Operating Systems And Concurrency, Lecture 1: Processes and Threads

Dr Amanda Coles

Slides based on Chapters 3-5 of "Operating Systems Concepts" Silberschatz, Galvin and Gagne.

Overview

- Introduction to OSC.
- What is a Process?
- Process Scheduling.
- Operations on Processes.

Module Information

- Lecturers Amanda Coles and Andrew Coles.
- Wednesday 10:00 – 13:00
  - 5 lectures on Operating Systems
  - 5 lectures on Concurrency
  - 1 lecture on either of the above
- Assessment:
  - Weekly coursework (15%): starts this week, 18/01/16 (see next slide)
  - Examination in May (85%).
- Recommended books:
  - “Operating Systems Concepts”, Silberschatz, Galvin and Gagne; Wiley, 8th Edition (2009)
  - Will also use some of “Principles of Concurrent and Distributed Programming”, Ben-Ari, 2nd Edition (2006)

Weekly Coursework

- 15% of the module mark is from coursework
- You must pass the coursework to pass the module
  - e.g. 39% in the coursework, 100% in the exam = fails the module, need to resit

- 10 weekly exercises available through KEATS all weeks count:
  - The 15% is split evenly between these (don't worry about the total marks for each week they are just to balance the weighting of the questions within the quiz).
- The coursework is assessed and must be your own work. We are happy to review topics that you are struggling with but will not directly assist with coursework questions.
  - Use the tutorial to read your questions through, ask any questions you have about the lecture material and make sure you understand them.

Deadlines

- Each task is released on Wednesday at approximately noon.
- Each task is due in the following Tuesday, at 23:59:
  - Meeting the deadline is your responsibility: do not assume that acceptance of submission on KEATS implies that the work is on time.
- Feedback released a week after the deadline.
- Normal college assessment rules apply for each individual task:
  - Submissions up to 24h late: capped at 40% (N.B. KEATS will release your actual mark but this will be capped in calculating your coursework mark for the module).
  - If you have mitigating circumstances:
    - Submit an NEA as soon as possible.
    - Extensions will not normally be possible, but alternative assessments may be offered. If you defer more than 2 assignments from either half of the course then the assessment-sub board may require you to sit the module again next year.
- If you have any questions about the marking of your coursework we will answer them in tutorials.

Overview

- Introduction to OSC.
- What is a Process?
- Process Scheduling.
- Operations on Processes.

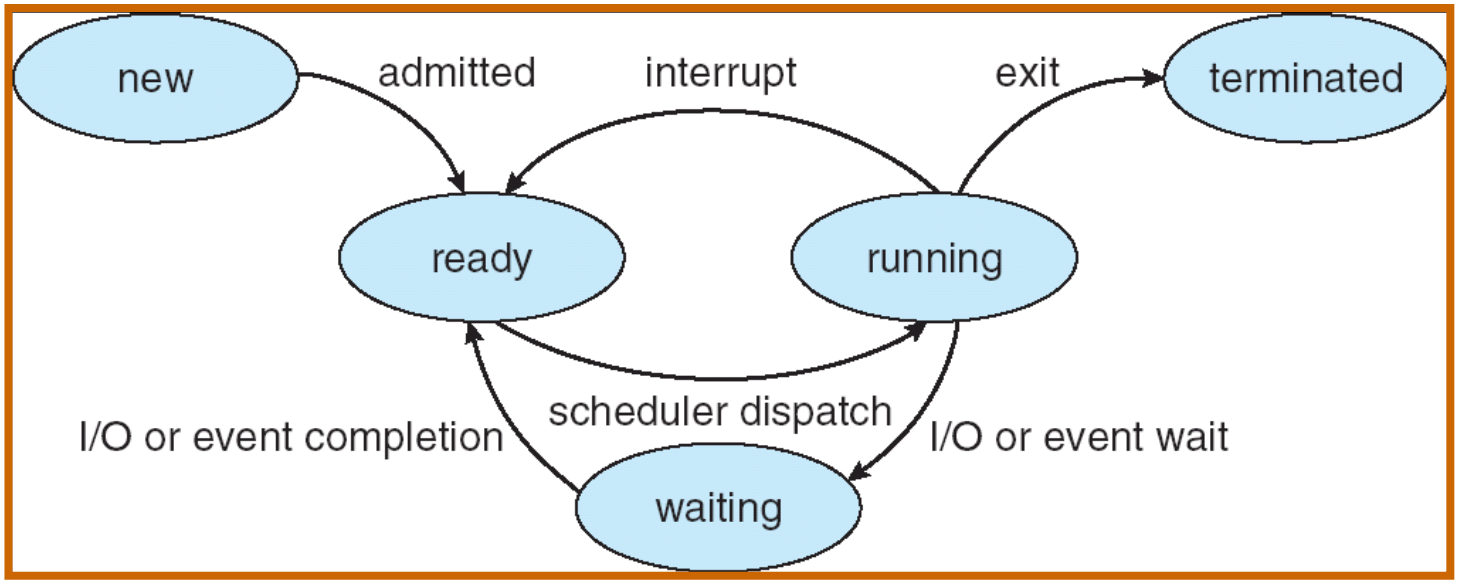
What is a Process?

- “A program in execution”.
  - The program counter: what is the next instruction to execute;
  - The current values of variables at this point during execution.

- Distinct from a ***program*** which is actually an unchanging collection of instructions.
  - For our purposes a program is a binary: the output of a compiler: a set of machine code instructions stored on disk (or in memory) that can be executed.

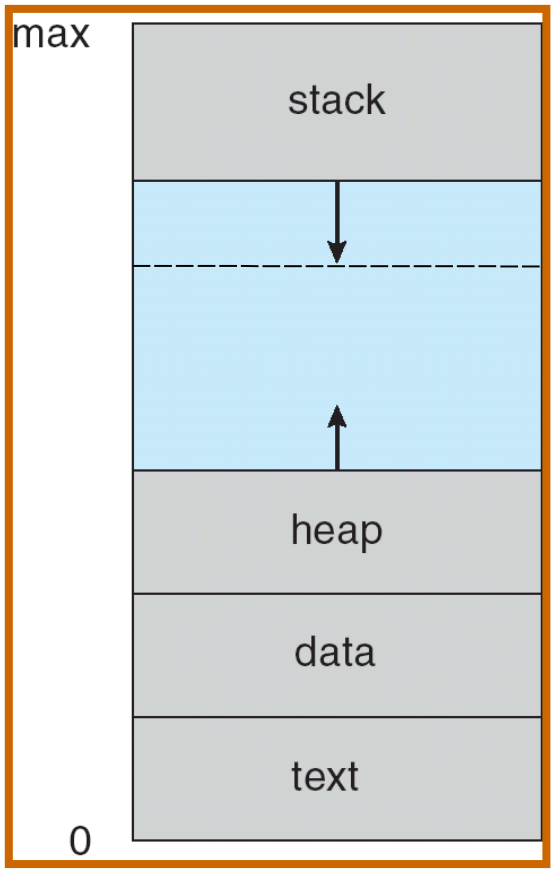
- To clarify the distinction it is possible that a user may run several instances of a single program, creating multiple processes for a single program:
  - e.g. You might run 3 copies of notepad:
    - 1 copy of the program (the binary) will be loaded into memory;
    - 3 processes are created each maintaining their own position in the source code, and current values of variables.

