
/* release exclusive access to 'rc' */  
/* access the data */  
/* get exclusive access to 'rc' */  
/* one reader fewer now */  
/* if this is the last reader ... */  
/* release exclusive access to 'rc' */  
/* noncritical region */
```

```
}  
}
```

# Semaphore: Reader- writer problem

```
void writer(void)
{
    while (TRUE) {                                /* repeat forever */
        think_up_data();                          /* noncritical region */
        down(&db);                               /* get exclusive access */
        write_data_base();                       /* update the data */
        up(&db);                                 /* release exclusive access */
    }
}
```

# Dining philosopher problem



Situation:

- Five philosophers at table
- Want to eat spaghetti
- Must have 2 forks to eat
- Only 5 forks at the table
- Each philosopher:
- Must grab fork one at a time
- Can act at any time (concurrently with other philosophers)
- Is either thinking, hungry, or eating

# Dining philosopher problem

- Everyone follows:
- Grab left fork
- Grab right fork
- Eat spaghetti
- If fork not available, wait until available
- Release fork when done eating
- What if all grab left fork at same time?



DEADLOCK



# Dining philosopher problem

- One person follows:
  - Grab right fork
  - Grab left fork
  - Eat spaghetti
- Everyone else follows:
  - Grab left fork
  - Grab right fork
  - Eat spaghetti
- Avoids deadlock!

# Semaphore: Dining philosopher problem

```
#define N                5                /* number of philosophers */
#define LEFT             (i+N-1)%N        /* number of i's left neighbor */
#define RIGHT            (i+1)%N          /* number of i's right neighbor */
#define THINKING         0                /* philosopher is thinking */
#define HUNGRY           1                /* philosopher is trying to get forks */
#define EATING           2                /* philosopher is eating */
typedef int semaphore;    /* semaphores are a special kind of int */
int state[N];             /* array to keep track of everyone's state */
semaphore mutex = 1;      /* mutual exclusion for critical regions */
semaphore s[N];           /* one semaphore per philosopher */

void philosopher(int i)    /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {         /* repeat forever */
        think();           /* philosopher is thinking */
        take_forks(i);     /* acquire two forks or block */
        eat();             /* yum-yum, spaghetti */
        put_forks(i);      /* put both forks back on table */
    }
}
```

---

## Semaphore: Dining philosopher problem

```
void take_forks(int i)                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                      /* enter critical region */
    state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
    test(i);                          /* try to acquire 2 forks */
    up(&mutex);                        /* exit critical region */
    down(&s[i]);                       /* block if forks were not acquired */
}
```

---

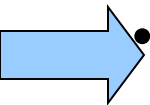
# Semaphore: Dining philosopher problem

```
void put_forks(i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                /* enter critical region */
    state[i] = THINKING;                         /* philosopher has finished eating */
    test(LEFT);                                  /* see if left neighbor can now eat */
    test(RIGHT);                                 /* see if right neighbor can now eat */
    up(&mutex);                                  /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

