Western

UNIVERSITY · CANADA

Chapter 5b – Algorithms

Spring 2023

Western

Algorithms

- https://en.wikipedia.org/wiki/Scheduling_(computing)
- How does the scheduler choose which process in the ready queue to run next?
  - CPU utilization – Can we keep the CPU(s) as busy as possible?
  - Throughput – How many processes can we run in a time frame? The more the merrier
  - Turnaround time – How long are processes taking? From **New** to **Terminated**
  - Waiting time – How long are processes waiting in the ready queue
  - Response time – How long are processes taking to just start giving responses?

Algorithms

- Fairness – The general goal is to:
    - Maximize CPU utilization
    - Maximize throughput
    - Minimize turnaround time
    - Minimize waiting time
    - Minimize response time
- Some systems may value some criteria more than others

0 CPU time

- No matter what, we don't want any processes to be starved out (never gets run!)

Algorithms

- First-Come, First-Served Scheduling (FIFO) (Queue)
- Last-Come, First Served Scheduling (LIFO) (Stack)
- Shortest-Job-First Scheduling
- Round-Robin Scheduling
- Priority Scheduling
- Lottery Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

First-Come, First-Served

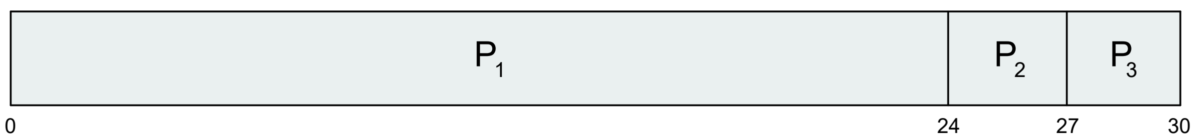
- The first process to enter the ready queue gets to run first
- A process continues to run until it needs I/O or until it terminates
  - Is this preemptive or nonpreemptive?
- Pro – The simplest algorithm to understand and implement
- Con – The waiting time can be long

First-Come, First-Served

- Consider the following processes in the following order:

P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is:



- Waiting time for **P_1 = 0**, **P_2 = 24**, **P_3 = 27**. Average is **(0 + 24 + 27)/3 = 17**

- Turnaround time for **P_1 = 24**, **P_2 = 27**, **P_3 = 30**. Average is **(24 + 27 + 30)/3 = 27**

First-Come, First-Served

- Suppose we had a different order:

P_2	3
P_3	3
P_1	24

- The Gantt chart for the schedule is:

- Waiting time for $\mathbf{P_1 = 6}$, $\mathbf{P_2 = 0}$, $\mathbf{P_3 = 3}$. Average is **(6 + 0 + 3)/3 = 3**

- Turnaround time for $\mathbf{P_1 = 30}$, $\mathbf{P_2 = 3}$, $\mathbf{P_3 = 6}$. Average is **(30 + 3 + 6)/3 = 14**

First-Come, First-Served

- Suppose we had a different order:

P_2	3
P_1	24
P_3	3

- What does the Gantt chart look like?

- What is the waiting time for $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$? Average waiting time? $(0+3+27)/3=10$

- What is the turnaround time for $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$? Average turnaround time? $(3+27+30)/3=20$

First-Come, First-Served

- Summary:

	P_1, P_2, P_3	P_2, P_3, P_1	P_2, P_1, P_3
Average Waiting Time	17	3	(3 + 0 + 27)/3 = 10
Average Turnaround Time	27	14	(27 + 3 + 30)/3 = 20

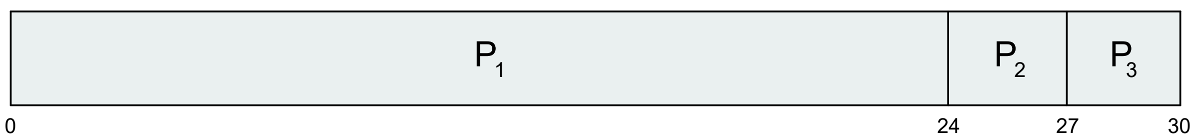
- **Convoy effect** – Short processes stuck behind long processes
  - We want to avoid this!