Process States



Processes in Memory

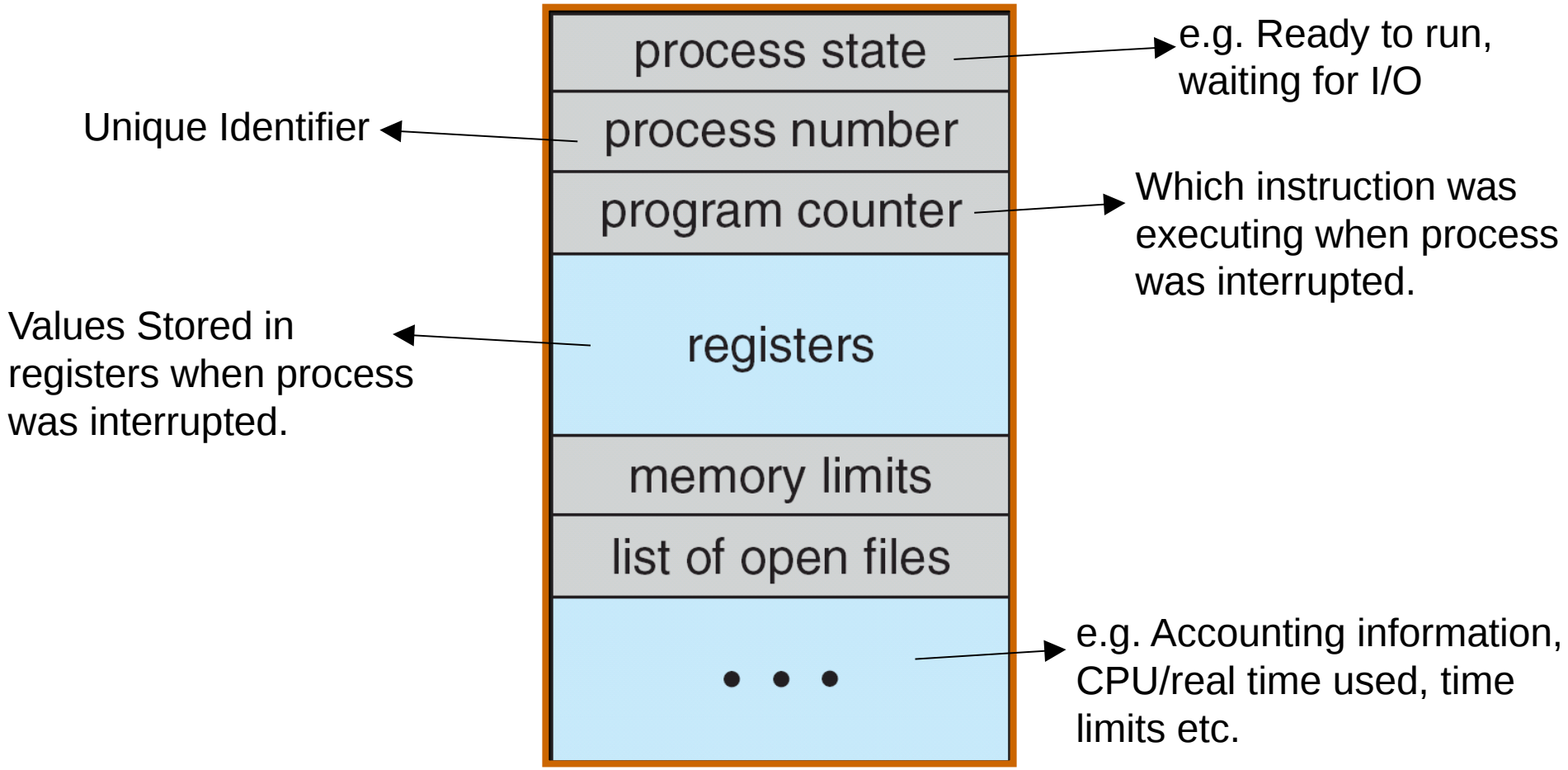
- **Stack:** Contains temporary data: function parameters, return addresses, local variables.
- **Heap:** Dynamically allocated memory (e.g. Objects created using new): the memory the process can use.
- **Data:** Global Variables (incl. e.g. Java static variables).
- **Text:** Code (compiled binary) for the program the process is executing.



N.B. Register values and programme counters are stored either in the registers (when the process is executing); or in the PCB (when it is not).

Process Control Block

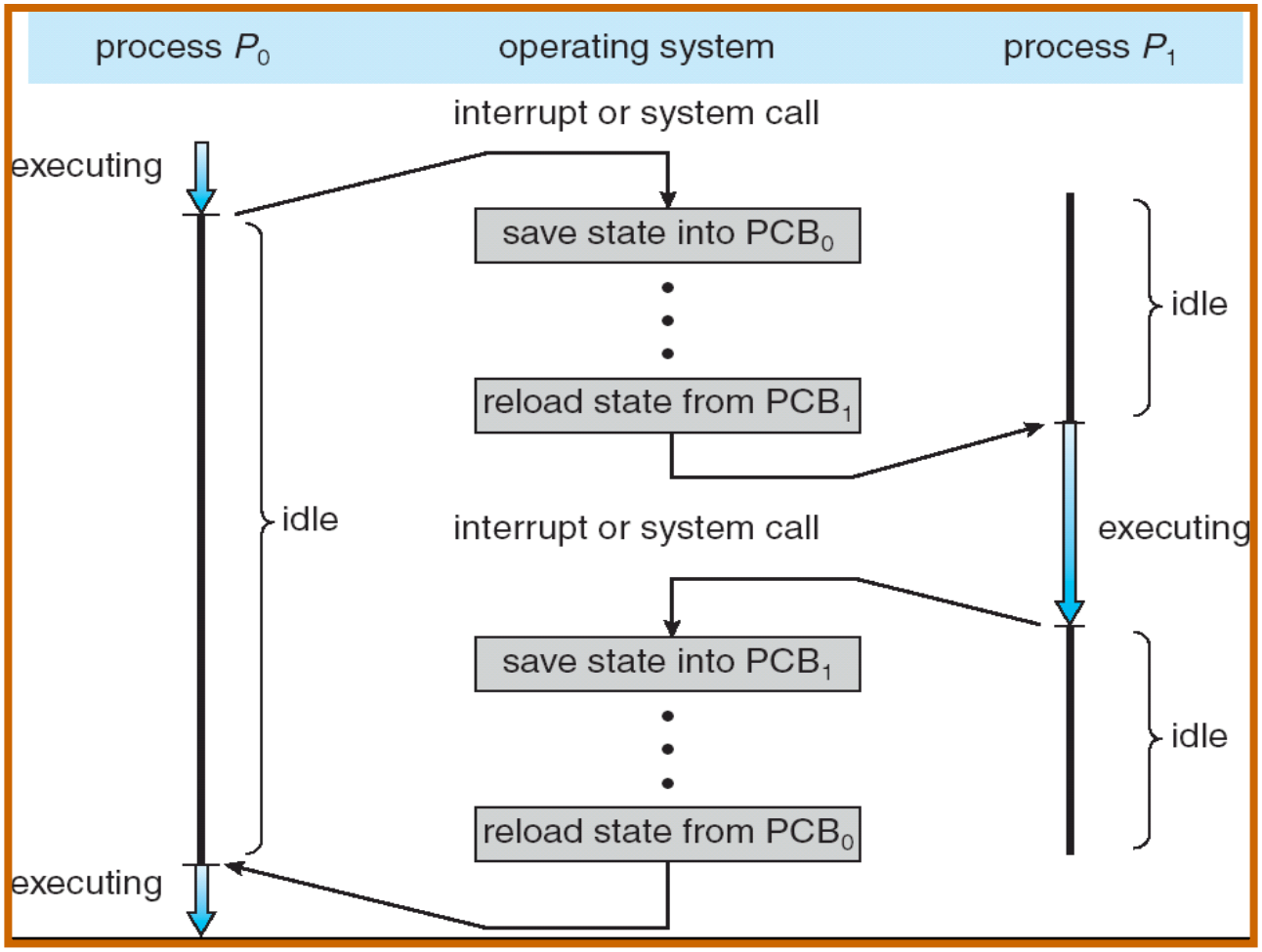
A Structure maintained by the operating system to represent scheduling and state information for processes: stored separately from the process itself.



Context Switching

- When an operating system changes which process is running this is referred to as a ***context switch***;
- The values in the registers, including the program counter, must be stored in the PCB for the process that is being stopped.
  - This ensures it can be resumed in the future from where it finished.
- The values stored in the PCB for the program counter and register values must be loaded into the appropriate registers.
  - This process can now be resumed from where it finished.
- Doing context switching takes time, in which no useful work is done, i.e. it is an ***overhead***.

Switching Between Processes



Why Context Switch?

- Context switching has overheads;
- However, it allows us to give the illusion of several applications running concurrently even on a single CPU:
  - In fact, there are currently 80 processes running on my (2 core) machine.
  - I don’t have to close PowerPoint to view my web-browser, or to listen to music.
  - If I copy a large file from one place to another I can carry on working whilst that completes in the background.
- Context switching allows us to get more work done: I/O bound processes can run at the same time as CPU bound ones. Most processes only require short CPU bursts anyway, so switching between several is a good solution.
- In reality only one process can run on one processor at any given time.

Overview

- Introduction to OSC.
- What is a Process?
- Process Scheduling.
  - Introduction
  - Single Processor Scheduling Algorithms
  - Multi-processor Scheduling
  - Operating System Examples
- Operations on Processes.

