Scheduling algorithms

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

First-Come, First-Served (FCFS) Scheduling

- Process that requests CPU first, is allocated the CPU first
- Ready queue=>FIFO queue
- Non preemptive
- Simple to implement

Performance evaluation

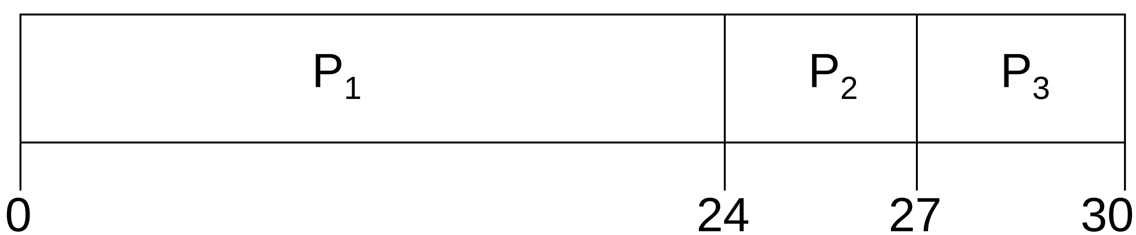
- Ideally many processes with several CPU and I/O bursts

- Here we consider only one CPU burst per process

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



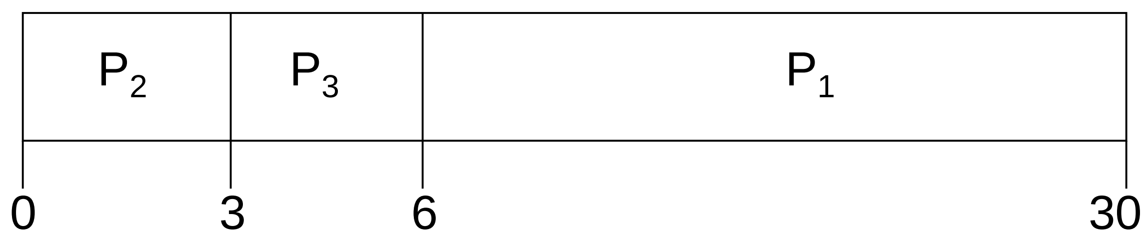
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- Average waiting time under FCFS heavily depends on process arrival time and burst time
- **Convoy effect** - short process behind long process
  – Consider one CPU-bound and many I/O-bound processes

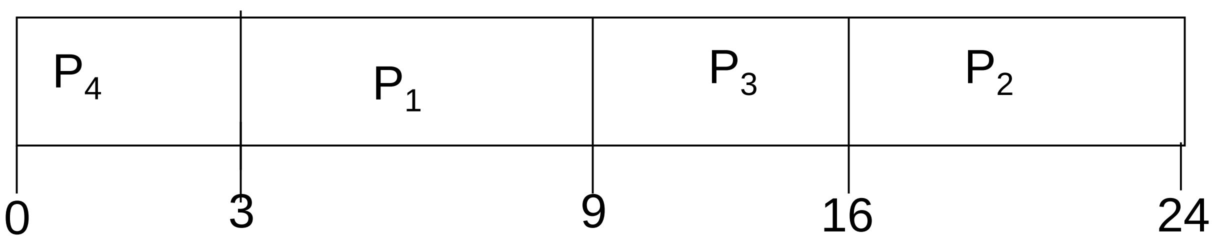
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Allocate CPU to a process with the smallest next CPU burst.
  - Not on the total CPU time
- Tie=>FCFS

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Avg waiting time for FCFS?

SJF

- SJF is optimal – gives minimum average waiting time for a given set of processes (Proof: home work!)
- The difficulty is knowing the length of the next CPU request
- Useful for Long term scheduler
  - Batch system
  - Could ask the user to estimate
  - Too low value may result in “time-limit-exceeded error”

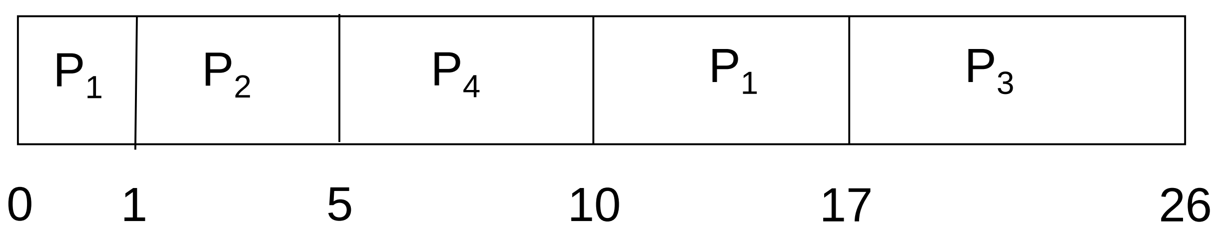
Preemptive version

Shortest-remaining-time-first

- Preemptive version called **shortest-remaining-time-first**
- Concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

Avg waiting time for non preemptive?

