Scheduling
The art and science of allocating processors and other resources to processes & threads

Professor Hugh C. Lauer
CS-3013, Operating Systems

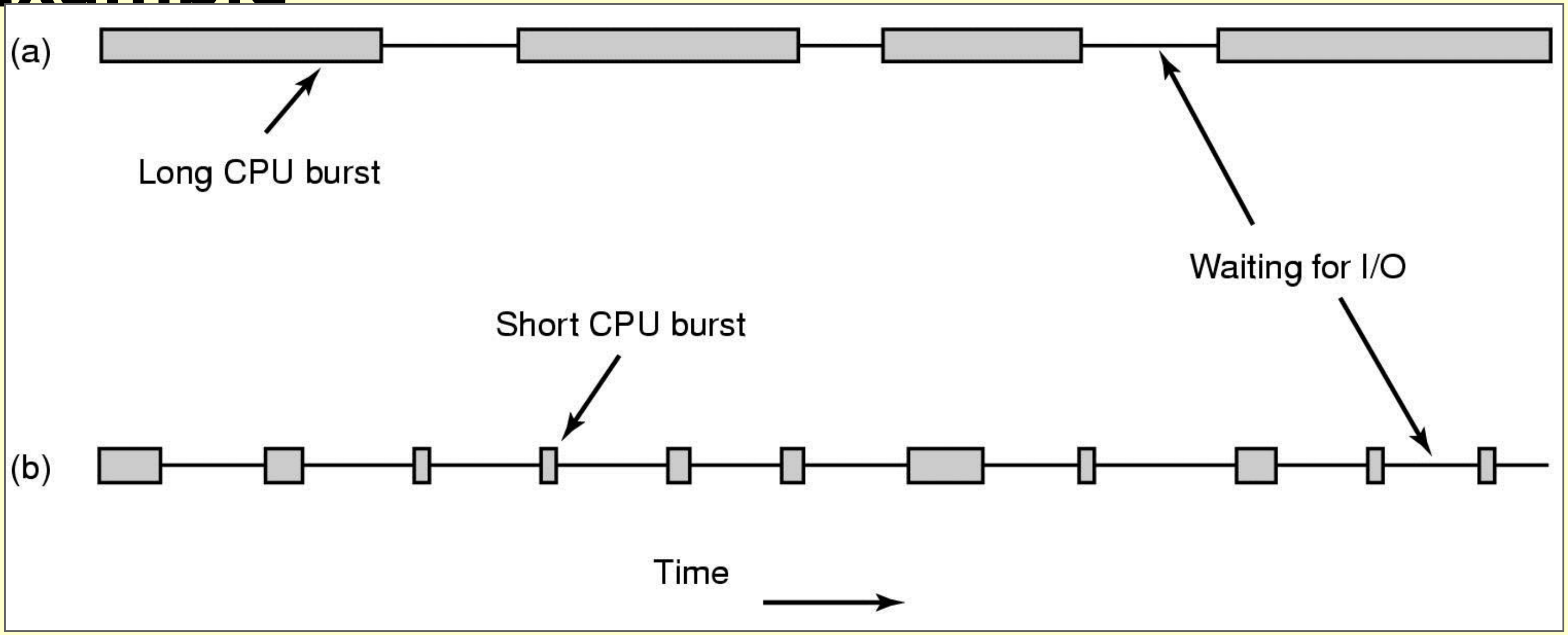
Why scheduling?

- We know *how* to switch processors among processes or threads, but …

- How do we decide *which* to choose next?

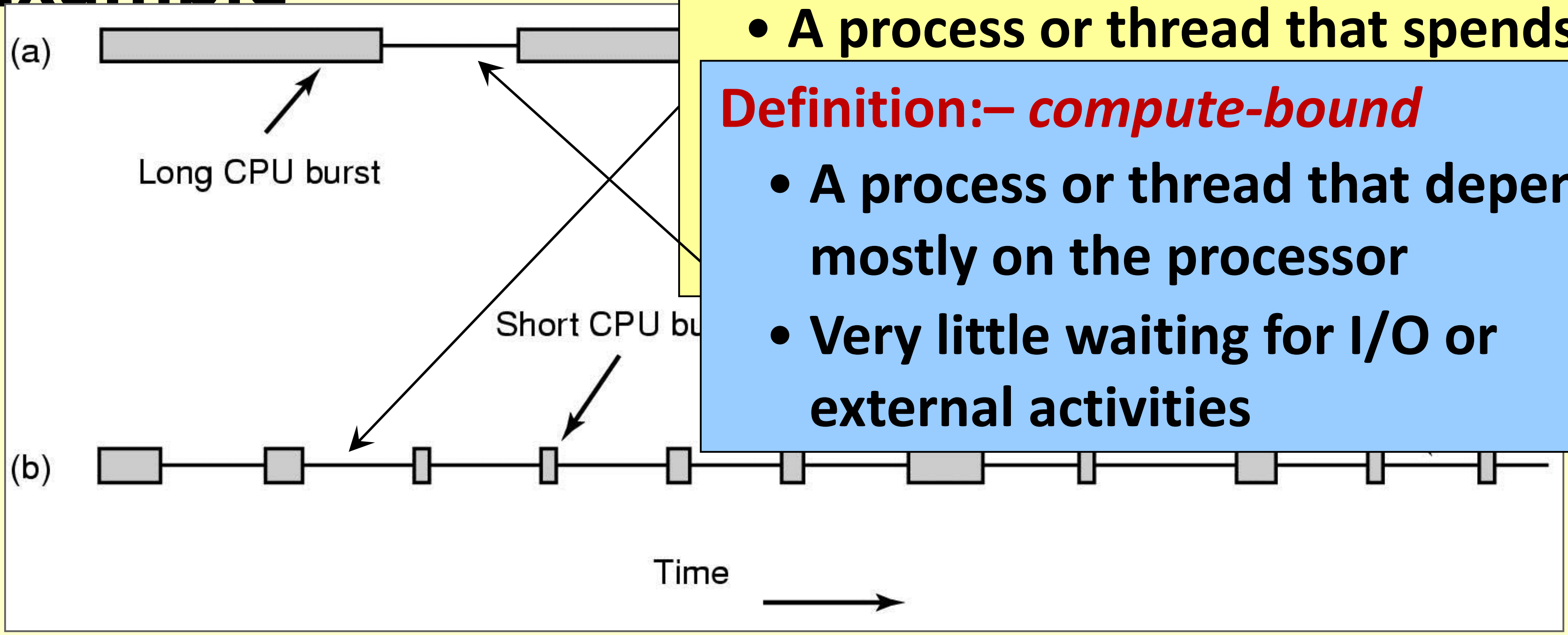
- Reading Assignment – OSTEP §7-10

Example



- **Bursts of processor usage alternate with periods of I/O wait**
  - a compute-bound process (a)
  - an I/O bound process (b)
- **Which process/thread should have preferred access to a processor?**
- **Which process/thread should have preferred access to I/O or disk?**
- **Why?**

