ET0023 Operating Systems

5. Process Management & Scheduling

Process Management & Scheduling

- In a multiprogramming or multitasking system processes will be competing for
  - System resources
  - Memory
  - Files (I/O devices)
- OS has to manage these resources in a fair and efficient manner
- OS has to provide for Process Management & Scheduling

Process Creation

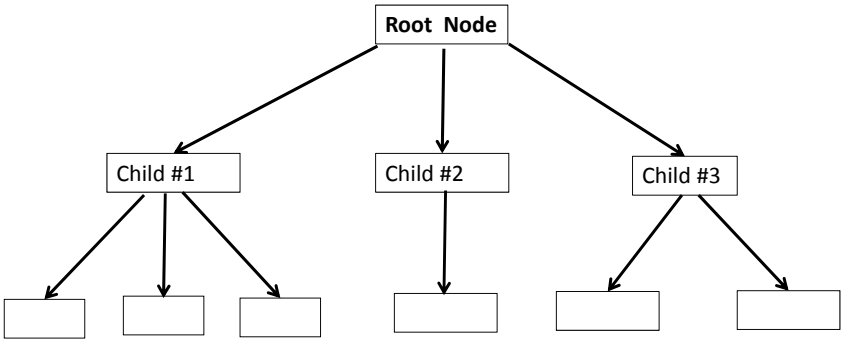
To create a new process, the OS has to

1. Name the process
2. Create a new Process ID and Process Control Block
3. Locate the program to be executed on disk and allocate memory for the code segment in RAM
4. Load the program into code segment, initialize PCB registers
5. Prioritize the process (default values)
6. Schedule the process for execution.

Process Creation

- Using CLI to create a process, new process becomes child of the command interpreter.
- Processes form a hierarchy and are linked by a tree structure.
- If Parent is signalled or killed, usually all Children received the same signal and are destroyed together with parent.
- When a Child is created, the process may
  - Duplicate the Parent process
  - Load a completely new program
- The Parent may also
  - Continue executing along side its Children
  - Wait for some/all to finish before proceeding

Process Hierarchies



Process Termination

- The process after creation, starts running and completes its job.
- The process must now terminate.
- Termination is usually one of the following
  1. Normal exit (voluntary)
  2. Error exit (voluntary)
  3. Fatal error (involuntary)
  4. Killed by another process (involuntary)

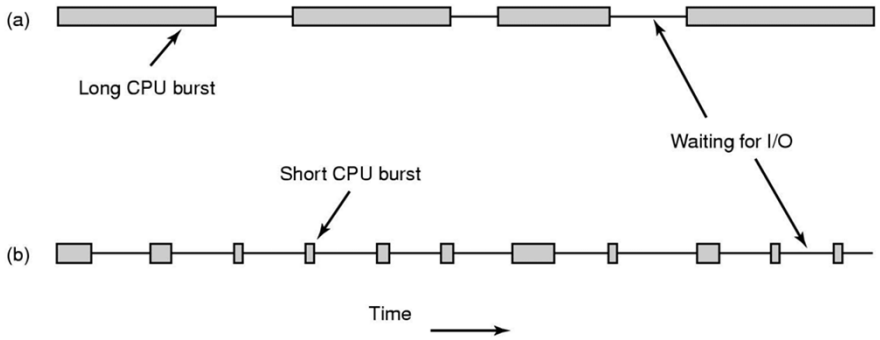
Process Scheduling

- To keep the CPU as busy as possible, the OS must perform scheduling.
- When a process blocks, it should be replaced.
- To let a process run for a maximum amount of time, an interrupt must be in place.
- Any stopped process must have its PCB saved and the next PCB restored.
- Process switching is expensive, hence the scheduler must be efficient.

CPU-I/O Burst Cycle

- All processes alternate bursts of computing (CPU cycles) with I/O requests.
- I/O wait is when a process enters the blocked state waiting for an external device to complete its work.
- CPU-bound: processes that use more time computing
- I/O-bound: processes that use more time waiting for I/O

How does a Process behave?



