

CS343 - Operating Systems

Module-2A

Introduction to Process Concept & Process States



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Session Outline

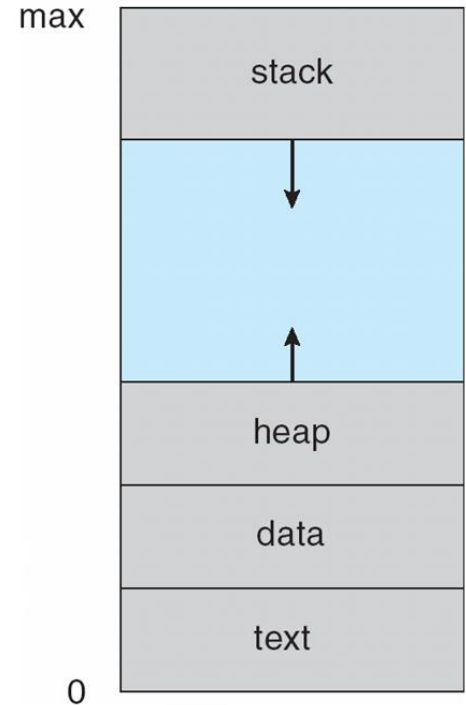
- ❖ **Process Concept**
- ❖ **Process State Diagram**
- ❖ **Process Control Block**
- ❖ **Context Switching between Processes**
- ❖ **Process Scheduling**
- ❖ **Long Term Vs Short Term Scheduler**

Process Concept

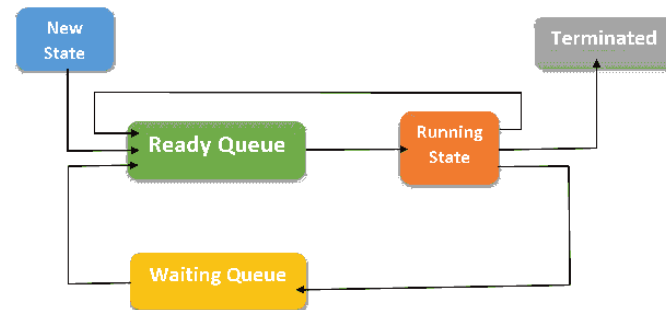
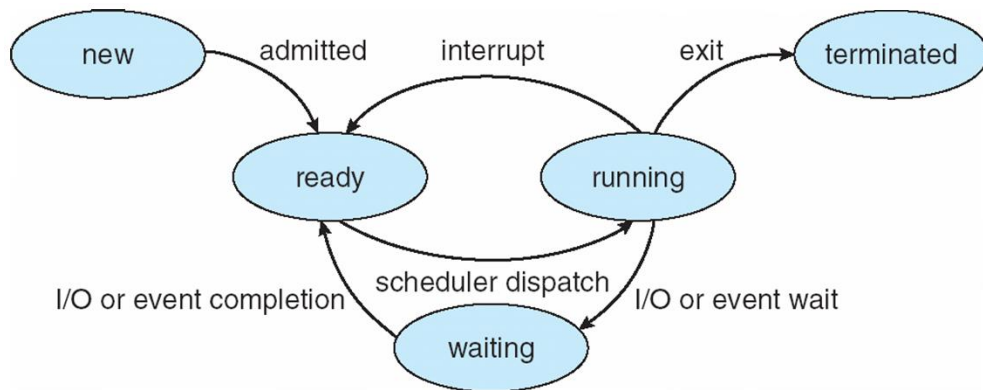
- ❖ A process is a **program in execution**
- ❖ It is a unit of work within the system
- ❖ Program is a **passive entity**, process is an **active entity**
- ❖ Process needs resources to accomplish its task
- ❖ These resources include **CPU, memory, I/O, files**, etc.
- ❖ Program becomes process when executable file loaded into memory
- ❖ Program execution is initiated by GUI mouse clicks / command line entry

Process Concept

- ❖ One program can have several processes
- ❖ **Process** has multiple parts
 - ❖ The program code, also called **text section**
 - ❖ Current activity - **program counter**, registers
 - ❖ **Stack** containing temporary data like function parameters, return addresses, local variables
 - ❖ **Data section** containing global variables
 - ❖ **Heap** -dynamically allocated memory during run time



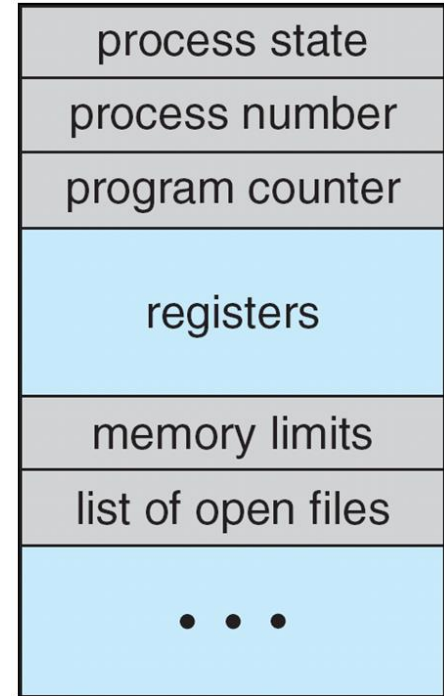
Process State Diagram



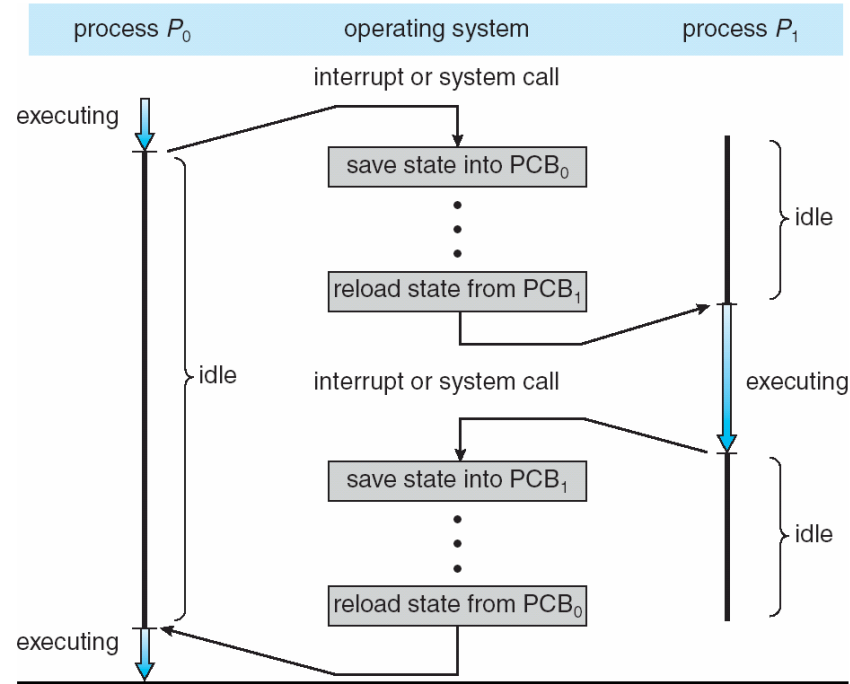
- ❖ **new**: The process is being created
- ❖ **running**: Instructions are being executed
- ❖ **waiting**: The process is waiting for some event to occur
- ❖ **ready**: The process is waiting to processor assignment.
- ❖ **terminated**: The process has finished execution

Process Control Block (PCB)

- ❖ Process state – running, waiting, etc
- ❖ Program counter – location of instruction to next execute
- ❖ CPU registers – contents of all process-centric registers
- ❖ CPU scheduling information- priorities, scheduling queue pointers
- ❖ Memory-management information – memory allocated to the process
- ❖ Accounting information – CPU used, clock time elapsed since start, time limits
- ❖ I/O status information – I/O devices allocated to process, list of open files



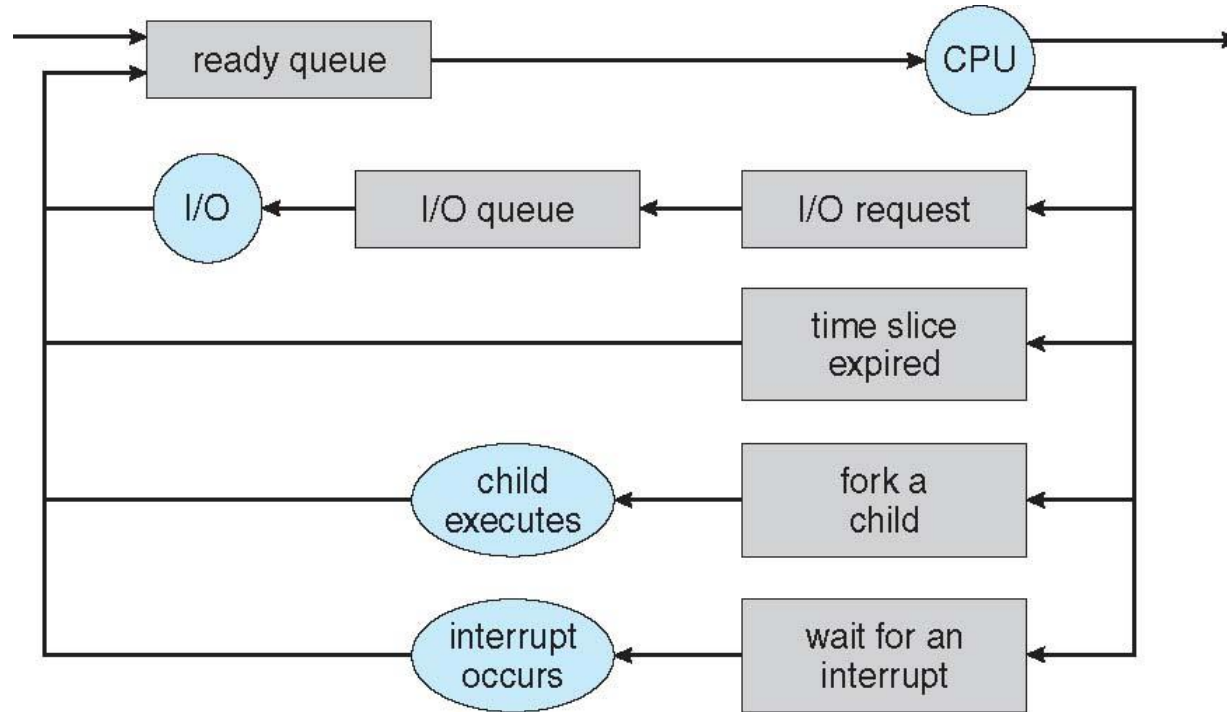
Context Switch From One Process to Another