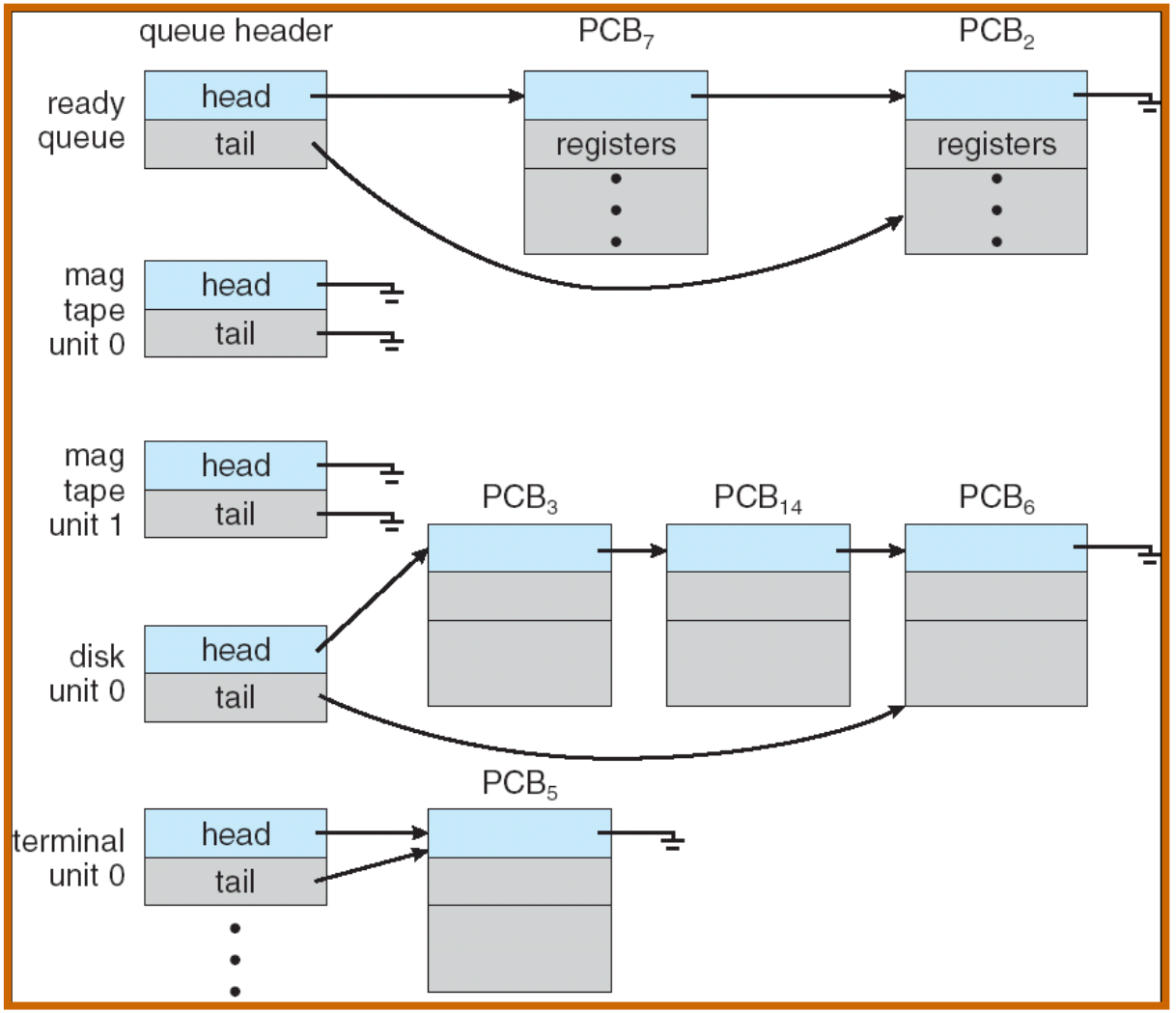
Various Queues in an OS

Job Queue: all processes in the system.

Ready Queue: Processes waiting for the CPU.

Device Queues: One per device, processes waiting for that device.

During the execution of a process it will move between these queues depending on its execution.

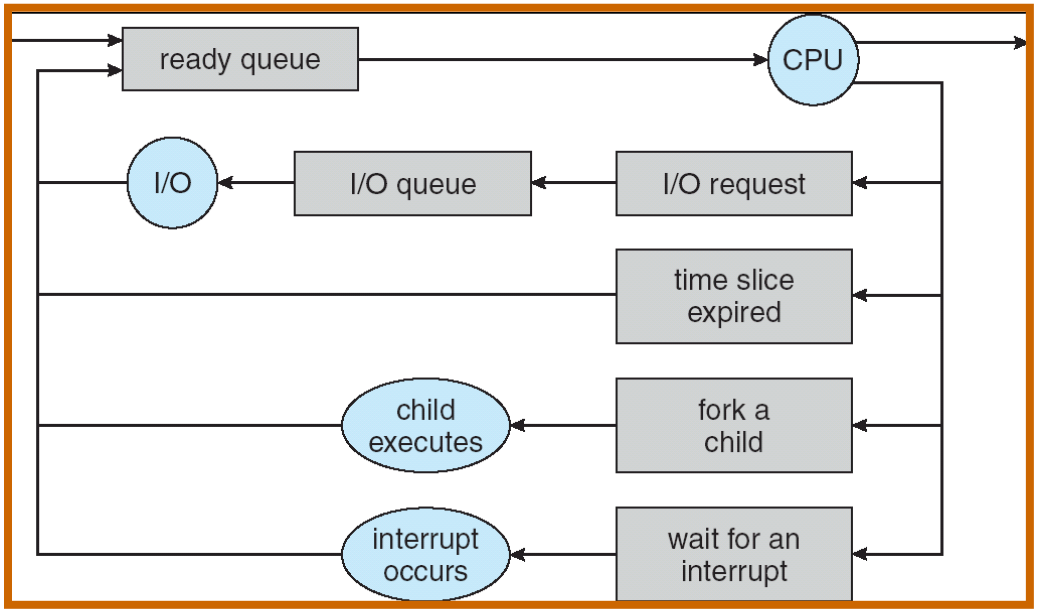


Phases of Scheduling

- Scheduling can be broken down into two separate stages, long and short-term:
  - Long-term scheduler:
    - Determines which processes are put on the ready queue;
    - Runs infrequently (seconds);
    - Determines how many processes can run at once (degree of multiprogramming).
  - Short-term (CPU) scheduler:
    - Decides which process from the ready queue will get to run on the CPU;
    - Runs frequently (milliseconds) so must be fast.

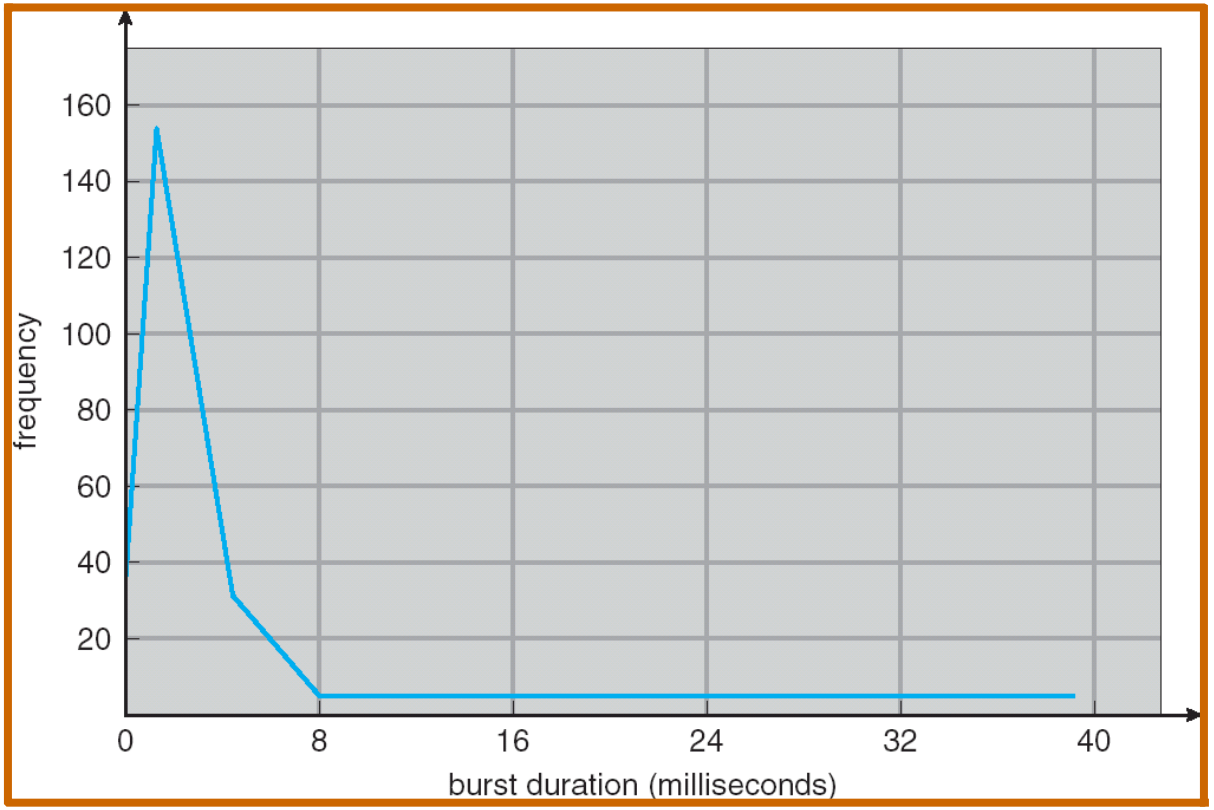
Process Scheduling

Everything starts on the ready queue when added by the long-term scheduler, executes on the CPU for some time, then is moved to other queues as appropriate.