Determining Length of Next CPU Burst

- Estimation of the CPU burst length – should be similar to the previous burst
  - Then pick process with shortest predicted next CPU burst
- Estimation can be done by using the length of previous CPU bursts, using time series analysis

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

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

Boundary cases α=0, 1

- Commonly, α set to ½

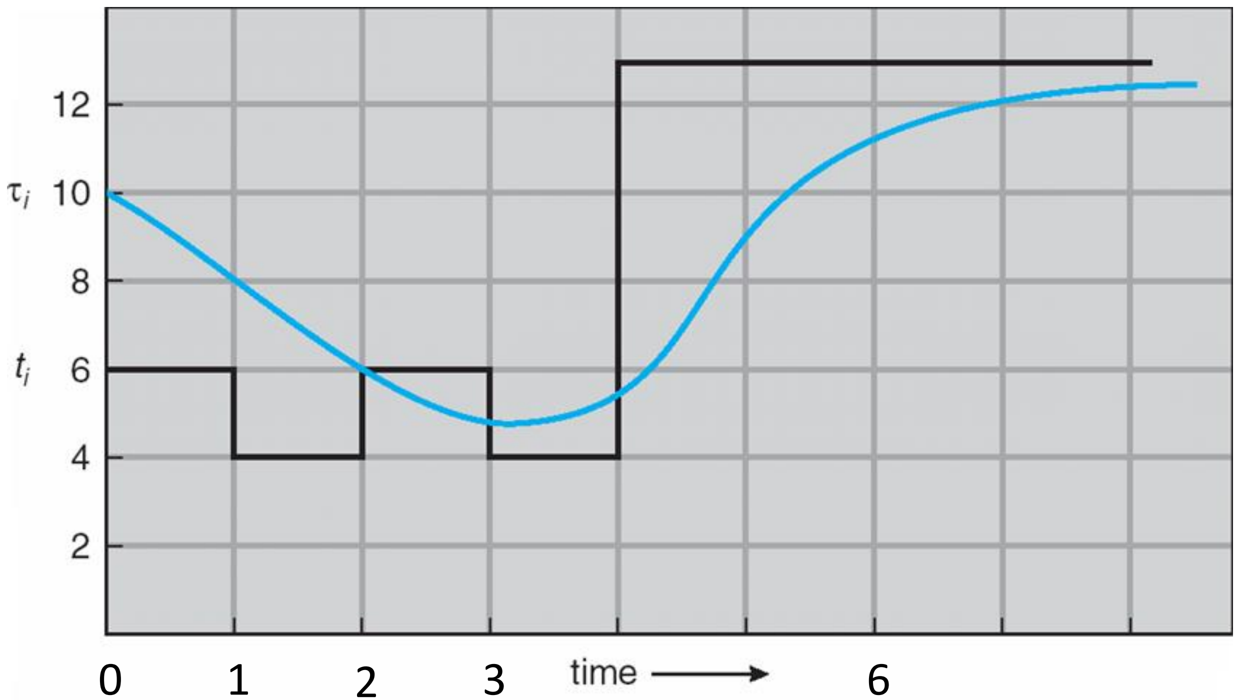
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent burst time does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst



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

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)

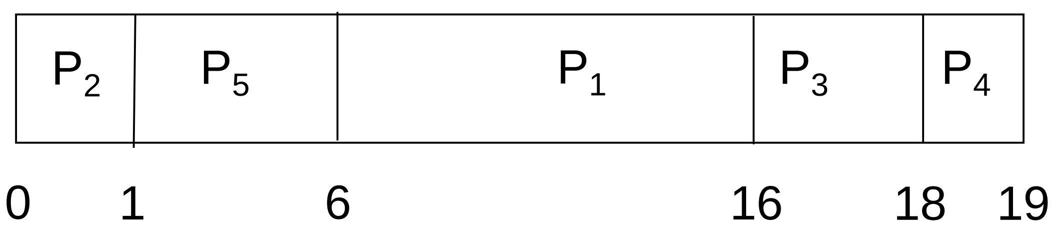
nice

- Set priority value
  - Internal (time limit, memory req., ratio of I/O Vs CPU burst)
  - External (importance, fund etc)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Two types
  - Preemptive
  - Nonpreemptive
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

Round Robin (RR)

- Designed for time sharing system
- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Implementation
  - Ready queue as FIFO queue
  - CPU scheduler picks the first process from the ready queue
  - Sets the timer for 1 time quantum
  - Invokes despatcher
- If CPU burst time < quantum
  - Process releases CPU
- Else Interrupt
  - Context switch
  - Add the process at the tail of the ready queue
  - Select the front process of the ready queue and allocate CPU

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

- Avg waiting time = ((10-4)+4+7)/3=5.66

Round Robin (RR)