Process Scheduling

- ❖ Maximize CPU use, quickly switch processes onto CPU for time sharing
- ❖ **Process scheduler** selects among available processes for next execution on CPU
- ❖ Maintains **scheduling queues** of processes
 - ❖ **Job queue** – set of all processes in the system
 - ❖ **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - ❖ **Device queues** – set of processes waiting for an I/O device
 - ❖ Processes migrate among the various queues

Representation of Process Scheduling



Schedulers

- ❖ **Short-term scheduler (CPU scheduler)** – selects which process should be assigned to CPU for execution
 - ❖ Short-term scheduler is invoked frequently
- ❖ **Long-term scheduler (Job scheduler)** – selects which processes should be brought into the ready queue (RAM)
 - ❖ Long-term scheduler is invoked less frequently
 - ❖ It controls the **degree of multiprogramming**
 - ❖ **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - ❖ **CPU-bound process** – spends more time doing computations; few very long CPU bursts
 - ❖ Long-term scheduler strives for good ***process mix***

Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-2B

Introduction to Process Scheduling



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

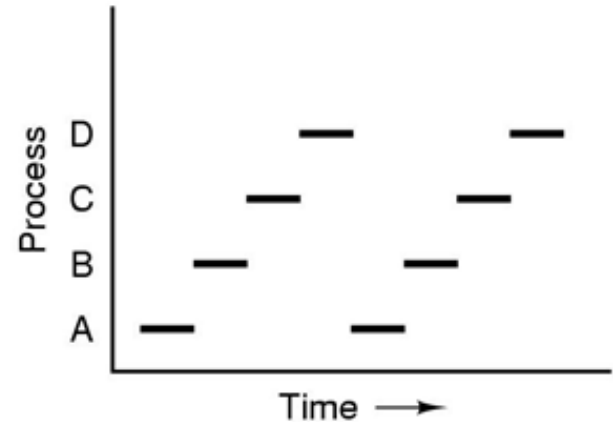
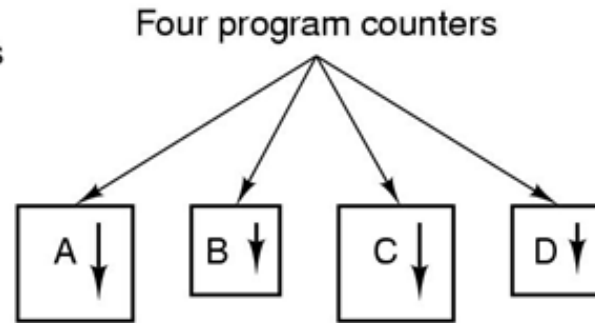
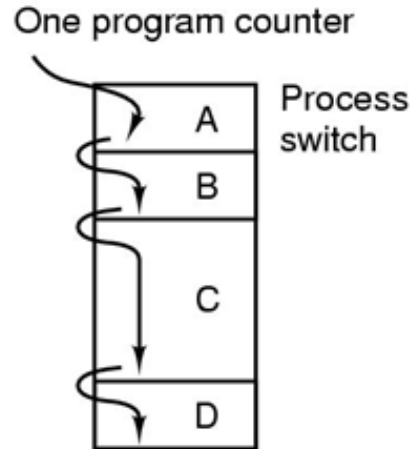
<http://www.iitg.ac.in/johnjose/>

Session Outline

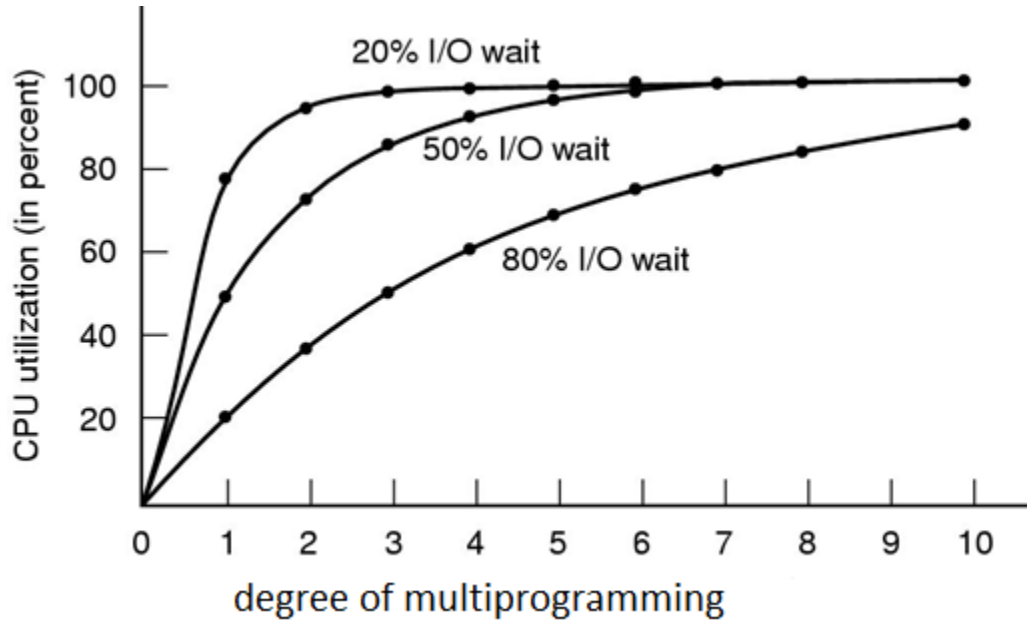
- ❖ **Concept of Multiprogramming**
- ❖ **CPU vs I/O bound Processes**
- ❖ **CPU vs Job Schedulers**
- ❖ **CPU Scheduling**
- ❖ **Dispatcher**
- ❖ **Preemptive vs Non-preemptive Scheduling**
- ❖ **Scheduling Criteria**

Multiprogramming

- ❖ In a uni-processor system, CPU is running only one process.
- ❖ In a multiprogramming system, CPU switches from processes quickly.