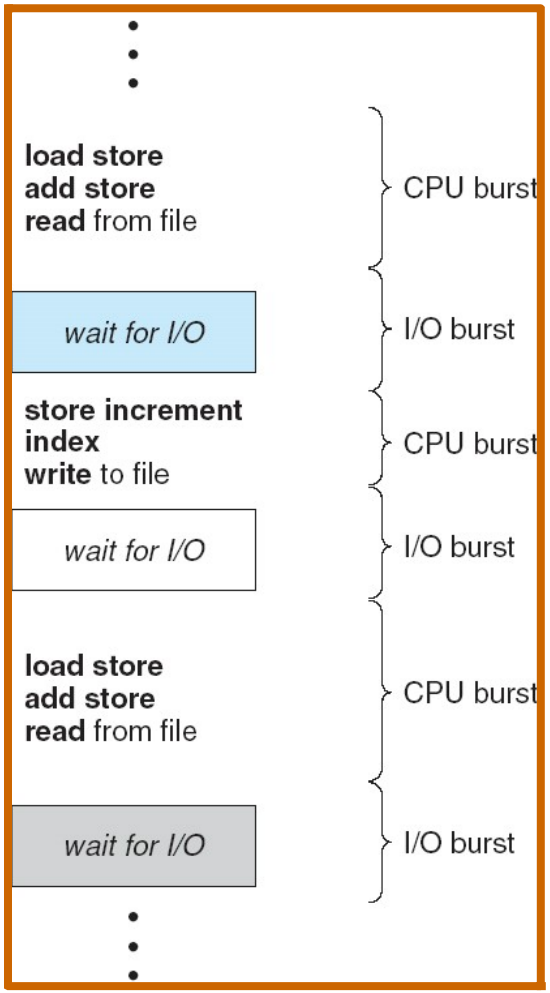
If the queue comprises a mixture of CPU bound and IO bound processes then good efficiency can be achieved by allowing one process to do IO whilst the other uses the CPU.

CPU Burst Time



Typically processes have short bursts of CPU usage interspersed with I/O access. When we talk about the ‘duration’ of a process we are scheduling we will mean the duration of the next CPU burst (and that will be all we consider scheduling).

Typical Process Behaviour



CPU Scheduling Algorithms

- Efficient CPU scheduling algorithms are essential for good utilisation of the processor and responsiveness of the operating system.
  - Keep many processes in memory;
  - Schedule one to run for some time;
  - When one process has finished running, requested I/O (or ran out of time) another is ready to run.
- Here we will consider several scheduling algorithms.
- Different algorithms are better for different types of processes depending on duration (CPU bound/IO bound).
- The properties of the algorithms therefore make them better suited in different situations: what measures can we use to asses their efficiency?

Criteria for Scheduling Algorithm Performance

- ***CPU Utilisation***: the percentage of the time the CPU is executing a process.
- ***Waiting time***: the total time a process spends waiting in the ready queue.
- ***Turnaround time***: the time between arrival of a process in the ready queue and it completing execution.
- ***Response time***: the time between arrival of the process and the production of the first response (depends on how long a process takes to make a response, it may not need to complete in order to do so).
- ***Throughput***: the number of processes completed per unit time (clearly affected by process duration).

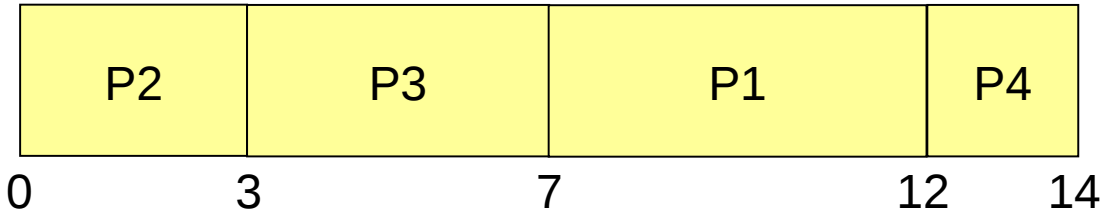
- We can also ask for ***average (mean)*** waiting/turnaround/response time or ***total (sum)*** waiting/turnaround/response time, or **maximum/minimum** waiting/turnaround/response time for a set of processes.

Scheduling Algorithms

- First Come, First Served (FCFS);
- Shortest Job First (SJF);
- Priority Scheduling;
- Round Robin (RR).

FCFS

Process	Arrival	Duration
P1	4	5
P2	0	3
P3	2	4
P4	12	2



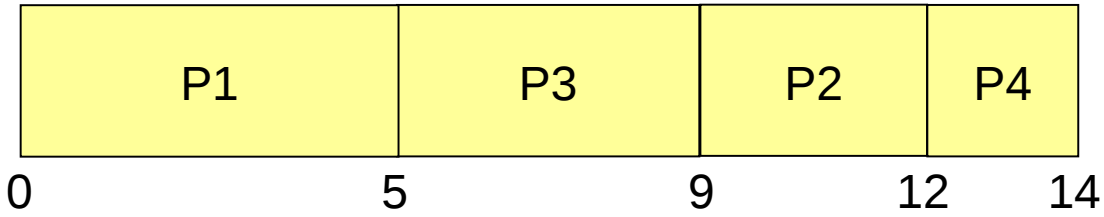
Average Waiting Time = (0 + (3 – 2) + (7 – 4) + (12 – 12)) / 4 = 1

Average Turnaround Time = (3 + (7 – 2) + (12 – 4) + (14 – 12)) / 4 = 4.5

Throughput = 4 / 14 = 0.29

FCFS: Order Sensitivity

Process	Arrival	Duration
P1	0	5
P2	4	3
P3	2	4
P4	12	2



Average Waiting Time = (0 + (5 – 2) + (9 – 4) + (12 – 12)) / 4 = 2

Average Turnaround Time = (5 + (9 – 2) + (12 – 4) + (14 – 12)) / 4 = 5.5

Throughput = 4 / 14 = 0.29

FCFS: Advantages/Disadvantages

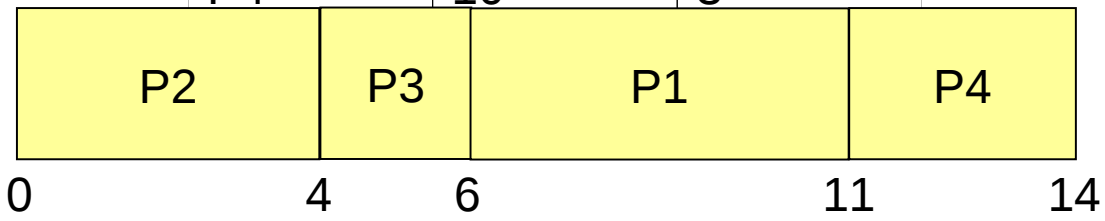
+ Simple algorithm;
+ Only one context switch per process.

− Average waiting time typically poor.
− Average waiting time highly variable: dependent on order in which processes arrive.
− CPU Bound processes can hog the CPU: imagine a system with one CPU bound process and many I/O bound processes.

Shortest Job First (SJF)

Remember: Jobs can only be scheduled when they have arrived!

Process	Arrival	Duration
P1	0	5
P2	0	4
P3	1	2
P4	10	3



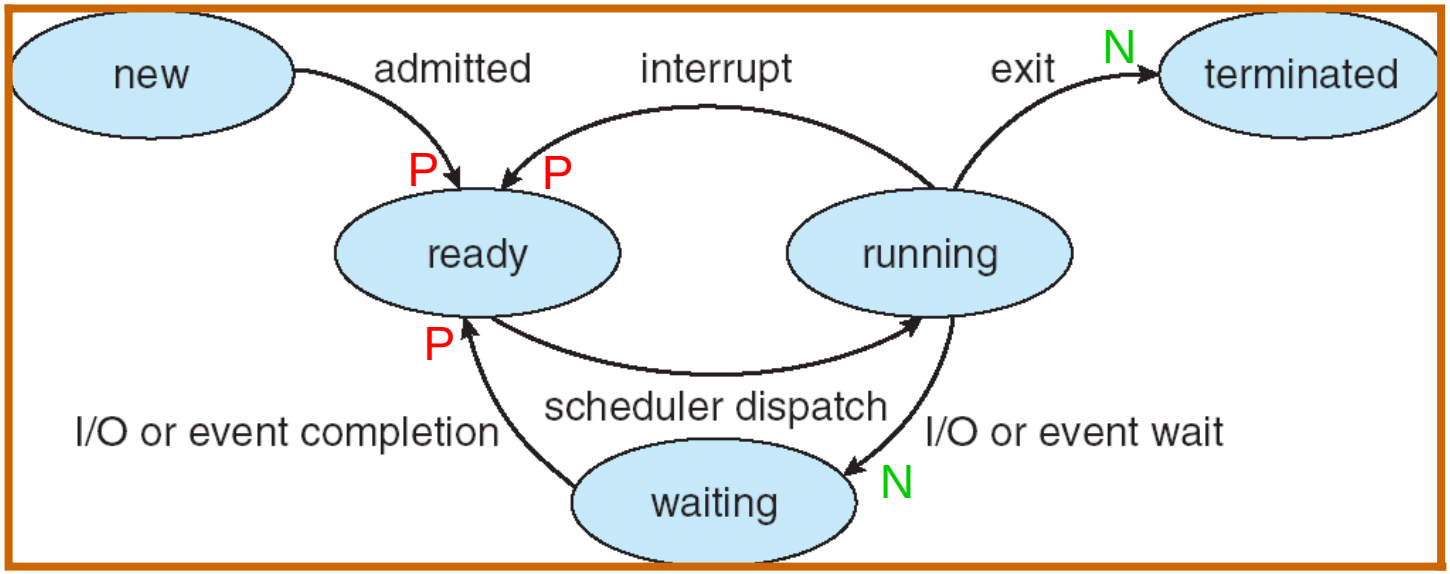
Average Waiting Time = (0 + (4 – 1) + (6 – 0) + (11 – 10)) / 4 = 2.5

