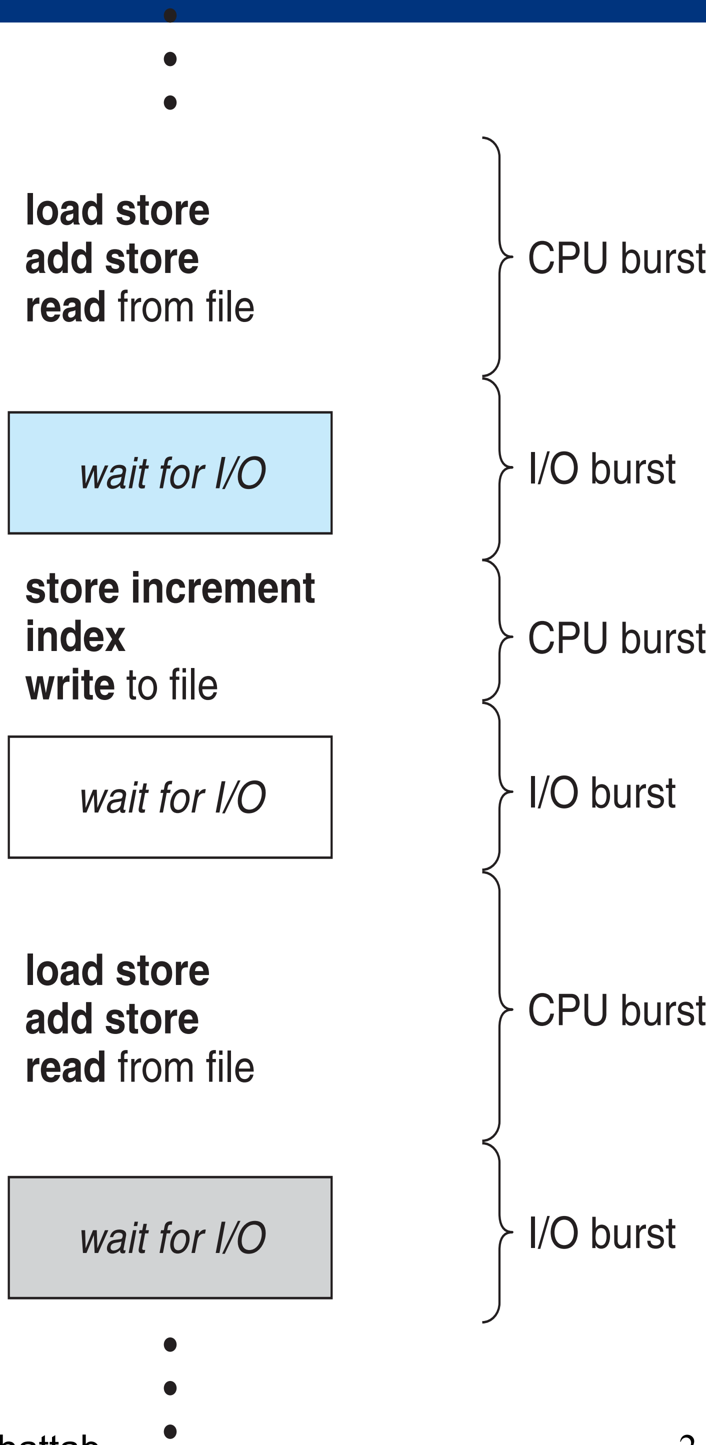
CPU Scheduling

Daniel Mosse

(Most slides are from Sherif Khattab and **Silberschatz, Galvin and Gagne ©2013)**