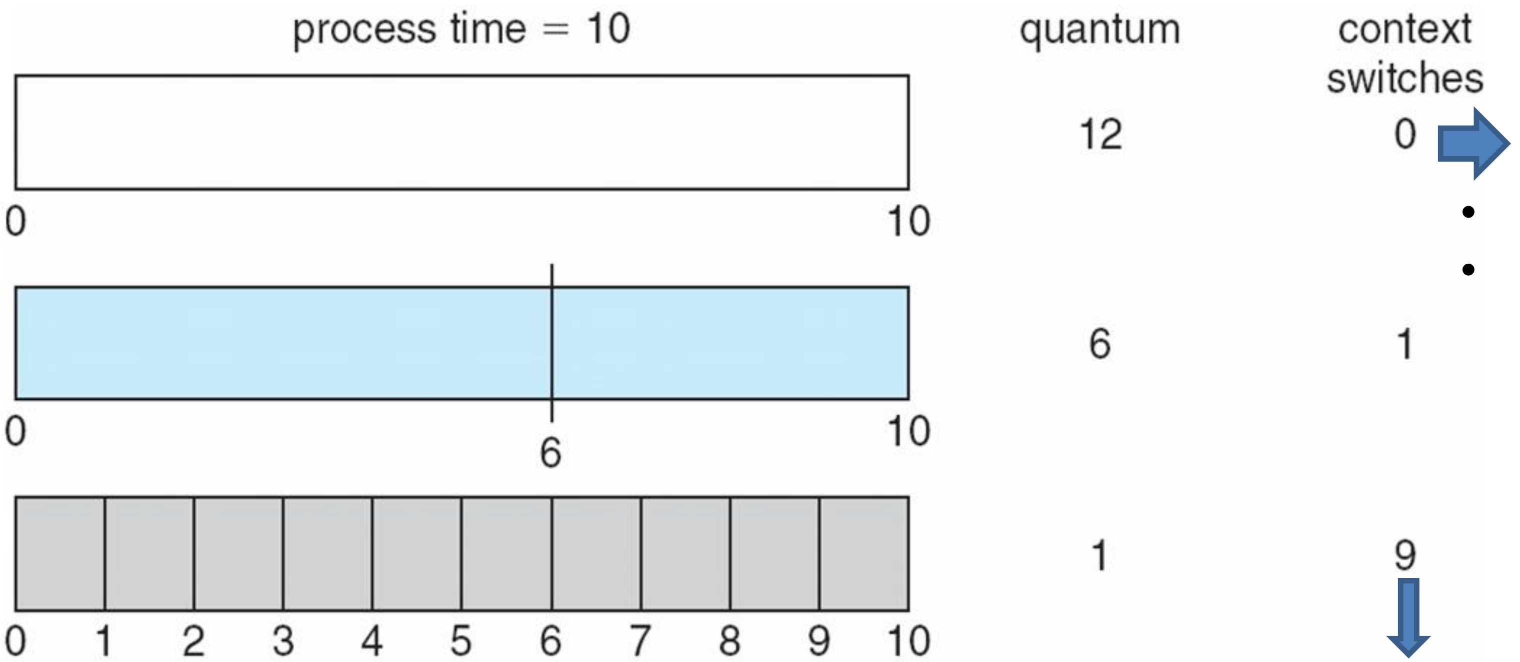
- Each process has a time quantum *T* allotted to it
- Dispatcher starts process P_0, loads a external counter (timer) with counts to count down from *T* to 0
- When the timer expires, the CPU is interrupted
- The context switch ISR gets invoked
- The context switch saves the context of P_0
  - PCB of P_0 tells where to save
- The scheduler selects P_1 from ready queue
  - The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1
- The dispatcher reloads the counter (timer) with *T*
- The ISR returns, restarting P_1 (since P_1's PC is now loaded as part of the new context loaded)
- P_1 starts running

Round Robin (RR)

- 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.
- Timer interrupts every quantum to schedule next process
- Performance depends on time quantum q
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ Processor sharing (n processes has own CPU running at 1/n speed)

Effect of Time Quantum and Context Switch Time

Performance of RR scheduling



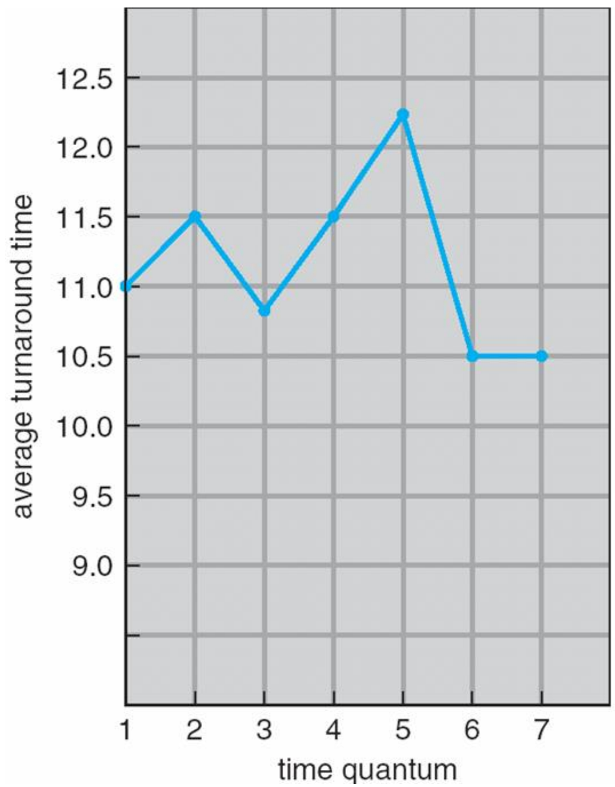
- No overhead
- However, poor response time

- Too much overhead!
- Slowing the execution time

- *q* must be large with respect to context switch, otherwise overhead is too high
- q usually 10ms to 100ms, context switch < 10 microsec

