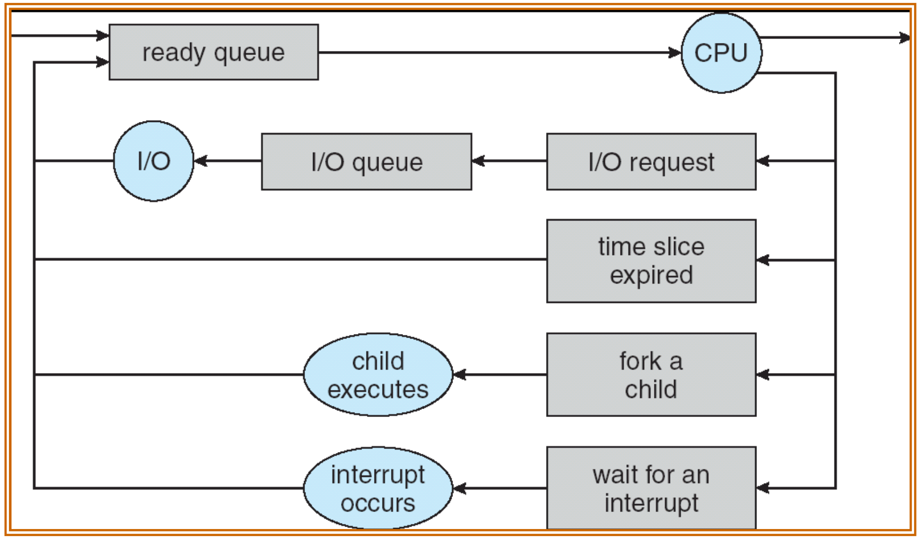
Operating System

CPU Scheduling

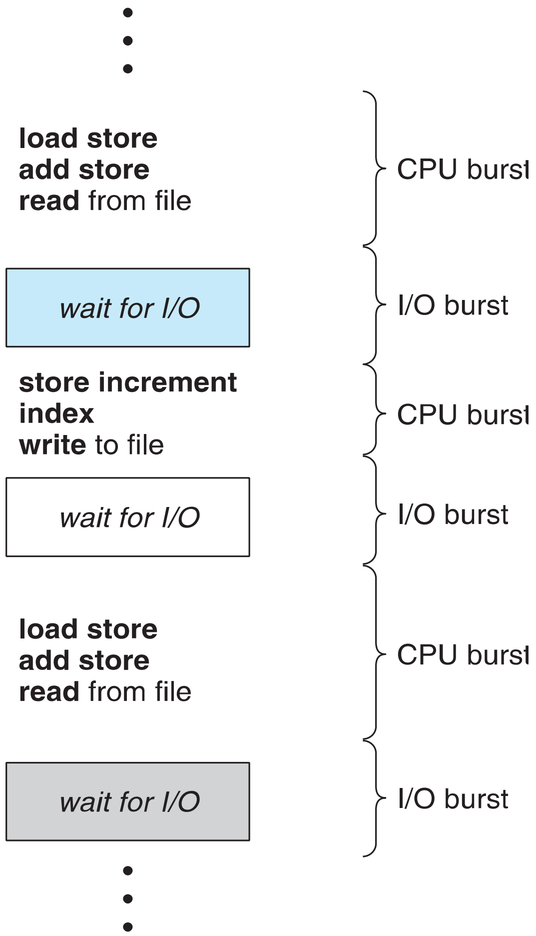
Recall: Scheduling

- Question: How is the OS to decide which of several tasks to take off a queue?
- Scheduling: deciding which threads are given access to resources from moment to moment
  - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access



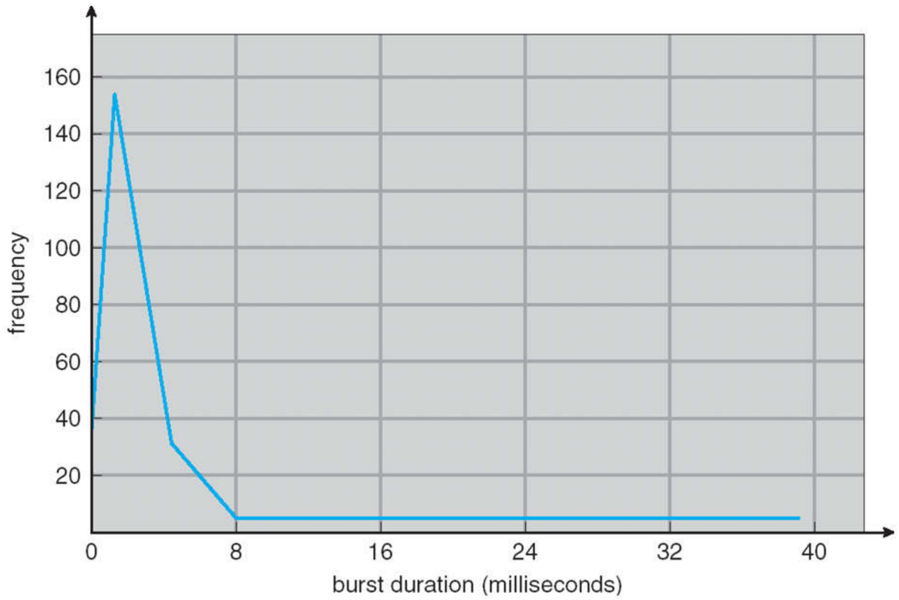
Motivation

- To make computer more productive
  - Maximum CPU utilization obtained with multiprogramming
- Process scheduling or Thread scheduling
- Having different sequence of IO or CPU
  - 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