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



load store
add store
read from file

wait for I/O

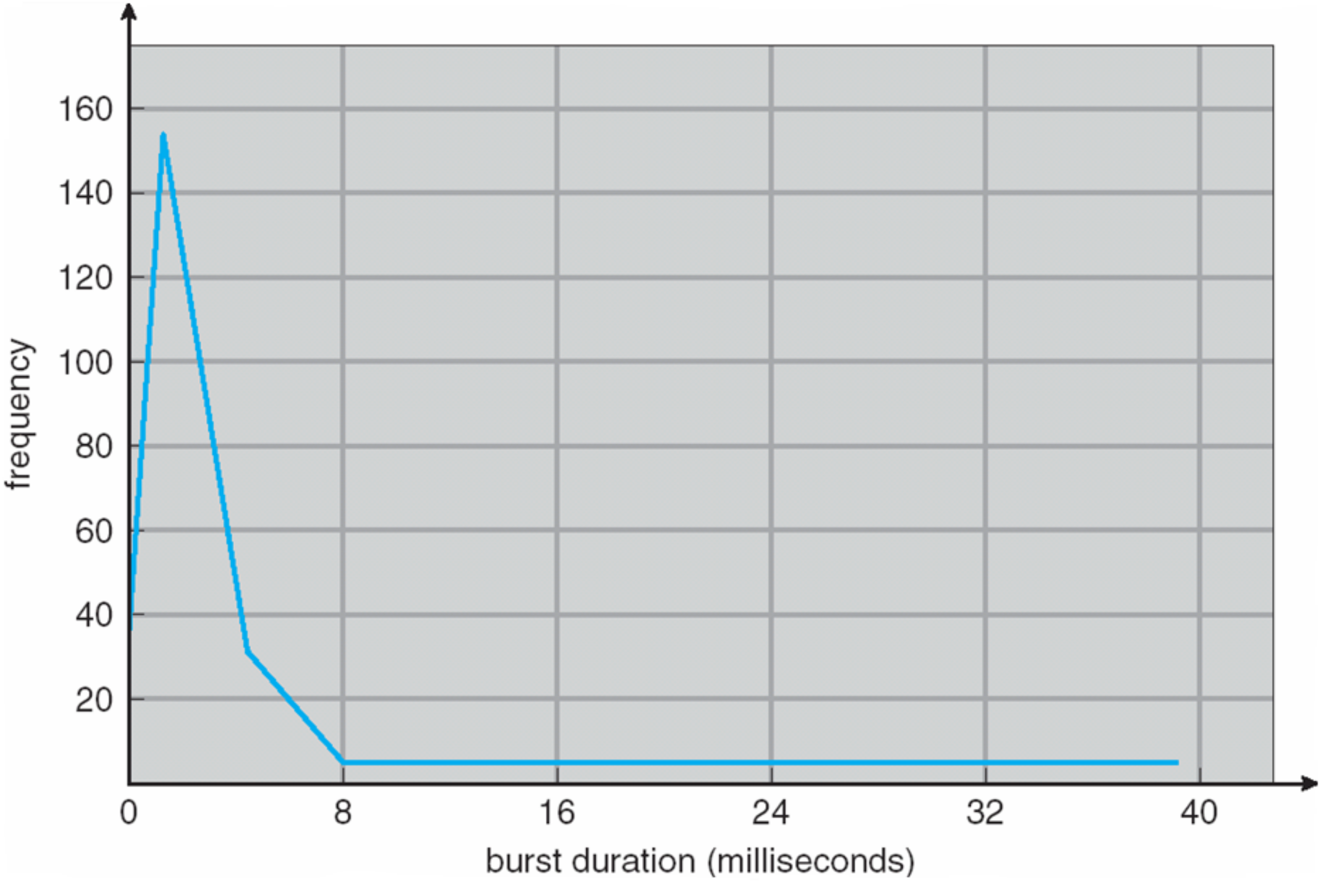
store increment
index
write to file

wait for I/O

load store
add store
read from file

wait for I/O

Histogram of CPU-burst Times



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

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: $(0 + 24 + 27)/3 = 17$

Determining Length of Next CPU Burst

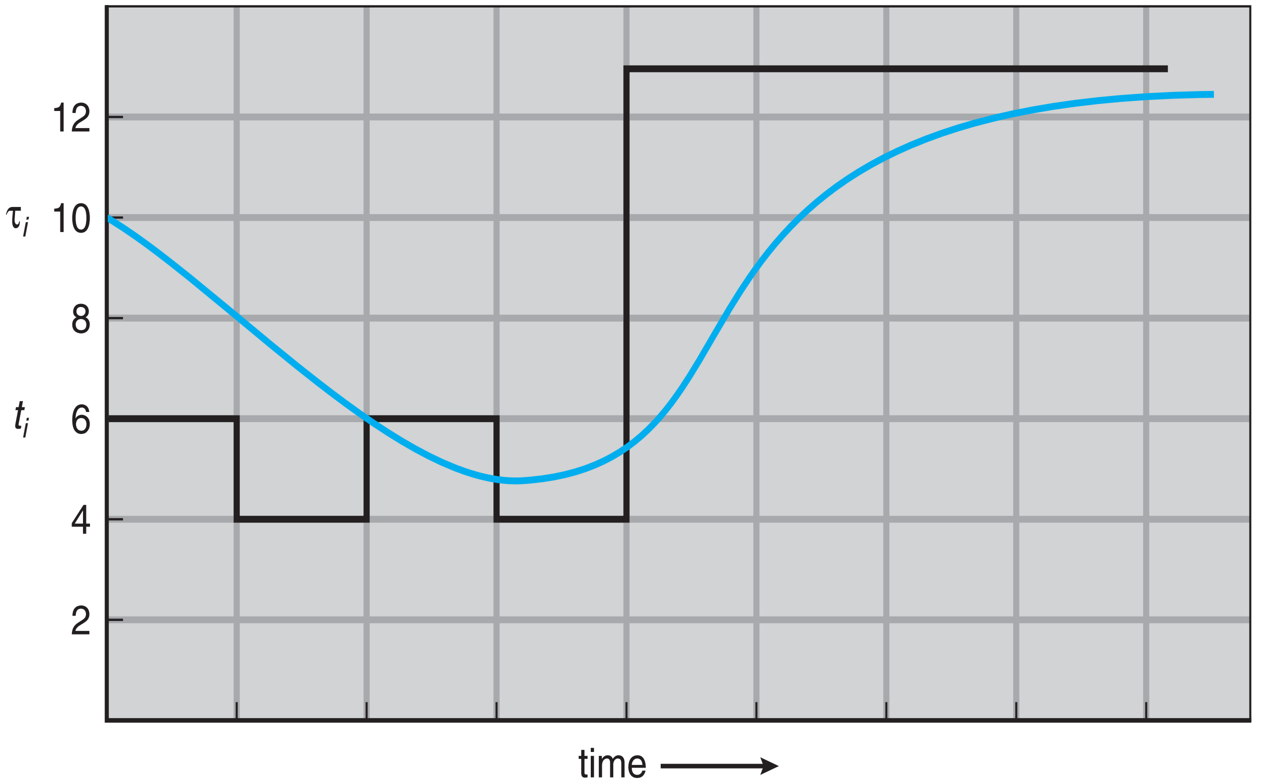
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

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