Effect on Turnaround Time

- TT depends on the time quantum and CPU burst time
  - Better if most processes complete there next CPU burst in a single q



process	time
P_1	6
P_2	3
P_3	1
P_4	7

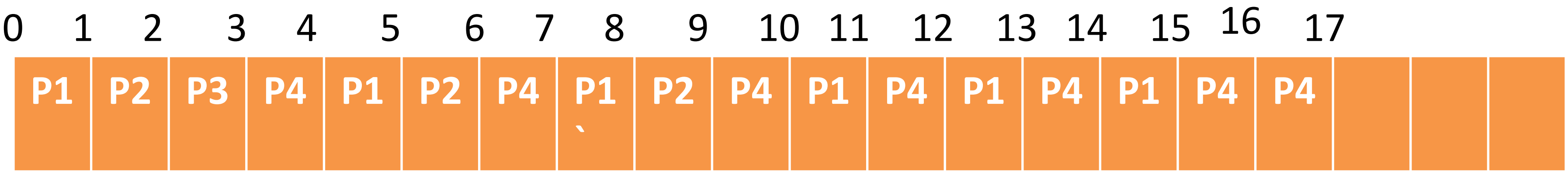
- Large q=> processes in ready queue suffer
- Small q=> Completion will take more time

80% of CPU bursts should be shorter than q

Response time Typically, higher average turnaround than SJF, but better *response time*

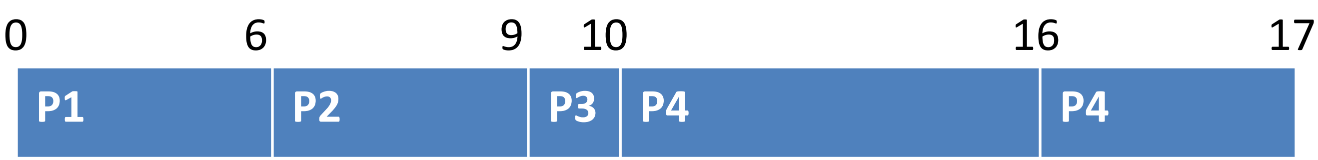
Turnaround Time

q=1



Avg Turnaround time=
(15+9+3+17)/4=11

process	time
P_1	6
P_2	3
P_3	1
P_4	7



q=6

(6+9+10+17)/4=10.5

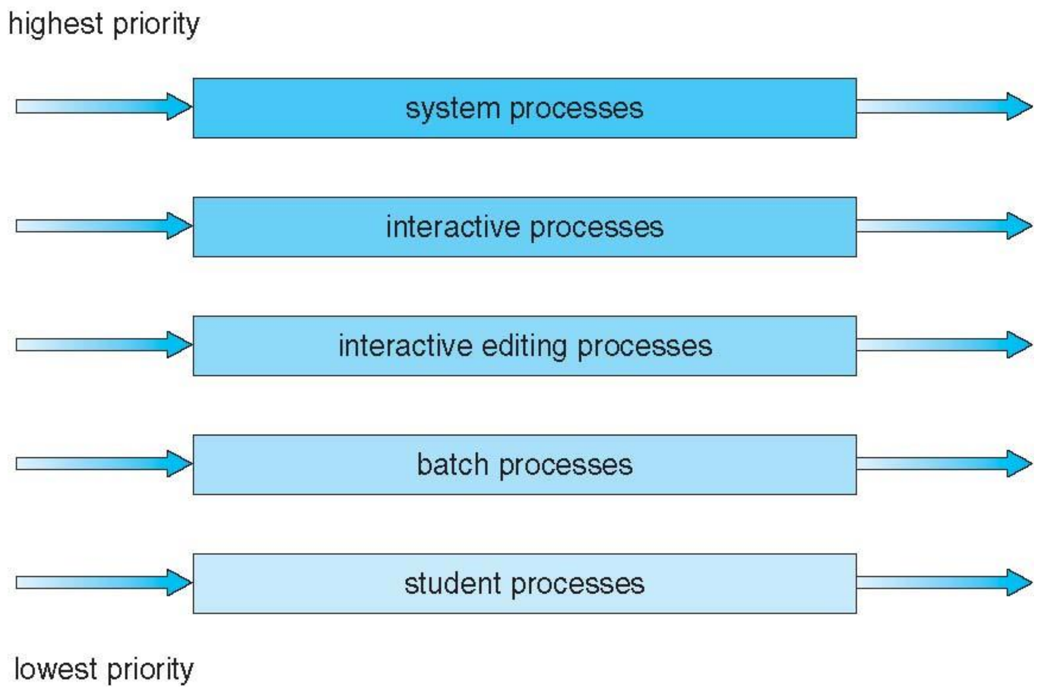
Process classification

- Foreground process
  - Interactive
  - Frequent I/O request
  - Requires low response time
- Background Process
  - Less interactive
  - Like batch process
  - Allows high response time
- Can use different scheduling algorithms for two types of processes ?

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently assigned in a given queue
  - Based on process type, priority, memory req.
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background).
  - Possibility of starvation.