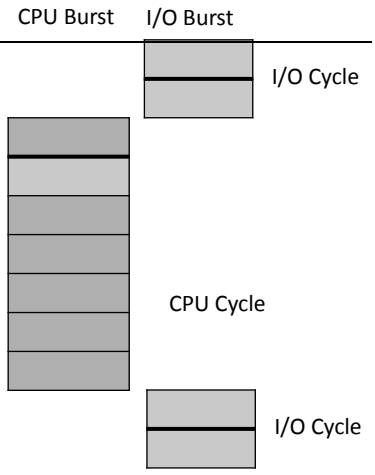
Bursts of CPU usage alternate with periods of IO wait

(a) A CPU-bound process

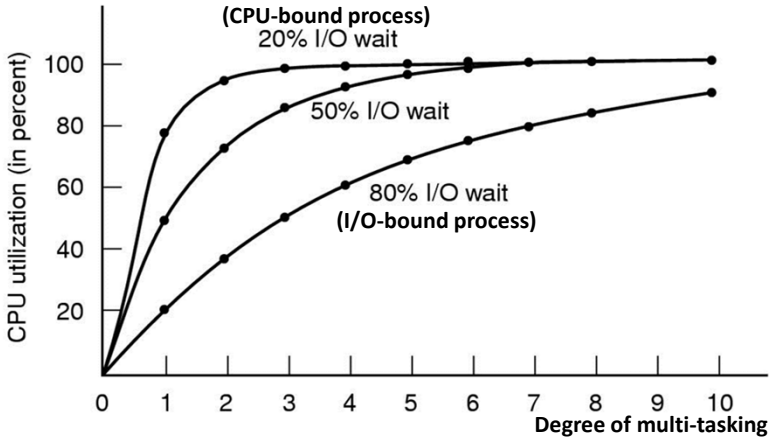
(b) An IO-bound process

Example

```
cout << "Enter sides: ";
cin >> a >> b >> c;
if (a == 0) {
cout << "Error!";
return -1;
}
sqrv = b*b - 4*a*c;
root1 = (-b + sqrv) / 2*a;
root2 = (-b - sqrv) / 2*a;
cout << "Root1 = " << root1;
cout << "Root2 = " << root2;
```



Modelling Multitasking



CPU utilization as a function of number of processes in memory

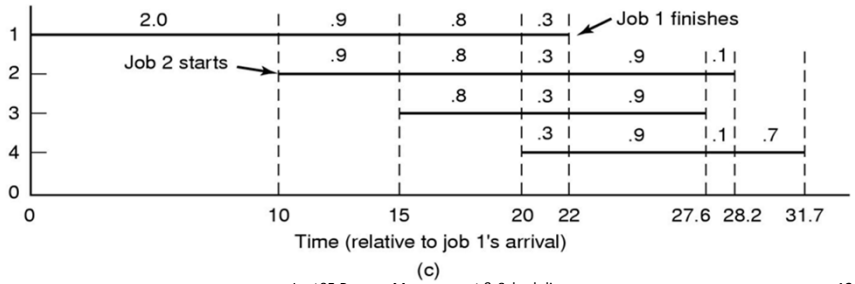
Analysis of Multi-Tasking Performance

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

Process Scheduling Policy

- An algorithm the Policy Scheduler uses to determine
  - How much CPU time does a process get
  - Who gets the process next (Priority)
  - When the process will be interrupted

Scheduling Criterion

- CPU Utilization: Objective to keep CPU as busy as possible
- Throughput: How many processes are completed per time unit
- Turnaround time: How long it takes to execute a process (from submission to completion)
- Waiting time: The time a process has to wait before execution starts
- Response time: Time from submission to obtaining the first results