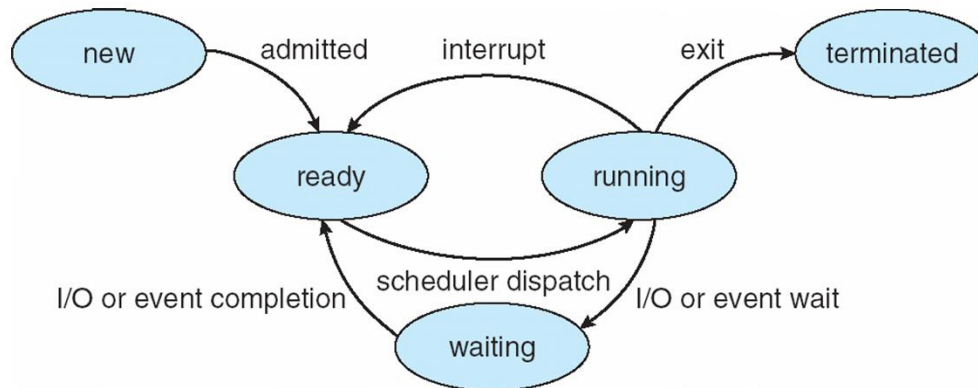


Modeling Multiprogramming



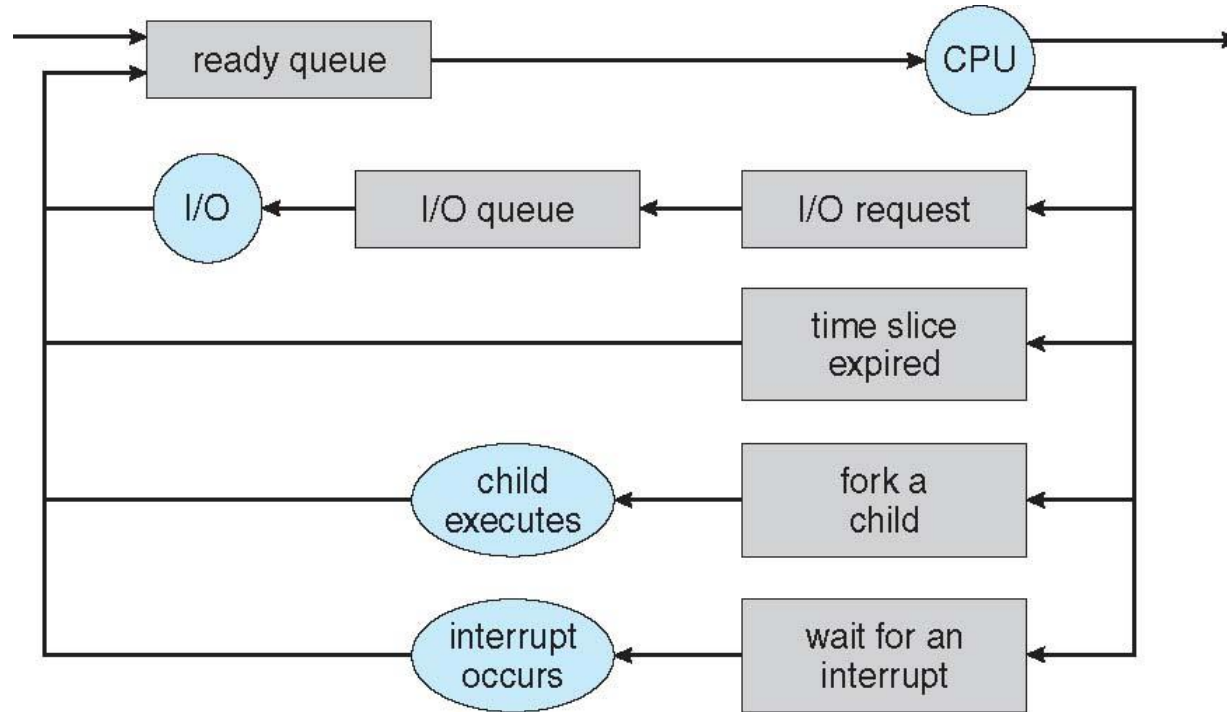
CPU utilization as a function of the number of processes in memory.

Process State Diagram

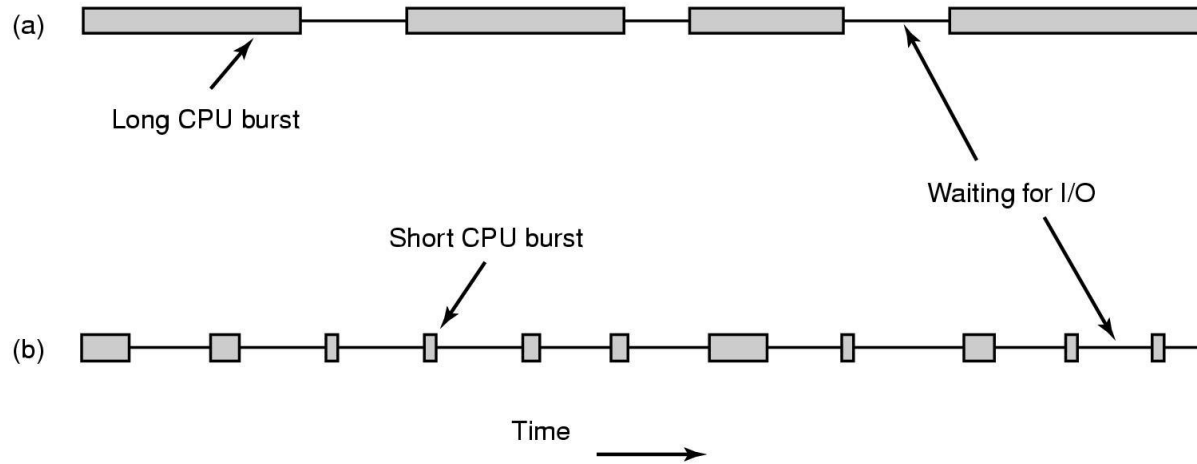


- ❖ **new**: The process is being created
- ❖ **running**: Instructions are being executed
- ❖ **waiting**: The process is waiting for some event to occur
- ❖ **ready**: The process is waiting to processor assignment
- ❖ **terminated**: The process has finished execution

Representation of Process Scheduling



CPU vs I/O Bound Processes

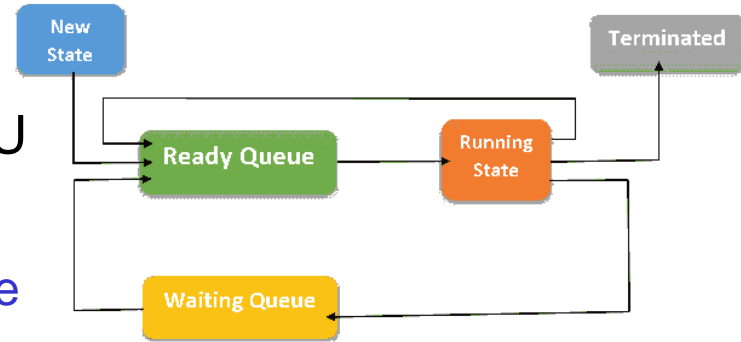


Bursts of CPU usage alternate with periods of waiting for I/O.
(a) A CPU-bound process. (b) An I/O-bound process.

CPU vs Job Schedulers

- ❖ **Short-term scheduler (CPU scheduler)** selects from among the processes that are ready to execute and allocates the CPU to one of them.

- ❖ Selection from **Ready state** to **Running state**
- ❖ Short-term scheduler is invoked frequently



- ❖ **Long-term scheduler (Job scheduler)** – selects which processes should be brought into the ready queue from the job queue
- ❖ Long-term scheduler is invoked less frequently
- ❖ It controls the **degree of multiprogramming**

CPU Scheduler

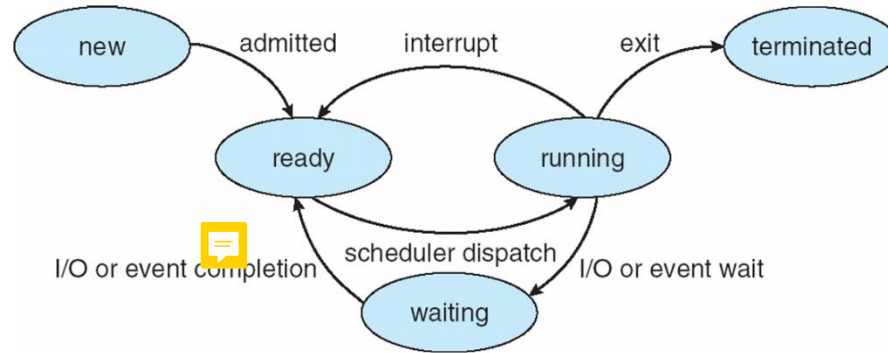
- ❖ Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.
- ❖ The selection process is carried out by the **CPU scheduler** of the OS.
- ❖ The scheduler selects a process that is in ready state and allocates the CPU to that process.
- ❖ Ready queue can be implemented as
 - ❖ FIFO queue
 - ❖ Priority queue
 - ❖ Tree
 - ❖ Unordered linked list

Dispatcher

- ❖ The **dispatcher** is the module that gives control of the CPU to the process selected by the CPU scheduler.
- ❖ Function of dispatcher involves the following:
 - ❖ Switching context
 - ❖ Switching to user mode
 - ❖ Initiate restarting/ resumption of selected user program
- ❖ Dispatcher should be as fast as possible, since it is invoked during every process switch.
- ❖ The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Preemptive vs Non-preemptive Scheduling

❖ CPU-scheduling decisions happen during the four circumstances:



1. When a process switches from the running state to the waiting state
2. When a process switches from the running state to the ready state
3. When a process switches from the waiting state to the ready state
4. When a process terminates

Preemptive vs Non-preemptive Scheduling

- ❖ CPU-scheduling decisions happen during the four circumstances:
 1. When a process switches from the running state to the waiting state
 2. When a process switches from the running state to the ready state
 3. When a process switches from the waiting state to the ready state
 4. When a process terminates
- ❖ 1 & 4, the scheduling is **non-preemptive** (cooperative)
- ❖ A running process keeps the CPU until it releases the CPU (terminating or by switching to the waiting state).
- ❖ 2 & 3, the scheduling is **pre-emptive**
- ❖ The process is removed from running state forcefully.

Scheduling Criteria

- ❖ Different CPU-scheduling algorithms have different properties.
- ❖ Certain characteristics/criteria are used for comparing various CPU scheduling algorithms.
 - ❖ CPU Utilization
 - ❖ Throughput
 - ❖ Turnaround time
 - ❖ Waiting Time
 - ❖ Response Time

Scheduling Criteria

- ❖ **CPU Utilization** – Percentage time CPU is busy executing process. ↑
- ❖ **Throughput** - Number of processes that are completed per time unit. ↑
- ❖ **Turnaround time** - The interval from the time of submission of a process to the time of completion. ↓
- ❖ **Waiting Time** - Amount of time that a process spends waiting in the ready queue. ↓
- ❖ **Response Time** - Time from the submission of a request until the first response is produced. ↓

Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-2C

CPU Scheduling Algorithms - 1



Dr. John Jose



Assistant Professor

Department of Computer Science & Engineering

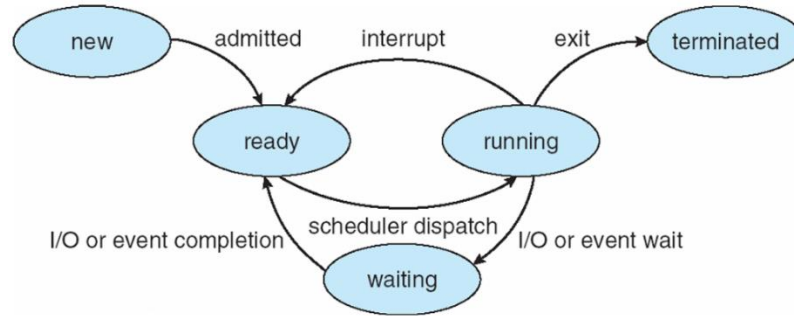
Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Session Outline