CPU burst curve

- Exponential or hyperexponential

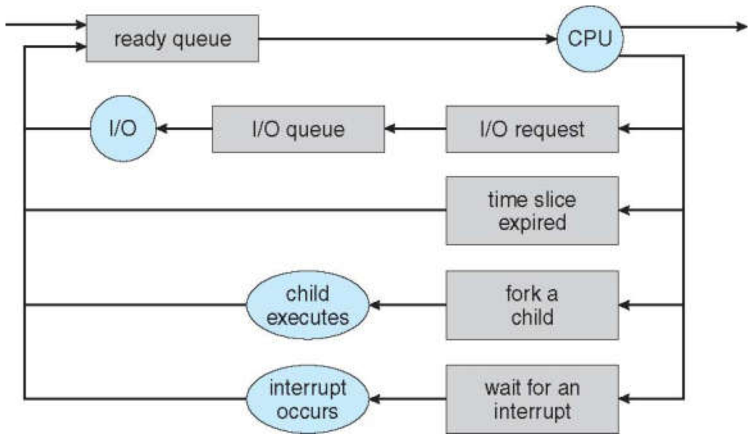


- a large number of short CPU bursts
- a small number of long CPU bursts

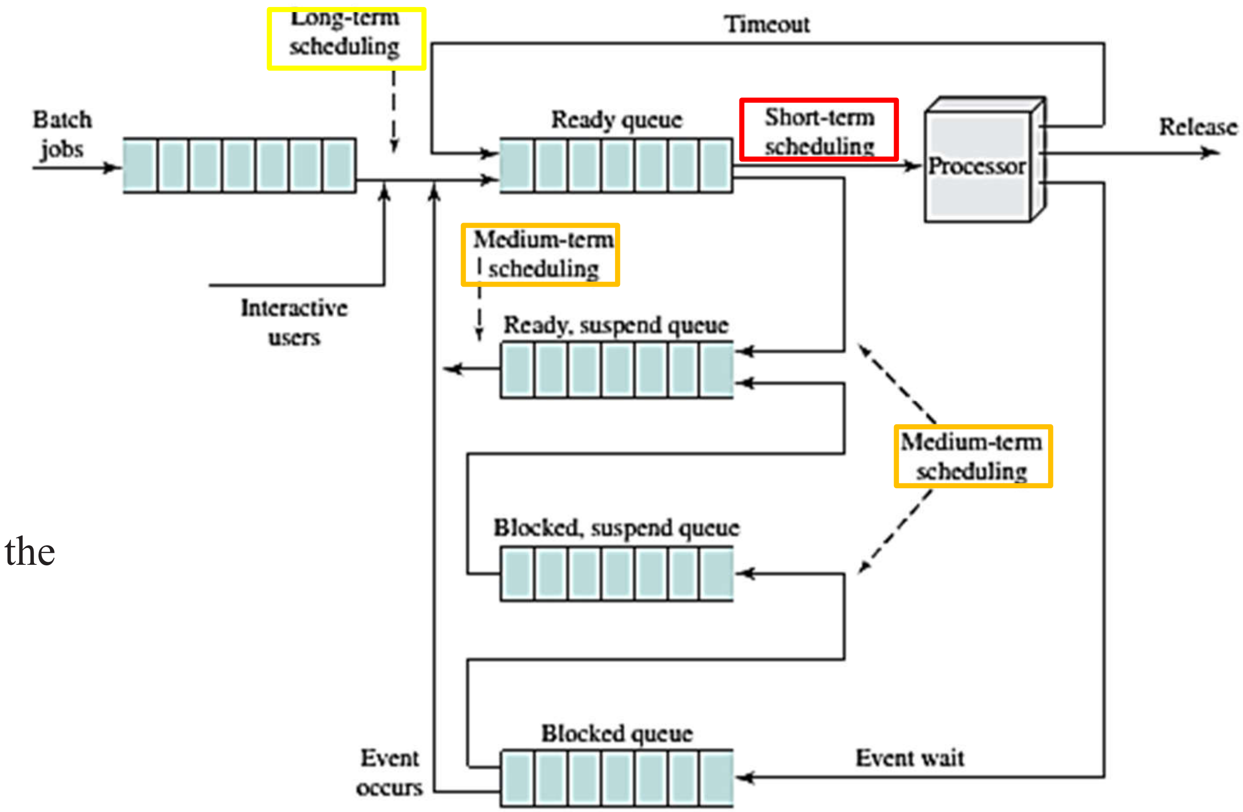
- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

CPU scheduler

- Whenever CPU is idle, it must select another process from ready queue (short-term scheduler).
- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
- Ready queue: FIFO, priority queue, tree, unordered linked list…
  - Consisted of PCBs of processes



Schedulers



- Long-term scheduling is performed when a new process is created.
- Medium-term scheduling is a part of the swapping function.
- Short-term scheduling is the actual decision of which ready process to execute next.