Average Turnaround Time = (4 + (6 – 1) + (11 – 0) + (14 – 10)) / 4 = 6

Throughput = 4 / 14 = 0.29

Pre-emptive Scheduling

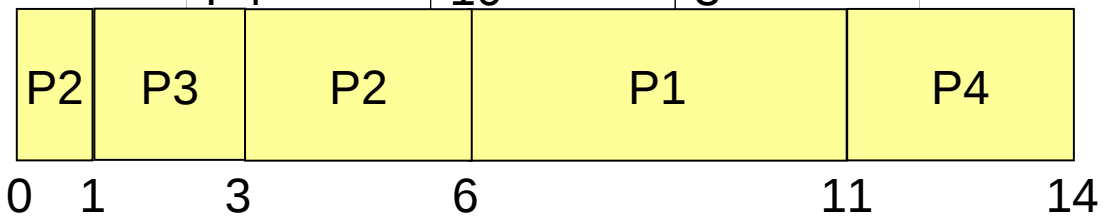
- **Non-pre-emptive** scheduling: The process that is executing on the CPU continues until it has finished its CPU burst.
- **Pre-emptive** scheduling: If another process arrives the process currently running can be stopped, and the new process started.
- Scheduling decisions occur when a new process is added to the ready queue (the arcs marked P below can cause new scheduling decisions).



Pre-emptive Shortest Job First
Shortest Remaining Time First (SRTF)

If a new job arrives that is shorter than the remaining time of the current job then execute it.

Process	Arrival	Duration
P1	0	5
P2	0	4
P3	1	2
P4	10	3



Average Waiting Time = (6 + 2 + 0 + 1) / 4 = 2.25 *(vs 3.5 for FCFS)*

Average Turnaround Time = ((11 – 0) + (6 – 0) + (3 – 1) + (14 – 10) / 4 = 5.75

SJF: Advantages/Disadvantages

+ Short processes do not have to wait for long processes to complete before executing (especially if pre-emptive scheduling is used). That is, CPU Bound processes cannot hog the CPU.
+ Maximises throughput (for this imagine a queue where processes continually arrive).
+ Average waiting time is smaller (for the same set of processes) since no process waits for the longest to complete.

− Risk of starvation for long processes; or alternatively long waiting times.
− Multiple context switches for each process if pre-emptive.
− Can we reliably estimate process duration before execution?
− Overheads in inserting processes in a sorted queue.

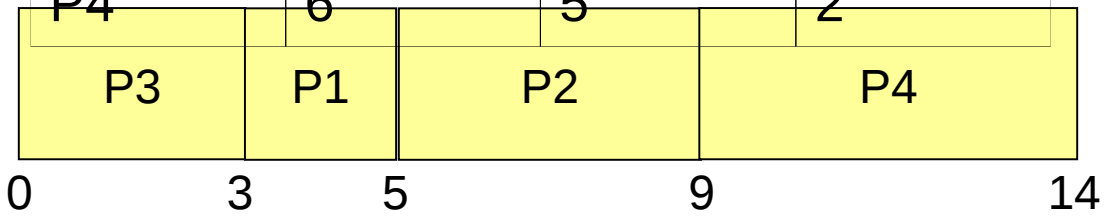
Priority Scheduling

- Associate a priority with each process:
  - An integer: lower number is higher priority.
- Schedule the process with the highest priority first.
  - Can be pre-emptive or non-pre-emptive;
- SRTF is a special case of priority scheduling where the priority is equal to the remaining execution time of the process.

Non-pre-emptive Priority Scheduling

Add jobs to at the appropriate position in the ready queue according to their priority.

Process	Arrival	Duration	Priority
P1	2	2	3
P2	0	4	4
P3	0	3	0
P4	6	5	2



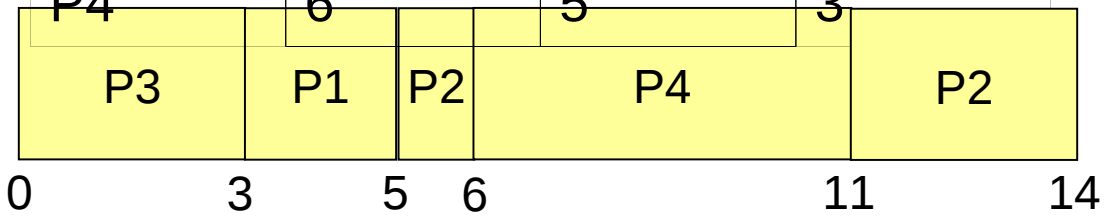
Average Waiting Time = (1 + 5 + 0 + 3) / 4 = 2.25

Average Turnaround Time = ((5 – 2) + (9 – 0) + (3 – 0) + (14 – 6) / 4 = 5.75

Pre-emptive Priority Scheduling

If a process arrives with higher priority than the one currently executing, execute it; otherwise insert it in the appropriate position in the ready queue according to priority.

Process	Arrival	Duration	Priority
P1	2	2	7
P2	0	4	14
P3	0	3	0
P4	6	5	3



Average Waiting Time = (1 + 10 + 0 + 0) / 4 = 2.75
(notice individual waiting time vs priority)

Average Turnaround Time = ((5 – 2) + (14 – 0) + (3 – 0) + (11 – 6) / 4 = 6.25

Priority Scheduling: Advantages/Disadvantages

+ Short waiting times for high priority processes (e.g. interactive user programs).
+ Users (or admin) have some control over the scheduler.
+ Deadlines can be met by giving processes high priority.

− Risk of starvation for low priority processes; or alternatively long waiting times.
  + Can alleviate starvation risk by using an aging scheme: gradually increase the priority of a process over time.
− Multiple context switches for each process if pre-emptive.
− Overheads in inserting processes in a sorted queue.

Round Robin Scheduling (RR)

- Features a ***time quantum***: the length of time each process is allowed to run for (usually 10-100ms);

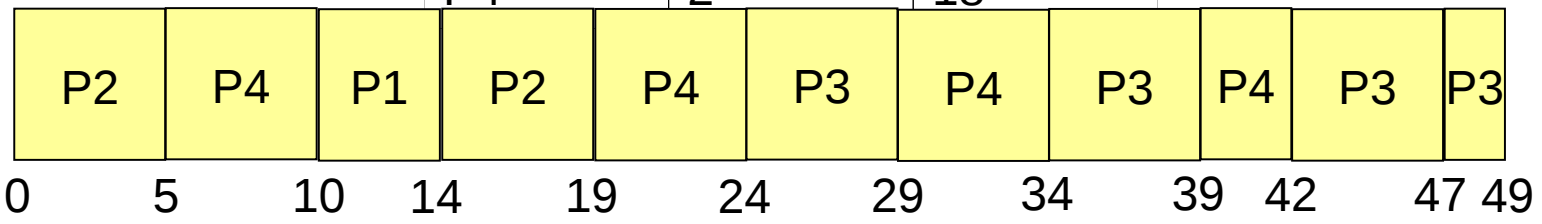
- When a process has run for this length of time it is pre-empted and another process from the ready queue can be scheduled.

- Performance varies according to q:
  - If q is large round robin tends to First Come First Served;
  - If q is small, context switching overheads become significant.
  - A heuristic for efficiency: a quantum that is longer than 80% of CPU bursts should be chosen.

Round Robin (q = 5)

*Take the first job from the queue (FCFS) and schedule it to run for q units. Then place it on the **back** of the queue. New processes join the **back** of the queue **on arrival**.*