- ❖ CPU scheduling
- ❖ Categories of scheduling algorithms
- ❖ FCFS scheduling algorithm
- ❖ SJF scheduling algorithm 
- ❖ SRTF scheduling algorithm 
- ❖ Round Robin scheduling algorithm
- ❖ Priority scheduling algorithm

Preemptive vs Non-preemptive Scheduling





1. When a process switches from the running state to the waiting state
 2. When a process switches from the running state to the ready state
 3. When a process switches from the waiting state to the ready state
 4. When a process terminates
- ❖ 1 & 4, the scheduling is **non-preemptive** (cooperative)
 - ❖ 2 & 3, the scheduling is **preemptive**

Scheduling Criteria



- ❖ Different CPU-scheduling algorithms have different properties.
- ❖ Certain characteristics/criteria are used for comparing various CPU scheduling algorithms.
 - ❖ CPU Utilization
 - ❖ Throughput
 - ❖ Turnaround time
 - ❖ Waiting Time
 - ❖ Response Time

Categories of Scheduling Algorithms & Goals

❖ Batch System

- ❖ Complete maximum number of jobs per unit time. 
- ❖ Minimize time between submission and termination 
- ❖ Keep CPU busy all time

❖ Interactive System

- ❖ Response to requests quickly 
- ❖ Reduce waiting time 
- ❖ Meet user expectations

❖ Real time System

- ❖ Meeting deadlines
- ❖ Ensure quality constraints

CPU Scheduling Algorithms


Batch Systems

- ❖ First-come first-served
- ❖ Shortest job first
- ❖ Shortest remaining Time next

Interactive Systems

- ❖ Round-robin scheduling
- ❖ Priority scheduling
- ❖ Multiple queues
- ❖ Shortest process next
- ❖ Guaranteed scheduling
- ❖ Lottery scheduling
- ❖ Fair-share scheduling

FCFS Scheduling

- ❖ Simplest CPU-scheduling algorithm
- ❖ First-come, first-served - process that requests the CPU first is allocated the CPU first
- ❖ FCFS policy is managed with a FIFO queue 
- ❖ When a process enters the ready queue, its PCB is linked onto the tail of the FIFO queue
- ❖ When the CPU is free, it is allocated to process at the head of the queue
- ❖ The running process is then removed from the queue
- ❖ It is **non-preemptive**, once scheduled it will complete
- ❖ Short jobs wait for long

FCFS Scheduling

❖ Example: Three processes arrive in order P1, P2, P3 all at time 0.

❖ P1 burst time: 24

❖ P2 burst time: 3

❖ P3 burst time: 9



❖ Waiting Time

❖ P1: 0, P2: 24, P3: 27


❖ Completion Time:

❖ P1: 24, 2: 27, P3: 36

❖ Average Waiting Time: $(0+24+27)/3 = 17$

❖ Average Completion Time: $(24+27+36)/3 = 29$

SJF Scheduling

- ❖ Shortest (in terms of CPU time) job available is scheduled first
- ❖ Shorter processes makes progress
- ❖ SJF policy is managed with a priority queue with burst time as input
- ❖ When a process enters the ready queue, its PCB is linked onto the priority queue at the appropriate entry
- ❖ When the CPU is free, it is allocated to the process at the head of the priority queue
- ❖ If too many short jobs, long processes will starve
- ❖ SJF is non-preemptive; once allotted the process will complete
- ❖ Lowest turnaround time 

SJF Scheduling

❖ Consider 3 process P2, P3, P1 all arriving at time T0.

❖ P1 burst time: 24

❖ P2 burst time: 3

❖ P3 burst time: 9



❖ Waiting Time

❖ P1: 12, P2: 0, P3: 3

❖ Completion Time:

❖ P1: 36, P2: 3, P3: 12

❖ Average Waiting Time: $(12+0+3)/3 = 5$ (compared to 17)

❖ Average Completion Time: $(36+3+12)/3 = 17$ (compared to 29)



SRTF Scheduling

- ❖ Shortest Remaining Time First (SRTF) job is scheduled first
- ❖ **Preemptive** scheduling algorithm
- ❖ A priority queue with remaining time is used as input
- ❖ When a process enters/re-enters the ready queue, its PCB is linked onto the priority queue at the appropriate entry
- ❖ When the CPU is free, it is allocated to the process at the head of the priority queue
- ❖ Newly arriving short process may forcefully preempt currently running process.
- ❖ Longer process may have multiple context switch before completion

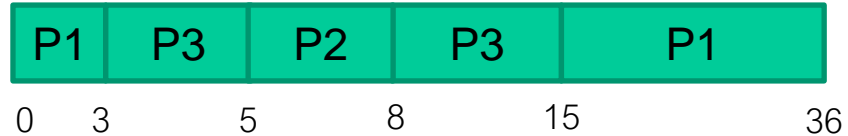
SRTF Scheduling

❖ Consider the following process arriving at different time slots

❖ P1 burst time: 24 arrives at 0

❖ P2 burst time: 3 arrives at 5

❖ P3 burst time: 9 arrives at 3



❖ Waiting Time

❖ P1: $(15-3) = 12$, P2: 0, P3: $(8-5) = 3$

❖ Completion Time:

❖ P1: 36, P2: 8, P3: 15

❖ Average Waiting Time: $(0+3+12)/3 = 5$

❖ Average Completion Time: $(8+15+36)/3 = 19.6$

Round Robin Scheduling

- ❖ Modified version of preemptive FCFS
- ❖ Each process gets a small unit of CPU time (time quantum)
- ❖ FIFO queue is used as input
- ❖ When a process enters/re-enters the ready queue, its PCB is linked the tail of the queue
- ❖ After quantum expires, the process is preempted and added to the tail of the ready queue (Hence, preemptive scheduling algorithm)
- ❖ CPU is allocated to the process at the head of the queue
- ❖ Longer process may have multiple context switch before completion

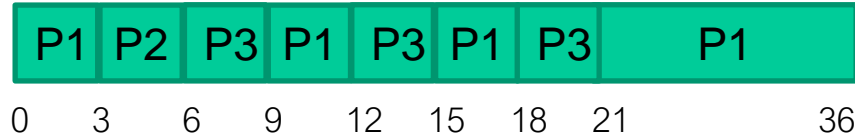
Round Robin Scheduling

❖ Consider the following process arriving at T_0 , time quantum of 3 units

❖ P1 burst time: 24

❖ P2 burst time: 3

❖ P3 burst time: 9



❖ Waiting Time

❖ P1: $(6+3+3) = 12$, P2: 0, P3: $(6+3+3) = 12$

❖ Completion Time:

❖ P1: 36, P2: 3, P3: 21

❖ Average Waiting Time: $(0+12+12)/3 = 8$

❖ Average Completion Time: $(3+21+36)/3 = 20$

Round Robin Scheduling

- ❖ RR scheduling is better for short jobs and fair
- ❖ Shorter response time, good for interactive jobs
- ❖ Context-switching time adds up for long jobs
- ❖ Context switching takes additional time and overhead
- ❖ If the chosen quantum is
 - ❖ too large, response time suffers
 - ❖ infinite, performance is the same as FIFO
 - ❖ too small, throughput suffers and percentage overhead grows

Priority Scheduling

- ❖ Each process has a priority number
- ❖ Highest priority process is scheduled first; if equal priorities, then FCFS
- ❖ Managed with a priority queue with priority value as input
- ❖ When a process enters the ready queue, its PCB is linked onto the priority queue at the appropriate entry
- ❖ CPU is allocated to the process at the head of the queue
- ❖ It can have 2 variants; non-preemptive and preemptive
- ❖ Arrival of a new process with a higher priority can preempt the currently running process.

Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-2D

CPU Scheduling Algorithms - 2



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Session Outline

- ❖ **CPU Scheduling**
- ❖ **Limitations of Priority Scheduling**
- ❖ **Priority Inversion**
- ❖ **Multilevel Feedback Queue Scheduling**
- ❖ **Lottery Scheduling**
- ❖ **Summary of Scheduling algorithms**

Scheduling Criteria

- ❖ Different CPU-scheduling algorithms have different properties.
- ❖ Certain characteristics/criteria are used for comparing various CPU scheduling algorithms.
 - ❖ CPU Utilization
 - ❖ Throughput
 - ❖ Turnaround time
 - ❖ Waiting Time
 - ❖ Response Time

CPU Scheduling Algorithms

Batch Systems

- ❖ First-come first-served
- ❖ Shortest job first
- ❖ Shortest remaining Time next

Interactive Systems

- ❖ Round-robin scheduling
- ❖ Priority scheduling
- ❖ Multiple queues
- ❖ Shortest process next
- ❖ Guaranteed scheduling
- ❖ Lottery scheduling
- ❖ Fair-share scheduling

Priority Scheduling

- ❖ Each process has a priority number (integer)
- ❖ Lower the integer, higher the priority
- ❖ Highest priority process is scheduled first; if equal priorities, then FCFS
- ❖ It can have 2 variants; non-preemptive and preemptive
- ❖ Arrival of a new process with a higher priority can preempt the currently running process.

Issues with Priority Scheduling

- ❖ Consider a scenario in which there are three processes, a high priority (H), a medium priority (M), and a low priority (L).
- ❖ Process L is running and successfully acquires a resource file.
- ❖ Process H begins; since we are using a preemptive priority scheduler, process L is preempted for process H.
- ❖ Process H tries to acquire L's resource, and blocks (held by L).
- ❖ Process M begins running, and, since it has a higher priority than L, it is the highest priority ready process. It preempts L and runs, thus starving high priority process H.
- ❖ This is known as priority inversion. What can we do?