Example

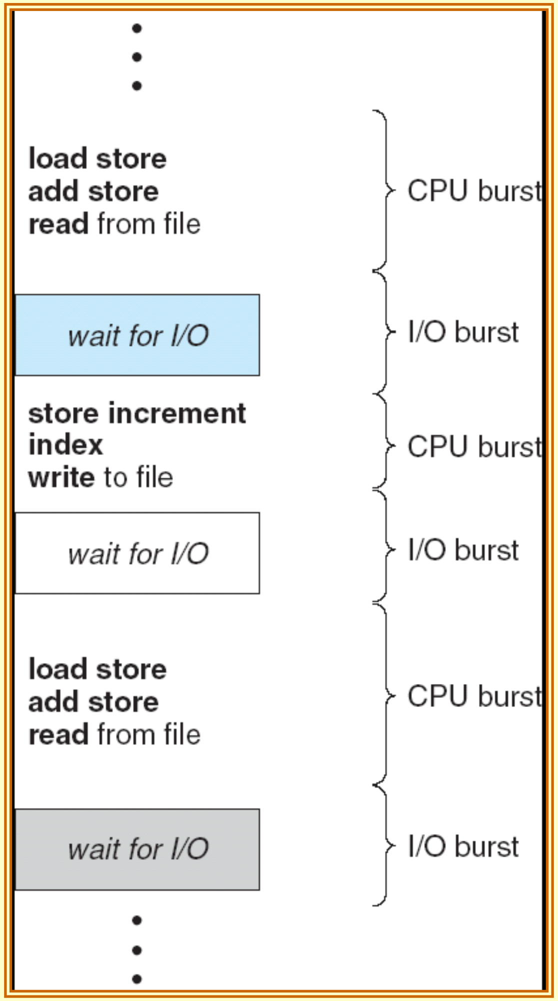


Definition:– *I/O-bound*
- **A process or thread that spends**

Definition:– *compute-bound*
- **A process or thread that depends mostly on the processor**
- **Very little waiting for I/O or external activities**

- **Bursts of processor usage alternate with periods of I/O wait**
  - a compute-bound process (a)
  - an I/O bound process (b)
- **Which process/thread should have preferred access to a processor?**
- **Which process/thread should have preferred access to I/O or disk?**
- **Why?**

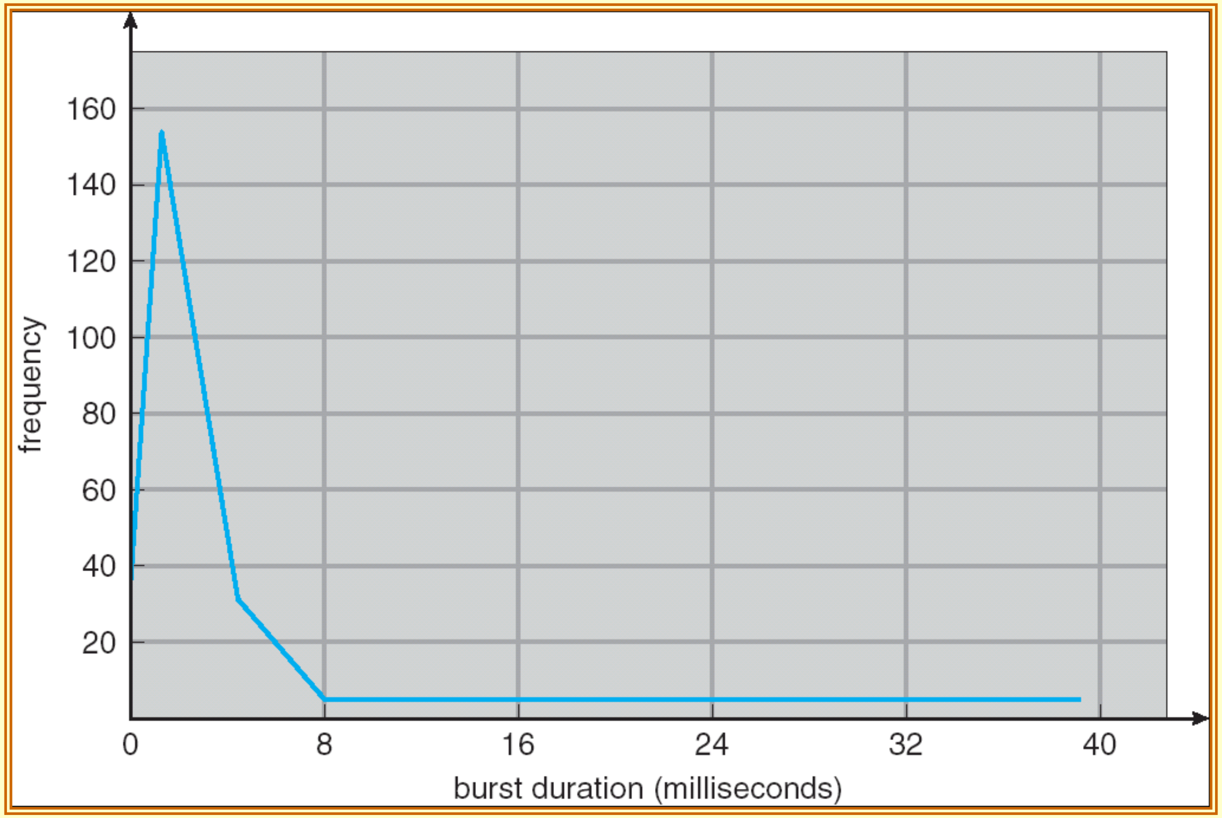
Alternating sequence of processor and I/O bursts