First-Come, First-Served

- We should be considering arrival time

Process	Arrival time	Burst time (milliseconds)
P_1	0	24
P_2	1	3
P_3	2	3

- The Gantt chart for the schedule is:



- Waiting time for **P_1 = 0**, **P_2 = 24-1 = 23**, **P_3 = 27-2 = 25**. Average is **(0 + 23 + 25)/3 = 16**

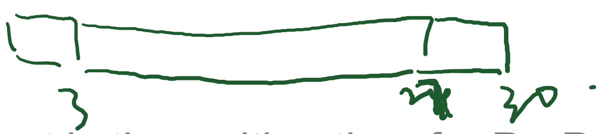
- Turnaround time for **P_1 = 24**, **P_2 = 27-1 = 26**, **P_3 = 30-2 = 28**. Average is **(24 + 26 + 28)/3 = 26**

First-Come, First-Served

- Suppose we had a different order

Process	Arrival time	Burst time (milliseconds)
P_2	0	3
P_1	1	24
P_3	2	3

- What does the Gantt chart look like?



- What is the waiting time for $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$? Average waiting time?

$[0 + (3-2) + (27-1)]/3$
$= 9$

- What is the turnaround time for $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$? Average turnaround time?

$[3+(27-1)+(30-2)]/3 = 19$

Last-Come, First Served

- The latest process to enter the ready queue gets to run first
- Pro – Improves response time for newly created processes
- Con – Risk of starvation. Some processes may never be picked up

Shortest-Job-First

- (More accurate to call it "Shortest-next-CPU-burst", since we are based off CPU bursts, not the entire job)
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - Generally, nonpreemptive but preemptive is, of course, possible
- Pro – If the CPU burst time is accurate, SJF is optimal. Guaranteed to provide the minimum average waiting time for a given set of processes

23333

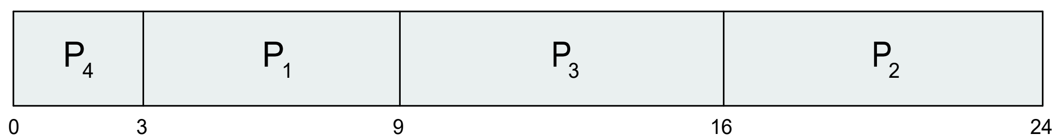
- Con – How do we determine the length of the next CPU burst? We could ask the user/developer to set it, or we could make an estimate based on previous history

Shortest-Job-First

- Consider the following processes

P_1	6
P_2	8
P_3	7
P_4	3

- What does the Gantt chart look like?



- Waiting time for **P_1 = 3**, **P_2 = 16**, **P_3 = 9, P_4 = 0**. Average is **(3 + 16 + 9 + 0)/4 = 7**
- Turnaround time for **P_1 = 9**, **P_2 = 24**, **P_3 = 16, P_4 = 3**. Average is **(9 + 24 + 16 + 3)/4 = 13**

Shortest-Job-First

- With arrival time

Process	Arrival time	Burst time (milliseconds)
P_1	0	6
P_2	1	8
P_3	2	7
P_4	3	3

- What does the Gantt chart look like?

- What is the waiting time for $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$, $\mathbf{P_4}$? Average waiting time?

- What is the turnaround time for $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$, $\mathbf{P_4}$? Average turnaround time?

Shortest-Job-First

- How do we estimate the next CPU burst? **Exponential smoothing** (or Exponential averaging)
  - It probably won't be the same as the previous CPU burst, but it will probably be similar

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Shortest-Job-First

- How do we estimate the next CPU burst?
  - Set α to 0 – The most recent history has no bearing. Pick an estimate and never update it

$$\tau_{n+1} = \tau_n$$

  - Set α to 1 – Only the most recent history has any bearing

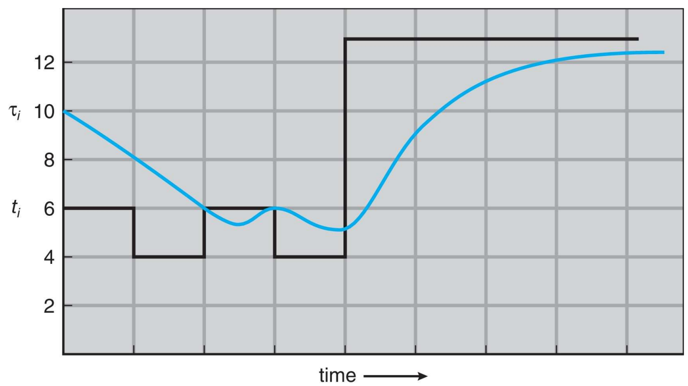
$$\tau_{n+1} = t_n$$

  - More commonly, set α to ½ so we get the most recent history and all the past history
    - Set α to 0.5 – 1.0 to put higher weight on recent history, 0 – 0.5 to put higher weight on past history