Example

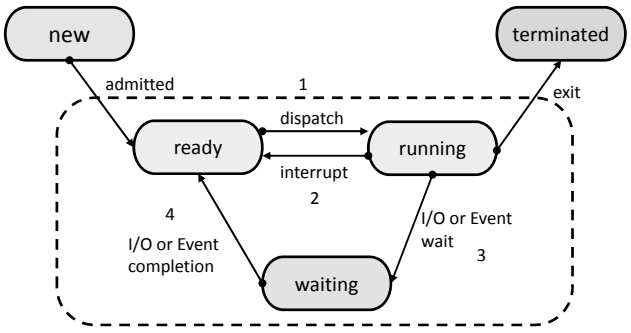
Four processes are submitted at time 0

P1	Waiting time	0
	Turnaround time	3
P2	Waiting time	3 + 3 = 6
	Turnaround time	9
P3	Waiting time	4 + 4 = 8
	Turnaround time	10
P4	Waiting time	5
	Turnaround time	7
Average waiting time		(0+6+8+5)/4 = 4.75
Average turnaround time		(3+9+10+7)/4 = 7.25

Process Scheduling Policy

- A good algorithm should
  - Maximize throughput
  - Minimize response time
  - Minimize turnaround time
  - Minimize waiting time
  - Maximize CPU efficiency
  - Ensure fairness for all processes

Process States



1 Scheduler picks this process
2 Scheduler picks another process
3 Process blocks for input
4 Input becomes available

Scheduling Algorithms

- A non-preemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on IO or waiting for another process) or until it voluntarily releases the CPU.
  - Does not require a clock.
- A preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of time interval, it is suspended and the scheduler picks another process to run (if one is available).
  - Requires a clock.

Scheduling Environments

- Different environments require different scheduling algorithms
  - Batch → Background
  - Interactive → Foreground
  - Realtime → Multimedia
- There is no one size fits all!

Scheduling Goals

- All systems
  - Fairness – giving each process a fair share of the CPU
  - Policy enforcement – seeing that stated policy is carried out
  - Balance – keeping all parts of the system busy
- Batch systems
  - Throughput – maximize jobs per hour
  - Turnaround time – minimize time between submission and termination
  - CPU utilization – keep the CPU busy all the time
- Interactive systems
  - Response time – respond to requests quickly
  - Proportionality – meet user's expectations
- Real-time systems
  - Meeting deadlines – avoid losing data
  - Predictability – avoid quality degradation in multimedia systems

The "Perfect" Scheduler

- Minimize Latency
  - Response/Job completion time
- Maximize Throughput
  - Maximize job/time
- Maximize Utilization
  - Keep all devices busy
- Fairness
  - Every job makes progress, no job starves

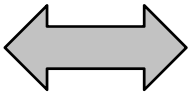
Problem Analogy

- Imagine you are the cook at a "Chi Char" stall
  - Customers continually enter and place their orders
  - Dishes take varying amounts of time to prepare
- What is your goal?
- What strategy achieves your goal?

"Chi Char Stall" vs Multitasking

Chi Char Stall

- Waitress
- Cook
- Client
- Order
- Dish



Multi-tasking OS

- CPU Scheduler
- CPU
- User
- Process
- CPU or I/O Burst

CPU / I-O Burst

- A Process alternates between CPU and I/O Bursts

CPU bound: Long CPU bursts

Matrix Multiply

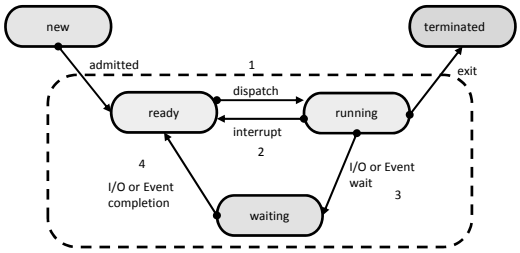
I-O bound: Short CPU bursts

Word Processor software

During an I/O burst = process is idle, switch to another "for free"

CPU Scheduler

- Processes and Threads migrate among queues (Ready, Device Wait, Run)
- Scheduler selects one from the ready queue to run
  - Which one?
  - When?