I/O bound **= short bursts of processing & long I/O waits**

Processor *bound* = long processor bursts & short I/O waits

Histogram of processor-burst times



Implementation of scheduling

- **Scheduler**
  - Policy

- **Dispatcher**
  - Mechanism

Sche

I.e., processes & threads

- **Selects from among the tasks in memory that are ready to execute, and allocates a processor to one of them**
- **Processor scheduling decisions may take place when a task:**
  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 *non-preemptive***
- **Scheduling under 2 and 3 is *preemptive***

Dispatcher

- **Dispatcher module gives control of a processor to the task selected by the scheduler:–**
  - switching context (registers, etc.)
  - Loading the PSW to switch to user mode and restart the selected program

- **Dispatch latency – time it takes for the dispatcher to stop one task and start another one running**
  - Non-trivial in some systems

Potential scheduling criteria

- *Processor utilization* – keep the processor(s) as busy as possible
- *Throughput* – # of tasks that complete their execution per time unit
- *Turnaround time* – amount of time to execute a particular task
- *Waiting time* – amount of time task has been waiting in the ready queue
- *Response time* – amount of time from request submission until first response is produced
- …

Considerations in scheduling policies

- **Issues**
  - *Fairness* – don't starve some tasks in favor of others
  - *Priorities* – most important first
  - *Deadlines* – task (or burst) X must be done by time *t*
  - *Optimization* – e.g. throughput, response time
- **Reality — No universal scheduling policy**
  - Many models
  - Determine what to optimize (define *metrics*)
  - Select an appropriate one and adjust based on experience

Note

- **In many situations, *scheduling* is not so important as it once was because …**
  a. Desktop, smart phones, and embedded systems focus on one or a few tasks at a time

  OR

  a. Systems have so much processing power relative to other subsystems that processors go idle

Scheduling – metrics

- *Simplicity* – easy to implement
- *Job latency* – time from start to completion
- *Interactive latency* – time from action start to expected system response
- *Throughput* – number of jobs completed
- *Utilization* – keep processor and/or subset of I/O devices busy
- *Determinism* – insure that jobs get done before some time or event
- *Fairness* – every job makes progress

Some task scheduling strategies

- **First-Come, First-Served (FCFS)**
- **Round Robin (RR)**
- **Shortest Job First (SJF)**
  - *Variation:* Shortest Completion Time First (SCTF)
- **Priority**
- **Real-Time**

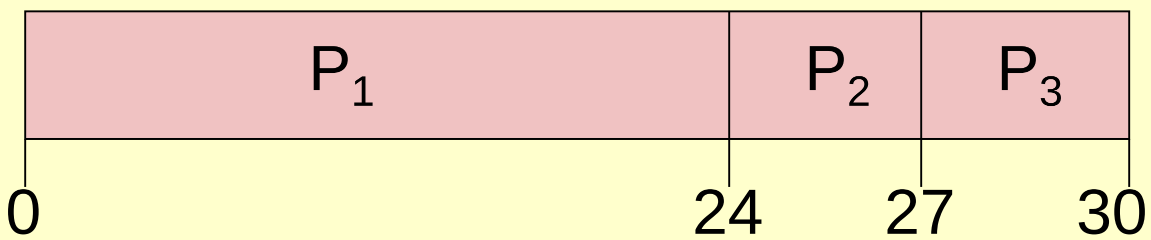
Scheduling policies — first come, first served (FCFS)

- **Easy to implement**
- **Non-preemptive**
  - I.e., no task is moved from *running* to *ready* state in favor of another one
- **Minimizes context switch overhead**

Example — FCFS scheduling

Task	Burst Time
P_1	24
P_2	3
P_3	3

- **Suppose that tasks arrive in the order: P_1 , P_2 , P_3**
- **The time line for the schedule is:–**



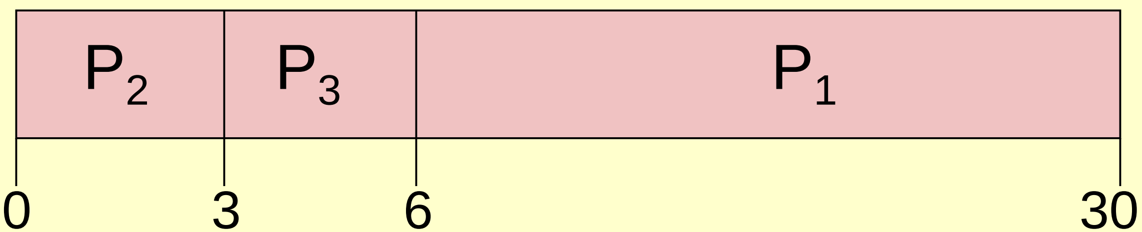
- **Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$**
- ***Average waiting time:*** **$(0 + 24 + 27)/3 = 17$**

Example: FCFS Scheduling (continued)

Suppose instead that the tasks arrive in the order

$$P_2 , P_3 , P_1$$

- **The time line for the schedule becomes:–**



- **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**
- **Previous case exhibits the *convoy effect***
  - short tasks stuck behind long task

FCFS scheduling (summary)

- **Favors *compute bound* jobs or tasks**
- **Short tasks penalized**
  - I.e., once a longer task gets the processor, it stays in the way of a bunch of shorter task
- **Appearance of random or erratic behavior to users**
- **Does not help in real situations**

Scheduling policies – round robin

- **Round robin (RR)**
  - *FCFS with preemption* based on time limits
  - Ready tasks given a *quantum* of time when scheduled
  - Task runs until quantum expires or until it blocks (whichever comes first)
  - Suitable for interactive (timesharing) systems
  - Setting quantum is critical for efficiency