Multilevel Queue Scheduling



- No process in batch queue could run unless upper queues are empty
- If new process enters
  - Preempt

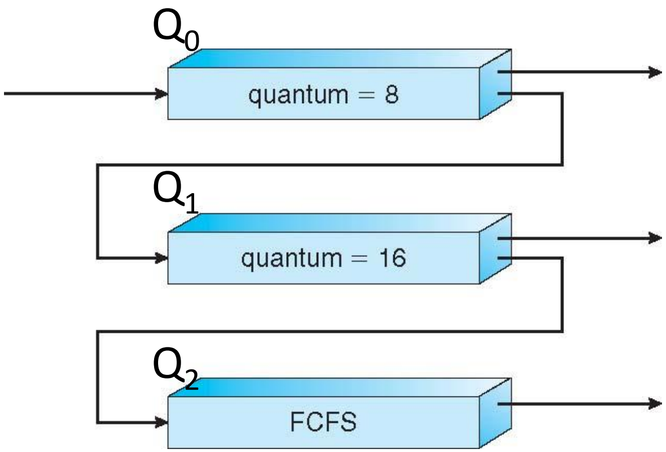
Another possibility

- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
- 20% to background in FCFS

Multilevel Feedback Queue

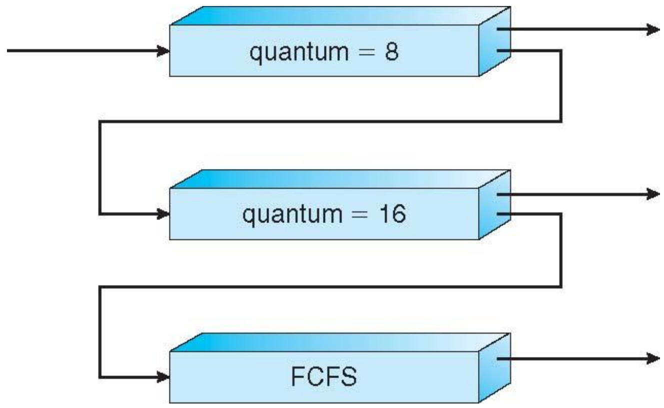
- So a process is permanently assigned a queue when they enter in the system
  - They do not move
- **Flexibility!**
  - Multilevel-feedback-queue scheduling
- A process can move between the various queues;
- Separate processes based of the CPU bursts
  - Process using too much CPU time can be moved to lower priority
  - Interactive process => Higher priority
- Move process from low to high priority
  - Implement aging

Example of Multilevel Feedback Queue



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again receives 16 milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Multilevel Feedback Queues



- Highest Priority to processes CPU burst time <8 ms
- Then processes >8 and <24

- 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

Multiple-Processor Scheduling

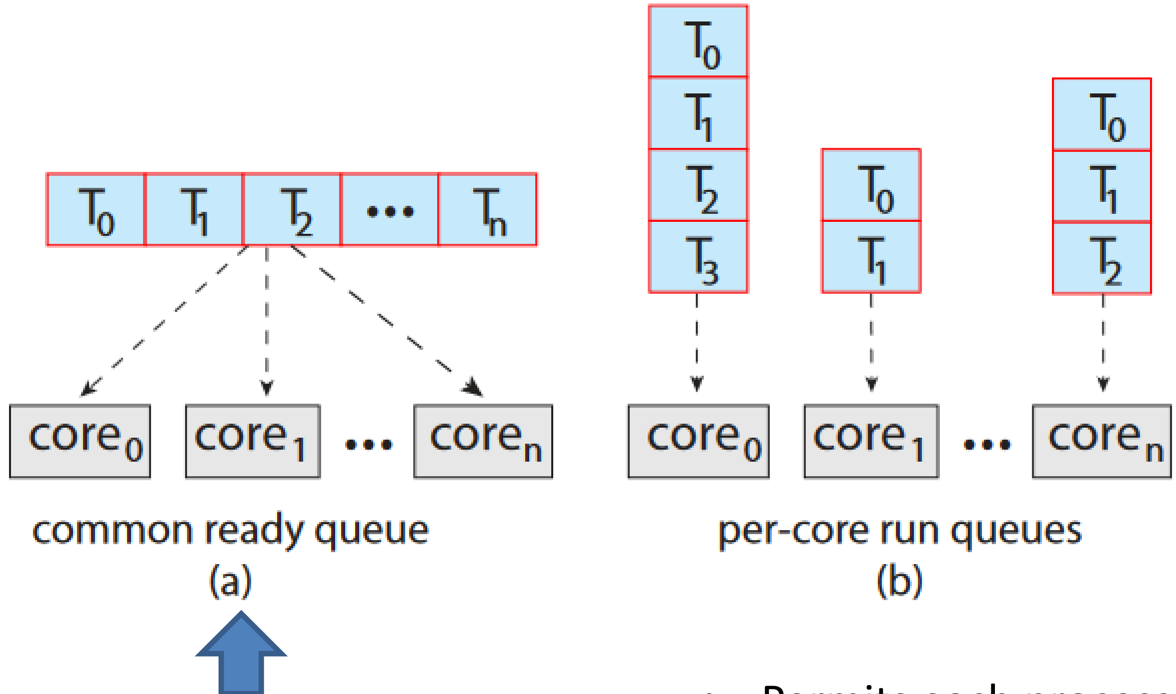
- If multiple CPUs are available, multiple processes may run in parallel

- However **scheduling** issues become correspondingly more **complex**.
- Many possibilities have been tried
- As we saw with CPU scheduling with a single-core CPU
  - there is no one best solution

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** –
  - Master server
  - only **one processor** accesses the **system data** structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,
- all processes in **common ready queue**, or each has its **own private queue** of ready processes
- Scheduler for each processor **examine the ready queue**
  - select a process to run.