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

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)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

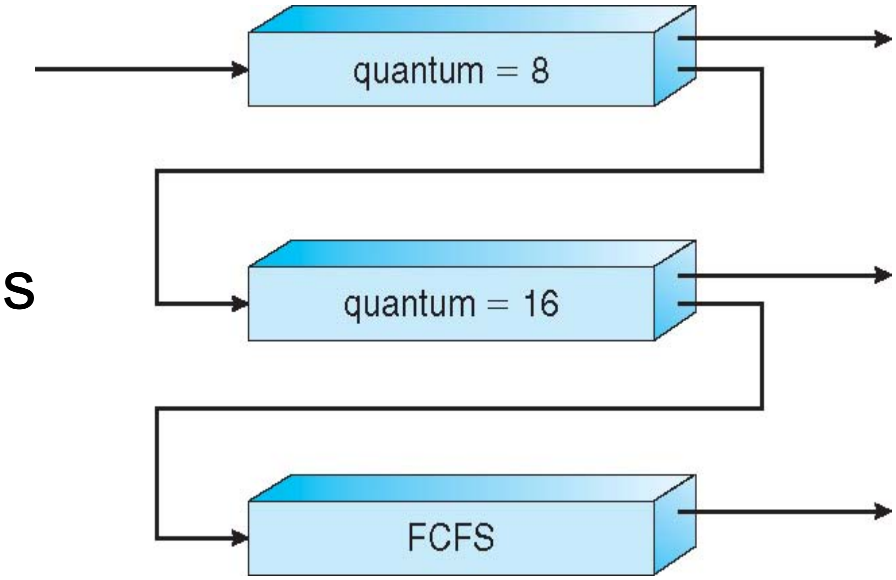
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 4-10 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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.
- Performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- 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.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes;
    - 80% to foreground in RR
    - 20% to background in FCFS