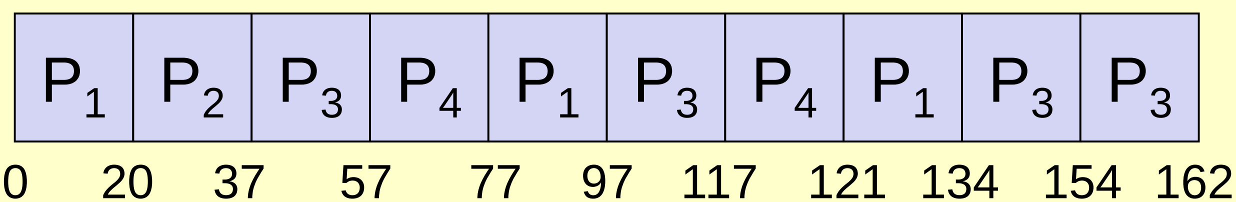
Round robin (continued)

- **Each task gets small unit of processor time (*quantum*), usually 10-100 milliseconds.**
  - After quantum has elapsed, task is preempted and added to end of ready queue.
- **If n tasks in ready queue and quantum = q, then each task gets $1/n$ of processor time in chunks of $\leq q$ time units.**
  - No task waits more than $(n-1)q$ time units.
- **Performance**
  - q large $\Rightarrow$ equivalent to FCFS
  - q small $\Rightarrow$ may be overwhelmed by context switches

Example of RR with time quantum = 20

Task	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- **The time line is:**



- **Typically, higher average turnaround than SJF, but better *response***

Comparison of RR and FCFS

Assume: **10 jobs each take 100 seconds – look at when jobs complete**

- **FCFS**
  - Job 1: 100s, job 2: 200s, … job 10:1000s
- **RR**
  - 1 sec quantum
  - Job 1: 991s, job 2 : 992s , … job 10:1000s
- **RR good for short jobs – worse for long jobs**

Application of round robin

- **Time-sharing systems**
- ***Fair* sharing of limited resource**
  - Each user gets *1/n* of processor
- **Useful where each user has *one* process to schedule**
  - Very popular in 1970s, 1980s, and 1990s
- **Not appropriate for desktop systems!**
  - *One* user, many processes and threads with very different characteristics

Shortest-job-first (SJF) scheduling

- **For each task, identify duration (i.e., length) of its next processor burst.**
- **Use these lengths to schedule task with shortest burst**

- **Two schemes:–**
  - *Non-preemptive* – once processor given to the task, it is not preempted until it completes its processor burst
  - *Preemptive* – if a new task arrives with burst length less than remaining time of current executing task, preempt.
    - This scheme is known as the Shortest-Remaining-Time-First (SRTF)

- **…**

Shortest-job-first scheduling (continued)

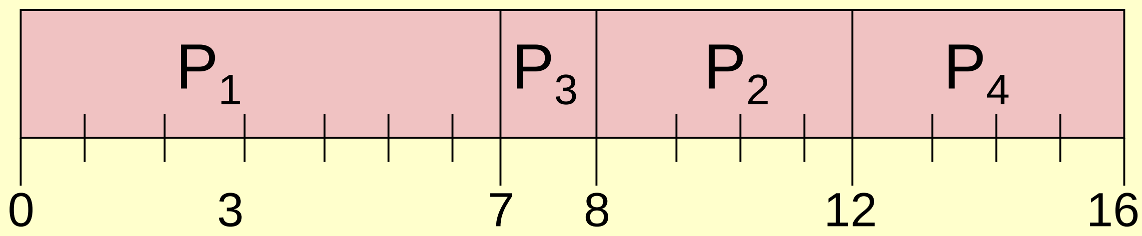
- **…**

- **SJF is provably optimal – gives *minimum average waiting time* for a given set of task bursts**
  - Moving a short burst ahead of a long one reduces wait time of short task more than it lengthens wait time of long one.

Example of non-preemptive SJF

Task	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **SJF (non-preemptive)**

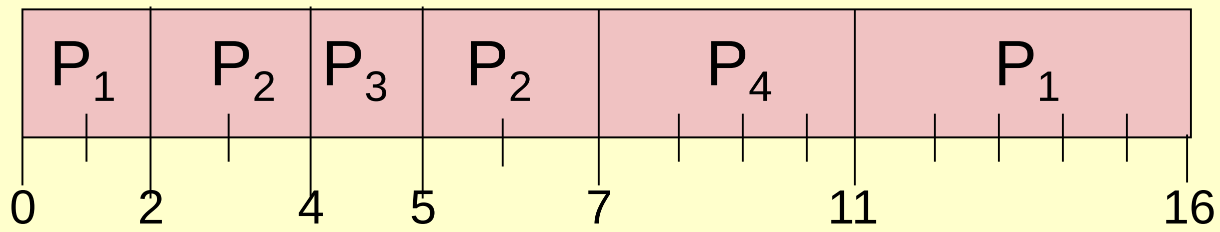


- **Average waiting time = (0 + 6 + 3 + 7)/4 = 4**

Example of preemptive SJF

Task	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **SJF (preemptive)**



- **Average waiting time = (9 + 1 + 0 +2)/4 = 3**

Determining length of next processor burst

- *Predict* from previous bursts
    - exponential averaging
- Let
    - t_n = actual length of n^{th} processor burst
    - τ_n = predicted length of n^{th} processor burst
    - α in range $0 \leq \alpha \leq 1$
- Then define
    - i.e., the weighted average of t_n and τ_n