Shortest-Job-First

- How do we estimate the next CPU burst?
  - Pick any value for τ_0 to get us started (say, 10) set α to ½

$$T_{n+1} = \frac{1}{2} \times 6 + \frac{1}{2} \times 10$$
$$= 8$$
$$T_{n+1} = \frac{1}{2} \times 4 + \frac{1}{2} \times 8 = 6$$



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Shortest-Job-First

- How do we estimate the next CPU burst?
  - What if we give higher weight to the most recent burst and set α to 0.75

t_i		6	4	6	4	13	13	13
τ_0	10	4.5+2.5 7	3+1.8 4.8	4.5+1.2 5.7	3+1.4 4.4	9.8+1.1 10.9	9.8+2.8 12.6	9.8+3.2 13

  - What if we give higher weight to past history and set α to 0.25

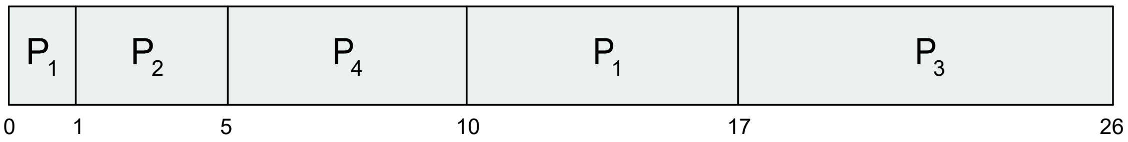
t_i		6	4	6	4	13	13	13
τ_0	10							

Shortest-Job-First

- With arrival time and preemption ("shortest-remaining-time-first")

Process	Arrival time	Burst time (milliseconds)
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- What does the Gantt chart look like? $\mathbf{P_1}$ has 7ms left but $\mathbf{P_2}$ has 4:



- **P_1 = 10-1, P_2 = 1-1**, **P_3 = 17-2, P_4 = 5-3** Average waiting time is 6.5
- **P_1 = 17-0, P_2 = 5-1**, **P_3 = 26-2, P_4 = 10-3**? Average turnaround is 13

Round-Robin

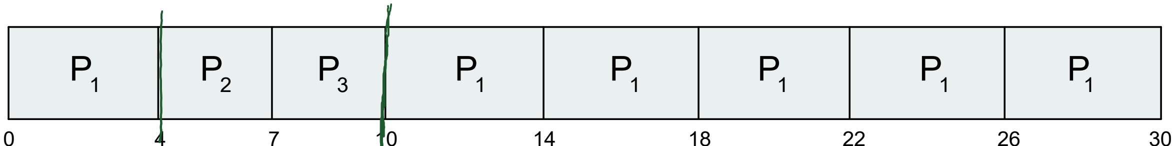
- Similar to FCFS scheduling except we add a preemption using a timer
- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Pro – If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.
- Con – If q is very high, this is just FCFS with extra steps! If q is very low, you may spend more time context switching than actually executing tasks! Typically, q ~10-100 milliseconds. Context switches <10 microseconds

Round-Robin

- Consider the following processes with time quantum 4

P_1	24
P_2	3
P_3	3

- What does the Gantt chart look like?