Priority Inversion

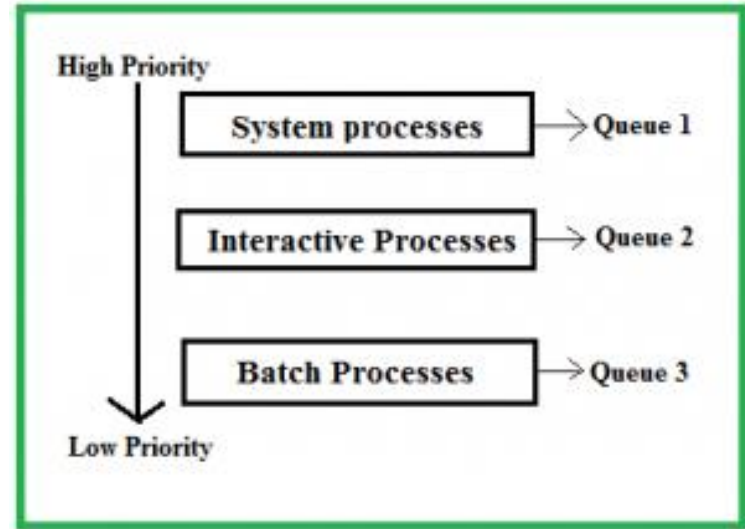
- ❖ Process L should, in fact, be temporarily of higher priority than process M, on behalf of process H.
- ❖ Process H can donate its priority to process L, which, in this case, would make it higher priority than process M.
- ❖ This enables process L to preempt process M and run.
- ❖ When process L is finished, process H becomes unblocked.
- ❖ Process H, now being the highest priority ready process, runs, and process M must wait until it is finished.

Multilevel Queue

- ❖ Ready queue is partitioned into separate queues:
 - ❖ Foreground, interactive process → RR scheduling
 - ❖ Background, batch process → FCFS scheduling
- ❖ A process is permanently assigned to one queue
- ❖ Each queue has its own scheduling algorithm
- ❖ Can be preemptive

Multilevel Feedback Queue Scheduling

- ❖ Scheduling must be done between the queues.
- ❖ Fixed priority scheduling 
- ❖ Serve all from foreground then from background
- ❖ Possibility of starvation
- ❖ Time slice
- ❖ Each queue gets a certain amount of CPU time which it can schedule among its processes
 - ❖ i.e.: 80% Vs 20% 



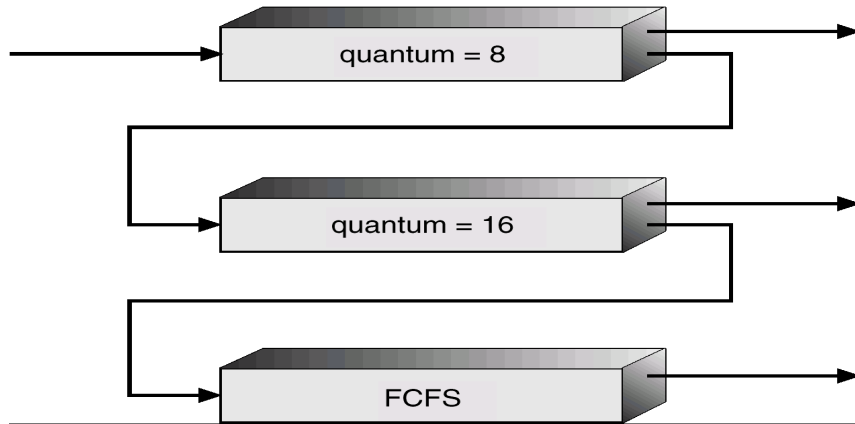
Multilevel Feedback Queue Scheduling

- ❖ **A process can move between the various queues. (Aging)**
- ❖ Multilevel-feedback-queue scheduler defined by the following parameters:
 - ❖ number of queues
 - ❖ scheduling algorithms for each queue
 - ❖ method used to determine when to upgrade a process
 - ❖ method used to determine when to demote a process
 - ❖ method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

❖ Three queues:

- ❖ Q0 – time quantum 8 milliseconds, FCFS
- ❖ Q1 – time quantum 16 milliseconds, FCFS
- ❖ Q2 – FCFS



- ❖ A new job enters queue Q0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q1.
- ❖ At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2.

Lottery Scheduling

- ❖ Each job some number of lottery tickets are issued
- ❖ On each time slice, randomly pick a winning ticket
- ❖ On average, CPU time is proportional to number of tickets given to each job over time
- ❖ How to assign tickets?
 - ❖ To approximate SRTF, short-running jobs get more, long running jobs get fewer
 - ❖ To avoid starvation, every job gets at least one ticket (everyone makes progress)

Example: Lottery Scheduling

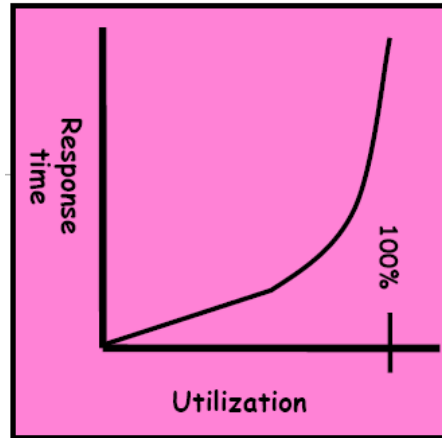
- ❖ Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs / # long jobs	% of CPU each short job gets	% of CPU each long job gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%



Conclusion

- ❖ Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- ❖ When do the details of the scheduling policy and fairness really matter?
 - ❖ When there aren't enough resources to go around



Conclusion

- ❖ FCFS scheduling, FIFO Run Until Done:
 - ❖ Simple, but short jobs get stuck behind long ones
- ❖ RR scheduling:
 - ❖ Give each thread a small amount of CPU time when it executes, and cycle between all ready threads
 - ❖ Better for short jobs, but poor when jobs are the same length
- ❖ SJF/SRTF:
 - ❖ Run whatever job has the least amount of computation to do / least amount of remaining computation to do
 - ❖ Optimal (average response time), but unfair; hard to predict the future

Conclusion

- ❖ Multi-Level Feedback Scheduling:
 - ❖ Multiple queues of different priorities
 - ❖ Automatic promotion/demotion of process priority to approximate SJF/SRTF
- ❖ Lottery Scheduling:
 - ❖ Give each thread a number of tickets (short tasks get more)
 - ❖ Every thread gets tickets to ensure forward progress / fairness
- ❖ Priority Scheduling:
 - ❖ Preemptive or Non-preemptive
 - ❖ Priority Inversion

Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-2E

Introduction to Threads



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

<http://www.iitg.ac.in/johnjose/>

Session Outline

- ❖ **Process vs Threads**
- ❖ **Thread model**
- ❖ **Multithreaded programs**
- ❖ **User and Kernel threads**
- ❖ **Multithread mapping models**

Concept of Threads

- ❖ Thread is a flow of control within a process.
 - ❖ single-threaded process, multi-threaded process.
- ❖ It is a basic unit of CPU utilization, which comprise
 - ❖ a thread ID, program counter, register set, stack.
- ❖ Shares with other threads belonging to the same process its code section, data section, and other OS resources (open files and signal)
- ❖ If a process has multiple threads of control, it can perform more than one task at a time.