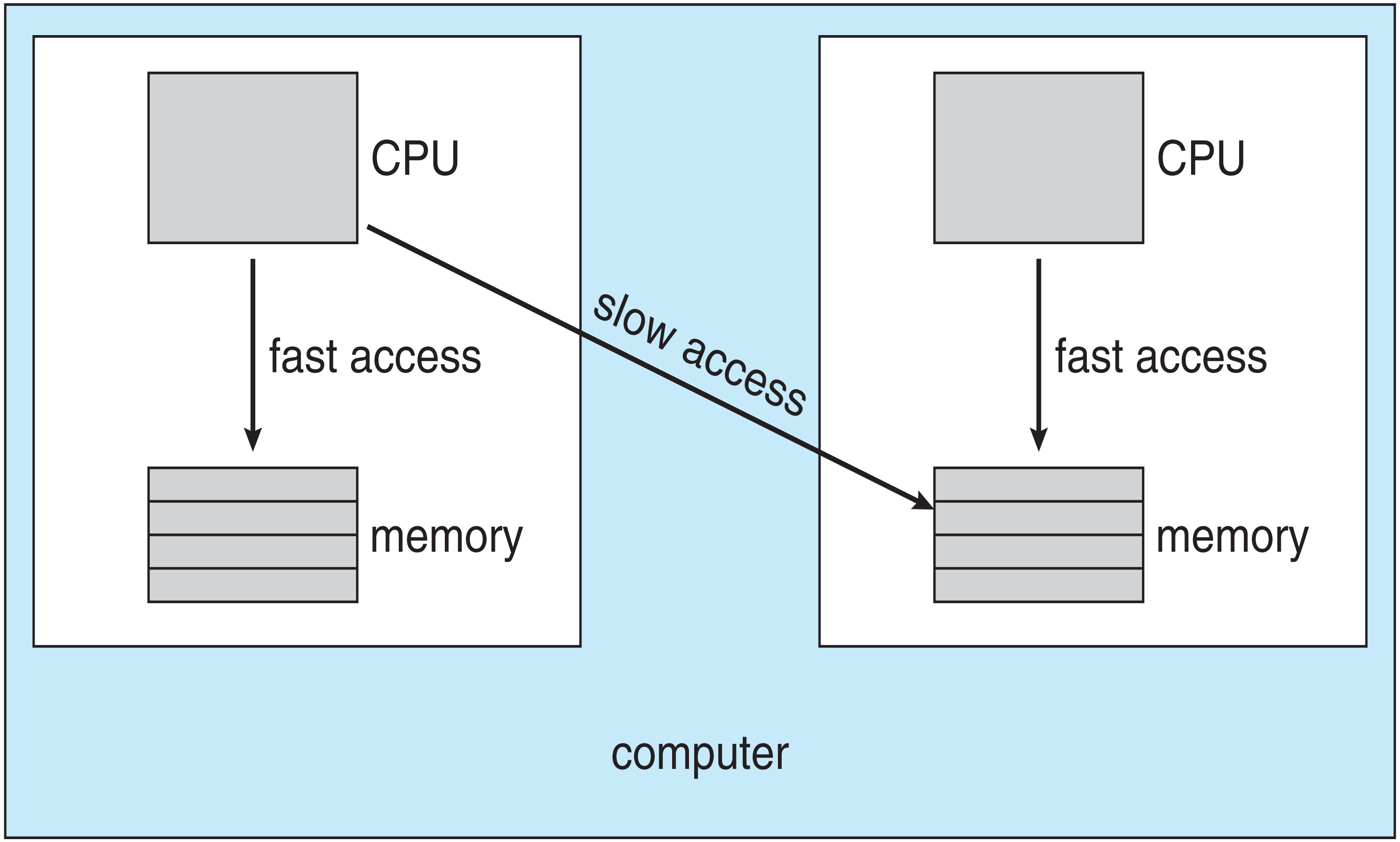
Multilevel Feedback Queue (by example)



- **Three queues:**
  - Q_0 – RR; quantum 8 milliseconds
  - Q_1 – RR; 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 receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Multiple-Processor Scheduling