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

Underlying principle:– *Estimate* behavior from past behavior

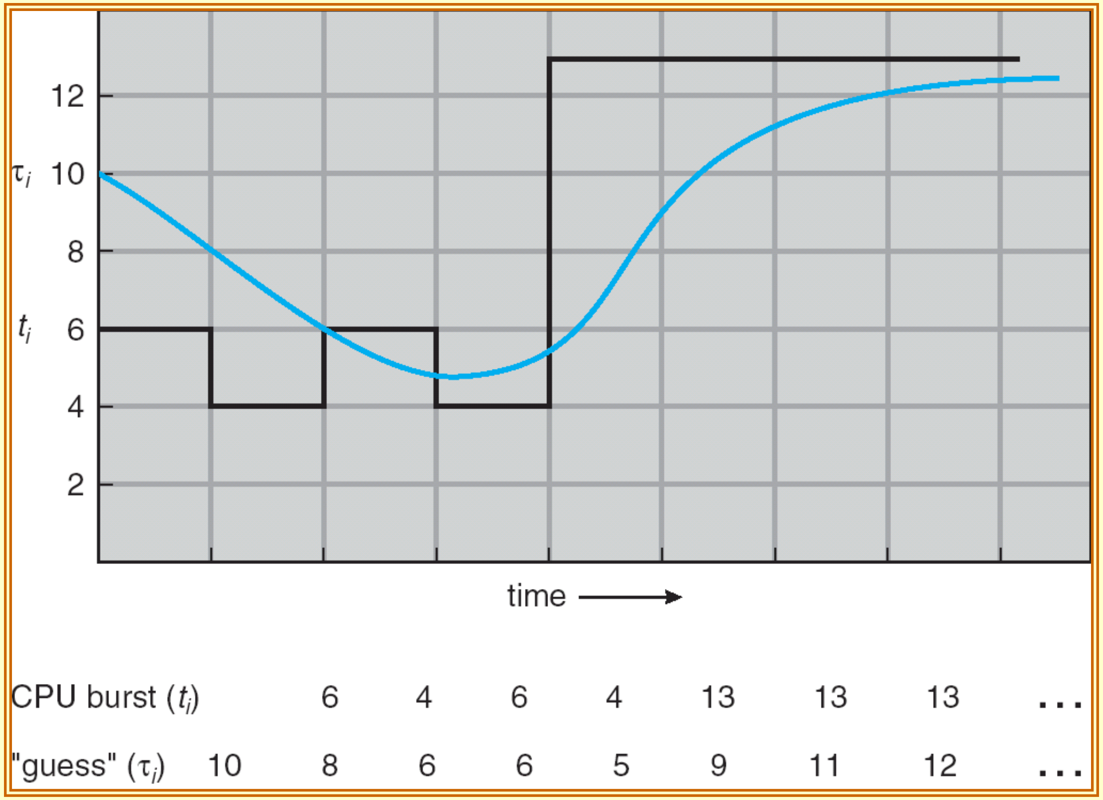
Note

- This is called *exponential averaging* because

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

- $\alpha = 0 \Rightarrow$ history has no effect
- $\alpha = 1 \Rightarrow$ only most recent burst counts
- Typically, $\alpha = 0.5$ and τ_0 is system average

Predicted length of the next processor burst



- **Notice how predicted burst length lags reality**
  - α defines how much it lags!

Applications of SJF scheduling

- **Multiple desktop windows active at once**
  - Document editing
  - Background computation (e.g., Photoshop)
  - Print spooling & background printing
  - Sending & fetching e-mail
  - Calendar and appointment tracking
- **Desktop word processing (at thread level)**
  - Keystroke input
  - Display output
  - Pagination
  - Spell checker

Some task scheduling strategies

- First-Come, First-Served (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)
  - *Variation:* Shortest Completion Time First (SCTF)
- Priority
- Real-Time

Priority scheduling

- **A priority number (integer) is associated with each task**
- **Processor is allocated to the task with the highest priority (smallest integer ≡ highest priority)**
  - Preemptive
  - nonpreemptive

Priority scheduling

- **(Usually) preemptive**
- **Tasks are given *priorities* and ranked**
  - Highest priority runs next
  - May be done with multiple queues – *multilevel*
- ***SJF* ≡ priority scheduling where priority is next predicted processor burst time**
- **Recalculate priority – many algorithms**
  - E.g. increase priority of I/O intensive jobs
  - E.g. favor tasks in memory
  - Must still meet system goals – e.g. response time

Priority scheduling issue #1

- **Problem: *Starvation***
  - I.e., low priority tasks may never execute

- **Solution: *Aging***
  - As time progresses, increase priority of waiting tasks

Priority scheduling issue #2

- ***Priority inversion***
  - *A* has high priority, *B* has medium priority, *C* has lowest priority
  - *C* acquires a resource that *A* needs to progress
  - *A* attempts to get resource, fails and busy waits
    - *C* never runs to release resource!

or

  - *A* attempts to get resource, fails and blocks
    - *B* (medium priority) enters system & hogs processor
    - *C* never runs!
- **Priority scheduling can't be naive**

Definition:– *Priority Inversion*

A high priority task blocked by a lower priority task

Solution

- **Some systems increase the priority of a process/task/job to**
    - Match level of resource

      or
    - Match level of waiting task

- **Some variation of this is implemented in almost all real-time operating systems**

Priority scheduling (conclusion)

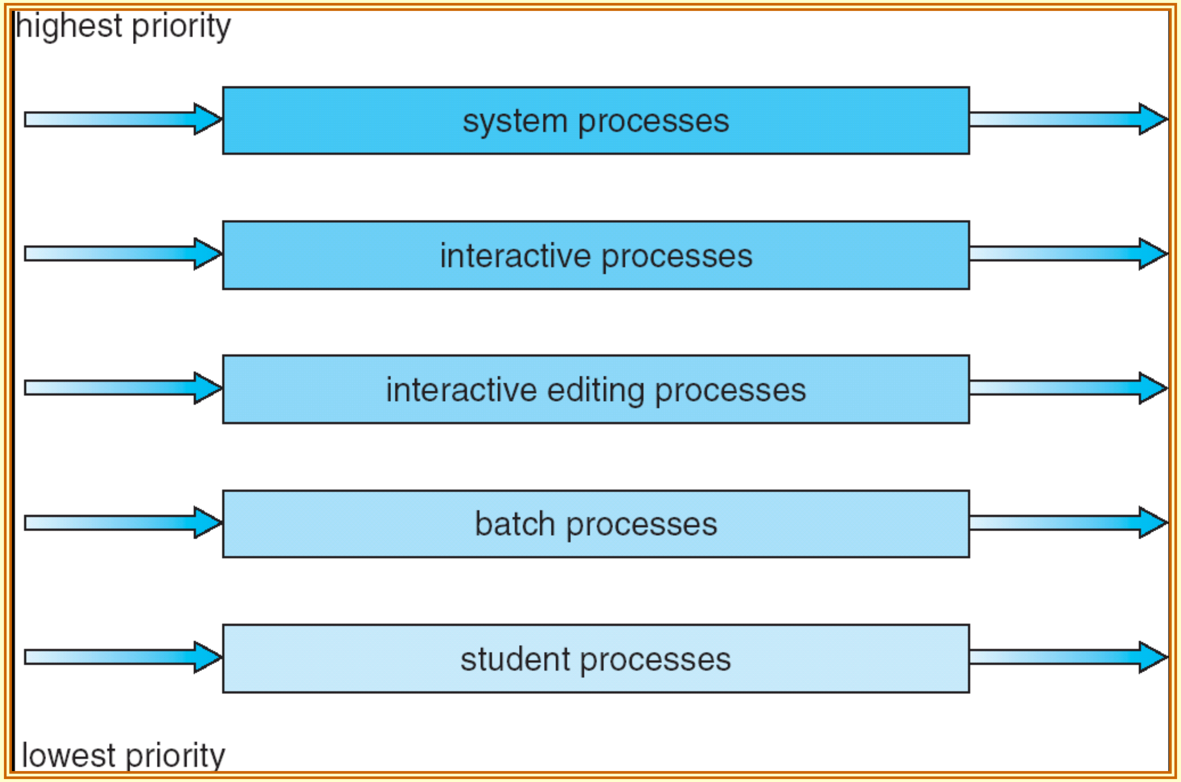
- Very useful if different kinds of tasks can be identified by level of importance