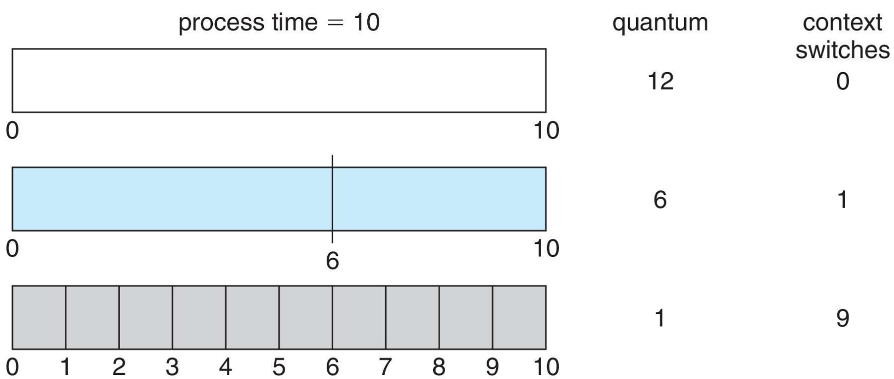


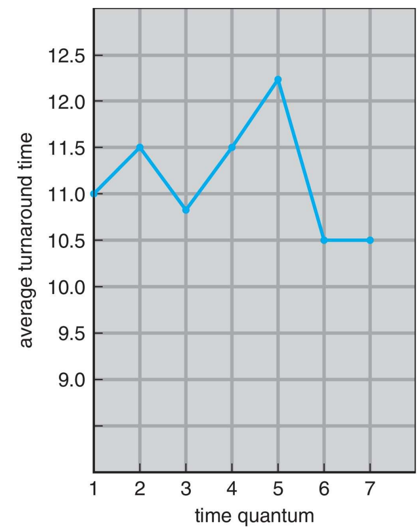
- Waiting time for **P_1 = 10-4**, **P_2 = 4**, **P_3 = 7**. Average is **(6 + 4 + 7)/3 = 5.66**

- Turnaround time for **P_1 = 30**, **P_2 = 7**, **P_3 = 10**. Average is **(30+7+10)/3 = 15.66**

Round-Robin

- A smaller quantum generally means more context switches which generally means longer turnaround time
- Ideally, ~80% of CPU bursts should be shorter than the quantum





process	time
P_1	6
P_2	3
P_3	1
P_4	7

Priority

- Assign a priority number (integer) to each process
- The CPU is allocated to the process with the highest priority (We will treat the smallest integer as highest priority, but this could be implemented the other way too)
- Generally, a preemptive algorithm but nonpreemptive could be possible
- FCFS is priority scheduling where all processes have the same priority
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time (The larger the CPU burst, the lower the priority, and vice versa)
- Pro – More control over CPU scheduling
- Con – **Starvation** is possible and needs to be accounted for

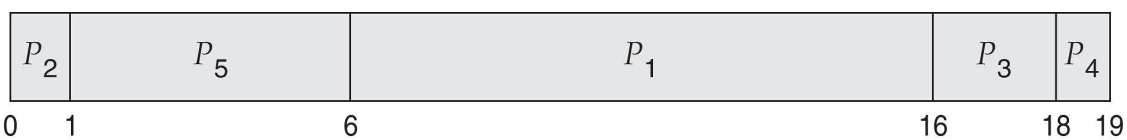
most resources are put to higher priority.

Priority

- Consider the following processes with the following priorities

Process	Burst time (milliseconds)	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- What does the Gantt chart look like?



- Average waiting time is 8.2. Average turnaround time is 12

Priority

- Consider the following processes with the following priorities with round-robin with time quantum 2

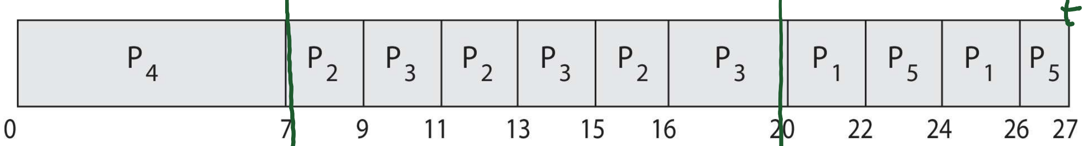
Process	Burst time (milliseconds)	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

if two processes have same priority, then it would run FCFS, then round-robin.

- What does the Gantt chart look like?

2

3.



Priority

- A process that is ready to run but may be in the ready queue indefinitely
  - It may EVENTUALLY run (e.g. 2am on a Sunday)
  - The computer may reboot or crash and the process may never get run
- Implement **aging** – As time progresses, increase the priority of the process

avoid of starvation.

