OPERATING SYSTEM

Process Management

TO CONNECT ER.AMAR PANCHAL SCAN

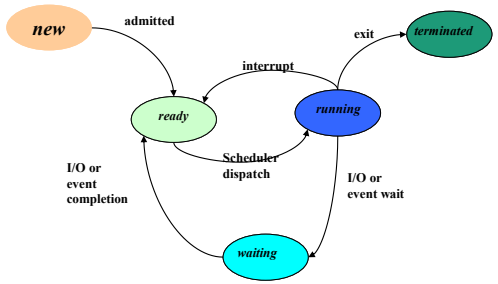
Program-Process

- Process - a program in execution
  - process execution proceeds in a sequential fashion

TO CONNECT ER.AMAR PANCHAL SCAN

Process State

- A process changes state as it executes.



TO CONNECT ER.AMAR PANCHAL SCAN

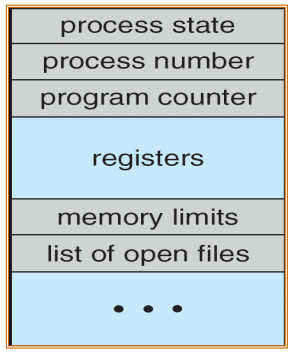
Process States

- New - The process is being created.
- Running - Instructions are being executed.
- Waiting - Waiting for some event to occur.
- Ready - Waiting to be assigned to a processor.
- Terminated - Process has finished execution.

TO CONNECT ER.AMAR PANCHAL SCAN

Process Control Block

- **Contains information associated with each process**
  - **Process State - e.g. new, ready, running etc.**
  - **Process Number – Process ID**
  - **Program Counter - address of next instruction to be executed**
  - **CPU registers - general purpose registers, stack pointer etc.**
  - **CPU scheduling information - process priority, pointer**
  - **Memory Management information - base/limit information**
  - **Accounting information - time limits, process number**
    - **I/O Status information - list of I/O devices allocated**



Process Control Block

TO CONNECT ER.AMAR PANCHAL SCAN

Scheduling Criteria

- CPU Utilization
  - Keep the CPU and other resources as busy as possible
- Throughput
  - # of processes that complete their execution per time unit.
- Turnaround time
  - amount of time to execute a particular process from its entry time.

Principles of Operating Systems - CPU Scheduling

TO CONNECT ER.AMAR PANCHAL SCAN

Scheduling Criteria (cont.)

- Waiting time
  - amount of time a process has been waiting in the ready queue.
- Response Time (in a time-sharing environment)
  - amount of time it takes from when a request was submitted until the first response is produced, NOT output.

Optimization Criteria

- Max CPU Utilization
- Max Throughput
- Min Turnaround time
- Min Waiting time
- Min response time

Principles of Operating Systems - CPU Scheduling

TO CONNECT ER.AMAR PANCHAL SCAN

First Come First Serve (FCFS) Scheduling

- Policy: Process that requests the CPU *FIRST* is allocated the CPU *FIRST.*
  - FCFS is a non-preemptive algorithm.
- Implementation - using FIFO queues
  - incoming process is added to the tail of the queue.
  - Process selected for execution is taken from head of queue.
- Performance metric - Average waiting time in queue.
- Gantt Charts are used to visualize schedules.

Principles of Operating Systems - CPU Scheduling

TO CONNECT ER.AMAR PANCHAL SCAN

First-Come, First-Served(FCFS) Scheduling

- Example

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule

- Suppose the arrival order for the processes is
  - P1, P2, P3
- Waiting time
  - P1 = 0;
  - P2 = 24;
  - P3 = 27;
- Average waiting time
  - (0+24+27)/3 = 17

Principles of Operating Systems - CPU Scheduling

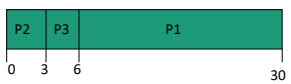
TO CONNECT ER.AMAR PANCHAL SCAN

FCFS Scheduling (cont.)

- Example

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule



- Suppose the arrival order for the processes is
  - P2, P3, P1
- Waiting time
  - P1 = 6; P2 = 0; P3 = 3;
- Average waiting time
  - (6+0+3)/3 = 3 , better..
- *Convoy Effect*:
  - short process behind long process, e.g. 1 CPU bound process, many I/O bound processes.

Shortest-Job-First(SJF) Scheduling

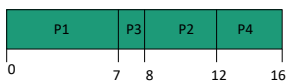
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two Schemes:
  - Scheme 1: Non-preemptive
    - Once CPU is given to the process it cannot be preempted until it completes its CPU burst.
  - Scheme 2: Preemptive
    - If a new CPU process arrives with CPU burst length less than remaining time of current executing process, preempt. Also called Shortest-Remaining-Time-First (SRTF).
  - SJF is optimal - gives minimum average waiting time for a given set of processes.

