ECE30021/ITP30002 Operating System

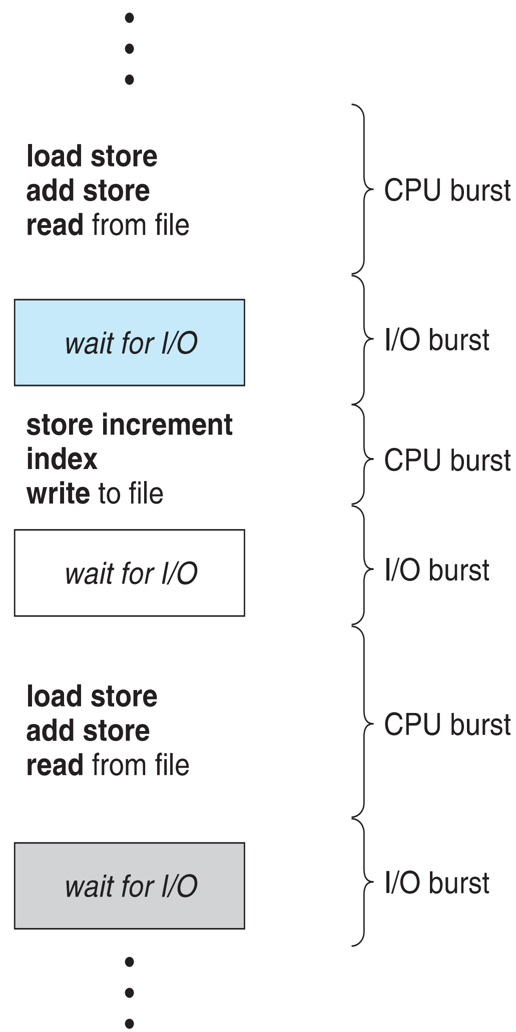
Process Scheduling

(OSC: Ch. 5)

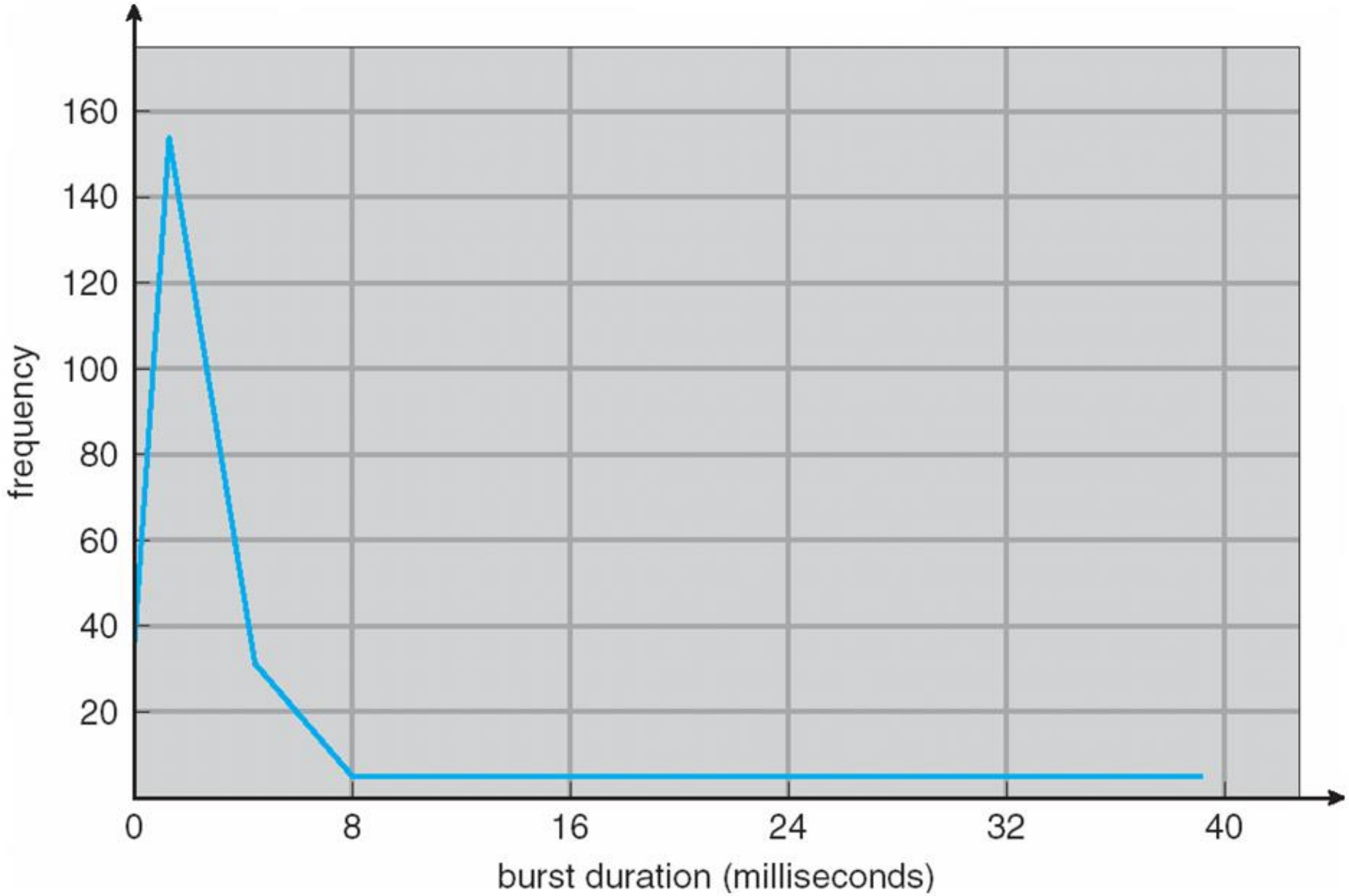
This lecture note is taken from the instructor's resource of Operating System Concept, 9/e and then partly edited/revised by Shin Hong.