Lottery

- The scheduler distributes "lottery tickets" to all the processes
- The scheduler then randomly selects a lottery number
- The process assigned the lottery number is the next process to run
- E.g. 100 tickets for 3 processes: Process A gets 50%, Process B gets 15%, Process C gets 35%
- Pro – If each process gets tickets, no process will ever starve get a chance to run
- Con – If there are many tickets for many processes, this may be inefficient

Multilevel Queue

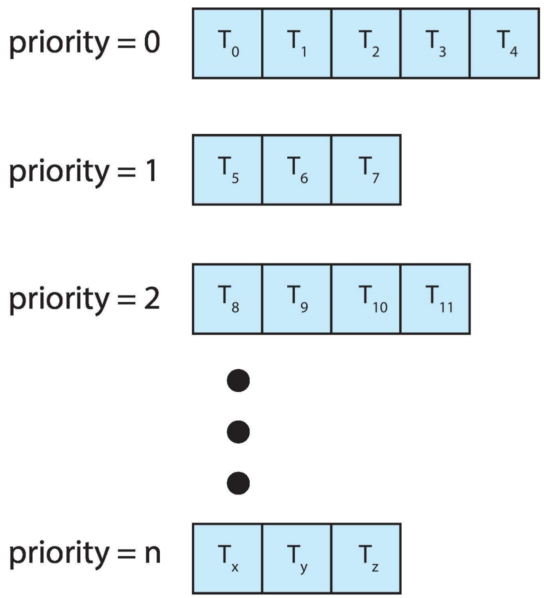
- The ready queue consists of multiple queues. This is how ready queues are usually implemented

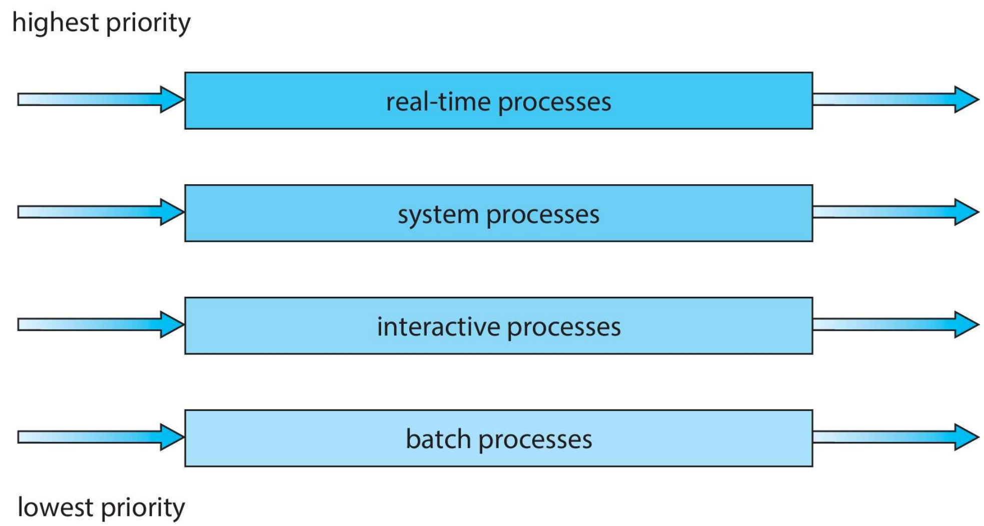
- Multilevel queue scheduler requires: A set of queues, a scheduling algorithms for each queue, a method used to determine which queue a process will enter when that process needs service, and scheduling among the queues

- Pro – The best of all worlds. Pick the most appropriate scheduling algorithm for the queue

- Con – Starvation is still possible. High priority processes may use up all the CPU time

Multilevel Queue



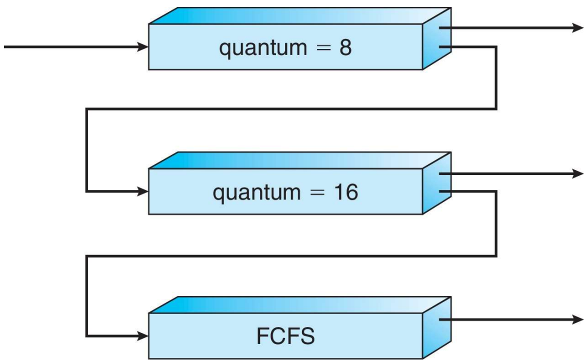


Multilevel Feedback Queue

- To address the issue of starvation in a multilevel queue, provide an aging mechanism so processes can be promoted to higher priority queues
- To address the issue of processes using too much CPU time, provide a mechanism to demote them to a lower priority queue

Multilevel Feedback Queue

- Consider an example with three queues
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- A new process enters queue Q_0 and receives 8 milliseconds
- If it does not finish in 8 milliseconds, the process is moved to queue Q_1
- At Q_1 job is again served in RR and receives 16 additional milliseconds
- If it still does not complete, it is preempted and moved to queue Q_2



Western

UNIVERSITY · CANADA