Preemption and preemptive scheduling

- Preemption
  - The act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time [wiki]
- Scheduler
  - Preemptive vs. Nonpreemptive (cooperative)
  - When CPU can switch?
    1. A process switches from running → waiting state (IO request, wait())
    2. A process switches from running → ready state (interrupt occurs)
    3. A process switches from waiting → ready state (completion of IO)
    4. A process terminates!
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Which one is good? preemptive or nonpreemptive

- Nonpremtive scheduler
  - Windows 3.1
  - No need of any special hardware mechanisms (timer, etc.)

- Preemptive scheduler
  - Windows 95, 98, ME, XP, 7, 8, 10
  - Mac OS X
  - Can result Race Condition! (why?)

Dispacher

- An OS module gives control of CPU to the process selected by short-term scheduler
  - Switching context
  - Switching to user mode
  - Jumping to proper location in the user program to resume it

- Should be fast.

- Dispatch latency
  - The time to stop one process and start another running

Which scheduler is the best?

- Criteria
  - CPU utilization
    - As busy as possible
    - A value from 0 to 100 (real system 40 to 90)
  - Throughput
    - Number of processes that are completed.
  - Turnaround time
    - Time from submission of a process to time of completion
    - Sum of periods spent waiting {to get memory, IO, CPU}, running in CPU, doing IO
  - Waiting time
    - Sum of periods spent waiting in the ready queue
  - Response time
    - Time from submission of a request until first response is produced.
    - Time it takes to start responding
- Which one is better?

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better *average* response time by making system *less* fair

Best scheduler?

- For interactive systems (desktop systems)
  - Minimizing variance in response time

- The main question:
  - Which one of processes in Ready Queue is to be allocated to CPU?

Scheduling Algorithms

1) First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - In early systems, FCFS meant one program scheduled until done (including I/O)
    - Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose 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
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process stuck behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P2 , P3 , P1
    Now, the Gantt chart for the schedule is:

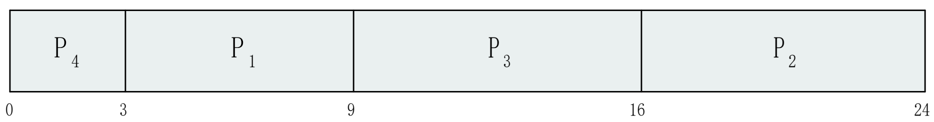
  - Waiting time for P1 = 6; P2 = 0; P3 = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average Completion time: (3 + 6 + 30)/3 = 13
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)

2) Shortest-Job-First scheduling

- Shortest-next-CPU-burst
- Decides based on the length of process's next CPU burst
- Is optimal; has min average waiting time!
- It cannot be implemented in short-term scheduler (why?)

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

- Prediction as exponential average

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

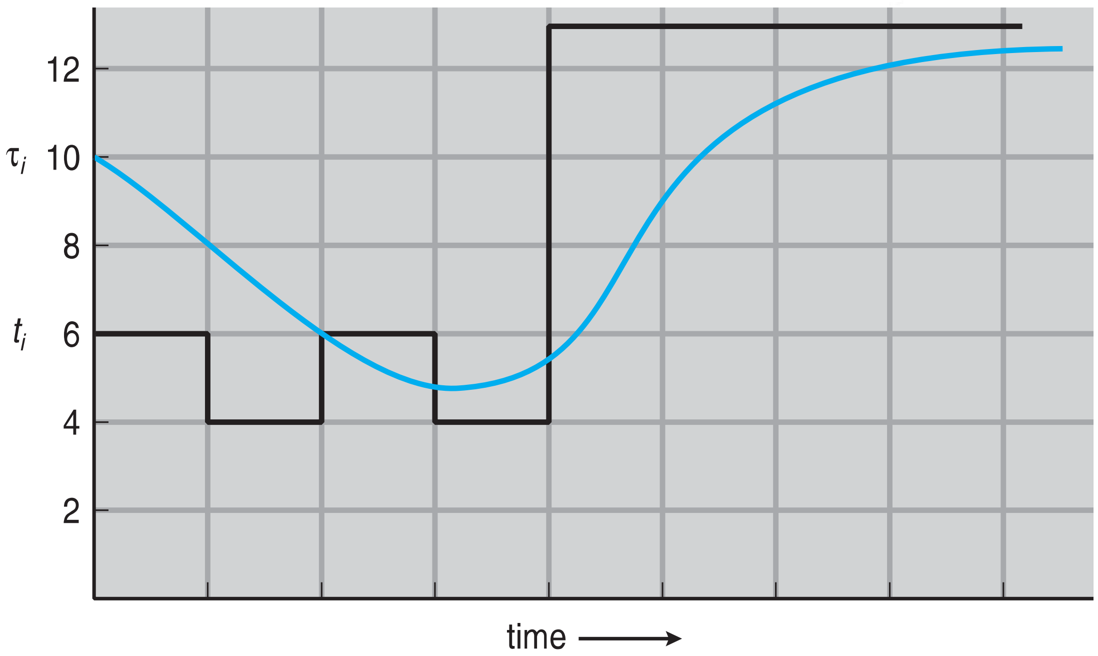
- Commonly, α set to ½

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

- Two implementations: Preemptive, Nonpreemptive
  – Preemptive SJF: shortest-remaining-time-first

Example

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



$\tau_0 = 10$
$\alpha = 0.5$

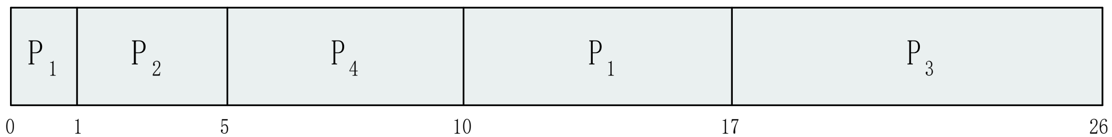
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Preemptive SJF

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

3) Priority scheduling

- General case of SJF (how?)