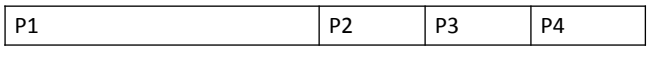
First Come First Served

- Jobs are scheduled in order of arrival
- Non-preemptive
- Problem: Average wait time depends on arrival order

P1 16
P2 4
P3 5
P4 5

Arrival order: P1,P2,P3,P4

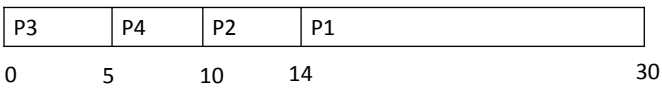
I



0 16 20 25 30

Arrival order: P3,P4,P2,P1

II



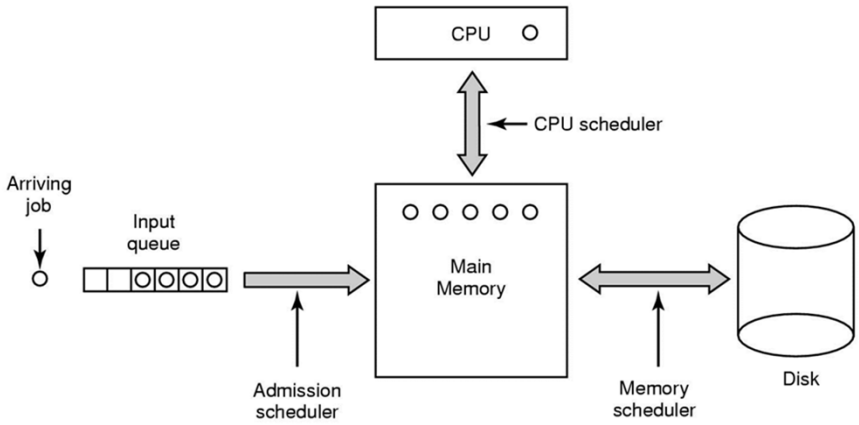
0 5 10 14 30

- Advantage: Really Simple!!

Batch Systems

- As jobs arrive at the system, they are initially placed in an input queue stored on the disk.
- The admission scheduler decides which jobs to admit to the system.
- The memory scheduler determines which processes are kept in memory and which on the disk.
- The CPU scheduler picks one of the ready processes in main memory to run next.

Scheduling in Batch Systems



Last-In-First-Out ?

- Newly arrived jobs are placed at the head of ready queue.
- Improves response time for newly created jobs

- Problem:
  - May lead to starvation (early processes may never get any CPU time!)

Round Robin

- FCFS with preemption
  - Often used for timesharing
  - Ready queue is treated as a circular queue (FIFO)
  - Each process is given a time slice (quantum)
  - Process runs for the quantum or until blocked
  - RR allocates the CPU uniformly (fairly) across participants
  - If average queue length is n, each quantum is 1/n

Round Robin

Process Burst Time (sec)

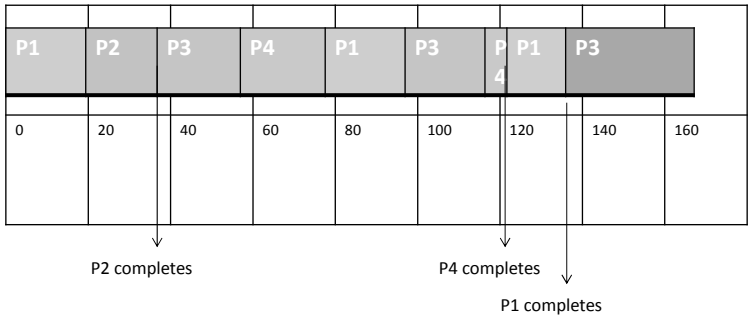
P1 53

P2 17

P3 68

P4 24

Time Quantum = 20

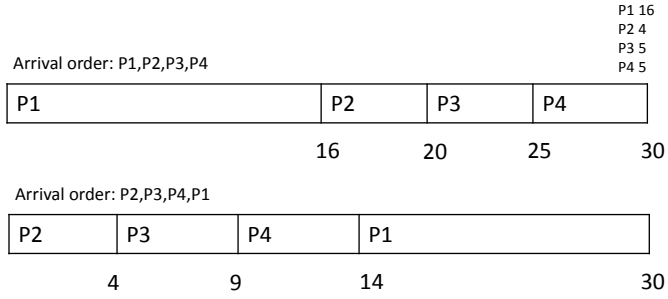


RR: Choice of Time Quantum

- Performance depends on length of time-slice
  - Context switching is not free
  - If time-slice is set too high
    - FCFS
    - Process will finish or block before their slice is up
  - If time-slice is set too low
    - Overhead is on the context switching between processes
- Usual Practice
  - Context switch is usually negligible (<1% per timeslice)
  - Context switch too often – lose all productivity