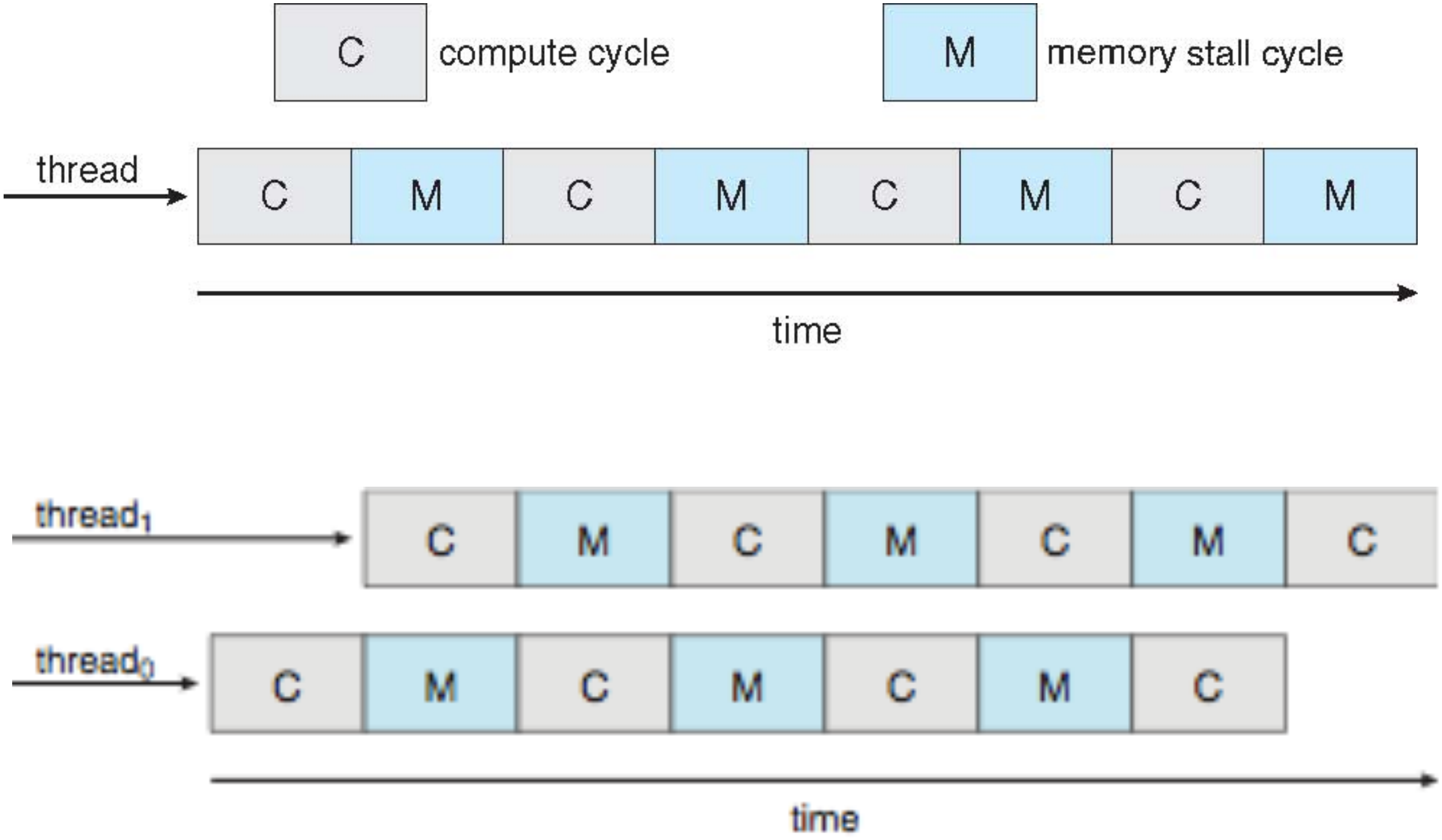
- NUMA



Multiple-Processor Scheduling

- **Symmetric multiprocessing** (**SMP**) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
- **Load balancing**
  - **Contradicts affinity?**