- Very irritating if used to create different classes of citizens

Multilevel queue — A variation on priority scheduling

- **Ready queue is partitioned into separate queues — e.g.,**
  - foreground (interactive)
  - background (non-interactive)
- **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 processor time to schedule amongst its tasks; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel queue scheduling



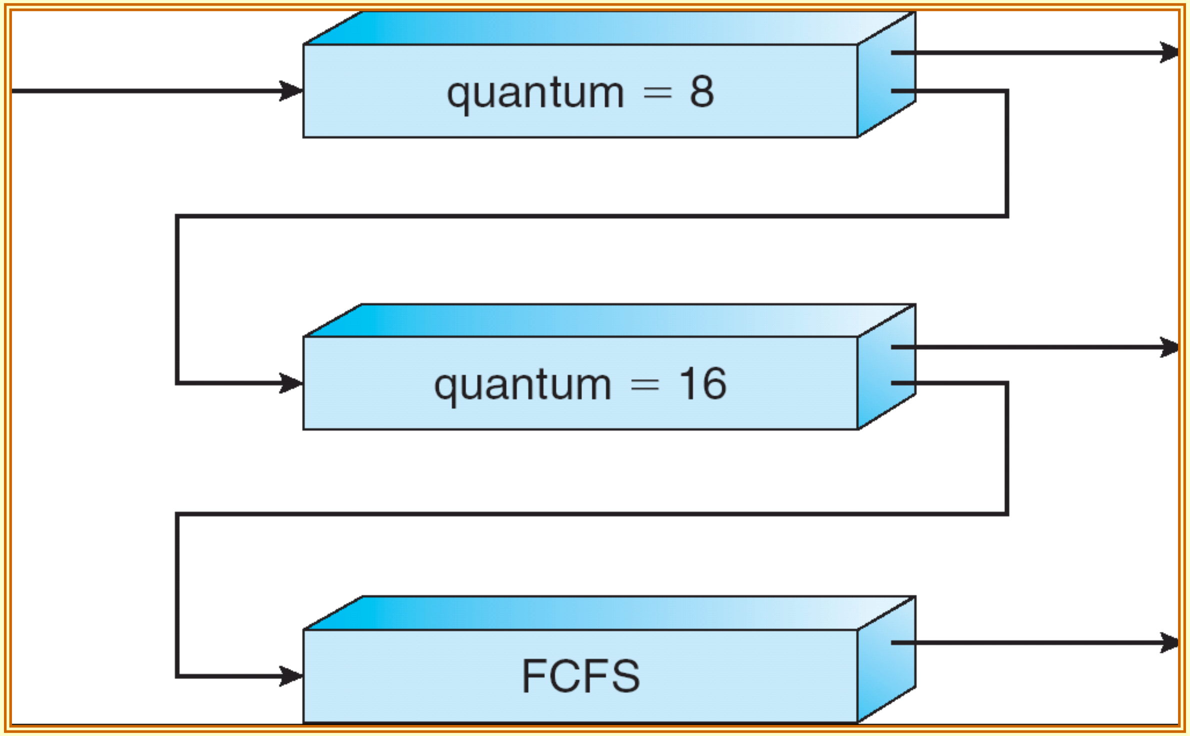
Multilevel feedback queue

- **A task 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 task
  - method used to determine when to demote a task
  - method used to determine which queue a task will enter when that task 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**
  - New job enters queue Q_0 (FCFS). When it gains processor, 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.

Multilevel feedback queues



Scheduling – examples

- **Unix – multilevel - many policies and many policy changes over time**
- **Linux – multilevel with 3 major levels**
  - Realtime FIFO
  - Realtime round robin
  - Timesharing
- **Windows Vista – two-dimensional priority policy**
  - *Process class* priorities
    - Real-time, high, above normal, normal, below normal, idle
  - *Thread* priorities relative to class priorities.
    - Time-critical, highest, …, idle

Reading Assignments

- **OSTEP**
  - §7-10: *Scheduling* (in four chapters)
- **Love, Chapter 4, *Process Scheduling***
  - Esp. pp. 47-50
- **Much overlap between the two**
  - OSTEP tends to be broader overview
  - Love tend to be more practical about Linux

Questions?