- A priority (number) is associated with each process
  - Internal: time limits, memory requirements, number of open files, ratio of IO burst to average CPU burst
  - External: outside of OS (importance of process, type of funds being paid, etc)

- Can be:
  - preemptive
  - nonpreemptive

3) Priority scheduling (cont'd)

- Main problem? Indefinite blocking or starvation
- Solution? To include aging

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

4) Round-Robin scheduler

- Round Robin Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - n processes in ready queue and time quantum is $q \Rightarrow$
    - Each process gets $1/n$ of the CPU time in chunks of at most q time units
    - No process waits more than $(n-1)q$ time units

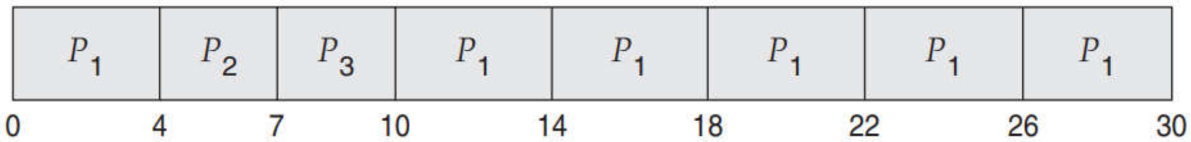
4) Round-Robin scheduler (cont'd)

Consider the following set of processes that arrive at time 0

Process	Burst Time
P_1	24
P_2	3
P_3	3

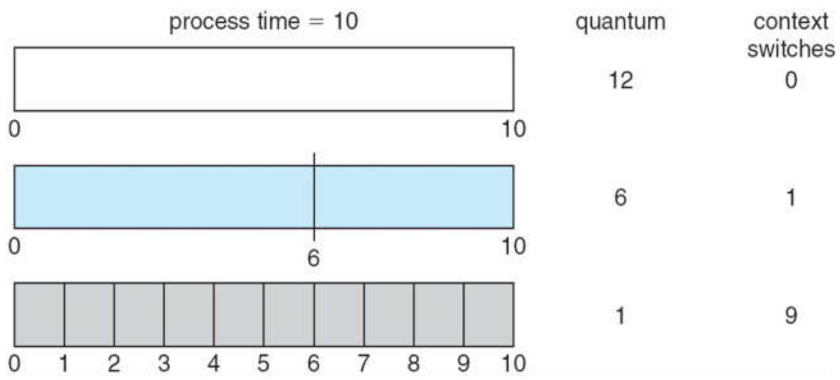
time quantum is 4 millisecond

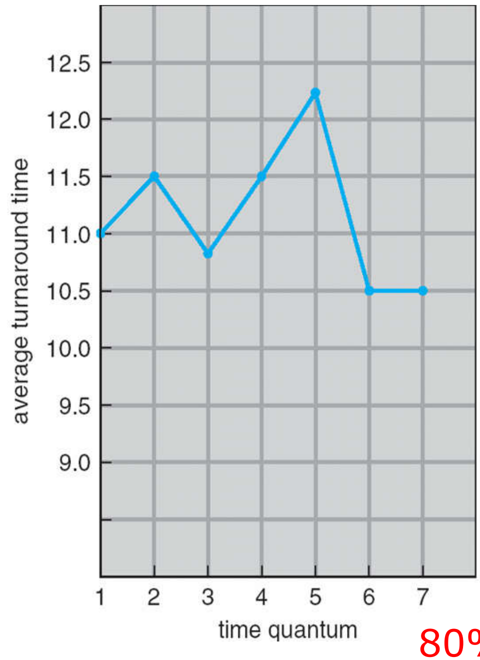
The Gantt chart is:



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

Time quantum & context switch time





process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - The Gantt chart is:

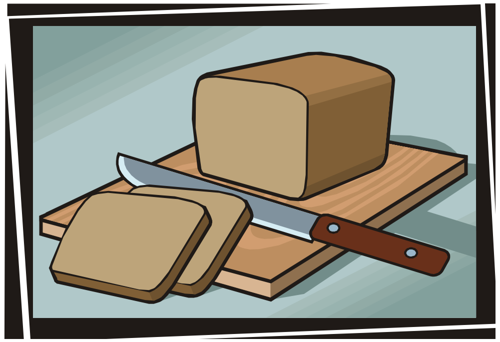
P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

  - Waiting time for
    - P_1=(68-20)+(112-88)=72
    - P_2=(20-0)=20
    - P_3=(28-0)+(88-48)+(125-108)=85
    - P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FIFO
  - What if time slice too small?
    - Throughput suffers!



- Actual choices of time slice:
  - Initially, UNIX time slice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - Need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:

  10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time

- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  – Both RR and FCFS finish at the same time
  – Average response time is much worse under RR!
    - Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  – Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

RR Scheduling (Cont.)

- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ Interleaved (really small $\Rightarrow$ hyperthreading?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

5) Multilevel Queue scheduler

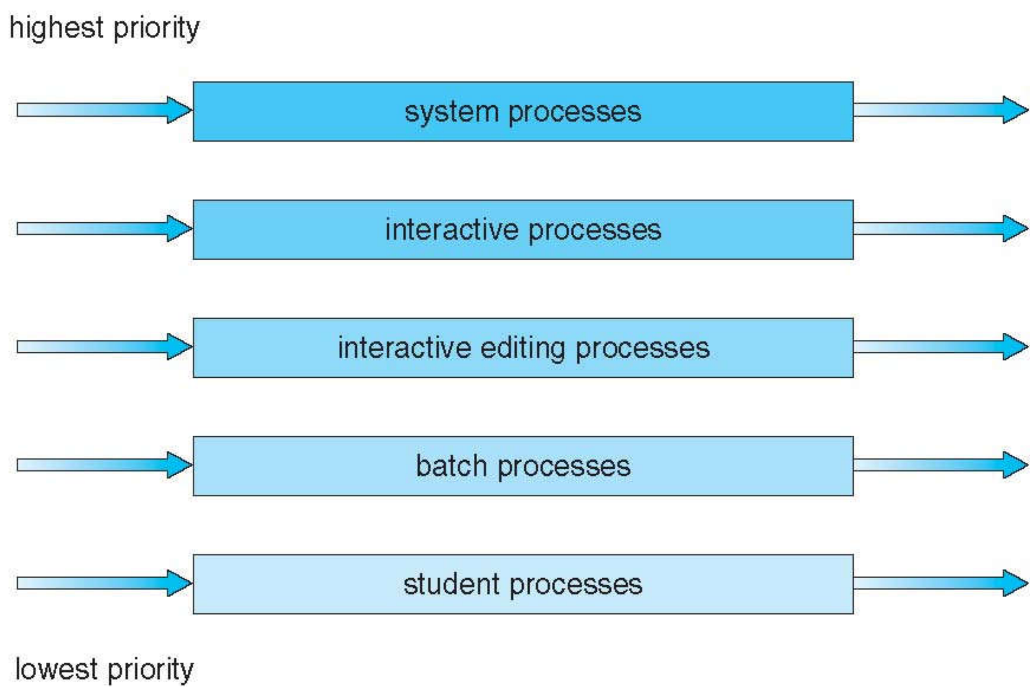
- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)

- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS

- For example, it has 5 queues:
  - System processes
  - Interactive processes
  - Interactive editing processes
  - Batch processes
  - Student processes

Example of multilevel queue scheduler

- Each queue has absolute priority over lower-priority queues



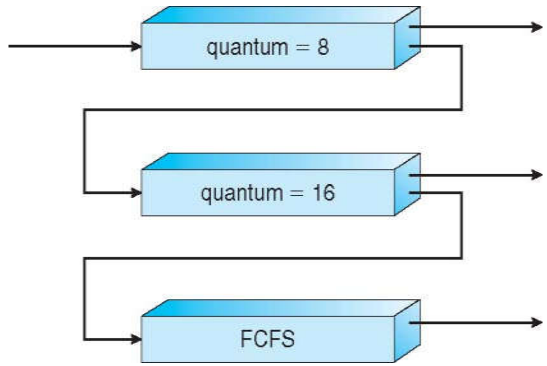
6) Multilevel Feedback Queue scheduler

- 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

Thread scheduling

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow

  PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

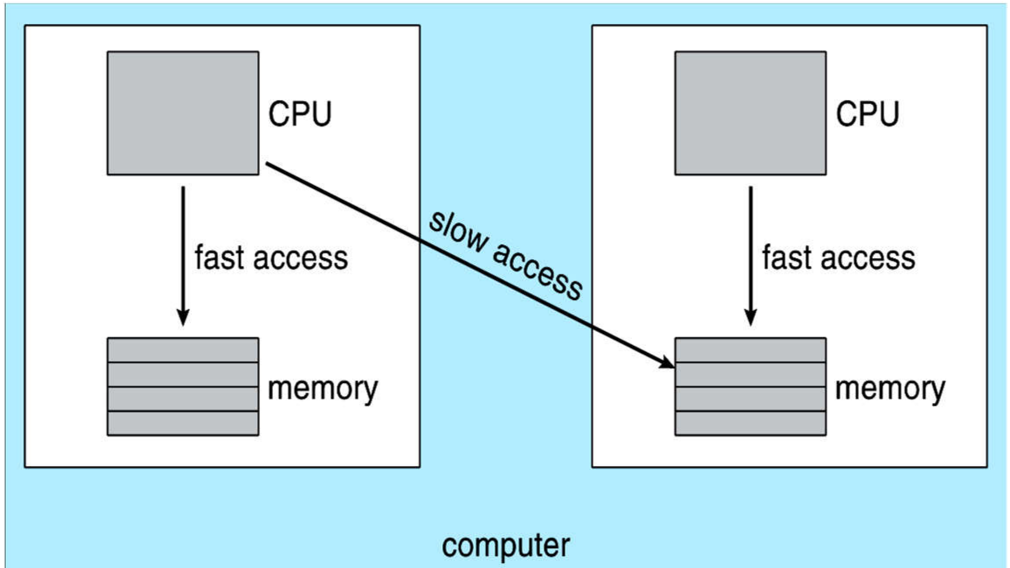
Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Multiple-processor scheduling