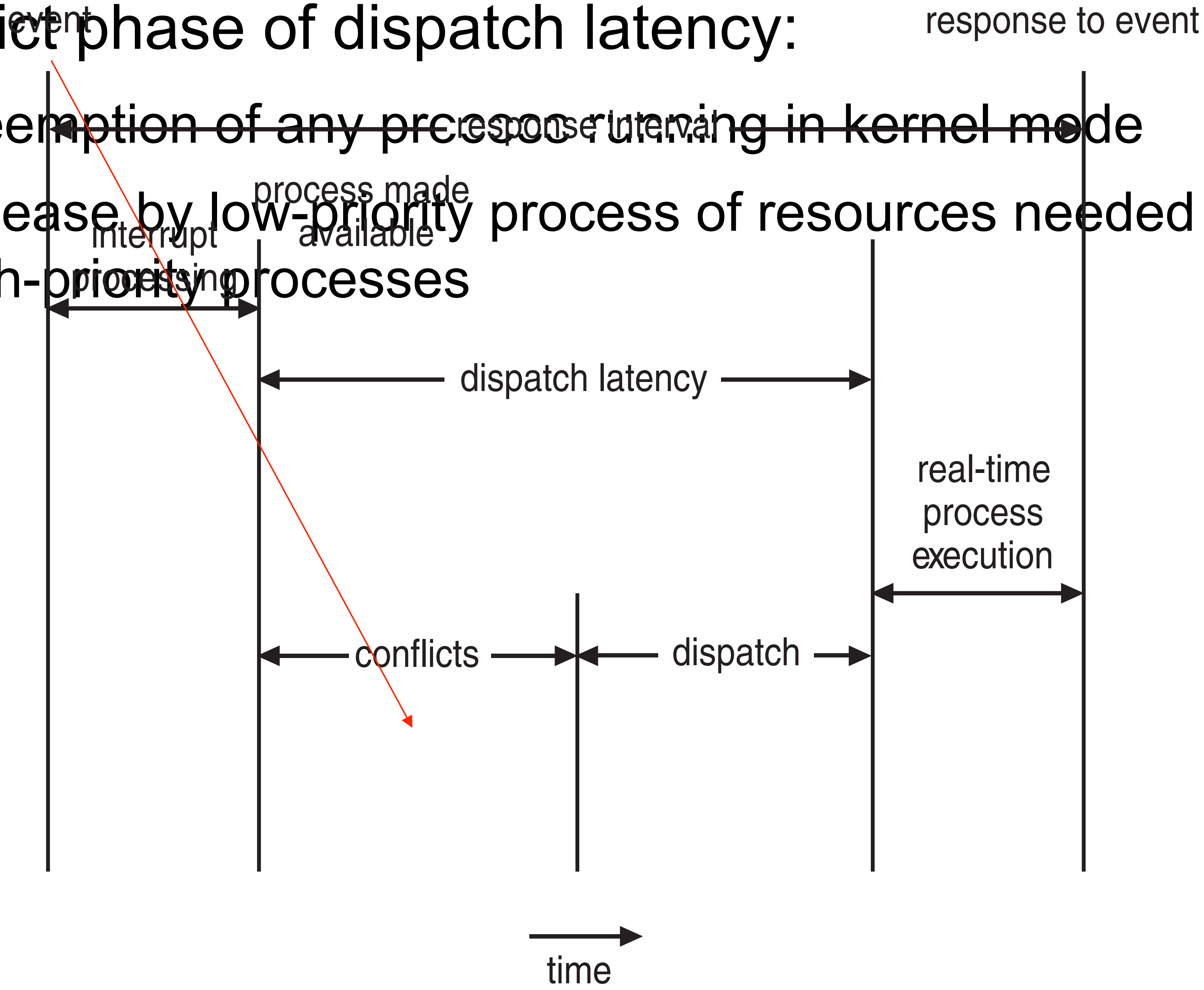
Multithreaded Multicore System



Real-Time CPU Scheduling

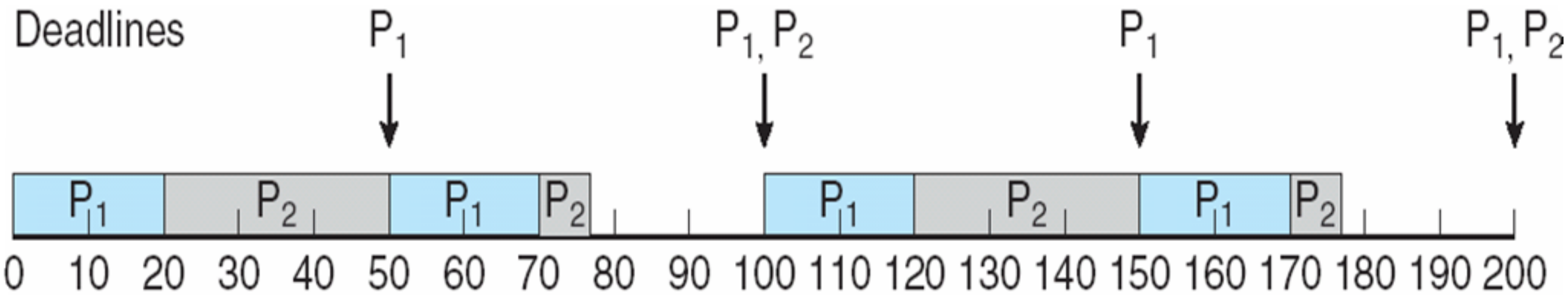
- Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes



Rate Montonic Scheduling

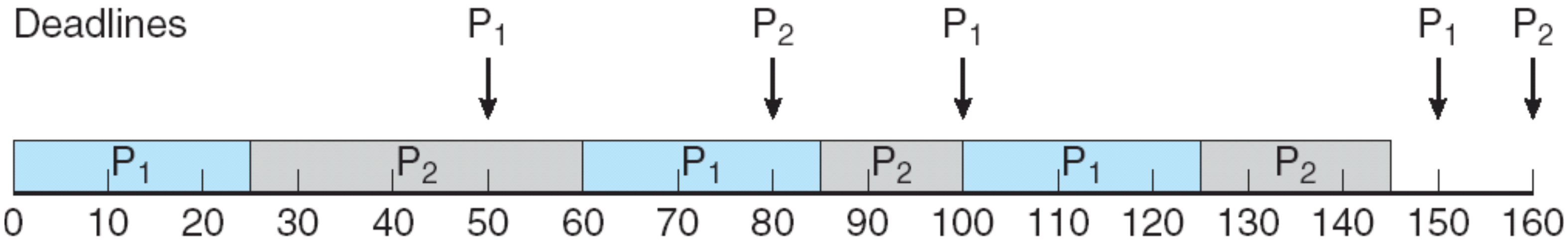
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



Earliest Deadline First Scheduling (EDF)

Priorities are assigned according to deadlines:

- the earlier the deadline, the higher the priority



Operating System Examples

- Linux scheduling

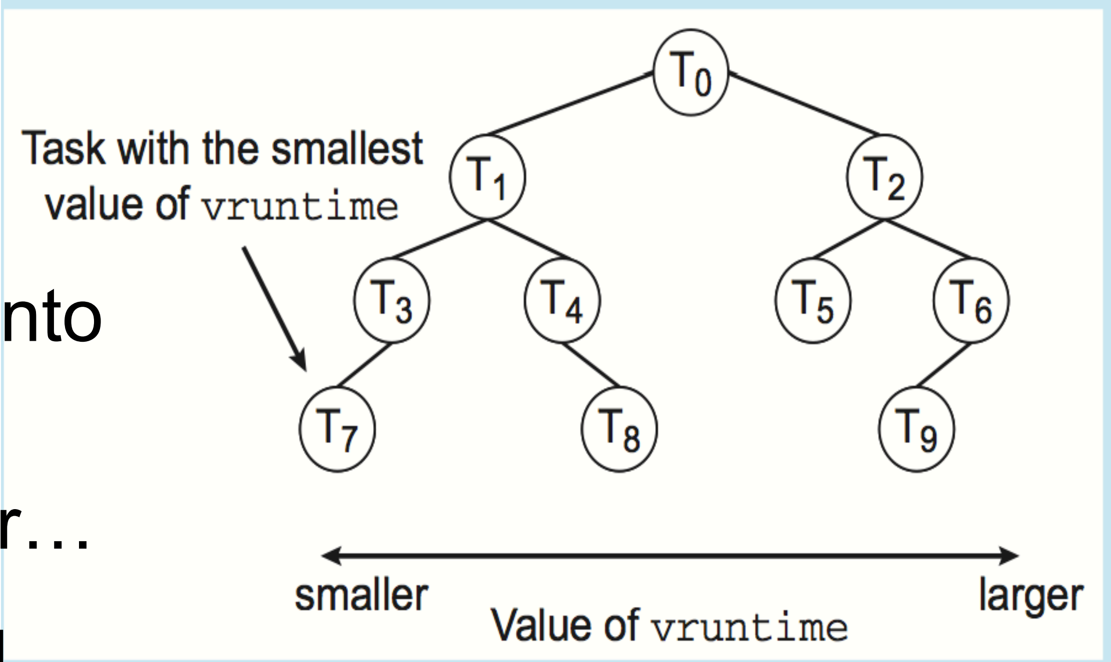
- Windows scheduling

Linux Scheduling in Version 2.6.23 +

- **Scheduling classes**
  - default: *Completely Fair Scheduler* (CFS)
  - real-time scheduling class (highest priority tasks)
- CFS
  - Quantum based on proportion of CPU time
  - per-task **virtual run time** in variable **`vruntime`**
    - vruntime += t, t is the amount of time it ran
    - Choose the task with the lowest vruntime
    - Normal default priority ➔ virtual run time = actual run time
    - decay factor based on priority of task – lower priority is higher decay rate ("bonus")
  - To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

- (Red-Black) Binary Search Tree, not queue
  - Insert finishing process into queue (n log n)
  - Pointer the lowest: faster…
  - RB tree is self-balancing



- Vruntime calculated based on **nice value** from -20 to +19
  - Lower value is higher priority; nice is static value
- What happens to I/O bound processes?
- Initialization value? vruntime = min_vruntime

User Mode Scheduling

- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like
    - C++ **Concurrent Runtime** (ConcRT) framework
- Linux has P-threads (and other thread packages)
- What happens when one thread blocks?

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
  - Deterministic
  - Proofs
  - queuing models
  - simulation
  - implementation

Deterministic Evaluation

- Group activity: calculate minimum average waiting time
  - FCFS
  - non-preemptive SJF
  - RR with quantum=10
  - Multilevel Feedback Queue

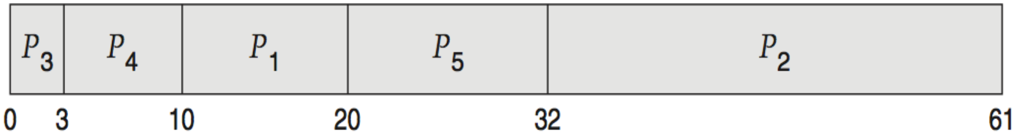
    (q0: 8; q1: 16; q2: FCFS)

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

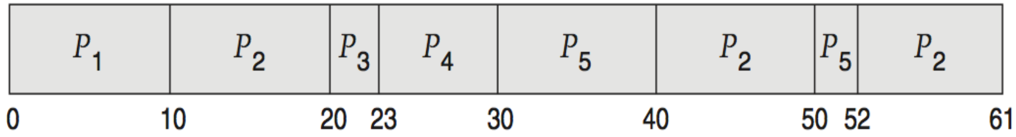
Simple and fast, but requires exact numbers for input, applies only to those inputs

- FCFS is 28ms:

- Non-preemptive SJF is 13ms:



- RR is 23ms:



Proofs

- Mathematical functions that you want to optimize
  - Metrics: response time, average response time, maximum response time, throughput, …
  - Optimize: minimize, maximize,
  - Assumptions: very important; realistic? Eg, all jobs available at time t=0
- Example: prove that SJF is optimal with respect to minimizing average response time

Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

  $$n = \lambda \times W$$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems
  - Event-driven or Time-Driven simulations

Implementation

- Even simulations have limited accuracy
- “Just” implement new scheduler and test in real systems
    - High cost, high risk
    - Environments vary
- Most flexible schedulers can be modified per-site or per-system
    - Or APIs to modify priorities
- But (again) environments vary and “can be modified” does not mean it’s easy ☺