The Thread Model

- ❖ Items shared by all threads in a process
- ❖ Items private to each thread

Per process items

Address space

Global variables

Open files

Child processes

Pending alarms

Signals and signal handlers

Accounting information

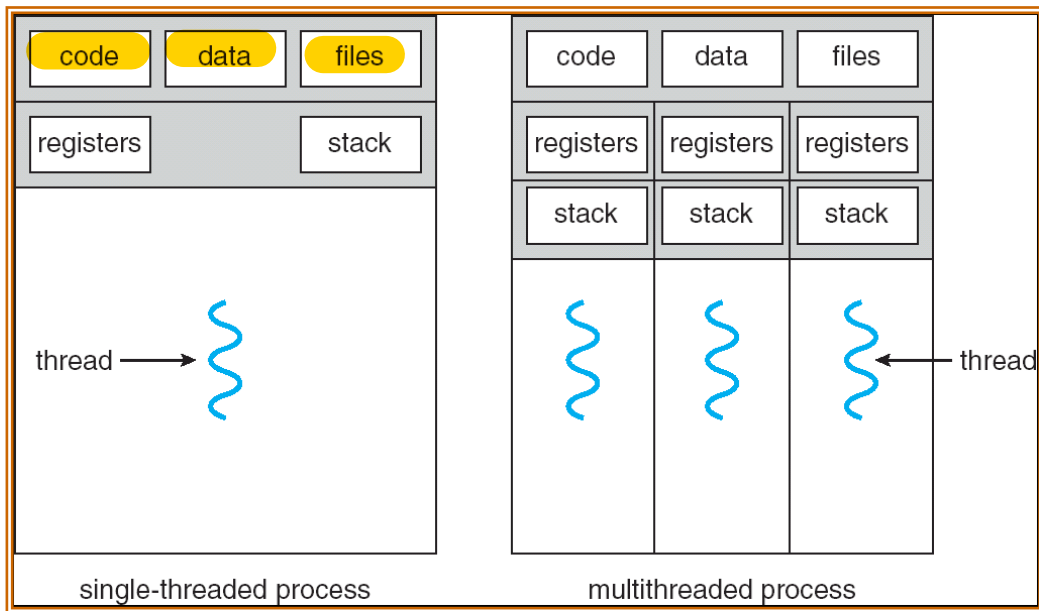
Per thread items

Program counter

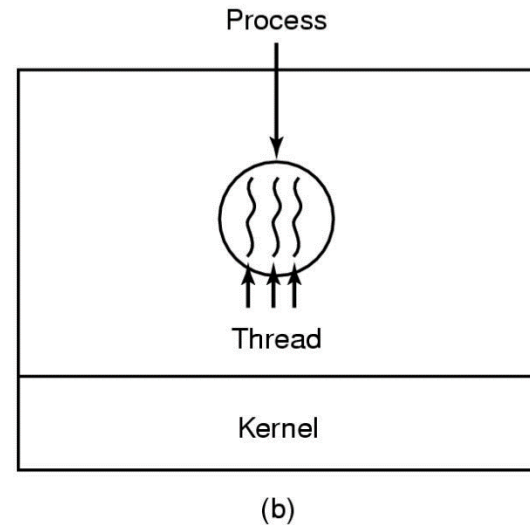
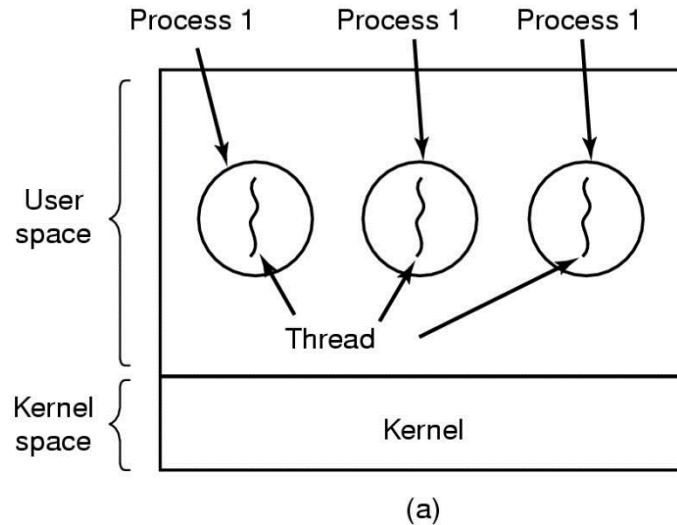
Registers

Stack

State



The Thread Model



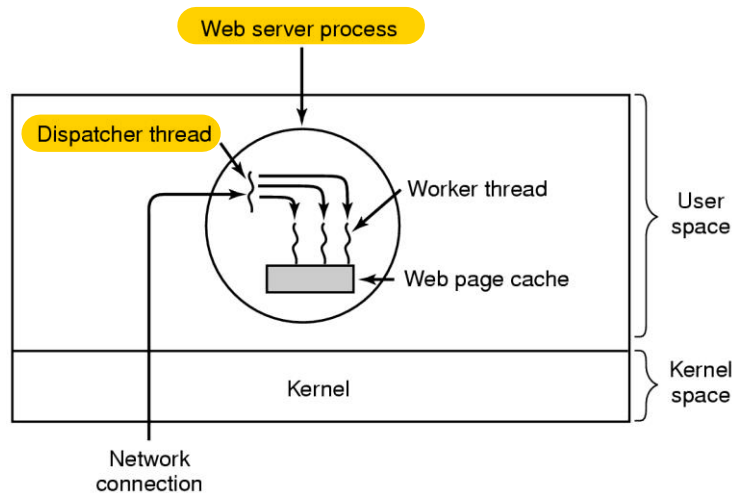
Three processes each with one thread

Vs

One process with three threads

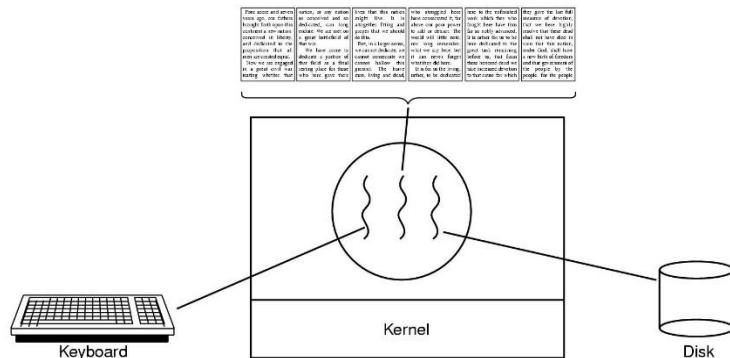
Multi-threaded programs

- ❖ Many software packages that run on modern OS are multi-threaded.
- ❖ A web browser might have
 - ❖ One thread display images or text
 - ❖ Another thread retrieves data from the network



Multi-threaded programs

- ❖ Many software packages that run on modern OS are multi-threaded.
- ❖ A word processor may have
 - ❖ A thread for displaying graphics
 - ❖ Another thread for responding to keystrokes from the user
 - ❖ A third thread for performing spelling and grammar checking



Multi-threaded programs

❖ Types of Web Server

- ❖ **Single-threaded web server**: a client might have to wait for its request to be serviced.
 - ❖ **Multi-processes web server**: used before threads become popular, much overhead in creating a new process.
 - ❖ **Multi-threaded web server**: less overhead in thread creation, concurrent service to multiple client.
- ## ❖ Many OS kernels are now multi-threaded
- ❖ **Several threads** operates in the kernel
 - ❖ Each thread performs a specific task, such as managing devices or interrupt handling.

Benefits of multi-threaded programming

❖ Responsiveness

- ❖ Multithreading an interactive application may allow a program to continue running even if part of it is blocked or doing a lengthy operation.

❖ Resource Sharing

- ❖ Threads share the memory and the resources of the process to which they belong.

❖ Economy

- ❖ Because threads in a process shares the resources, it is more economical to create and context-switch threads.

❖ Utilization of Multi-Processor Architectures

- ❖ Threads may be running in parallel on different processors.

Two types of threads

❖ User Thread

- ❖ User-level thread are threads that are visible to the programmer and are unknown to the kernel.
- ❖ User thread are supported above the kernel and are managed without kernel support.
- ❖ Thread management done by user-level threads library
- ❖ Three primary thread libraries:
 - ❖ **POSIX** Pthreads
 - ❖ **Win32** threads
 - ❖ **Java** threads

Two types of threads

❖ Kernel Thread

- ❖ OS kernel supports and manages kernel-level threads

- ❖ The threads are supported and managed directly by the operating system.

❖ Examples

- ❖ Windows 10

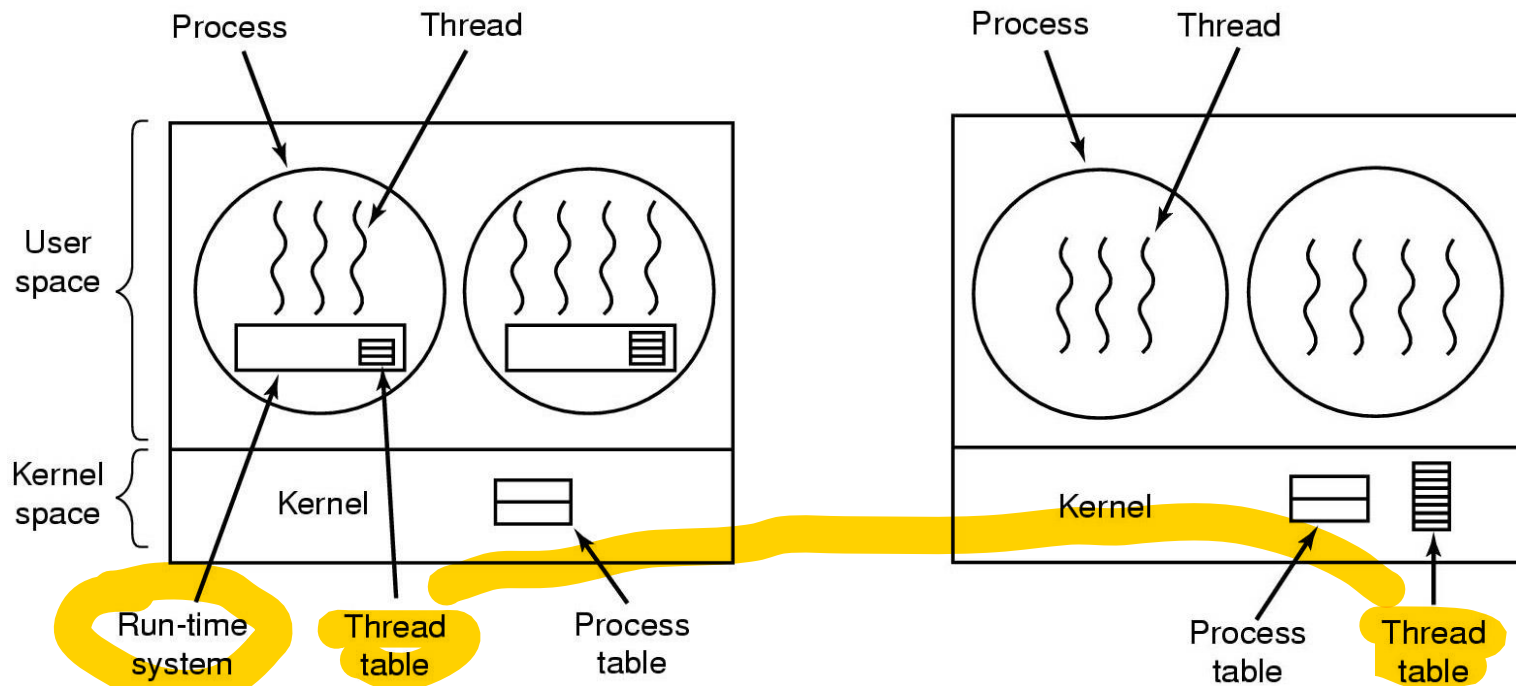
- ❖ Solaris

- ❖ Linux

- ❖ Tru64 UNIX

- ❖ Mac OS X

Implementing Threads in User Space



A user-level threads package

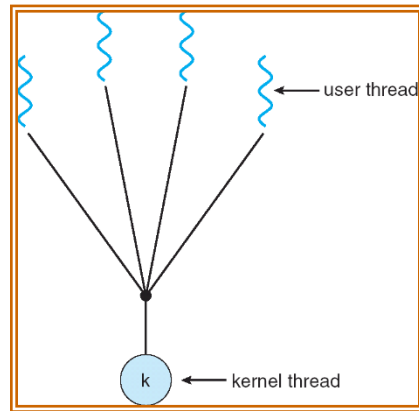
A threads package managed by the kernel

Multithreading Models

- ❖ A Relationship between user threads and kernel threads.
 - ❖ Many-to-One
 - ❖ One-to-One
 - ❖ Many-to-Many
 - ❖ Two Level Model