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



Histogram of CPU-burst Times



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency: time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization**: keep the CPU as busy as possible
- **Throughput**: # of processes that complete their executions per time unit
- **Turnaround time**: the interval from the submission (launch) to the completion of a process
- **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)
- **Waiting time**: the sum of the periods spent waiting in the ready queue for a 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 (Cond't)

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

Shortest-Job-First (SJF) Scheduling

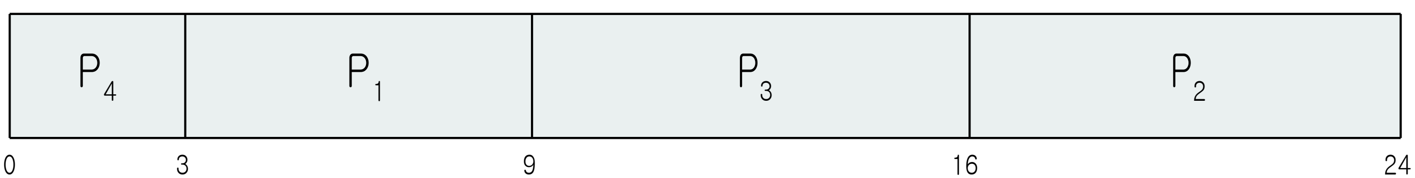
- Associate each process with the length of its next CPU burst, and schedule the process with the shortest time

- SJF is optimal: gives minimum average waiting time for a given set of processes
  - the difficulty is knowing the length of the next CPU request
  - could ask the user

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

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
  - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
  - Commonly, α set to ½
  - Preemptive version called **shortest-remaining-time-first**

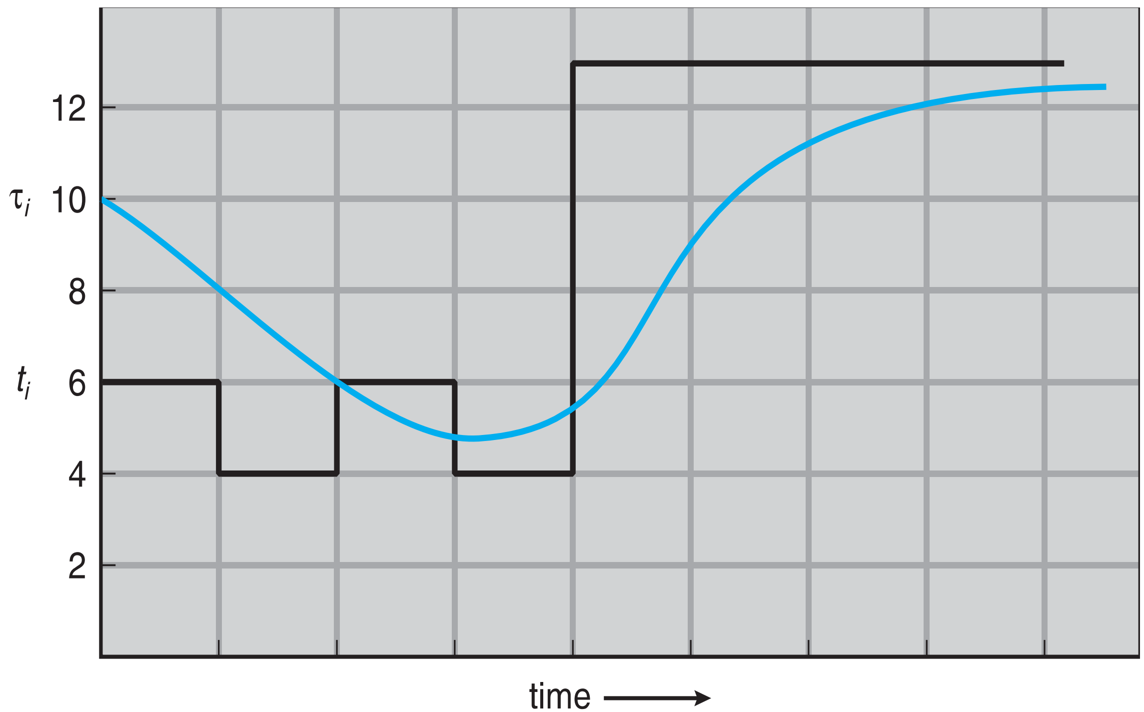
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - 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