Some Task Scheduling Strategies

- First-Come, First-Served (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)
  - *Variation:* Shortest Completion Time First (SCTF)
- Priority
- **Real-Time**

Real-time scheduling

- **When you need to meet deadlines in the physical world**
  - According to the “real world” clock
- **Audio or video player**
  - to avoid “jerky” presentation, blips and bleeps, etc.
- **Process control – to react to physical processes**
  - Power plants, refineries, steel mills, nuclear reactors
  - Aircraft control, autopilots, etc.
  - Automatic braking systems
  - …

Two common approaches

- Rate Monotonic Scheduling
- Earliest Deadline First

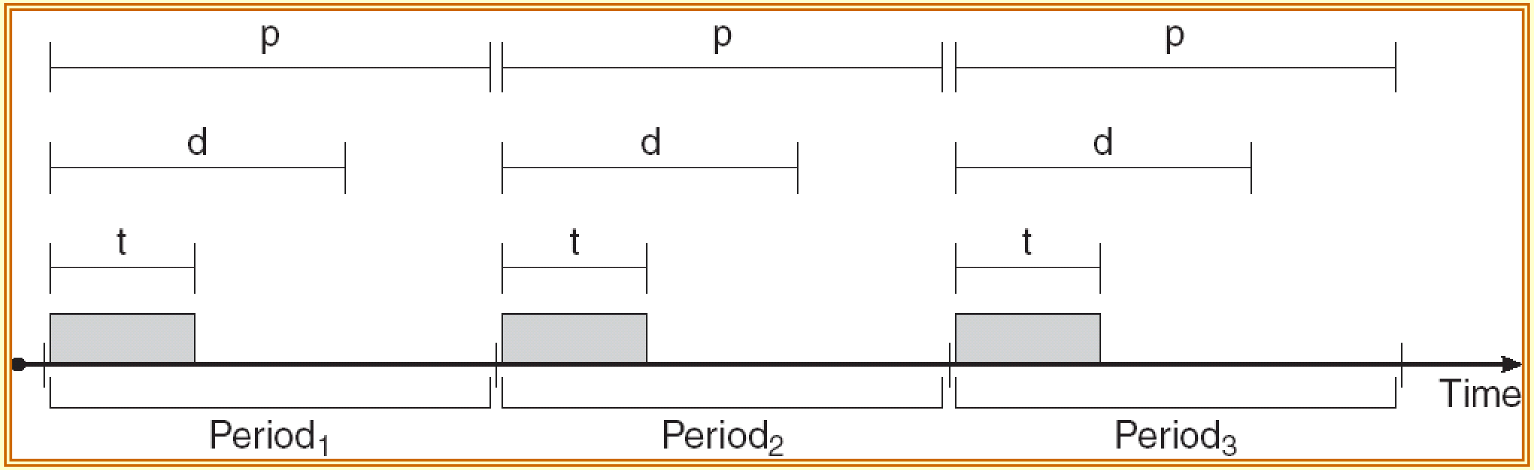
- Many variations
- Many analytic methods for proving QoS (*Quality of Service*)

Rate monotonic scheduling (RMS)

- **Assume *m* periodic processes**
  - Process *i* requires t_i milliseconds of processing time every p_i milliseconds .
  - Equal processing every interval — like clockwork!

Example

- **Periodic process *i* requires the processor at specified intervals (periods)**
- **p_i is the duration of the period**
- **t_i is the processing time**
- **d_i is the deadline by when the process must be serviced**
  - Often same as end of period



Rate monotonic scheduling (RMS)

- **Assume *m* periodic processes**
  - Process *i* requires t_i milliseconds of processing time every p_i milliseconds .
  - Equal processing every interval — like clockwork!
- **Assume** $$\sum_{i=1}^{m} \frac{t_i}{p_i} \leq 1$$
- **Assign priority of process *i* to be** $\frac{1}{p_i}$
  - Statically assigned
- **Let priority of non-real-time processes be *0***

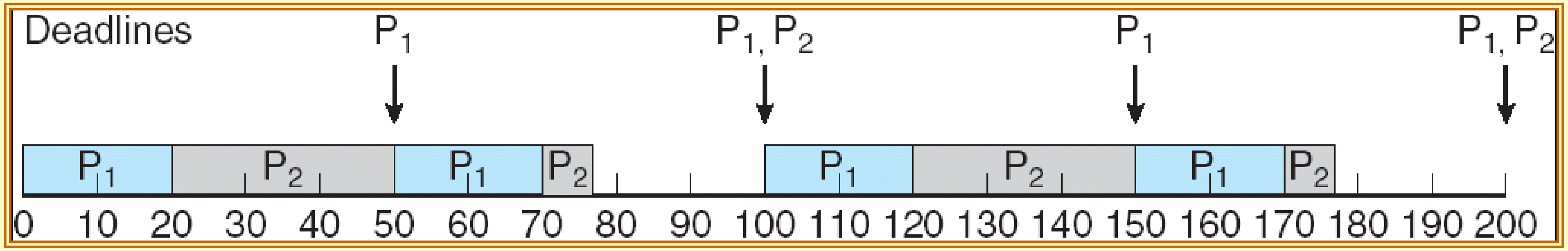
Rate monotonic scheduling (continued)

- **Scheduler simply runs highest priority process that is ready**
  - May pre-empt other real-time processes
  - Real-time processes become ready in time for each frame or sound interval
  - Non-real-time processes run only when no real-time process needs processor

Example

- p_1 = *50* msec; t_1 = 20 msec
- p_2 = *100* msec; t_2 = 35 msec
- Priority(p_1) > Priority(p_2)

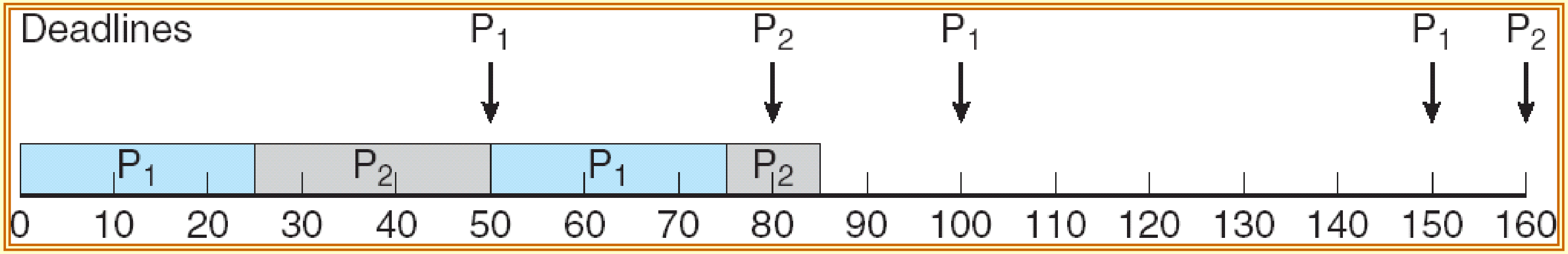
- Total compute load is 75 msec per every 100 msec.
- Both tasks complete within every period
  - 25 msec per 100 msec to spare



Example 2

- $p_1 = 50$ msec; $t_1 = 25$ msec
- $p_2 = 80$ msec; $t_2 = 35$ msec
- Priority(p_1) > Priority(p_2)

- Total compute load is ~ 94% of processor.
- Cannot complete both tasks within some periods
  - Even though there is still processor capacity to spare!