- Multiple processors
  - Load sharing
- Multiple-processor scheduling
  - AMP: only one processor accesses the system data structures, alleviating the need for data sharing
    - Master server (master processor)
  - SMP
    - Common ready queue
    - Private ready queue
      - Processor affinity: a process has an affinity for the processor on which it is currently running
        - » Soft affinity
        - » Hard affinity
    - Load balancing
      - To get best CPU utilization

NUMA and CPU scheduling



Note that memory-placement algorithms can also consider affinity

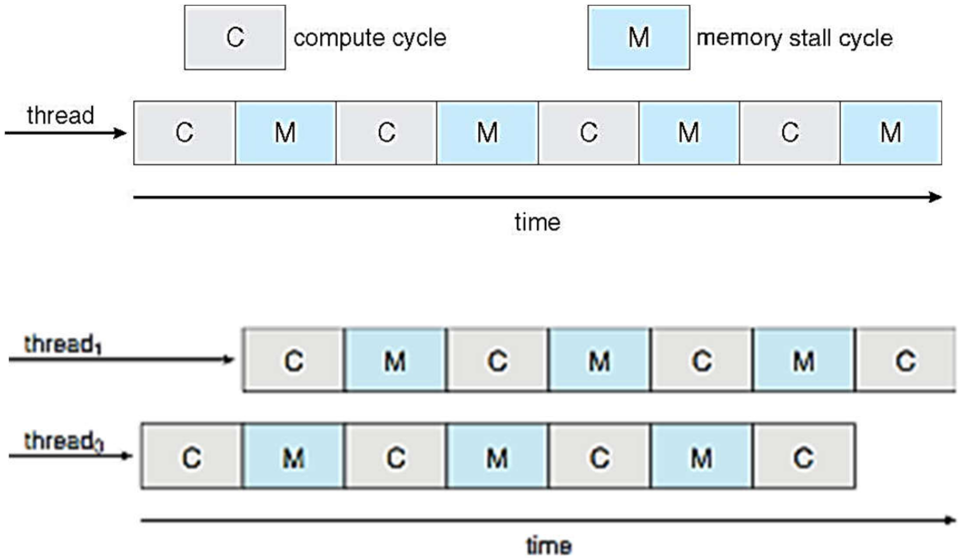
Load balancing in SMP

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
  - Push migration
    - Task periodically check the loads on each processor
  - Pull migration
    - Idle processor pulls a waiting task
- Problem to affinity
  - Using threshold for imbalancing

Multicore processor

- Faster, less power consumption (why?)
- Memory stall is costly
  - Hardware threads for each core
    - UlteraSPARC T3 CPU (16*8)
    - Intel Itanium (dual core)
  - Coarse-grained vs. fine-grained scheduling

Multithreaded multicore system



Real-Time Scheduling

Real-time CPU scheduling

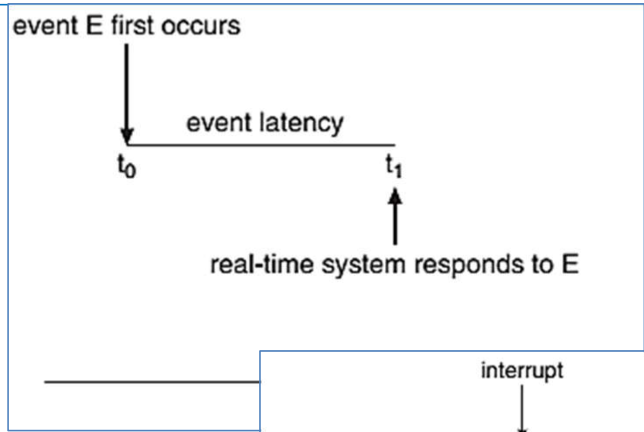
- Events
  - SW: timer
  - HW: external interrupts

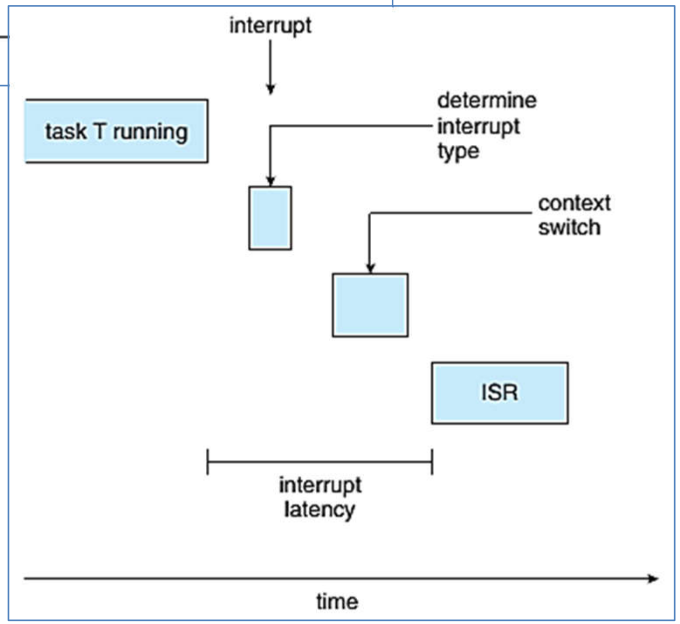
- **Soft real-time systems**
  - No guarantee as to when critical real-time process will be scheduled
  - Guarantee only critical processes have preference over noncritical ones.