Multiple-Processor Scheduling



- We have a possible **race condition**
- **Locking** to protect the common ready queue from this race condition.
- Accessing the shared queue would likely be a **performance bottleneck**

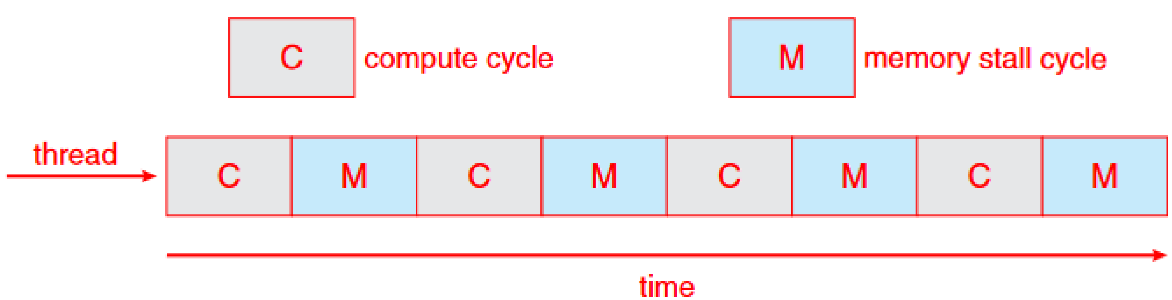
- Permits each processor to schedule process from its **private ready queue**
- **Does not suffer** from the possible **performance** problems
- Most common approach on systems supporting SMP.
- **Load balancing**

Multi core processors

- SMP systems have allowed several processes to run **in parallel** by providing **multiple physical processors**.
- Recently, **multiple computing cores** on the **same physical chip**, resulting in a **multicore processor**.
- **Each core** maintains its **architectural state** and thus appears to the operating system to be a separate **logical CPU**
- SMP systems that use multicore processors are **faster and consume less power**
  - than systems in which each CPU has its own physical chip

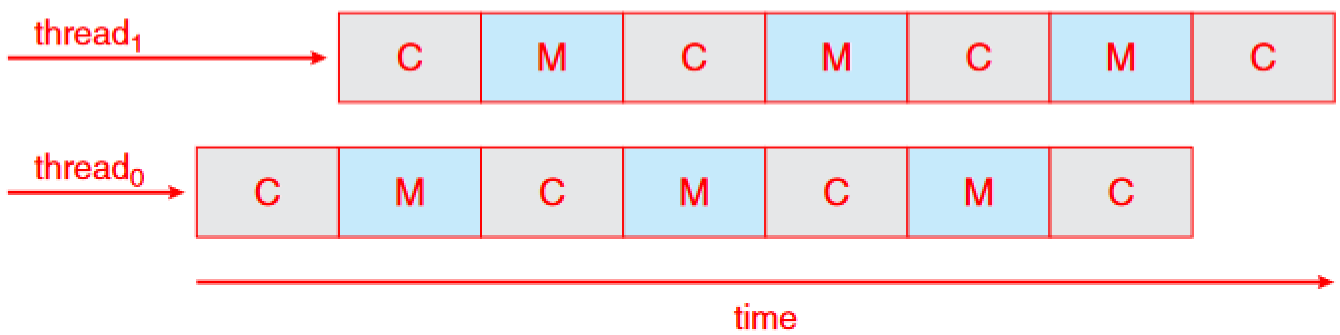
Challenge: Memory stall

- When a processor **accesses memory**, it spends a significant amount of **time waiting** for the data to become available.
- This situation, known as a **memory stall**, occurs primarily because
  - modern processors operate at much faster speeds than memory.
  - For cache miss

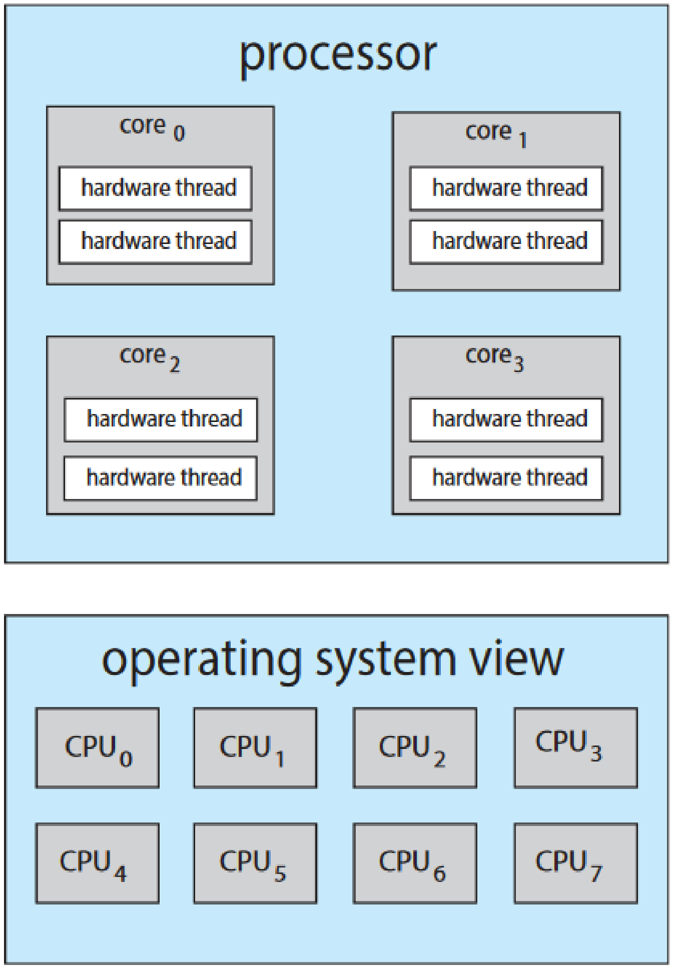


Solution: Hardware threads

- Recent hardware designs have implemented **multithreaded processing cores** in which two (or more) hardware threads are assigned to **each core**.
- That way, if **one hardware thread stalls** while waiting for memory, the core can **switch** to another thread.