Non-Preemptive SJF Scheduling

- Example

Process	Arrival Time	Burst Time
P1	0	7
P2	0.2	4
P3	4	1
P4	5	4

Gantt Chart for Schedule



Average waiting time =
(0+6+3+7)/4 = 4

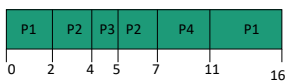
TO CONNECT ER.AMAR PANCHAL SCAN

Preemptive SJF Scheduling(SRTF)

- Example

Process	Arrival Time	Burst Time
P1	0	7
P2	0.2	4
P3	4	1
P4	5	4

Gantt Chart for Schedule



Average waiting time =
(9+1+0+2)/4 = 3

TO CONNECT ER.AMAR PANCHAL SCAN

Determining Length of Next CPU Burst

- One can only estimate the length of burst.
- Use the length of previous CPU bursts and perform exponential averaging.
  - t_n = actual length of nth burst
  - τ_{n+1} =predicted value for the next CPU burst
  - $\alpha = 0,\ 0 \leq \alpha \leq 1$
  - Define
    - $\tau_{n+1} = \alpha\, t_n + (1- \alpha)\, \tau_n$

Principles of Operating Systems - CPU Scheduling

TO CONNECT ER.AMAR PANCHAL SCAN

Exponential Averaging(cont.)

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$; Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$; Only the actual last CPU burst counts.
- Similarly, expanding the formula:
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\, \alpha t_{n-1} + \ldots + (1-\alpha)^j\, \alpha t_{n-j} + \ldots$ $(1-\alpha)^{(n+1)}\, \tau_0$
    - Each successive term has less weight than its predecessor.

TO CONNECT ER.AMAR PANCHAL SCAN

Priority Scheduling

- A priority value (integer) is associated with each process. Can be based on
  - Cost to user
  - Importance to user
  - Aging
  - %CPU time used in last X hours.
- CPU is allocated to process with the highest priority.
  - Preemptive
  - Nonpreemptive

Priority Scheduling (cont.)

- SJN is a priority scheme where the priority is the predicted next CPU burst time.
- Problem
  - Starvation!! - Low priority processes may never execute.
- Solution
  - Aging - as time progresses increase the priority of the process.

Principles of Operating Systems - CPU Scheduling

Round Robin (RR)

- Each process gets a small unit of CPU time
  - Time quantum usually 10-100 milliseconds.
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- *n* processes, time quantum = *q*
  - Each process gets 1/*n* CPU time in chunks of at most *q* time units at a time.
  - No process waits more than (*n*-1)*q* time units.
  - Performance
    - Time slice *q* too large - FIFO behavior
    - Time slice *q* too small - Overhead of context switch is too expensive.
    - Heuristic - 70-80% of jobs block within timeslice

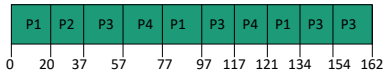
TO CONNECT ER.AMAR PANCHAL SCAN

Round Robin Example

- Time Quantum = 20

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

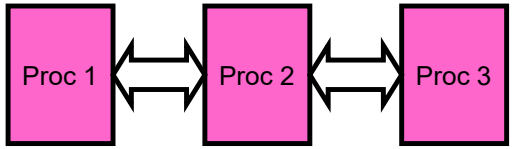
Gantt Chart for Schedule



Typically, higher average turnaround time than SRTF, but better response

TO CONNECT ER.AMAR PANCHAL SCAN

Interprocess Communication (IPC)



- Separate address space isolates processes
  - High Creation/Memory Overhead; (Relatively) High Context-Switch Overhead
- Mechanism for processes to communicate and synchronize actions.
  - Via shared memory - Accomplished by mapping addresses to common DRAM
    - Read and Write through memory
  - Via Messaging system - processes communicate without resorting to shared variables.
    - `send()` and `receive()` messages
    - Can be used over the network!
  - Messaging system and shared memory not mutually exclusive
    - can be used simultaneously within a single OS or a single process.

TO CONNECT ER.AMAR PANCHAL SCAN

The Critical-Section Problem

- N processes all competing to use shared data.
  - Structure of process P_i — Each process has a code segment, called the critical section, in which the shared data is accessed.

```
repeat
entry section    /* enter critical section */
      critical section  /* access shared variables */
  exit  section     /* leave critical section */
      remainder section  /* do other work */
until false
```

- Problem
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Principles of Operating Systems - Process Synchronization

TO CONNECT ER.AMAR PANCHAL SCAN

Solution: Critical Section Problem - Requirements

- **Mutual Exclusion**
  - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**
  - If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- **Bounded Waiting**
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Semaphore

- Semaphore *S* - integer variable (non-negative)
  - used to represent number of abstract resources
- Can only be accessed via two indivisible (atomic) operations

  wait (*S*): **while** *S* <= 0 **do** no-op

  S := *S*-1;

  signal (S): *S* := *S*+1;
  - *P* or *wait* used to acquire a resource, waits for semaphore to become positive, then decrements it by 1
  - *V* or *signal* releases a resource and increments the semaphore by 1, waking up a waiting P, if any
  - If *P* is performed on a *count* <= 0, process must wait for *V* or the release of a resource.