Rate Monotonic Theorems (without proof)

- *Theorem 1:* **using these priorities, scheduler can guarantee the needed *Quality of Service* (QoS), provided that**

$$\sum_{i=1}^{m} \frac{t_i}{p_i} \leq m(2^{1/m} - 1)$$

  - Asymptotically approaches *ln 2* as $m \rightarrow \infty$

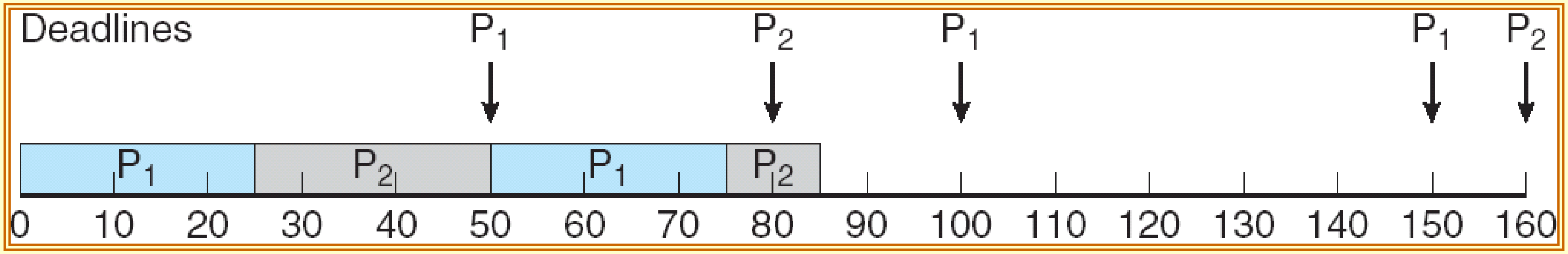
    ln 2 = 0.6931…

- *Theorem 2:* **If a set of processes can be scheduled by any method of static priorities, then it can be scheduled by Rate Monotonic method.**

Example 2 again

- Note that ***p₁*** pre-empts ***p₂*** in second interval, even though ***p₂*** has the earlier deadline!

$$\left(\frac{t_1}{p_1}+\frac{t_2}{p_2}\right)=\left(\frac{25}{50}+\frac{35}{80}\right)=0.9375>2\left(2^{1/2}-1\right)=0.828$$



More on rate monotonic scheduling

- Rate Monotonic assumes periodic processes

- MPEG-2 playback is *not* a periodic process!

Liu, C. L. and Layland, James W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery* (JACM), vol. 20, #1, January 1973, pp-46-61. (.pdf)

Earliest deadline first (EDF)

- **When each process *i* become ready, it specifies deadline D_i for its next *task*.**
- **Scheduler always assigns processor to process with earliest deadline.**
  - May pre-empt other real-time processes

Earliest deadline first scheduling (continued)

- **No assumption of periodicity**
- **No assumption of uniform processing times**

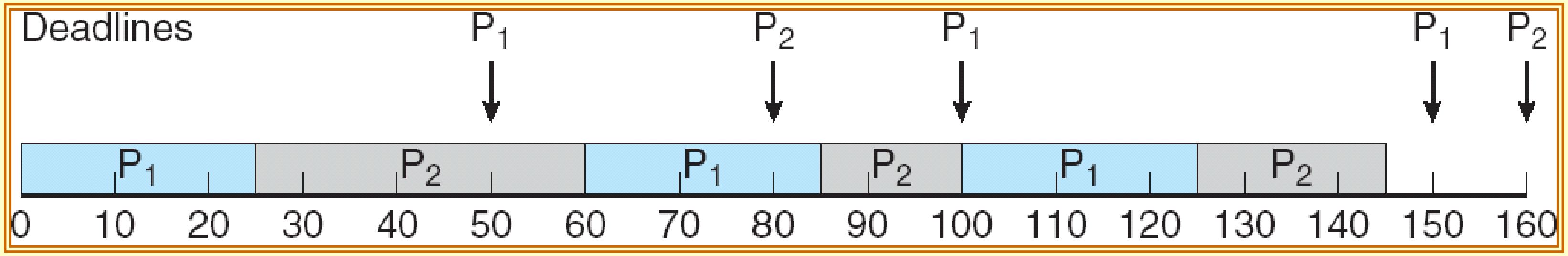
- **Theorem: If *any* scheduling policy can satisfy QoS requirement for a sequence of real time tasks, then EDF can also satisfy it.**
  - *Proof:* If i scheduled before $i+1$, but $D_{i+1} < D_i$, then i and $i+1$ can be interchanged without affecting QoS guarantee to either one.

Earliest deadline first scheduling (continued)

- **EDF is more complex scheduling algorithm**
  - Priorities are dynamically calculated
  - Processes must know deadlines for tasks
- **EDF can make higher use of processor than RMS**
  - Up to 100%
- **There is a large body of knowledge and theorems about EDF analysis**

Example 2 (again)

- **Priorities are assigned according to deadlines:**
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority.



Some task scheduling strategies

- First-Come, First-Served (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)
  - *Variation:* Shortest Completion Time First (SCTF)
- Priority
- Real-Time

Lots of other Scheduling Strategies for Different purposes

Scheduling – summary

- **General theme – what is the "best way" to run *n* tasks on *k* resources? (*k* < *n*)**
- **Conflicting Objectives – no one "best way"**
  - Latency vs. throughput
  - Speed vs. fairness
- **Incomplete knowledge**
  - E.g. – does user know how long a job will take
- **Real world limitations**
  - E.g. context switching takes processor time
  - Job loads are unpredictable

Scheduling – summary (continued)

- **Bottom line – scheduling is hard!**
  - Know the models
  - Adjust based upon system experience
  - Dynamically adjust based on execution patterns

Questions?