Process	Arrival	Duration
P1	4	4
P2	0	10
P3	17	17
P4	2	18



Average Waiting Time = (6 + 9 + 15 + 22) / 4 = 13

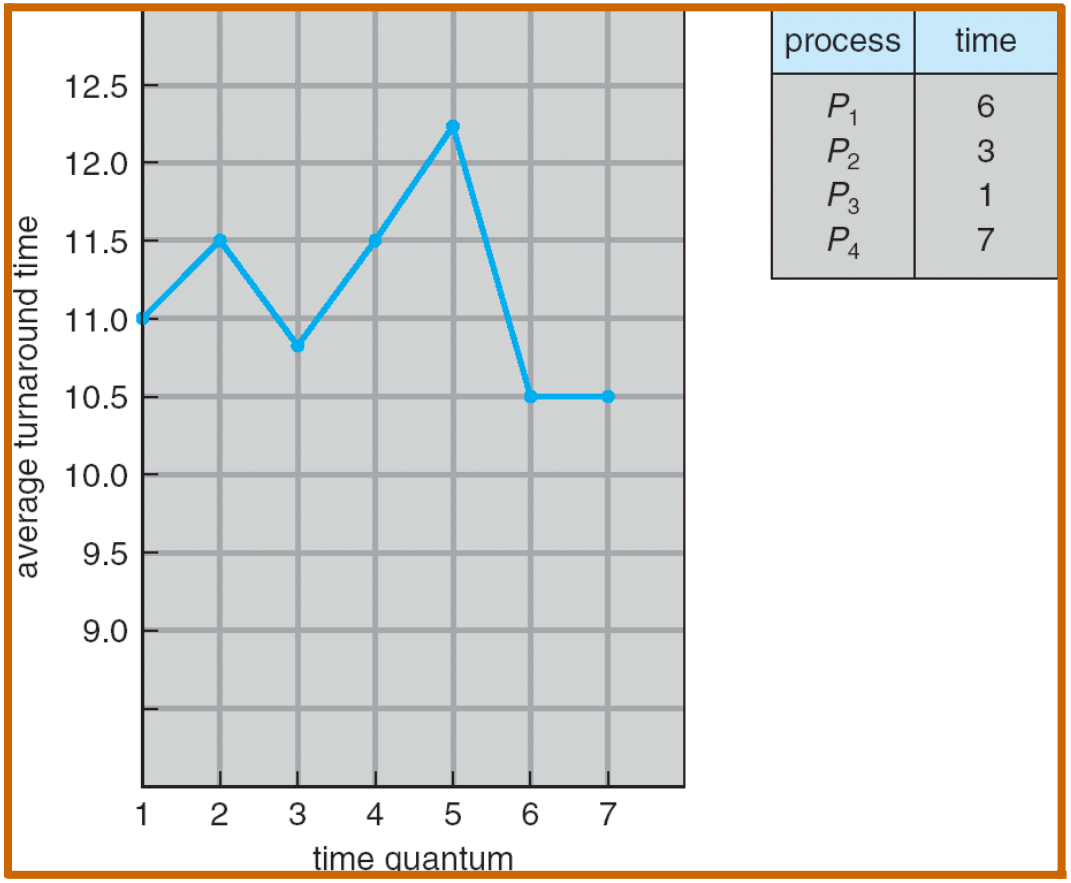
Average Turnaround Time = ((14 – 4) + (19 – 0) + (49 – 17) + (42 – 2) / 4 = 25.25

Round Robin: Advantages/Disadvantages

+ No Starvation.
+ Each process must wait no more than (n-1) * q time units before beginning execution.
+ Fair sharing of CPU between processes.

− Poor average response time
− Waiting time depends on number of processes, rather than priority.
− Particularly large number of context switches for each process so large overhead;
− Hard to meet deadlines because waiting times are generally high.

Quantum vs Average Turnaround Time



There is no monotonic relationship between quantum and average turnaround time. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

Multi-Level Queue Scheduling

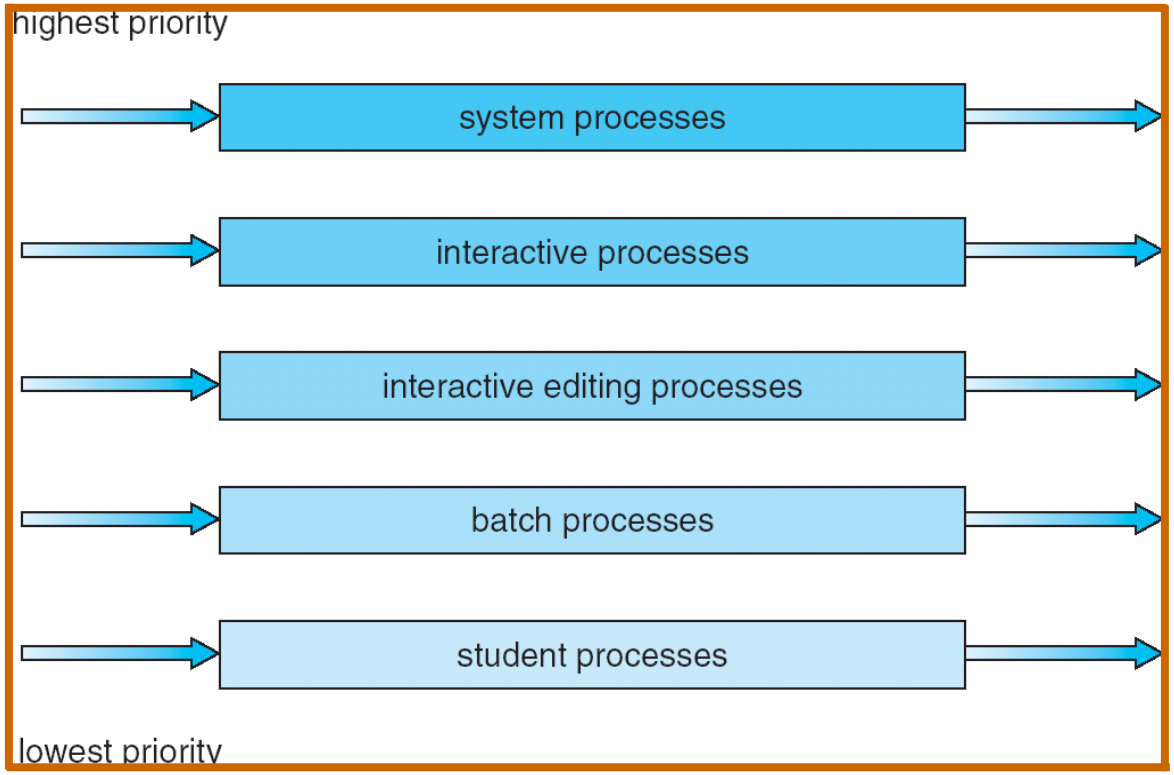
- If processes can be divided into groups we can use multi-level queue scheduling:
  - e.g. 1 finite number of priorities, one queue for each priority level;
  - e.g. 2 a queue for foreground processes (e.g. web browser) and one for background processes (e.g. disk defragmenter).
  - Different response time needs => different scheduling needs.

- Idea: Partition the ready queue in to several queues:
  - Each processes is assigned to a specific queue;
  - Each queue has a different scheduling algorithm:
    - e.g. foreground uses RR, background uses FCFS.
  - Also need a scheduling algorithm to choose between the queues:
    - Typically fixed priority pre-emptive.
    - Sometimes foreground may have absolute priority: background processes only run when the foreground queue is empty.

Multi-Level Queue Examples

Scenario 1:

- Each queue has absolute priority over lower priority queues
- e.g. no process in the batch queue can run unless the queues above it are empty
- This can result in starvation for the processes in the lower priority queues



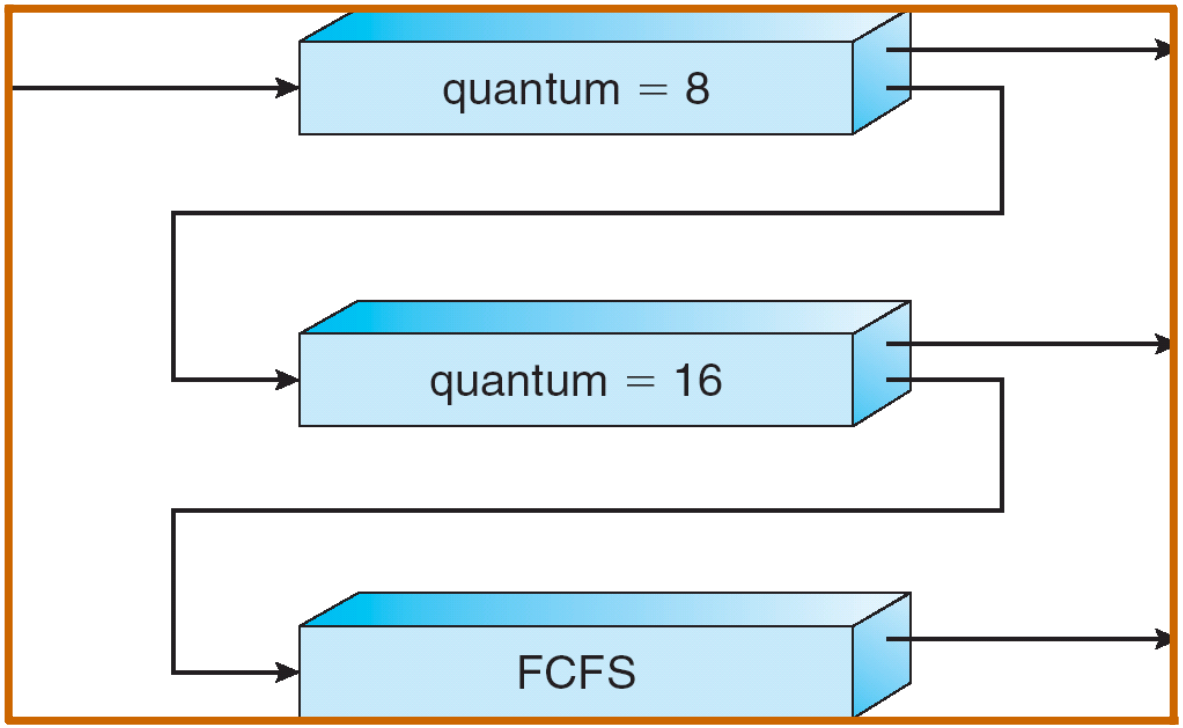
Scenario 2:

- Time slice amongst queues;
- Each gets a fixed percentage of CPU time.
- e.g.
  - 40% System
  - 30% Interactive
  - 20% Interactive Editing
  - 6% Batch
  - 4% Student
- Again each queue can use its own scheduling algorithm.