# CONTENTS

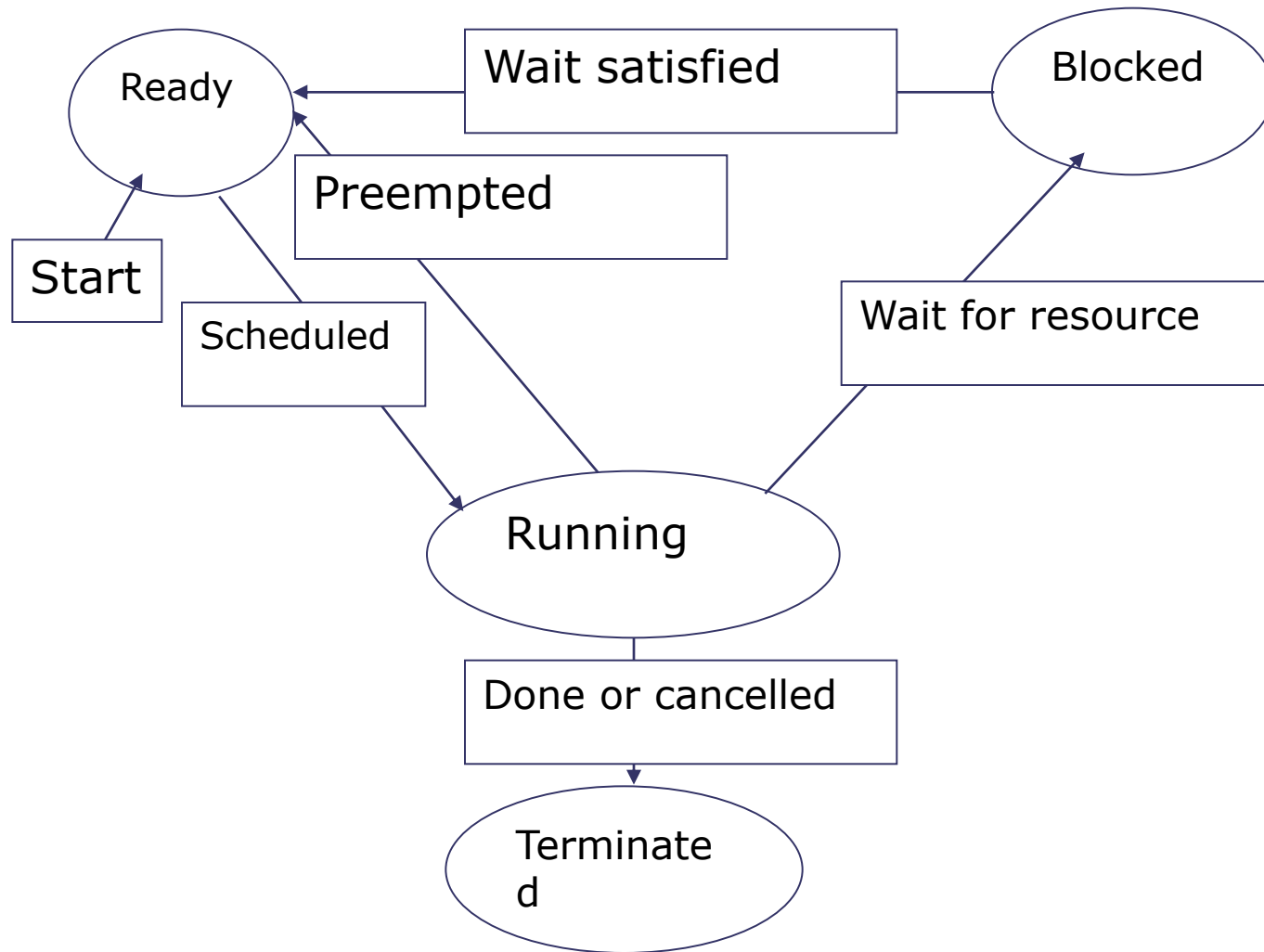
- Process Concept, Process States, Process Description
- Processes and Threads
- Symmetric Multiprocessing, Microkernel
- Concurrency: Principles of Concurrency
- Mutual Exclusion: S/W approaches, H/W Support
- Programming Language construct: Semaphores, Monitors
- Classical Problems of Synchronization: Readers-Writers problem, Producer Consumer problem, Dining Philosopher problem.
- Different useful functions



## Commonly used pThread API's

- `pthread_create( )`
  - `pthread_detach( )`
  - `pthread_equal( )`
  - `pthread_exit( )`
  - `pthread_join( )`
  - `pthread_self( )`
  - `sched_yield( )`
  - `pthread_cancel()`
- `pthread_mutex_init()`
  - `pthread_mutex_destroy()`
  - `pthread_mutex_lock()`
  - `pthread_mutex_trylock()`
  - `pthread_mutex_unlock()`

# Thread State Transitions



---

```
int pthread_create (  
pthread_t *tid,          // Thread ID returned by the system  
const pthread_attr_t *attr, // optional creation attributes  
void *(*start)(void *), // start function of the new thread  
void *arg                // Arguments to start function  
  
);
```

Description: Create a thread running the start function.

```
pthread_t pthread_self(void);
```

It return the thread ID of the calling thread.

This is the same value that is returned in \*thread in the pthread\_create() call that created this thread.

---



---

```
int pthread_exit(  
void *value_ptr,      // Return value.  
);
```

Description: Terminate the calling thread, returning the value value\_ptr to any joining thread.

```
int pthread_join(  
pthread_t      thread,      // ID of thread  
void           **value_ptr   // return value of thread  
);
```

Description: Wait for thread to terminate, and return thread's exit value if value\_ptr is not NULL. This also detaches thread on successful completion.