Hyper-threading



- **Each hardware thread** maintains its **architectural state**, such as **instruction pointer and register set**,
- Thus appears as a **logical CPU** that is available to run a **software process**.

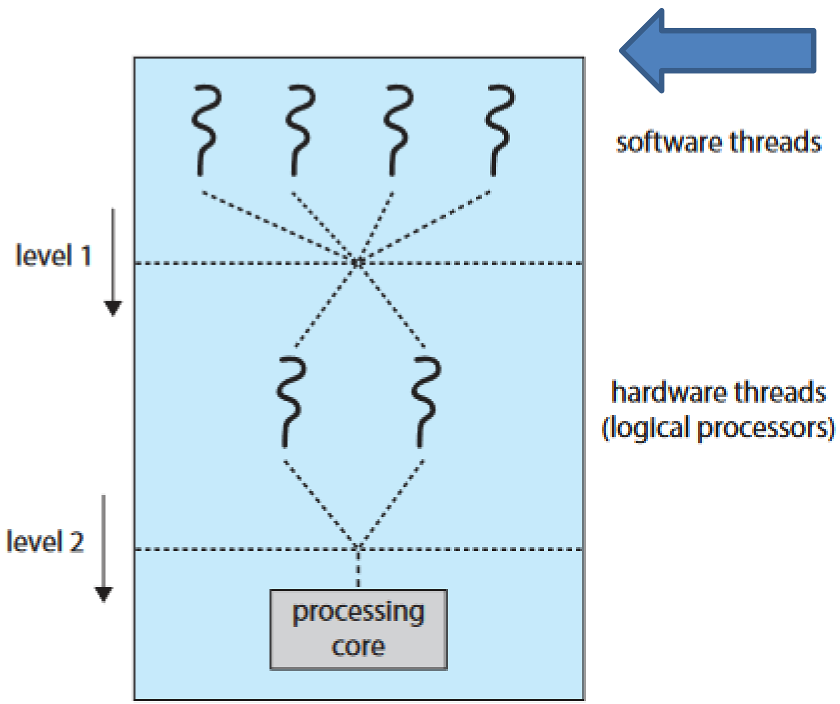
Contemporary Intel processors—such as the **Intel i7—support two threads per core**,

Oracle Sparc M7 processor supports **eight threads per core**, with eight cores per processor, thus providing the operating system with 64 logical CPUs

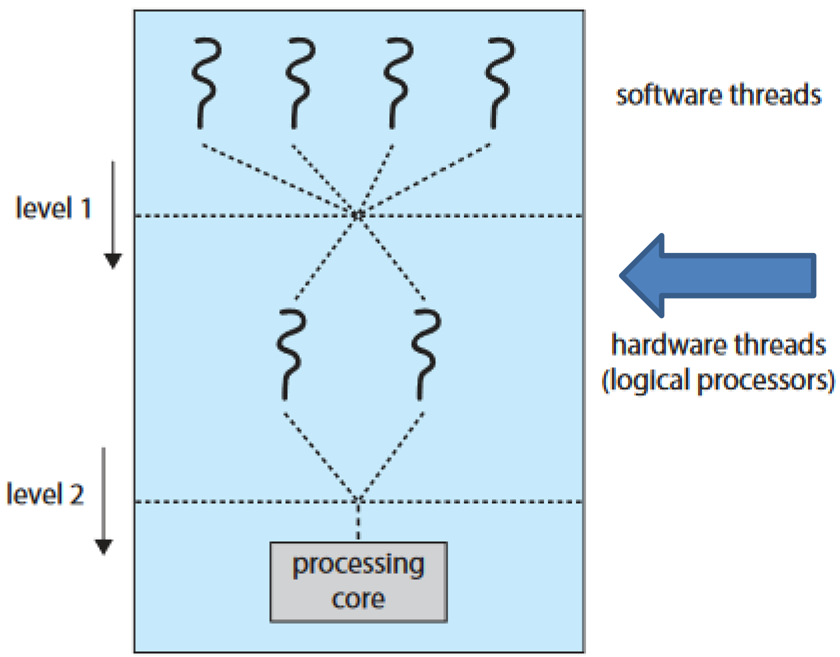
Dual scheduling

- It is important to note that
- a **processing core** can only execute **one hardware thread** at a time.
  - **resources of the physical core** (such as caches and pipelines) must be shared among its **hardware threads**
- Multithreaded, multicore processor actually requires **two different levels of scheduling**

Dual scheduling



- The **scheduling decisions** that must be made by the operating system as it **chooses which software process** to run on each **hardware thread** (logical CPU).
- For this level of scheduling, the operating system may choose **any scheduling algorithm**



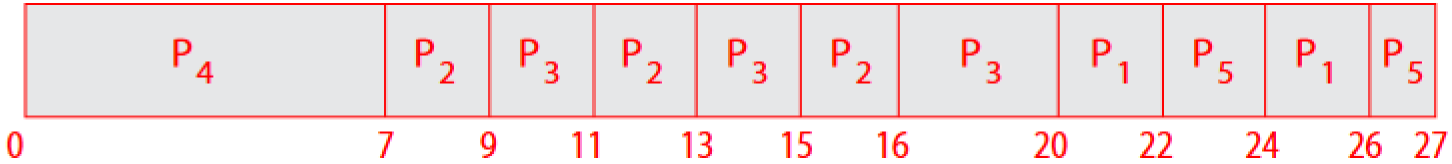
- A second level of scheduling specifies **which hardware thread to run on a core**.
- One approach is to use a **simple round-robin algorithm** to schedule a hardware thread to the processing core.
- This is the approach adopted by the UltraSPARC T3.
- Another approach is used by the Intel Itanium
- Assigned to each hardware thread is a dynamic urgency value ranging from 0 to 7

Problem 1

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling (q=2).