Many-to-One

- ❖ Many user-level threads mapped to single kernel thread
 - ❖ Thread management is done by the thread library in user space
 - ❖ Can create as many user threads as you wish.
 - ❖ The entire process will block when a thread makes a blocking system call.
 - ❖ Even on multiprocessors, threads are unable to run in parallel
- ❖ Examples:
 - ❖ Solaris Green Threads
 - ❖ GNU Portable Threads

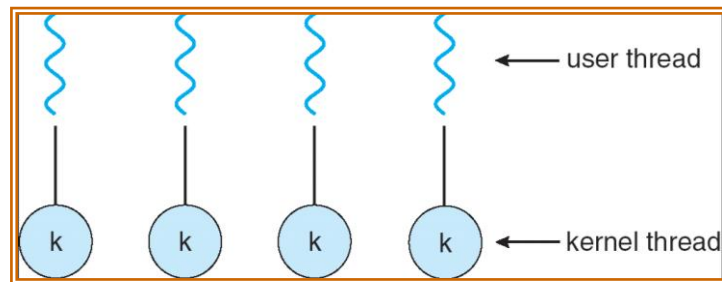


One-to-One

- ❖ Each user-level thread maps to a kernel thread
 - ❖ Provides **more concurrency** than the many-to-one model
 - ❖ Allows another thread to run when a thread is in blocking system call
 - ❖ Creating a user thread requires creating the corresponding kernel thread. (overhead)
 - ❖ The number of threads a process can create is smaller than many-to-one model. (careful not to create too many thread)

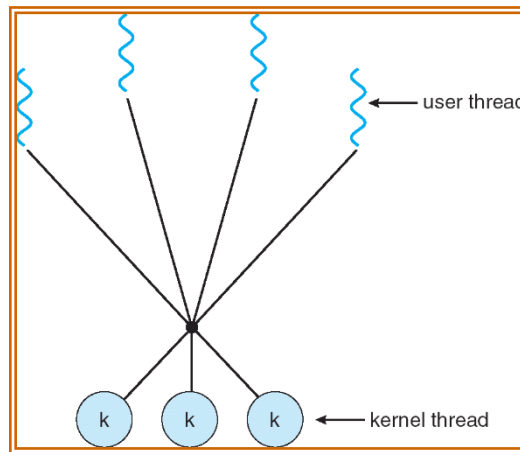
- ❖ Examples

- ❖ Windows NT/XP/2000
- ❖ Linux
- ❖ Solaris 9 and later



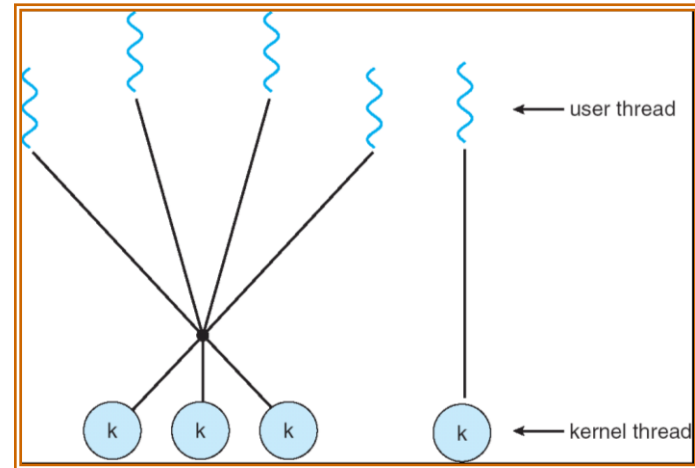
Many-to-Many Model

- ❖ Allows many user level threads to be mapped to smaller or equal kernel threads
 - ❖ Allows the OS to create a sufficient number of kernel threads
 - ❖ The number of kernel threads may be specific to either a application or machine
- ❖ Examples
 - ❖ Solaris prior to version 9
 - ❖ Windows NT/2000 with the ThreadFiber package



Two-Level Model

- ❖ One popular variation on many-to-many model
 - ❖ Similar to Many-to-Many model,
 - ❖ Many user-level threads are multiplexed to a smaller or equal number of kernel threads
 - ❖ But it allows a user thread to be **bound** to a kernel thread
- ❖ Examples
 - ❖ IRIX
 - ❖ HP-UX
 - ❖ Tru64 UNIX
 - ❖ Solaris 8 and earlier



Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>



CS343 - Operating Systems

Module-2F

Thread Libraries



Dr. John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

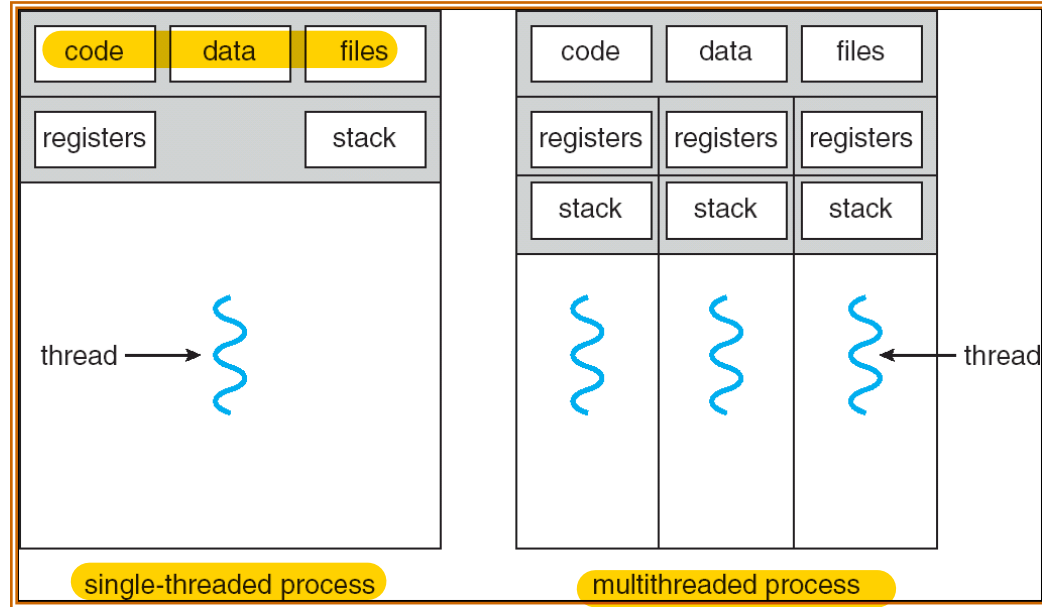
<http://www.iitg.ac.in/johnjose/>

Session Outline

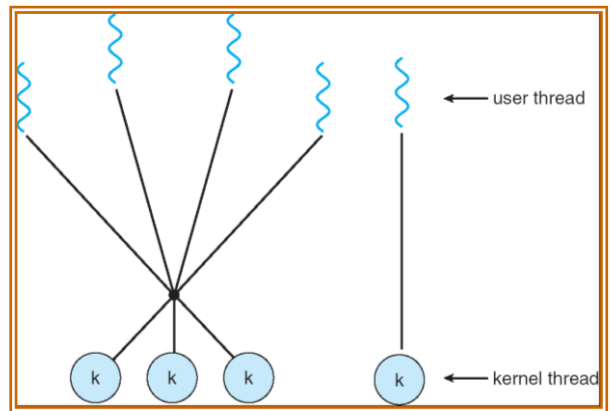
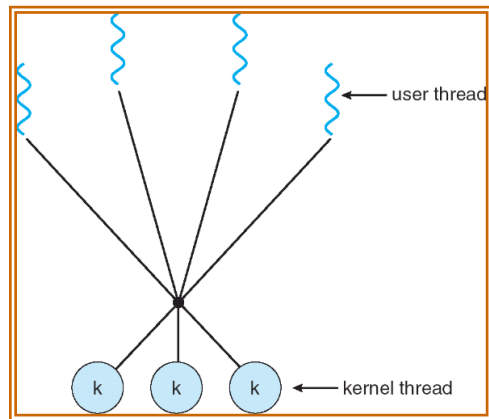
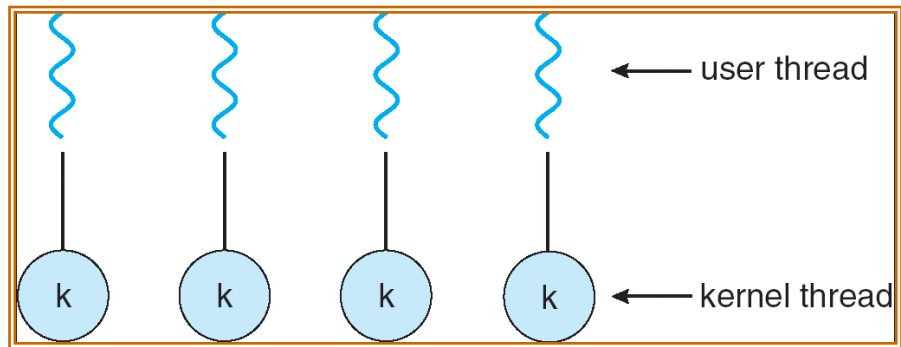
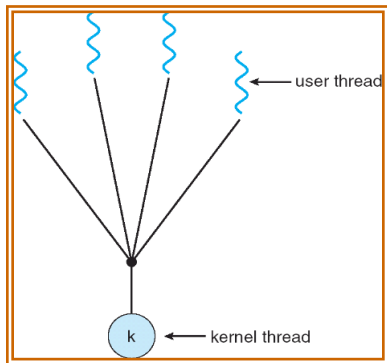
- ❖ Review of Thread model
- ❖ Light weight Process
- ❖ Thread libraries
- ❖ Semantics of `fork()` and `exec()` system calls
- ❖ Thread cancellation
- ❖ Signal handling
- ❖ Thread pools
- ❖ Scheduler activations