- **Hard real-time systems**
  - Task must be serviced by its deadline
  - Service after deadline is the same as no service at all.

Event latency

- Event latency
  - Time from when an event occurs to when it is serviced
  - For ABS: 3-5 ms
  - Interrupt latency
    - time from arrival of interrupt to start of routine that services interrupt
  - Dispatch latency
    - time for schedule to take current process off CPU and switch to another



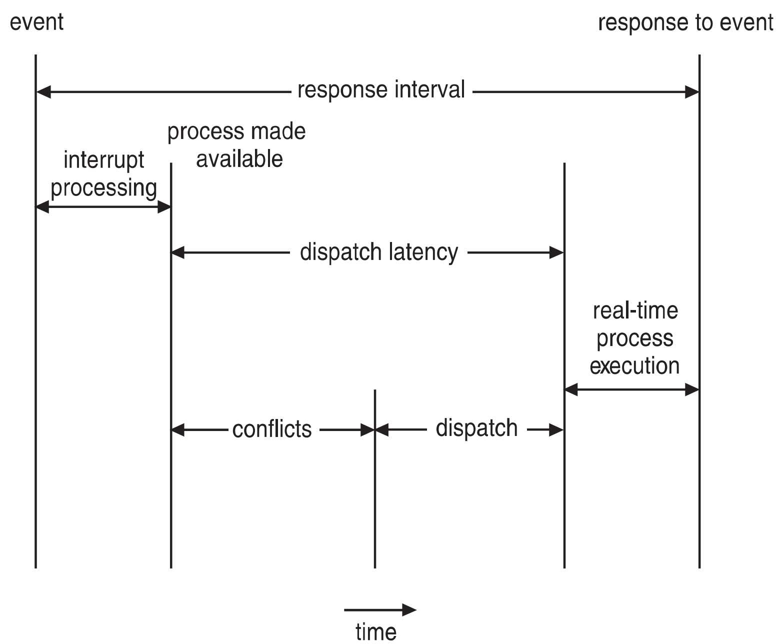


Dispatch latency

- Conflict phase of dispatch latency:

  1. Preemption of any process running in kernel mode

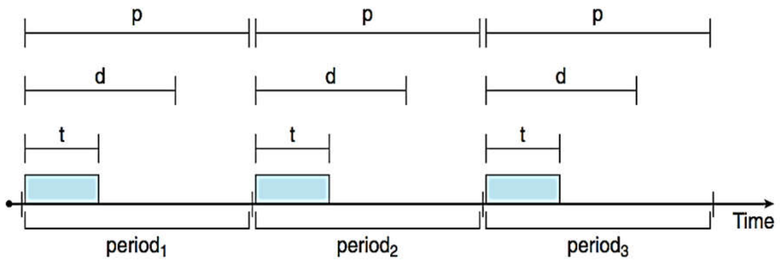
  2. Release by low-priority process of resources needed by high-priority processes



Priority-based scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
  - For hard real-time must also provide ability to meet deadlines

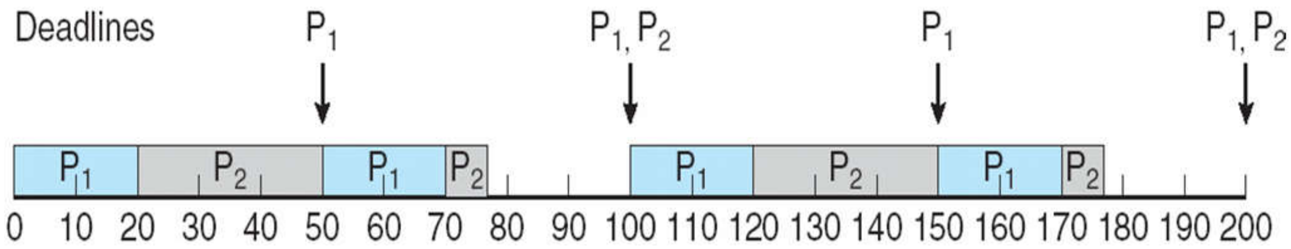
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



1) Rate-monotonic scheduling

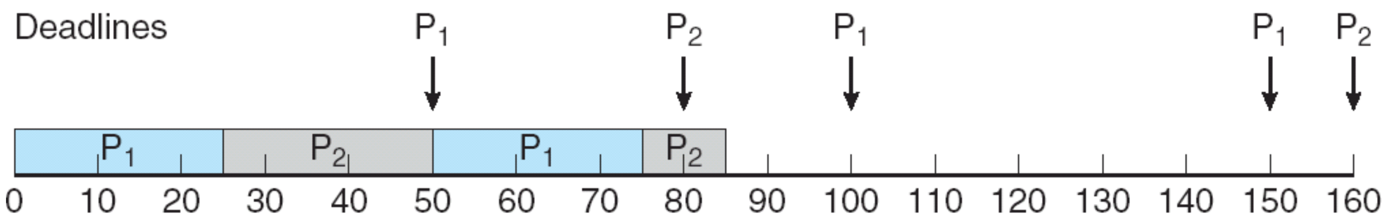
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2

Process	Period	CPU burst time	Utilization
P_1	50	20	20/50=0.4
P_2	100	35	35/100=0.35
			total=0.75



Missed deadlines with Rate Monotonic scheduling

Process	Period	CPU burst time	Utilization
P_1	50	25	25/50=0.5
P_2	80	35	35/80=0.44
			total=0.94



Limitation of CPU utilization in Rate-Monotonic

- CPU utilization in RM is bounded!