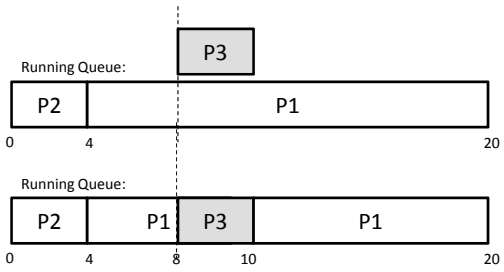
Shortest Job First

- Choose the job with the shortest next CPU burst.
- Optimal for minimizing average meeting time
- Problem: Impossible to know the length of next CPU burst.



Preemptive Shortest Job First

- SJF can be preemptive or non-preemptive
- When new job arrives and current job has long time to execute.
- Preemptive SJF = Shortest Remaining Time First (SRTF)



Priority Scheduling

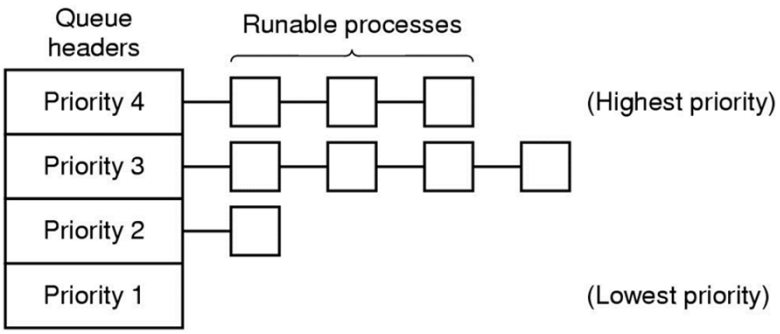
- Choose next job based on priority
- For SJF, priority = expected CPU burst
- Can be preemptive or non-preemptive
- Can approximate to other scheduling algorithms
  - P (arrival time) FIFO
  - P (now – arrival time) LIFO
  - P (job length) SJF
- Problem:
  - Starvation: jobs can wait indefinitely
  - Solution: Age processes, increase priority as a function of waiting time.

Priority Scheduling

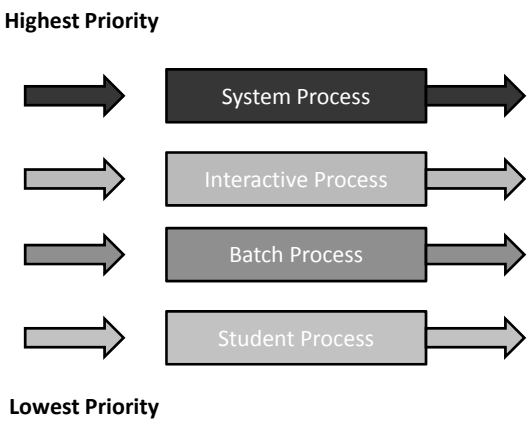
- Can use external characteristics of the process to assign priorities
  - Position of user (e.g. root/sysadmin)
  - Cost ($$$ - he who pays gets first)
- Can use internal characteristics of the process to assign priorities
  - Memory requirements
  - Number and type of peripheral devices
  - Estimated CPU time
  - Amount of time already spent in the system

Priority Scheduling using Multiple Queues

- Each process is assigned a priority, and the runnable process with the highest priority is allowed to run.
- Idea used by Windows and Linux.