Multi-Level Feedback Queue Scheduling

- Like Multi-Level Queue Scheduling but processes can move between queues;
- Can use this to implement ageing in priority scheduling;
- In addition to the queues and their scheduling algorithms we need:
  - Mechanism for deciding when to promote processes
  - Mechanism for deciding when to demote processes;
  - Mechanism for deciding which queue a process will join initially.

Multi-Level Feedback Queue Example



- All new jobs enter the top queue (RR q=8);
- If not complete after 8 time units of execution they move to the next (RR q = 16);
- If still not complete after 16 time units of execution the move to the final queue which operates FCFS.

Priority Scheduling via MLFQ

- Create n queues, one for each priority level;
  - Higher priority queues get allocated more CPU time.
- Each process enters in the queue corresponding to its priority;
- After a process has been in the system for some time, its priority is increased so that it moves to a higher priority queue.

Multi-Processor Scheduling

- Multiple CPUs can be used to share load
  - Problem becomes more complex if the CPUs have different capabilities (e.g. certain jobs can only run on some CPUs).
  - Here we focus on CPUs with the same capabilities.
    - Every process can run on any CPU.

- Two Variants:
  - Asymmetric multiprocessing (one processor controls all scheduling);
  - Symmetric multi-processing (each processor schedules itself).

Asymmetric vs Symmetric

- *Asymmetric multiprocessing (ASMP)*
  - One processor makes all scheduling decisions, and handles I/O processing, and other system activities
  - The other processors execute user code only.
  - The need for data sharing is reduced because only one processor handles all system processes.

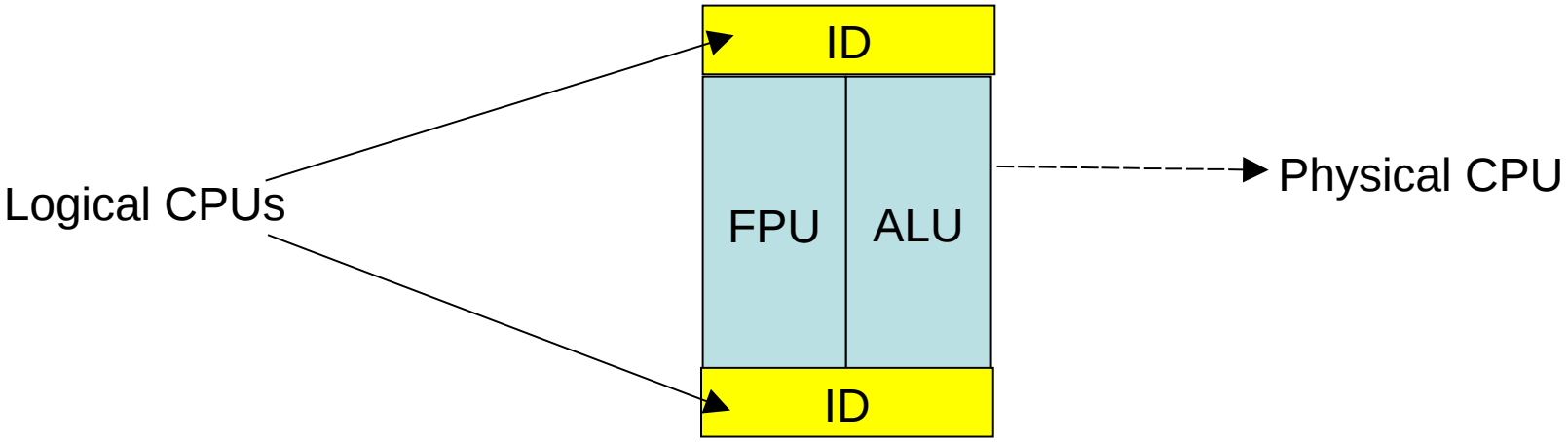
- Symmetric multiprocessing (SMP)
  - Each processor does its own scheduling.
  - There may be a single ready queue, or a queue for each processor.
  - Whichever is used, each processor selects a process to execute from the ready queue.
  - Efficient CPU use (high utilisation) requires load balancing for even distribution, two approaches:
    - **Push** migration: a specific process regularly checks the load on each processor and redistributes waiting processes.
    - **Pull** migration: an idle processor pulls a waiting job from the queue of a busy processor
  - All major modern operating systems support SMP.

Hyperthreading

- ***Multiprocessing*** uses multiple (physical) CPUs to run processes in parallel; ***Hyperthreading (AKA Symmetric Multithreading)*** *allows the same thing but using multiple* **logical** processors.

- Logical processors:
  - Share the hardware of the CPU: cache, busses etc.
  - Are responsible for their own interrupt handling.

- Appears as 2 CPUs instead of one: can do Floating point computations for one process, whilst doing Arithmetical/Logical computations for another.



Scheduling in Windows (XP)

- Pre-emptive priority scheduling, with time quanta. Priority levels are integers from 0 to 31 (unusually: higher number = higher priority).
- When a process is selected to run (highest priority ready thread) it executes, until either:
  - Its time quantum is used;
  - another higher priority process arrives;
  - It terminates;
  - It makes a system call e.g. I/O request;
- A queue is maintained for each priority level and a process is selected from the queue with the highest priority. If no thread is available for execution a special “system idle process” is executed.
- Pre-emption by higher priority threads ensures that real-time threads can access the CPU when required.

Priorities in Windows (XP)

- Processes start at the base priority of their class (inherited from parent);
- Priorities (excl. real time) are variable within their class.
  - When a process completes its time quantum its priority is lowered (never below the base priority for its class). Prevents CPU bound threads from hogging CPU.
  - When a process completes waiting its priority is boosted (e.g. Keyboard I/O large boost; Disk I/O smaller boost). Gives good response for interactive processes.
- Gives good I/O device utilisation; whilst allowing CPU bound processes to use spare CPU cycles.
- Current active window also given a boost for responsiveness.
- Interactive processes being run by the user typically all boosted: generally given a quantum 3 times as long.

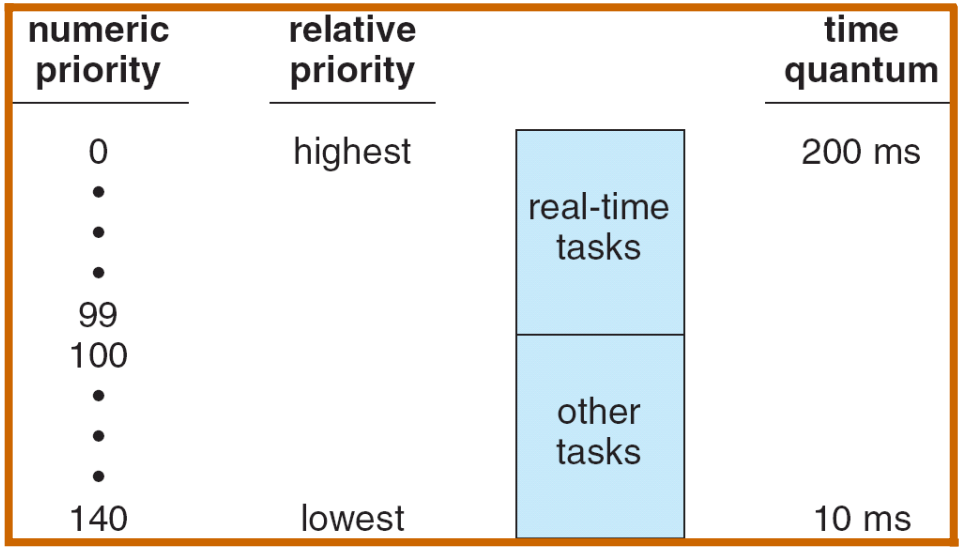
Win32 API Priority Class

Priority Within Class

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Scheduling in Linux (2.5)

- Linux uses a pre-emptive priority-based algorithm with two separate priority ranges:
  - A real-time range from 0 to 99
  - A nice value ranging from 100 to 140
- These two ranges map into a global priority scheme: lower values => higher priority.
- Higher priority tasks receive longer time quanta than lower priority ones.



Linux (2.5) Scheduling Algorithm

- A runnable task is eligible for execution if it has time remaining in its quantum;
- When a task has exhausted its quantum it is expired and not eligible for execution again until all other tasks have also exhausted their time quanta
- Because of its support for SMP, each processor maintains its own runqueue and schedules itself independently
- The scheduler selects the eligible task with the highest priority for execution;
- When all tasks have exhausted their time slices all tasks become eligible for execution.

Priorities in Linux (2.5)

- Real time processes have static priorities;
- All other tasks have dynamic priorities, based on their nice value and the number 5.
  - Interactivity of a task is determined by the time spent sleeping waiting for I/O;
  - I/O bound processes will sleep for more time than CPU bound processes.
  - For highly interactive processes (lots of I/O) 5 will be subtracted from their priority, thus it is increased;
  - For processes with very little interaction (CPU bound) 5 is added to their priority and thus it is lowered.
- Priority is recomputed on expiration of the time quantum.