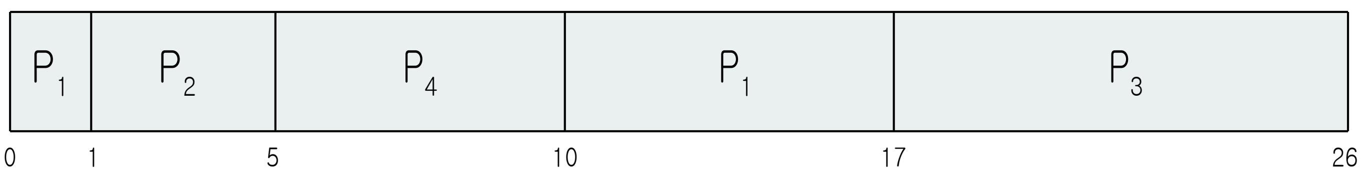
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Example of Shortest-remaining-time-first

- Now we add the 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

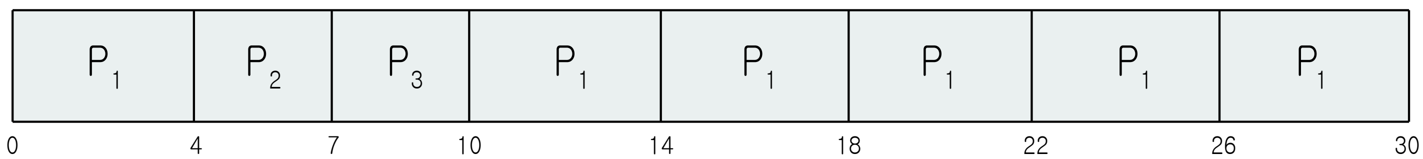
Round Robin (RR)

- 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.
- 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
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- *q* should be large compared to context switch time
  - q usually 10 ms to 100 ms, context switch < 10 usec

Priority Scheduling

- A priority number (integer) is associated with a process
- The CPU is allocated to the process with the highest priority (smallest integer indicates highest priority)
  - Preemptive
  - Non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Starvation problem: low priority processes may never
  - Aging: as time progresses upgrade 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

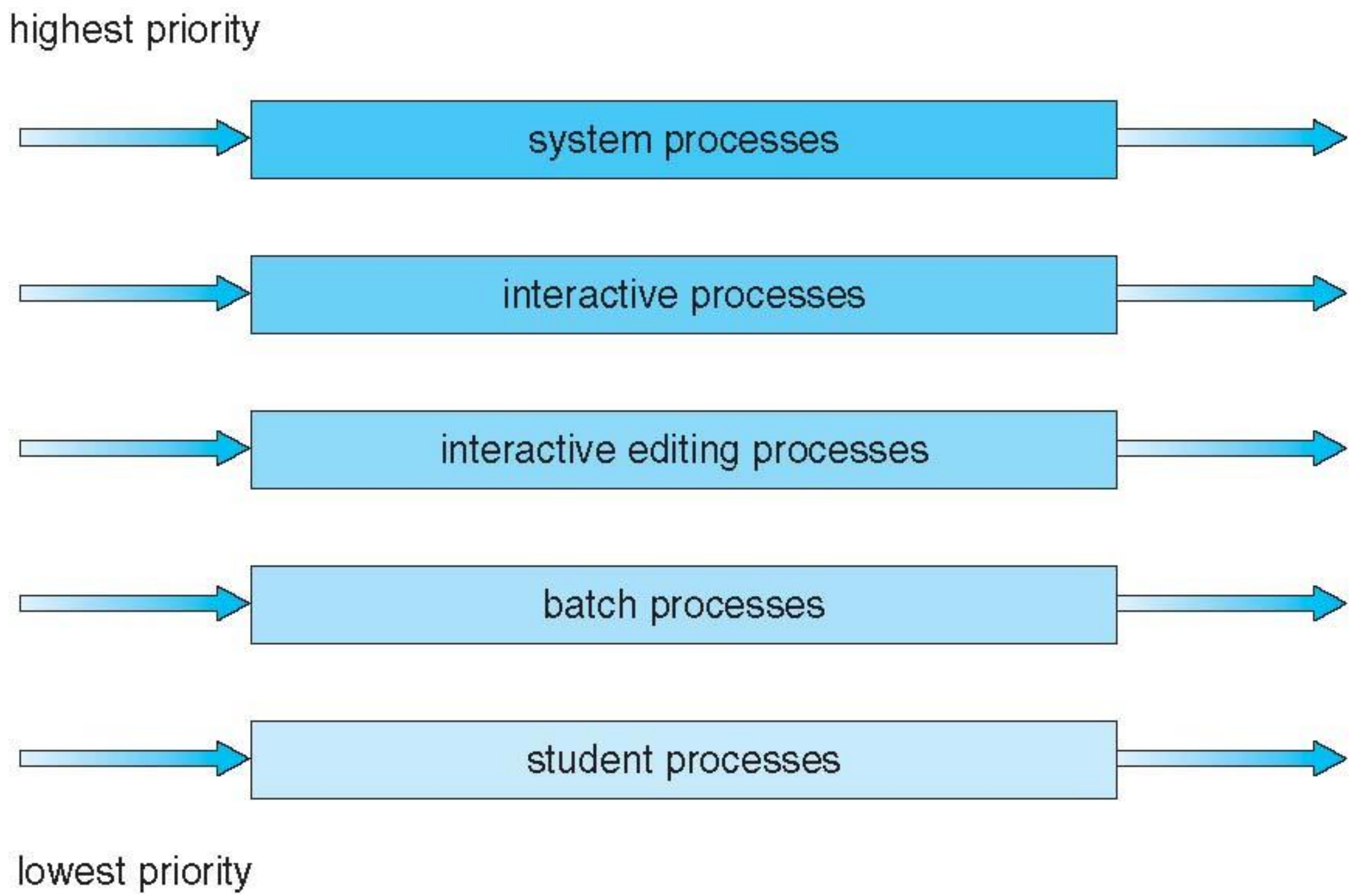
- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec

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; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling

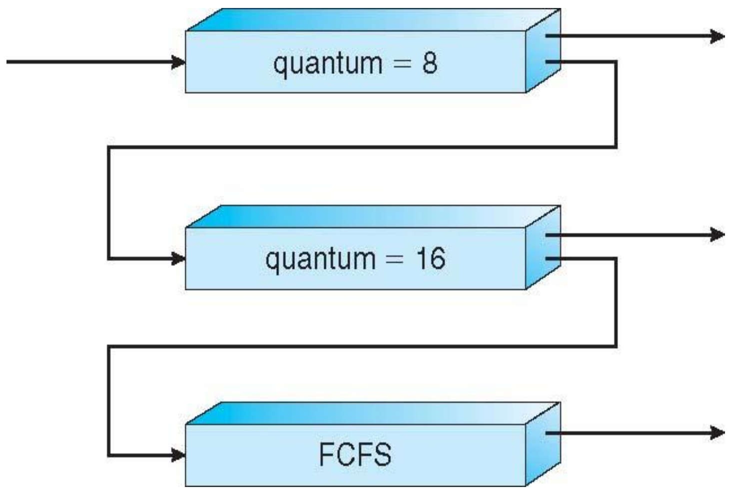


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- 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

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



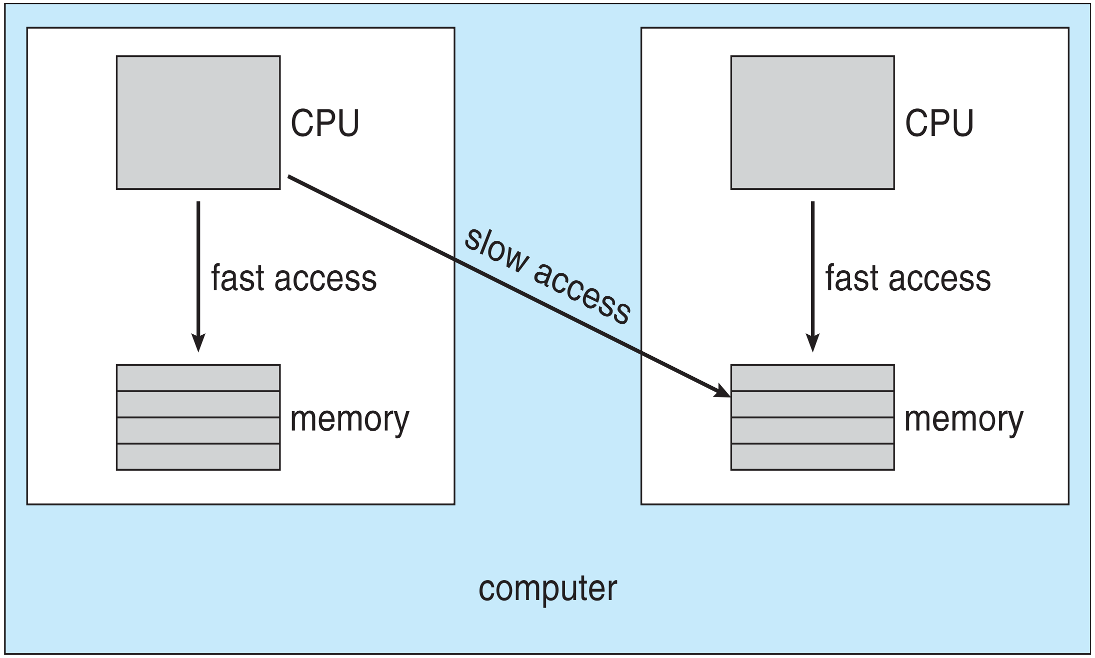
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

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
    - Currently, most common

Processor Affinity

- **Processor affinity:** a process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

