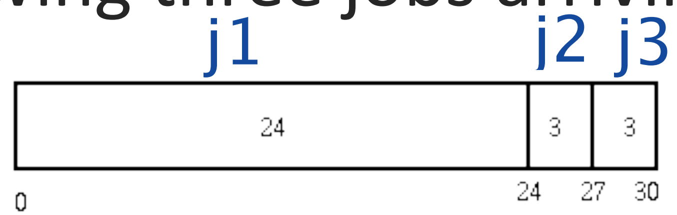
First Come First Serve

- This is a simple scheduling scheme where a process is placed at the end of the ready queue when it arrives
- FCFS is non-preemptive, i.e. the running process executes until its current CPU burst ends
- When the running process is done, the job at the head of the ready queue is selected as the next process to run
- The performance of FCFS varies greatly, according to the burst durations and arrival order of the processes

FCFS Example

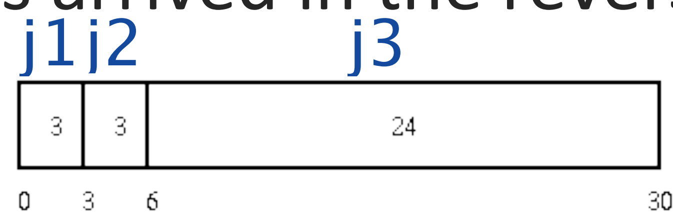
Consider the following three jobs arriving:



numbers represent time taken before i/o burst

Average turnaround time = (24 + 27 + 30)/3 = 27 units

Consider if the jobs arrived in the reverse order:



Average turnaround time = (3 + 6 + 30)/3 = 13 units

- The average turnaround time is generally not minimal, and depends on the order of arrival of the processes

Shortest Job First

- In SJF the length of the next CPU burst of each job is used to determine which goes next. E.g.: Ready queue:

Job	Next Burst Length
1	6
2	3
3	8
4	7

j2	j1	j4	j3

0 3 9 16 24

Avg turnaround time = (9 + 3 + 24 + 16) / 4 = 13 units

SJF is optimal for non-preemptive algorithms

- But it is impossible to implement because you cannot determine the length of the next CPU burst!
- We can use techniques such as *exponential averaging* to approximate the length of the next burst

Exponential Averaging

- We can estimate the duration of the next CPU burst by taking the exponential average of past burst lengths
- We use the formula:

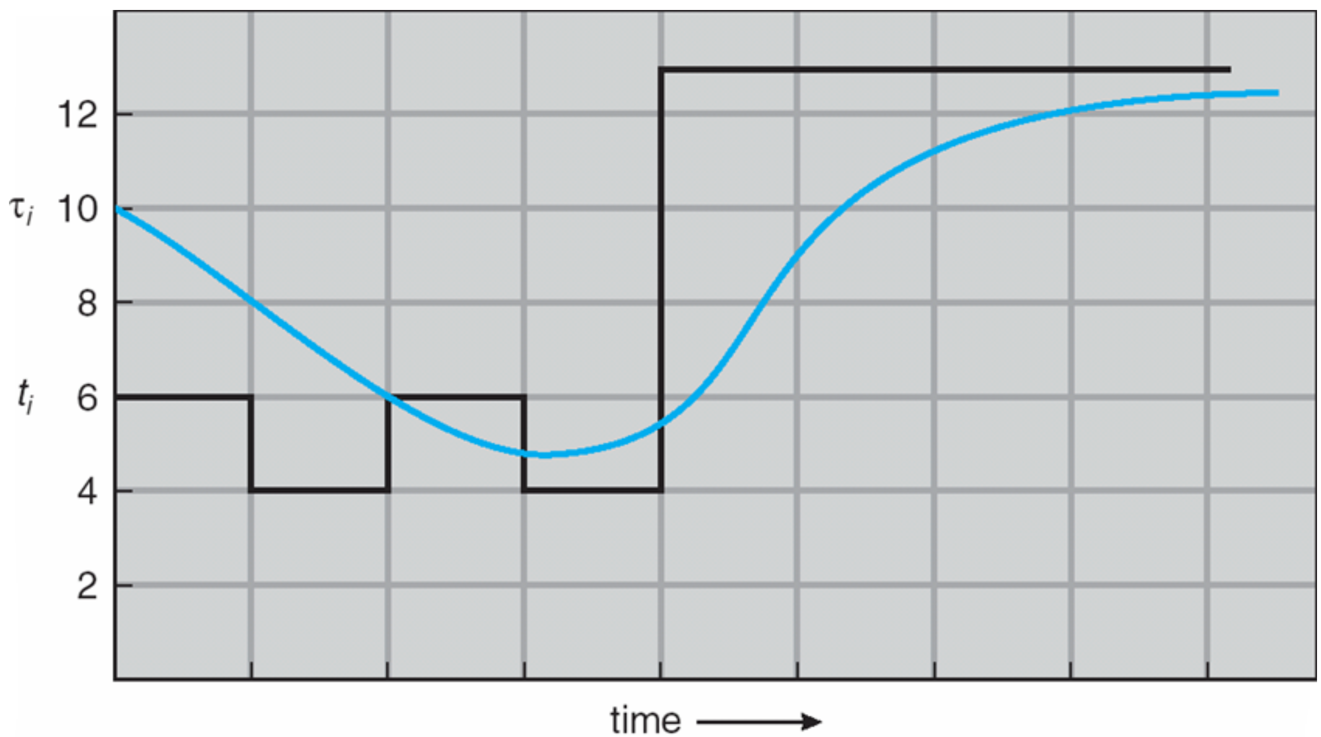
$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

- Where:

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

- For `n = 0` we can use a constant or system average

Example Prediction using Exponential Averaging



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

Exponential Averaging Example

- If we choose α close to 1 we base our prediction more on recent burst history

  E.g alpha = 1 T_n+1 = alphatn
  – only the last CPU burst length counts

- If we choose α close to 0 we base our prediction more on the distant past

  E.g. alpha = 0 Tn+1 = Tn
  – recent history doesn't count

Shortest Remaining Time First

- If a new job becomes ready and its next burst is shorter than the remaining burst of the running job, the running job will get preempted
  - i.e it will lose the CPU and go back on ready queue.

Job	Arrival Time	Burst time
1	0	8
2	1	4
3	2	9
4	3	5

J1 | J2 | J4 | J1 | J3

0 1 5 10 17 26

Avg turnaround time = (17 + 4 + 24 + 7)/4 = 13

- SRTF is optimal for preemptive algorithms
  - Again we must use exponential averaging to predict next burst length

Round Robin

- In this type of scheduling, the CPU is given to each process in the ready queue in turn for a duration up to a period of time called a time quantum (~10-100 ms)
- New processes are added to the end of the ready queue
- Scheduler sets a timer to go off after "quantum" time units and dispatches a ready process
  - If process voluntarily releases the CPU then it goes to the end of the appropriate blocked queue
  - If timer goes off, process goes to end of ready queue

Round Robin

- Round robin example: q = 4

Job	Burst
1	24
2	3
3	3

j1	j2	j3	j1	j1	j1	j1	j1

0 4 7 10 14 18 22 26 30

Avg turnaround time = (30 + 7 + 10) / 3 = 16

- Each process in a queue of n can get the CPU for at least `1/n` of the time. Each must wait a maximum of `(n - 1)*q` time units for the processor (`q` is the quantum used)

Round Robin

- The performance of RR depends on the quantum chosen:
  - If quantum is very large → RR turns into FCFS
  - If quantum is very small → RR turns into processor sharing