Concept of Threads

❖ Thread is a flow of control within a process.



Thread Mappings Models



Thread Libraries

- ❖ A **thread library** provides the programmer an API for creating and managing threads
- ❖ Two primary ways to implement
 - ❖ to **provide a library** entirely in user space with no kernel support.
 - ❖ All **code** and **data structures** for the library exist in user space
 - ❖ Every function call executes in user mode, not in kernel mode
 - ❖ to **implement a** kernel-level library supported by OS
 - ❖ All **code** and **data structures** exist in kernel space
 - ❖ **Invoking functions** result in a system call to the kernel

Thread Libraries

- ❖ A **thread library** provides the programmer **an API** for creating and managing threads
- ❖ **Three** main thread libraries
 - ❖ **POSIX Pthreads** - Solaris, Linux, Mac OS, Tru64 UNIX
 - ❖ **Win32 Thread** - Windows
 - ❖ **Java Thread** - Java 

Pthreads

- ❖ A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ❖ API specifies behavior of the thread library
- ❖ Implementation is up to development of the library
- ❖ Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Windows XP Threads

- ❖ Implements the one-to-one mapping
- ❖ Each thread contains a thread context that consists of
 - ❖ A thread id
 - ❖ Register set
 - ❖ Separate user and kernel stacks
 - ❖ Private data storage area

Linux Threads

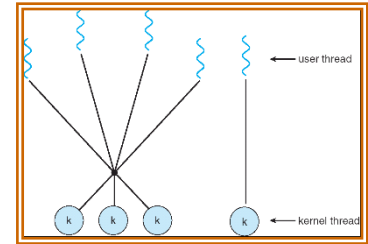
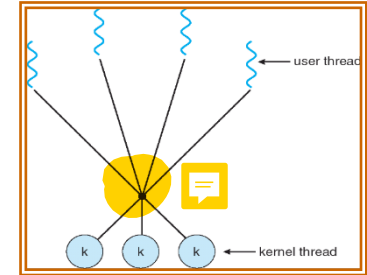
- ❖ Linux refers to tasks rather than threads
- ❖ Thread creation is done through **clone()** system call
- ❖ **clone()** allows a child task to share the address space of the parent task

Java Threads

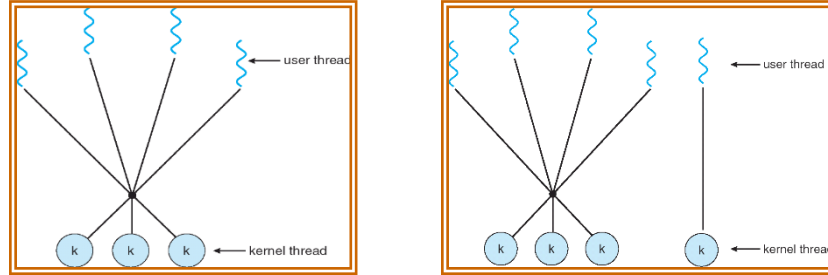
- ❖ Java threads are managed by the JVM
- ❖ Java threads may be created by:
 - ❖ Extending Thread class
 - ❖ Implementing the Runnable interface
- ❖ Because JVM is running on top of a host OS, the Java thread API is implemented using a thread library available on the host system.
 - ❖ On Windows system: using Win32 thread library
 - ❖ On Linux and Unix system: using Pthread library

Light Weight Processes (LWP)

- ❖ Most popular mapping model - many-to-many or two-level mode
- ❖ Light Weight Process (LWP)
 - ❖ An intermediate data structure between user and kernel threads
 - ❖ A user thread is attached to a LWP
 - ❖ Each LWP is attached to a kernel thread
 - ❖ OS schedules the kernel threads (not processes) to run on CPU
 - ❖ If a kernel thread blocks, LWP blocks, and the user thread blocks



Light Weight Processes (LWP)



- ❖ To the user thread library, LWP appears to be a virtual CPU on which the application can schedule a user thread to run.
- ❖ The user thread library is responsible for scheduling among user threads to the LWPs. It is similar to the CPU scheduling in kernel.
- ❖ In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread.

Threading Issues

- ❖ Semantics of fork() and exec() system calls
- ❖ Thread cancellation
- ❖ Signal handling
- ❖ Thread pools
- ❖ Scheduler activations

Semantics of fork() and exec()

- ❖ **fork()** - system call is used to create a duplicate process.
- ❖ **exec()** - system call is used to create a new separate process.
- ❖ Fork() starts a new process which is a copy of the one that calls it, while exec replaces the **current process image** with another new one.
- ❖ Both parent and child processes are executed simultaneously in case of fork()
- ❖ In exec() control never returns to the original program unless there is an exec() error.

Thread Cancellation

- ❖ Terminating a thread before it has finished by other threads
 - ❖ Ex, multiple threads search DB, one thread returns result. The remaining thread might be canceled.
- ❖ Two general approaches to cancel the target thread
 - ❖ **Asynchronous cancellation** terminates the target thread immediately
 - ❖ **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Cancellation

❖ Asynchronous cancellation

- ❖ The difficulty with cancellation occurs in situation where
 - ❖ resources have been allocated to a canceled thread, or
 - ❖ where a thread is canceled while in the midst of updating data it is sharing with other threads

❖ Deferred cancellation

- ❖ One thread indicates that target thread is to be canceled.
- ❖ Cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not

Signal Handling

- ❖ **Signals** are used in UNIX systems to notify a process that a particular event has occurred
- ❖ **Two types of signals**
 - ❖ Synchronous signals [illegal memory access, division by 0]
 - ❖ Asynchronous signals [Specific keystrokes (Ctrl-C), timer expire]
- ❖ What happen when a signal generated?
 - ❖ Signal is generated by particular event
 - ❖ Signal is delivered from kernel to a process
 - ❖ Signal is handled

Signal Handling

- ❖ Every **signal** may be handled by one of two possible handlers.
 - ❖ A **default signal handler**
 - ❖ A **user-defined signal handler**
- ❖ Every signal has a **default signal handler** that is run by the **kernel**
- ❖ The default action **can be overridden** by a **user-defined signal handler**
- ❖ Options when a **signal occurs on a multi-threaded process**
 - ❖ Deliver the signal to the thread to which the signal applies
 - ❖ Deliver the signal to every thread in the process
 - ❖ Deliver the signal to certain threads in the process
 - ❖ Assign a specific thread to receive all signals for the process

Thread Pools

- ❖ **Pool** of threads where they await work
- ❖ A process creates few threads at start up and place into a pool
- ❖ When receiving a request, server awakens a thread from the pool and passes the request to service
- ❖ Once the thread completes its service, it returns to the pool and await more work.
- ❖ If the pool contains no available thread, the server waits until one becomes free.
- ❖ Thread pools are faster to service a request with an existing thread than create a new thread
- ❖ Allows the **number of threads in the application(s)** to be bound to the size of the pool

Scheduler Activations

- ❖ Both Many-to-Many and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- ❖ Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- ❖ This communication allows an application to maintain the correct number kernel threads

Summary

- ❖ A thread is a flow of control within process.
- ❖ Multithreaded process contains several different flows of control within the same address space.
- ❖ Benefits of multi-threading includes
 - ❖ Increased responsiveness, Resource sharing within the process
 - ❖ Economy, Ability to take advantage of multiprocessor architecture
- ❖ User-level thread are thread that are visible to the programmer and are unknown to the kernel.

Summary

- ❖ OS kernel supports and manages kernel-level threads
- ❖ Three types of models relates user and kernel threads
 - ❖ One-to-one, many-to-one, many-to-many
- ❖ Thread libraries provide the application programmer with an API for creating and managing threads
 - ❖ POSIX Pthreads, Win32 threads, Java threads
- ❖ Multithreaded programs introduces several issues
 - ❖ fork()/exec(), thread cancellation, signal handling, thread pools and schedule activation.

Thank you

johnjose@iitg.ac.in

<http://www.iitg.ac.in/johnjose/>