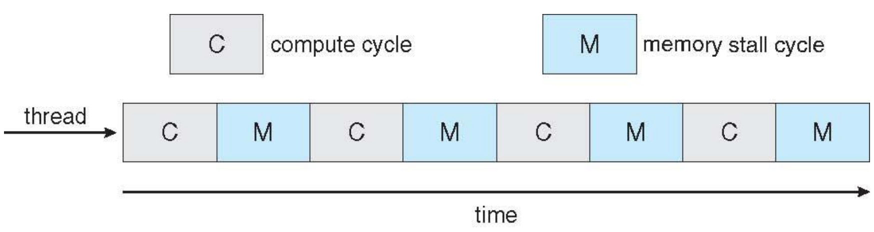


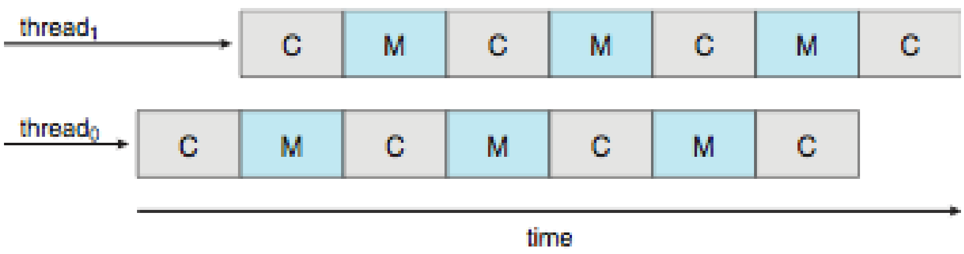
Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration: a periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- Pull migration: idle processors pulls waiting task from busy processor

Multicore Processors

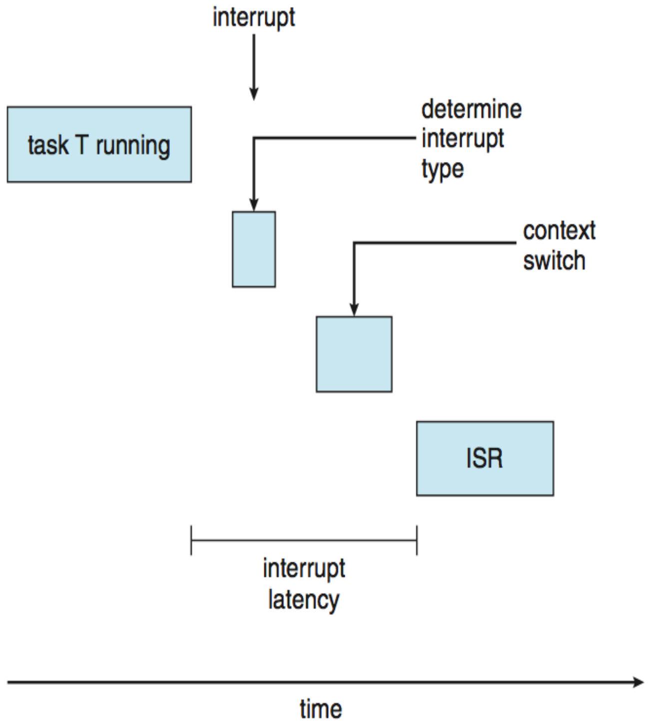
- Recent trend to place multiple processor cores on the same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens





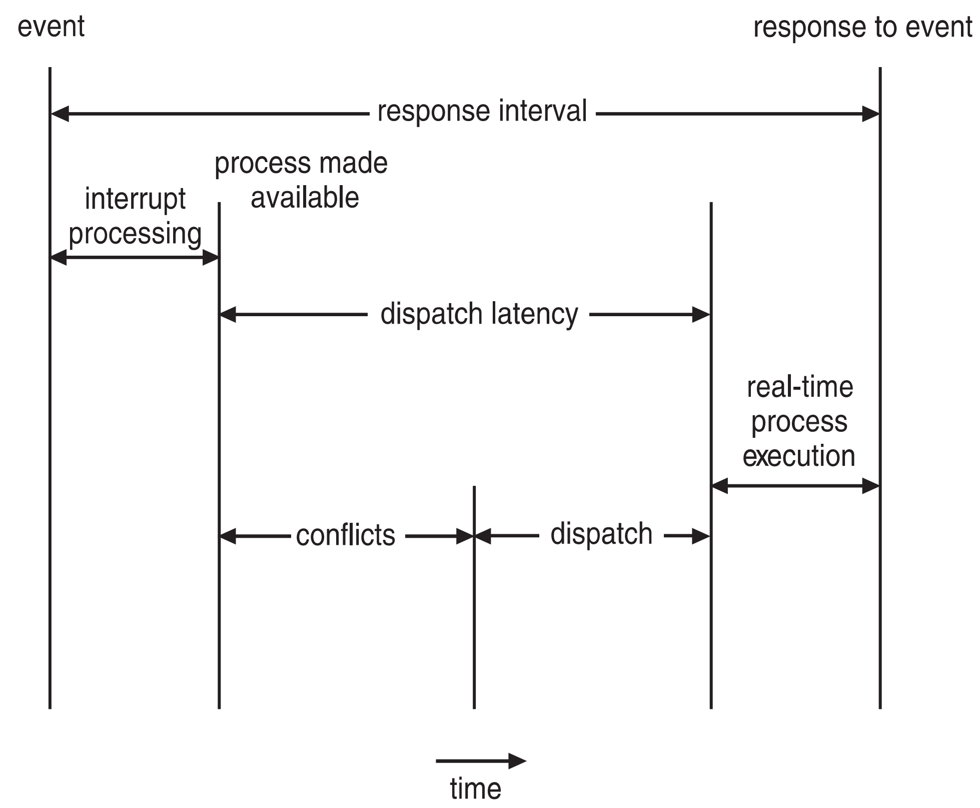
Real-Time CPU Scheduling

- **Soft real-time systems**: no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems**: task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency: time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency: time for schedule to take current process off CPU and switch to another



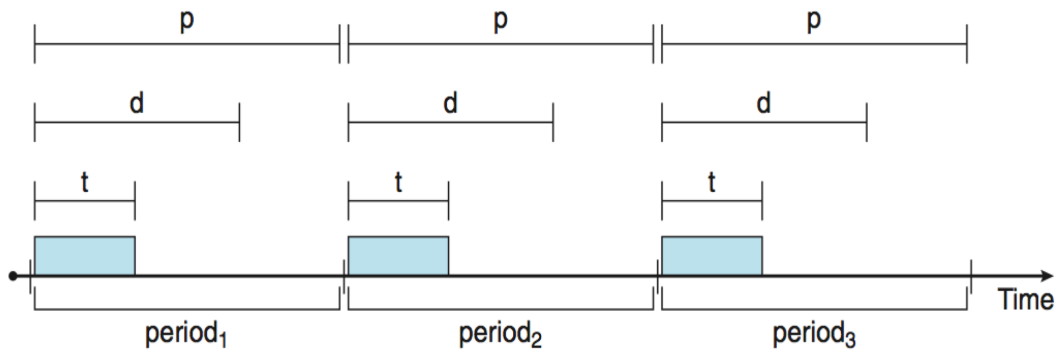
Real-Time CPU Scheduling (Cont.)

- 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



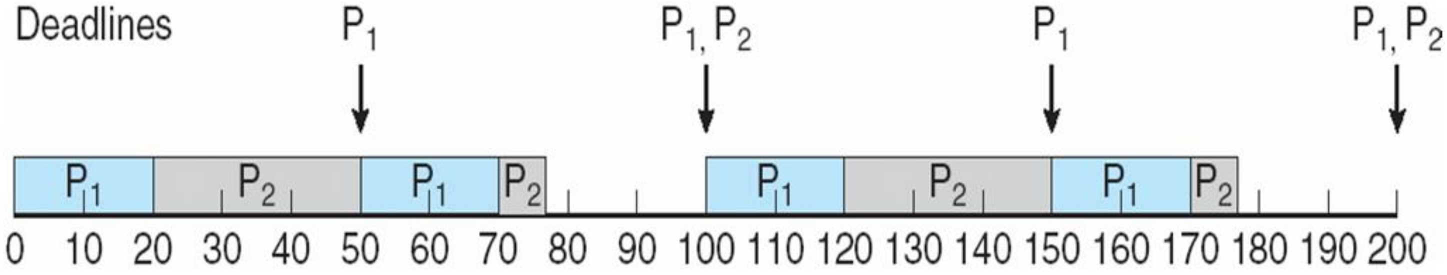
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
- For hard real-time must also provide ability to meet deadlines
  - admission-control
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



Rate Montonic Scheduling

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



$$t_1 = 20, \quad p_1 = 50$$
$$t_2 = 35, \quad p_2 = 100$$

RMS Schedulability

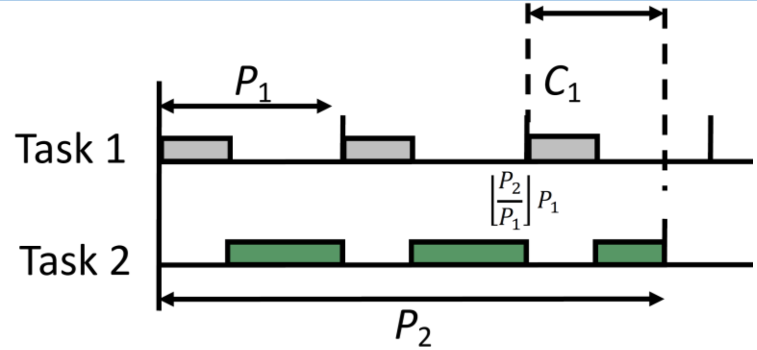
- It is possible to schedule given n tasks (e.g., CPU bursts) in hard-real time if they meet the following condition:

$$U \le n(2^{\frac{1}{n}} - 1)$$

  - C_i is the processing time of task i
  - P_i is the deadline and period of task i
  - $U = \frac{C_1}{P_1} + \frac{C_2}{P_2} + \cdots + \frac{C_n}{P_n}$