Solution 1



Problem 2

Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8 time units. All processes arrive at time zero. Consider the longest remaining time first (**LRTF**) scheduling algorithm. In LRTF ties are broken by giving priority to the process with the lowest process id. Compute average turn around time

Process	**AT**	**BT**	**TAT**
P0	0	2	
P1	0	4	
P2	0	8	

Solution 2

2. Consider three processes (process ID 0,1,2 respectively) with compute time bursts 2,4 and 8 time units. All processes arrive at time zero. Consider **the longest remaining time first (LRTF) scheduling algorithm**. In LRTF, ties are broken by giving priority to the process with the lowest process ID. The **average turnaround time is:**

P2	P2	P2	P2	P1	P2	P1	P2	P0	P1	P2	P0	P1	p2

PID	A.T	B.T	C.T	T.A.T	W.T
P0	**0**	**2**	**12**	**12**	**10**
P1	**0**	**4**	**13**	**13**	**9**
P2	**0**	**8**	**14**	**14**	**6**
TOTAL				**39**	**25**

A.T. Arrival Time
B.T. Burst Time
C.T. Completion Time.
T.A.T. Turn Around Time
W.T. Waiting Time.
Average TAT = 39/3 = 13 units

Problem 3

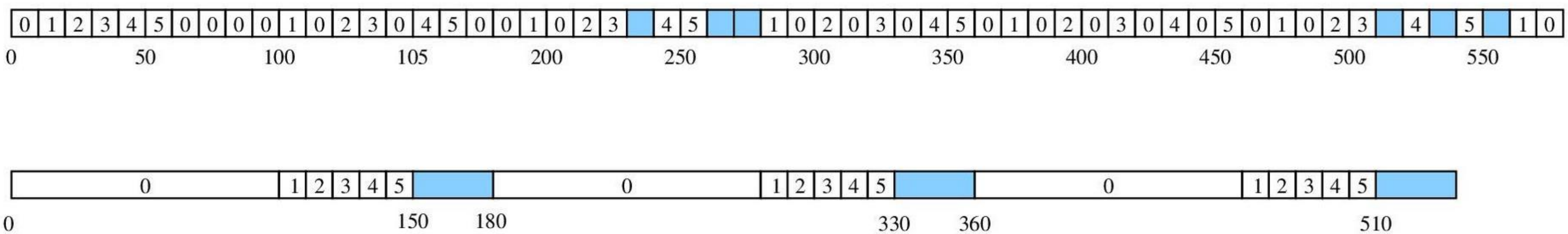
Process 0: CPU-bound (each CPU burst is 100ms)
Processes 1--5: IO bound (10ms CPU burst),IO time: 80ms

All processes are available at t = 0.

FCFS: find out when CPU becomes idle for the first time.

RR: Take time quantum q = 10ms. Again find out when the CPU becomes idle for the first time.

Solution 3



0 1 2 3 4 5 0 0 0 0 1 0 2 3 0 4 5 0 0 1 0 2 3 4 5 1 0 2 0 3 0 4 5 0 1 0 2 0 3 0 4 0 5 0 1 0 2 3 4 5 1 0
0 50 100 105 200 250 300 350 400 450 500 550
0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5
0 150 180 330 360 510

Problem 4

Consider the following set of processes, with the arrival times and the CPU-burst times given in milliseconds. What is the average turnaround time for these processes with the **preemptive shortest remaining processing time first (SRPT) algorithm?**

3. Consider the following set of processes, with the arrival times and the CPU-burst times given in milliseconds. What is the average turnaround time for these processes with the **preemptive shortest remaining processing time first (SRPT) algorithm?**

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	3
P4	4	1

Answer:

P1	P2	P4	P3	P1

0 **1** **4** **5** **8** **12**

Average turnaround time = 12 + 3 + 6 + 1 4 = 22 4 = 5.5

Process	Waiting Time = (Turnaround Time – Burst Time)	Turnaround Time = (Completion Time – Arrival Time)
P1	7	12
P2	0	3
P3	3	2
P4	0	1

Queue 1: SJF

Queue 2: FCFS

Queue 3: RR with Q=2

Q1 > Q2 > Q3

(priority)

PID	A.T	B.T	Queue
P_1	0	4	1
P_2	0	3	1
P_3	0	9	3
P_4	9	5	2
P_5	11	7	1
P_6	8	2	3

A

Q1 : P_1, P_2, P_5

Q2 : P_4

Q3 : P_3, P_6

P2	P1	P3	P4	P5	P4	P3	P6	P3	P3	P3	
0	3	7	9	11	18	21	23	25	27	29	30