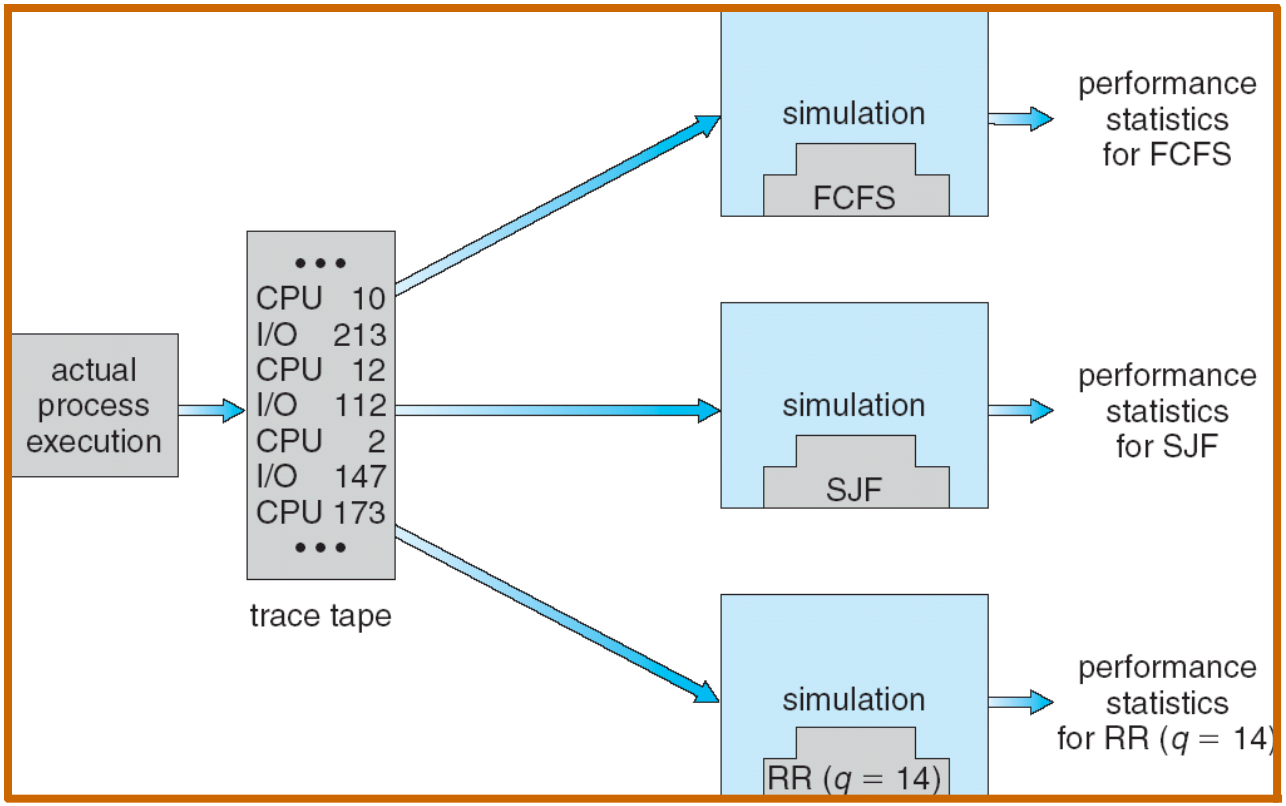
Scheduling Algorithm Evaluation

- **Deterministic modelling:**
  - Take a set of predetermined processes and run through the algorithm (like we did above).

- **Queueing Model:**
  - Use queueing theory to predict performance;
  - Knowing the ***arrival rates*** and service rates, we can compute utilisation, average queue length, average wait time, etc.
  - Little's formula **(*n* = λ x W):**
    - Queue length (n), arrival rate (λ), waiting time (W). e.g.
    - 7 processes arrive per second;
    - Queue length normally 14 processes;
    - Therefore average waiting time is 2 seconds.
  - **Do not use this formula for calculating average waiting times unless given an arrival rate, if given a set of processes use deterministic modelling.**

- **Problems with these approaches:** unrealistic, doesn't work with real processes; hard to model complex algorithms using queueing theory; arrival rate and waiting time are hard to represent accurately.

Evaluation via Simulation



Evaluation via Implementation

- The only way to accurately evaluate an algorithm is to implement it in the OS.
  - Lots of work in modifying the OS.
  - Users suffer inconsistent performance.
  - New processes may emerge that affect performance.
  - Users may adapt to ‘cheat’ scheduler, e.g. use interactive processes if they are favoured.
- The ideal is to have a scheduling algorithm that can be configured by the system administrator.
  - Allows tailoring to the specific system requirements: web server vs graphics workstation.
- Alternative is APIs that can allow users to change thread priorities:
  - Best performance for one application/system is not necessarily best for the system as a whole.

Overview

- Introduction to OSC.
- What is a Process?
- Process Scheduling.
- Operations on Processes.

Child Process Creation

- A process can create child processes (becoming their parent process) which, can themselves create other processes, forming a process tree.
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until some or all of its children have terminated
- Address space options
  - Child process is a duplicate of the parent process (same program and data)
  - Child process has a new program loaded into it

Creating Child Processes in Java: Process Builder

http://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html

```
public void execute(ArrayList<String> commmand) {
  try{
    ProcessBuilder builder = new ProcessBuilder(command);
    Map<String, String> environ = builder.environment();
    Process p = builder.start();
  } catch (Exception e){
    e.printStackTrace();
  }
}
```

```
ArrayList<String> commands = new ArrayList();
command.add(“explorer.exe”);
command.add(“myHelpFile.pdf”);
execute(commands);
```

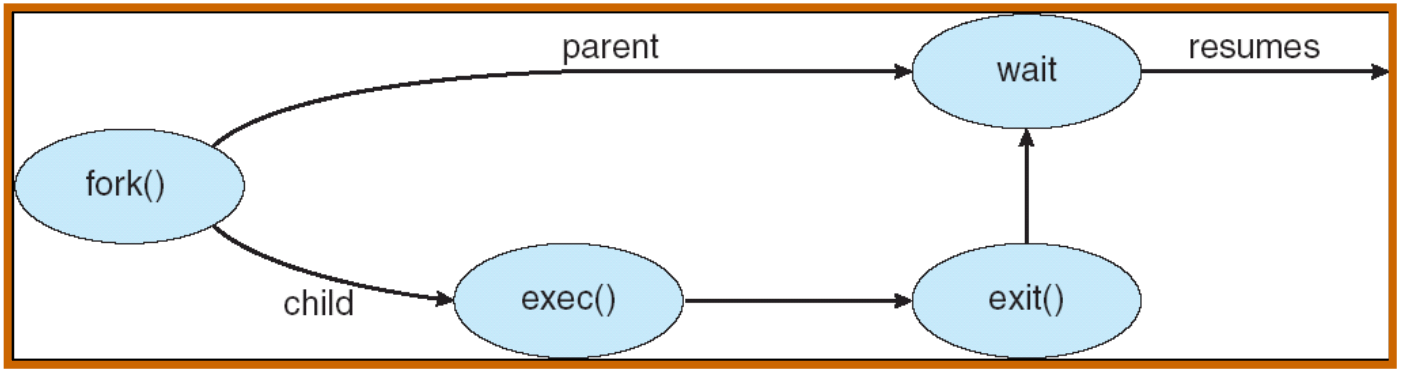
Creating Child Processes in Java: Process Builder

```
public void executeAndPrint(ArrayList<String> commmand) {
  try{
    ProcessBuilder builder = new ProcessBuilder(command);
    Map<String, String> environ = builder.environment();
    Process p = builder.start();
    p.waitFor();   //or not…
    BufferedReader br = new BufferedReader(new InputStreamReader(
                        p.getInputStream()));
    String line;
    while ((line = br.readLine()) != null){
    System.out.println(line);
    }
  } catch (Exception e){
    e.printStackTrace();
  }
  System.out.println("Program terminated!");
}
```

```
ArrayList<String> commands = new ArrayList();
command.add(“cat”);
command.add(“PawsTooSlippy”);
executeAndPrint(commands);
```

Process Creation in Unix

- fork() system call creates a new process
- exec() system call used after a fork() to replace the memory space of the process with a new program
- Can use these commands directly in C or implicitly using the above Java.



Process Termination

- Process executes last statement and asks the operating system to terminate it (**exit**)
  - Exit status value from child is received by the parent (via wait())
    - Indicates success or abnormal exit.
  - Process' resources are deallocated by operating system

- Parent may terminate execution of child processes (**kill**())
  - If a child has exceeded allocated resources;
  - If the task assigned to child is no longer required;
  - At will.

- If parent is exiting
  - Some operating system do not allow child to continue if its parent terminates;
  - All children terminated - cascading termination.

Overview

- Introduction to OSC.
- What is a Process?
- Process Scheduling.
- Operations on Processes.

Summary

- Introduction to OSC.
- What is a Process?
- Process Scheduling.
  - Introduction
  - Single Processor Scheduling Algorithms
  - Multi-processor Scheduling
  - Operating System Examples
- Operations on Processes.