RMS Schedulability (2/)

- $n = 2$
  - Case I: $C_1 \leqslant P_2 - \left\lfloor \frac{P_2}{P_1} \right\rfloor P_1$



$$C_2 \leq P_2 - C_1 \left\lceil \frac{P_2}{P_1} \right\rceil = P_2 - C_1 \left(\left\lfloor \frac{P_2}{P_1} \right\rfloor + 1 \right)$$

$$U = \frac{C_1}{P_1} + \frac{C_2}{P_2} \leq 1 + \frac{C_1}{P_2} \left(\frac{P_2}{P_1} - \left\lfloor \frac{P_2}{P_1} \right\rfloor - 1 \right)$$

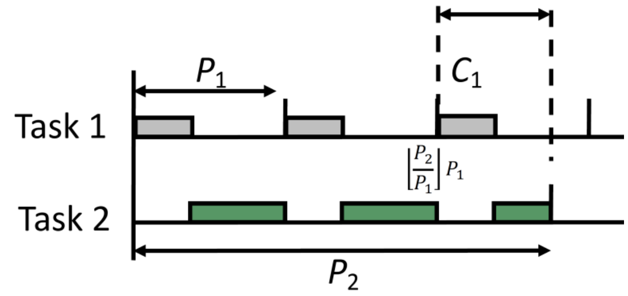
$$U \leq 1 + \frac{P_1}{P_2} \left(\frac{P_2}{P_1} - \left\lfloor \frac{P_2}{P_1} \right\rfloor \right) \left(\frac{P_2}{P_1} - \left\lfloor \frac{P_2}{P_1} \right\rfloor - 1 \right)$$

$$\boxed{\frac{P_2}{P_1} - \lfloor \frac{P_2}{P_1} \rfloor}$$

$$U \leq 1 - \frac{\left(\frac{P_2}{P_1} - \lfloor \frac{P_2}{P_1} \rfloor \right) \left(1 - \left(\frac{P_2}{P_1} - \lfloor \frac{P_2}{P_1} \rfloor \right) \right)}{P_2 / P_1} = 1 - \frac{G(1-G)}{G + \lfloor \frac{P_2}{P_1} \rfloor}$$

RMS Schedulability (3/)

- $n = 2$
  - Case 1: $C_1 \leqslant P_2 - \left\lfloor \frac{P_2}{P_1} \right\rfloor P_1$

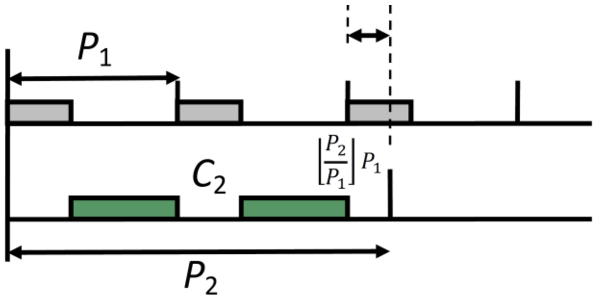


U is minimal when $\left\lfloor \frac{P_2}{P_1} \right\rfloor = 1$

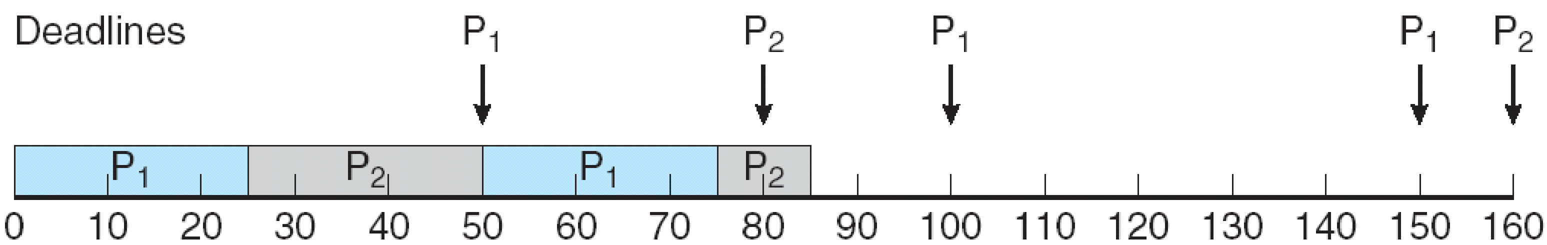
$$U \leq 1 + \frac{(P_2/P_1 - 1)(P_2/P_1 - 2)}{P_2/P_1} = 1 + \frac{(X-1)(X-2)}{X}$$

The minimal value of U is 0.83 when $X = \sqrt{2}$

  - Case 2: $C_1 > P_2 - \left\lfloor \frac{P_2}{P_1} \right\rfloor P_1$
    - The same result.