---

# Pthread Mutex

## States

### Locked

- Some thread holds the mutex

### Unlocked

- No thread holds the mutex

When several threads compete

One wins

The rest block

- Queue of blocked threads

## Synchronization(Mutexes)

A typical sequence in the use of a mutex

1. Create and initialize **mutex**
2. Several threads attempt to lock **mutex**
3. Only one succeeds and now owns **mutex**
4. The owner performs some set of actions
5. The owner unlocks **mutex**
6. Another thread acquires **mutex** and repeats the process
7. Finally **mutex** is destroyed

## Synchronization(Mutexes)

- `pthread_mutex_init()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `pthread_mutex_trylock()`
- `pthread_mutex_unlock()`

# Synchronization(Mutexes)

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
    const pthread_mutexattr_t *attr);
```

Initialize a pthread mutex: the mutex is initially unlocked

Returns

- 0 on success
- Error number on failure
  - **EAGAIN**: The system lacked the necessary resources; **ENOMEM**: Insufficient memory ; **EPERM**: Caller does not have privileges; **EBUSY**: An attempt to re-initialise a mutex; **EINVAL**: The value specified by attr is invalid

Parameters

- **mutex**: Target mutex
- **attr**:
  - NULL: the default mutex attributes are used
  - Non-NULL: initializes with specified attributes

```
#include <pthread.h>
```

```
int pthread_mutex_destroy (pthread_mutex_t  
    *mutex) ;
```

Destroy a pthread mutex

Returns

- 0 on success
- Error number on failure
  - **EBUSY**: An attempt to re-initialise a mutex; **EINVAL**: The value specified by attr is invalid

Parameters

- **mutex**: Target mutex

```
int pthread_mutex_lock (  
pthread_mutex_t *mutex );
```

Description: Lock a mutex.

If the mutex is currently locked, the calling thread is blocked until mutex is unlocked. On return, the thread owns the mutex until it calls pthread\_mutex\_unlock.

Parameters

**mutex**: Target mutex

```
int pthread_mutex_unlock (  
pthread_mutex_t *mutex );
```

Description: UnLock a mutex. The mutex becomes unwoned. If any threads are waiting for the mutex, one is awakened(scheduling policy SCHED\_FIFO and SCHED\_RR policy waiters are chosen in priority order, then any others are chosen in unspecified order.

Parameters

**mutex**: Target mutex

# POSIX Semaphores

- Data type

Semaphore is a variable of type `sem_t`

- Include `<semaphore.h>`

- Atomic Operations

```
int sem_init(sem_t *sem, int pshared, unsigned value);
```

```
int sem_destroy(sem_t *sem);
```

```
int sem_post(sem_t *sem);
```

```
int sem_trywait(sem_t *sem);
```

```
int sem_wait(sem_t *sem);
```



```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared, unsigned  
value);
```

Initialize an unnamed semaphore

Returns

0 on success

-1 on failure, sets **errno**

Parameters

**sem:**

- Target semaphore

**pshared:**

- 0: only threads of the creating process can use the semaphore
- Non-0: other processes can use the semaphore

**value:**

- Initial value of the semaphore

---

```
#include <semaphore.h>
int sem_destroy(sem_t *sem) ;
```

Destroy an semaphore

Returns

0 on success

-1 on failure, sets `errno`

Parameters

`sem`:

- Target semaphore

Notes

Can destroy a `sem_t` only once

Destroying a destroyed semaphore gives undefined results

Destroying a semaphore on which a thread is blocked gives undefined results

---

---

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

Unlock a semaphore - same as signal

Returns

0 on success

-1 on failure, sets `errno` (`== EINVAL` if semaphore doesn't exist)

Parameters

`sem`:

- Target semaphore
  - `sem > 0`: no threads were blocked on this semaphore, the semaphore value is incremented
  - `sem == 0`: one blocked thread will be allowed to run
-

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);
```

Lock a semaphore

Blocks if semaphore value is zero

Returns

0 on success

-1 on failure, sets `errno` (`== EINTR` if interrupted by a signal)

Parameters

`sem`:

- Target semaphore
- `sem > 0`: thread acquires lock
- `sem == 0`: thread blocks

---

```
#include <semaphore.h>
```

```
int sem_getvalue(sem_t *sem, int *sval);
```

places the current value of the semaphore pointed to `sem` into the integer pointed to by `sval`.

---