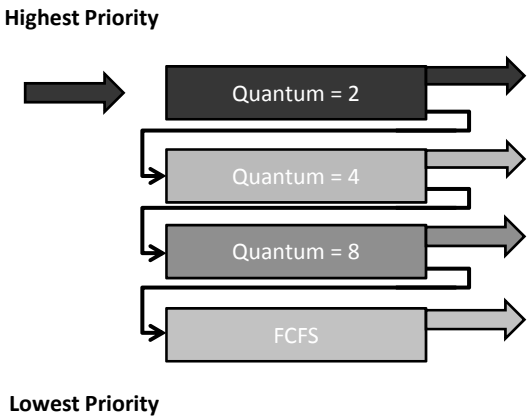
Multilevel Queue Scheduling



Multilevel Queue Scheduling

- Implement multiple ready queues based on job "type"
  - System processes
  - Interactive processes
  - Student programs
  - CPU-bound processes
  - Batch jobs
- Different queues may be scheduled using different algorithms
- Intra-queue CPU allocation is either **strict** or **proportional**
- Problem: classifying jobs into queues is difficult.

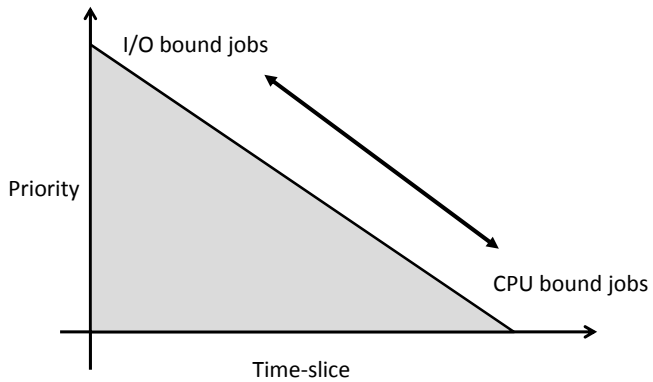
Multilevel Feedback Queues



Multilevel Feedback Queue

- Implement multiple ready queues
  - Different queues may be scheduled using different algorithms
  - Similar to multilevel queue scheduling but assignments are not static
- Jobs move from queue to queue based on feedback
  - Feedback = Behaviour of the job
  - e.g. full quantum for computation, or frequent I/O
- Very general algorithm, select parameters for
  - Number of queues
  - Scheduling algorithm for each queue
  - When to upgrade or downgrade a job

A Multi-level System



Real-time Scheduling

- Real-time processes have timing constraints, Expressed as
  - Deadlines
  - Rate requirements
- Common misconception:
  - Real time does not mean fast.

Real-time Scheduling

- Common RT scheduling policies
  - Rate monotonic
    - Just one scalar priority related to productivity of job
    - Priority = 1/rate
    - Static
  - Earliest Deadline First (EDF)
    - Dynamic but more complex
    - Priority = Deadline
- Both require admission control to provide guarantees

Real-Time System Scheduling

- Hard real-time
  - There are absolute deadlines that must be met, or else!
- Soft real-time
  - Missing an occasional deadline is undesirable, but nevertheless tolerable.

Thread Scheduling

- All thread share code & data segment
- Option 1: Ignore this fact
- Option 2: Two-level scheduling
  - User-level scheduling
  - Schedule process, and within each process, schedule threads
  - Reduce control switching overhead and improve cache hit ratio

Multiprocessor Scheduling

- Option 1: Ignore this fact
- Option 2: Space based affinity
  - Assign threads to processes
  - Control resource sharing
- Option 3: Gang scheduling
  - Run all threads belonging to a process at the same time
  - Low-latency communication
  - Greater distance (scheduling)

Problem Cases

- Blindness about job types
  - I/O goes idle
- Optimization involves favouring jobs of type "A" over "B"
  - Lots of "A"s, then "B" starve
- Interactive process trapped behind others
  - Response time suffers for no reason
- Priorities: A depends on B, A's priority > B's
  - B's never run

Convoy Effect

- CPU bound job will hold CPU until done, or it causes an I/O burst (rare)
  - Long periods where no I/O requests are issued, CPU is held
  - Result: Poor I/O device utilization
- Example: One CPU bound, many I/O bound
  - CPU-bound runs, I/O devices idle
  - CPU-bound blocks
  - I/O bound job(s) run, quickly blocks on I/O
  - CPU-bound runs again
  - I/O completes
  - CPU-bound still runs while I/O devices idle

- QUESTIONS