P():*"proberen"* (to test) ; V() *"verhogen"* (to increment) in Dutch

Example: Critical Section for n Processes

- Shared variables

  var *mutex*: semaphore
  initially *mutex* = 1

- Process P_i

```
repeat
        wait(mutex);
                critical section
        signal (mutex);
                remainder section
until false
```


Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* execute in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
  - Example 1
    - System has 2 tape drives. P1 and P2 each hold one tape drive and each needs the other one.
  - Example 2
    - Semaphores A and B each initialized to 1

P0	*P1*
wait(A)	*wait(B)*
wait(B)	*wait(A)*

TO CONNECT ER.AMAR PANCHAL SCAN

Definitions

- A process is *deadlocked* if it is waiting for an event that will never occur.
  - Typically, more than one process will be involved in a deadlock (the deadly embrace).
- A process is *indefinitely postponed* if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes,
  - i.e. the process is ready to proceed but never gets the CPU.

TO CONNECT ER.AMAR PANCHAL SCAN

Deadlock Management

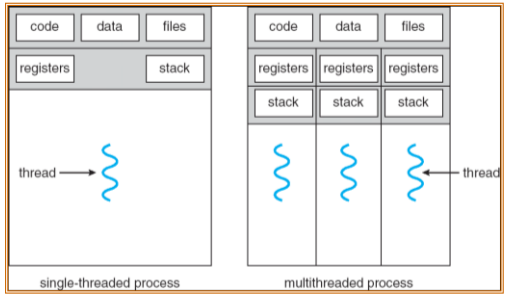
- Prevention
  - Design the system in such a way that deadlocks can never occur
- Avoidance
  - Impose less stringent conditions than for prevention, allowing the possibility of deadlock but sidestepping it as it occurs.
- Detection
  - Allow possibility of deadlock, determine if deadlock has occurred and which processes and resources are involved.
- Recovery
  - After detection, clear the problem, allow processes to complete and resources to be reused. May involve destroying and restarting processes.

Threads

- Processes do not share resources well
  - high context switching overhead

- Idea: Separate concurrency from protection
- Multithreading: *a single program made up of a number of different concurrent activities*
- A thread (or lightweight process)
  - basic unit of CPU utilization; it consists of:
    - program counter, register set and stack space
  - A thread shares the following with peer threads:
    - code section, data section and OS resources (open files, signals)
    - No protection between threads
  - Collectively called a task.
- Heavyweight process is a task with one thread.

Single and Multithreaded Processes



- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system

TO CONNECT ER.AMAR PANCHAL SCAN

Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

TO CONNECT ER.AMAR PANCHAL SCAN

Types of Threads

- Kernel-supported threads
- User-level threads
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

TO CONNECT ER.AMAR PANCHAL SCAN

Kernel Threads

- Supported by the Kernel
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Examples
  - Windows XP/2000, Solaris, Linux,Tru64 UNIX, Mac OS X, Mach, OS/2

TO CONNECT ER.AMAR PANCHAL SCAN

User Threads

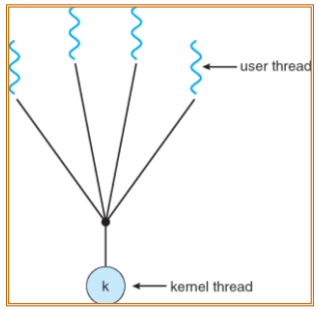
- Supported above the kernel, via a set of library calls at the user level.
  - Thread management done by user-level threads library
    - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-premptively relative to each other (only switch on yield())
- Advantages
  - Cheap, Fast
    - Threads do not need to call OS and cause interrupts to kernel
- Disadv: If kernel is single threaded, system call from any thread can block the entire task.
- Example thread libraries:
  - POSIX Pthreads, Win32 threads, Java threads

Multithreading Models

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

Many-to-One

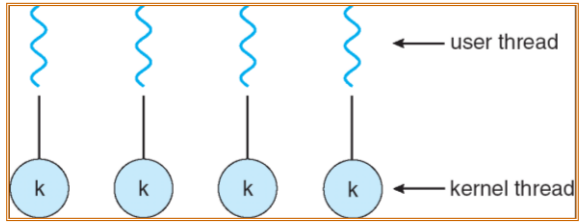
- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



TO CONNECT ER.AMAR PANCHAL SCAN

One-to-One

- Each user-level thread maps to kernel thread



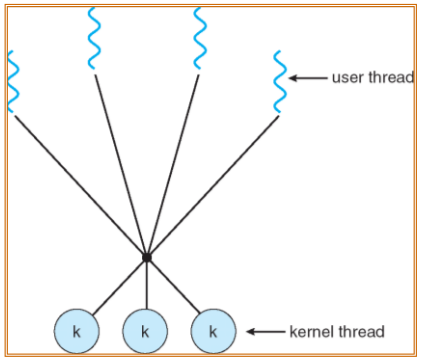
Examples

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

TO CONNECT ER.AMAR PANCHAL SCAN

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



TO CONNECT ER.AMAR PANCHAL SCAN