Tasks periodic with period P and computation C in each period: (P_i, C_i) for each task i

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

- N=1 → U(1) = 100%
- N=2 → U(2) = 83%
- N=inf → U(inf) = 69%

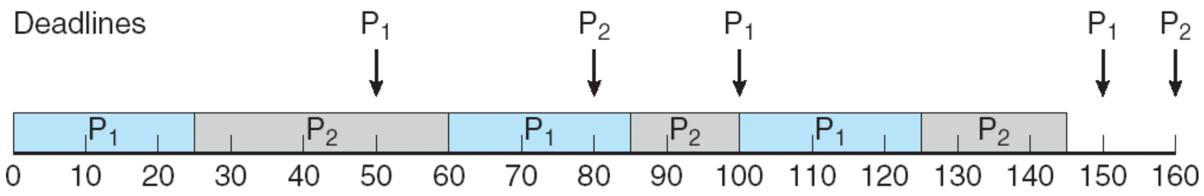
Liu, C. L.; Layland, J. (1973), "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of the ACM*, **20** (1): 46–61

2) Earliest-deadline-first scheduling (EDF)

- Priorities are assigned according to deadlines:

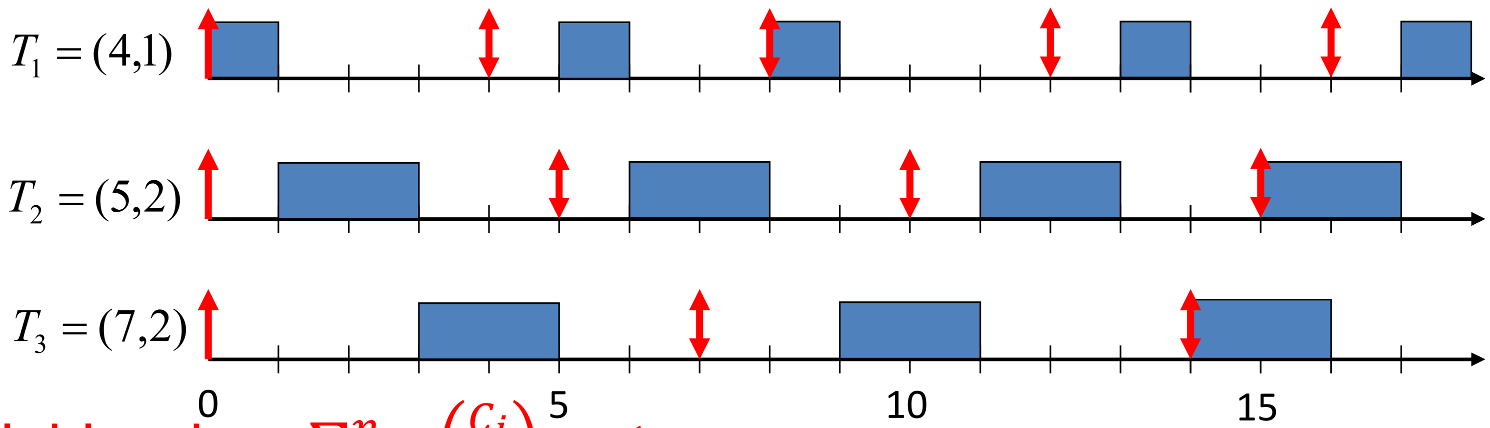
  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority

Process	Period	CPU burst time	Utilization
P_1	50	25	25/50=0.5
P_2	80	35	35/80=0.44
			total=0.94



Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P_i, C_i) for each task i
- Preemptive priority-based dynamic scheduling:
  - Each task is assigned a (current) priority based on how close the absolute deadline is (i.e. $D_i^{t+1} = D_i^t + P_i$ for each task!)
  - The scheduler always schedules the active task with the closest absolute deadline



- Schedulable when $\sum_{i=1}^{n} \left(\frac{C_i}{P_i}\right) \leq 1$

3) 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 $\boldsymbol{N} / T$ of the total processor time

Algorithm evaluation

- How to select CPU-scheduling algorithm for an OS?
  - Determine criteria, then evaluate algorithms
- Deterministic modeling
  - FCS is 28ms
  - Non-preemptive SFJ is 13ms
  - RR is 23ms

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

- Queueing models
  - 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:

    $n = \lambda \times W$
- Simulations
  - Programmed model of computer system

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

Example: Linux Scheduling

- Standard Linux kernels implement two scheduling classes:
- (1) a default scheduling class using the CFS scheduling algorithm
- (2) a real-time scheduling algorithm.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
     fprintf(stderr, "Unable to get policy.\n");
   else {
     if (policy == SCHED_OTHER)
      printf("SCHED_OTHER\n");
     else if (policy == SCHED_RR)
      printf("SCHED_RR\n");
     else if (policy == SCHED_FIFO)
      printf("SCHED_FIFO\n");
   }

   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
     fprintf(stderr, "Unable to set policy.\n");

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Summary (1 of 2)

- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

Summary (2 of 2)

- Realtime Schedulers such as EDF
  - Guaranteed behavior by meeting deadlines
  - Realtime tasks defined by tuple of compute time and period
  - Schedulability test: is it possible to meet deadlines with proposed set of processes?

Questions?