Missed Deadlines with Rate Monotonic Scheduling

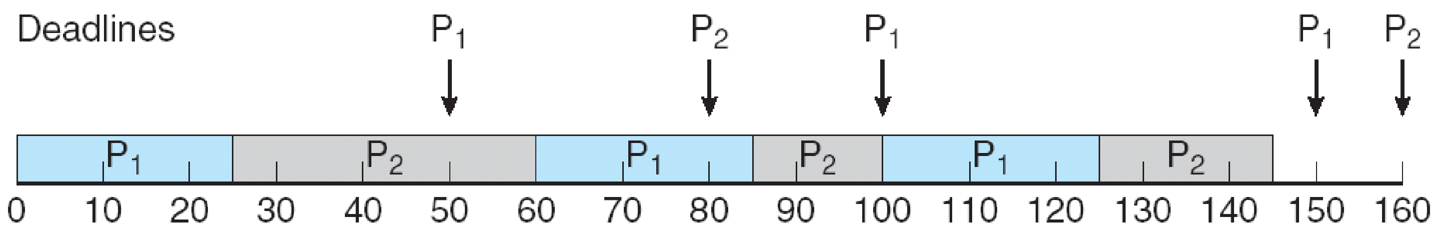


$$t_1 = 25, \quad p_1 = 50$$
$$t_2 = 35, \quad p_2 = 80$$

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority



$$t_1 = 25, \qquad p_1 = 50$$
$$t_2 = 35, \qquad p_2 = 80$$

Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive N / T of the total processor time

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

Operating System Examples

- Linux scheduling
- Windows scheduling

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
    - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
    - Higher priority gets larger q
  - Task runnable as long as time left in time slice (active)
  - If no time left (expired), not runnable until all other tasks use their slices
  - All runnable tasks tracked in per-CPU runqueue data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - when no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23+ (1/2)

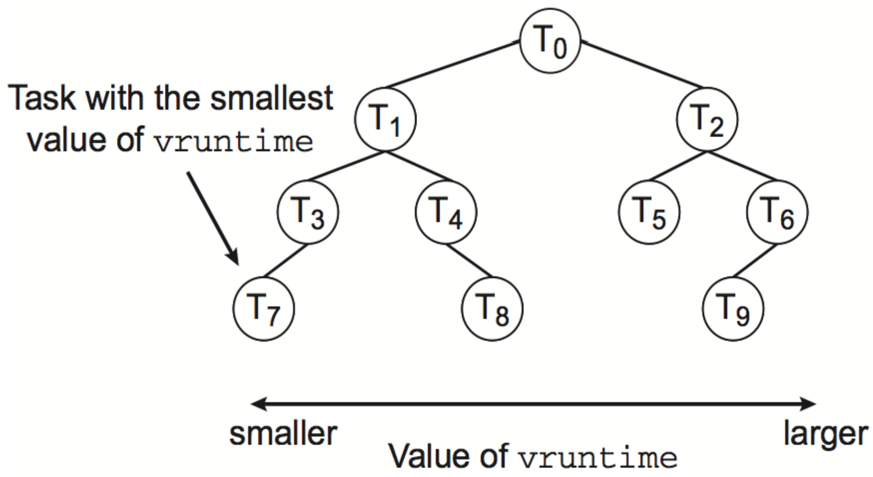
- Scheduling classes
  - Possible to run different scheduling schemes for different groups of processes
- Completely Fair Scheduling (CFS)
  - Default scheduling algorithm (i.e., SCHED_NORMAL)
  - For interactive applications such as GUI apps in desktop and web servers
- Quantum in CFS
  - CFS determines time slice given a runnable task (i.e., ready) by the following factors:
    - Minimum granularity: a minimum interval that a runnable task should run once it gets scheduled
    - The number of runnable tasks in a ready queue
    - Target latency: a time interval during which a runnable (i.e., ready) should run at least once
  - CFS determines a time slice of a runnable task as follows:
    - distribute a target latency to be propositional to the weights of runnable tasks
      - Weight of a task is determined by nice value (-20 to +19)
    - or, give at least minimum granularity

CFS: Completely fair process scheduling in Linux by Marty Kalin
https://opensource.com/article/19/2/fair-scheduling-linux

Linux Scheduling in Version 2.6.23+ (2/2)

- Priority in CFS
  - CFS dispatcher picks a task with smallest virtual runtime
  - Virtual runtime: the actual runtime of a task in the latest schedule
  - Smaller nice value (nicer), the higher scheduling priority

- CFS implements a ready queue as a red-black tree with vruntime as a key
  - one with highest priority is placed at the left-most leaf node
  - log(n) for insert and remove



CFS: Completely fair process scheduling in Linux by Marty Kalin
https://opensource.com/article/19/2/fair-scheduling-linux

Windows Scheduling

- Windows uses priority-based preemptive scheduling
  - Highest-priority thread runs next
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Real-time threads can preempt non-real-time
  - 32-level priority scheme
  - **Variable class** is 1-15, **real-time class** is 16-31
  - Priority 0 is memory-management thread
  - Queue for each priority
  - If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies priority classes to which a process can belong
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base