– appears as if each process has a CPU running at 1/n of real CPU speed

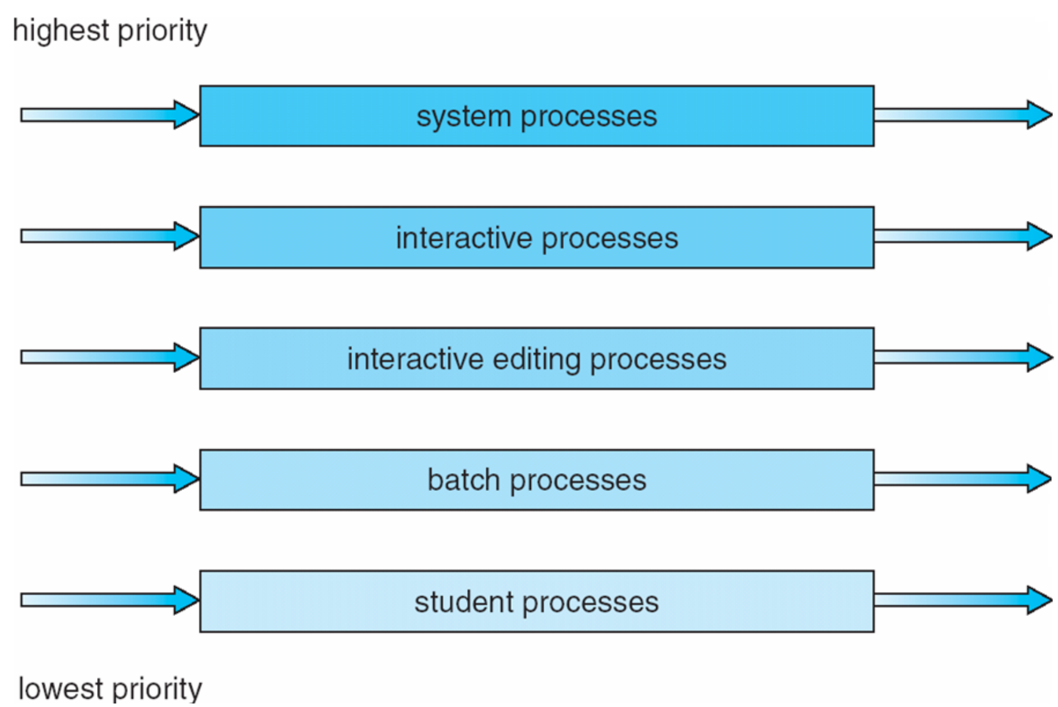
- We must choose the quantum so that it is:
  - Not so short as to cause undue context switching
  - Not so long as to unnecessarily increase the waiting time for ready processes

Round Robin

- The CPU efficiency is the percentage of time the CPU spends executing user processes, and helps us understand the implications of quanta choices
- E.g. for FCFS, if the context switch time is 1, and the average burst time is 5:
  - CPU efficiency = $5 / (5 + 1) = 83\%$
  - Same for SJF
- If we have time quantum q = 4, average burst time = 6, and switch time = 1:
  - # of switches per burst = $6/4$
  - Time taken by switches = $(6/4) * 1$
  - CPU efficiency = $6/(6 + (6/4) * 1) = 80\%$

Multilevel Queues

- Several different ready queues
- Each queue is for a different type of job
- Each queue has different scheduling characteristics
  - RR, FCFS, etc.
- Requires scheduling policy between queues



Multilevel Feedback Queues

- Several queues, but not based on job type
- Every job enters the top priority queue when it starts.
- Jobs might work their way down to lower priority queues, or up to higher priority queues
  - high to low: job didn't terminate during its first time quantum or CPU burst
  - low to high: age
- Very general scheme
  - configurable to accommodate different needs

Multilevel Feedback Queues

- Can be configured in many ways:
  - number of queues
  - scheduling algorithm for each queue
  - scheduling algorithm between queues
- Jobs can change queues
  - job type may determine initial queue

Multilevel Feedback Queues

- The point of Multilevel Queue's:
  - different processes have different scheduling needs
  - We can implement different scheduling algorithms based on the process type
- The point of Multilevel Feedback Queues:
  - the needs of a process may change over time
  - scheduling needs of a process may be unknown until it has been running for a while
  - adaptable scheduling strategy

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – 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
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**

no matter what, only want to go back to same core as it started out in