CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Spring 2022 Lecture 8

Scheduling

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

FAQ

- What exactly is a thread? A block of code?
  - A function when called within a new thread, runs concurrently with other threads.
- Process vs thread
  - A process is isolated from other processes. Processes can run concurrently. Can have multiple threads.
  - A thread is not isolated from other threads belonging to the same process. Runs concurrently with other threads.
- What is a pthread? POSIX compliant implementation of threads.
- Java threads? Most JVMs implement threads with native, OS level threads,
- Examples of threads: Self exercise set 4

FAQ

- Why use threads:
  - Parallelism if multiple cores/hyper-threading available.
  - Concurrency: quicker responses to some of the things like refreshing output, checking spelling as one types etc.
- Implicit threading: thread creating automated: compiler assisted higher level programming
- Unix signals vs interrupts: Signals are a limited form of inter-process communication. Interrupts are often initiated by hardware. In both cases, some specific routines respond.
- Hyper-threading: Requires additional hardware. Widely used
- Signals example (assume pid = 162): kill -9 162 or kill –s sigkill 162
- Pthread example: pthread_kill(ThreadID, SIGKILL);

Implicit Threading: OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
   for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Splits loop task in parallel threads

Compile using
gcc -fopenmp openmp.c

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

Self exercise 3, 4 available now.

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- A thread's cancelation type (mode) and state can be set.
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Is complexity always good?

- Is something that is
  - More advanced
  - More complex

  Generally better?

Colorado State University

Hyper-threading

"Hyper-threading": "simultaneous multithreading":

- Hardware support for multiple threads in the same core (CPU)

- Performance:
  - performance improvements are very application-dependent
  - Higher energy consumption ARM 2006
  - Not better than out-of-order execution Intel 2013
  - Intel has dropped it in some chips Core i7-9700K 2018 8 cores, 8 threads
  - May be enabled/disabled using firmware

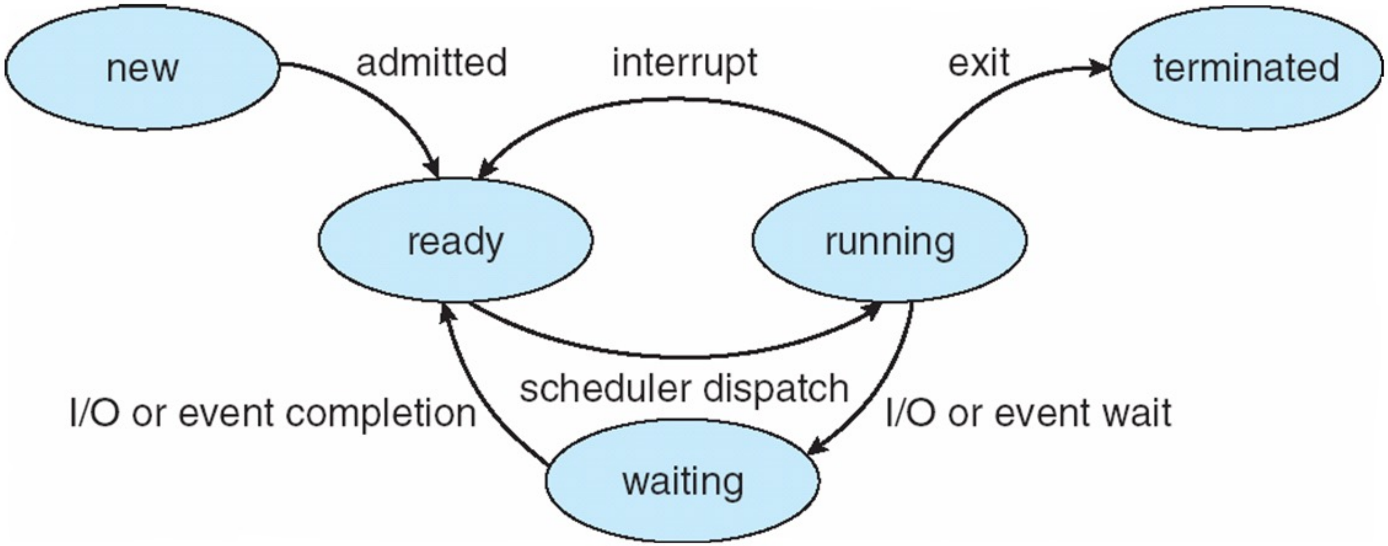
Forms of Parallelism

- Pipelining: instruction flows though multiple levels
- Multiple issue: Instruction level Parallelism (ILP)
  - Multiple instructions fetched at the same time
  - Static: compiler scheduling of instructions
  - Dynamic: hardware assisted scheduling of operations
    - “Superscalar” processors
    - CPU decides whether to issue 0, 1, 2, … instructions each cycle
- Thread or task level parallelism (TLP)
  - Multiple processes or threads running at the same time

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

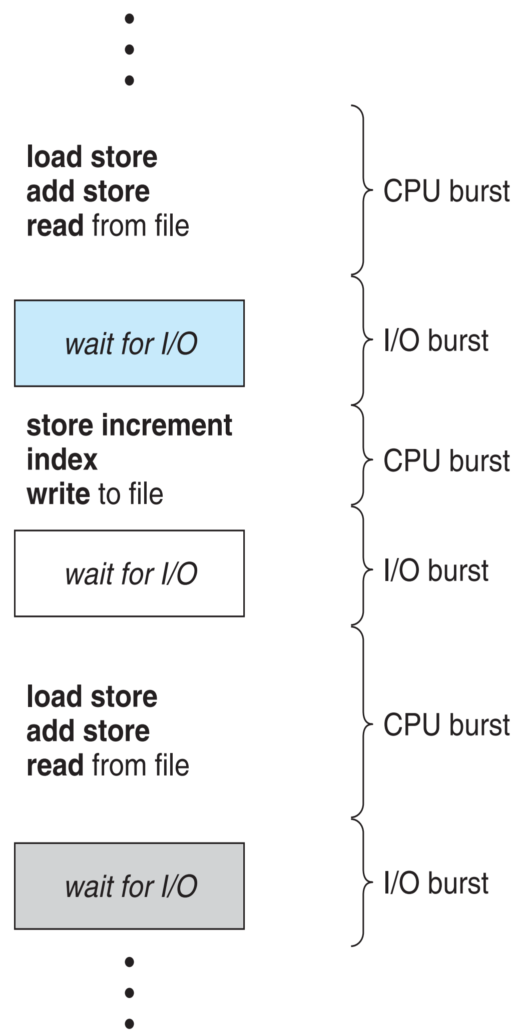
Diagram of Process State



Ready to Running: scheduled by scheduler
Running to Ready: scheduler picks another process, back in ready queue
Running to Waiting (Blocked) : process blocks for input/output
Waiting to Ready: Input available

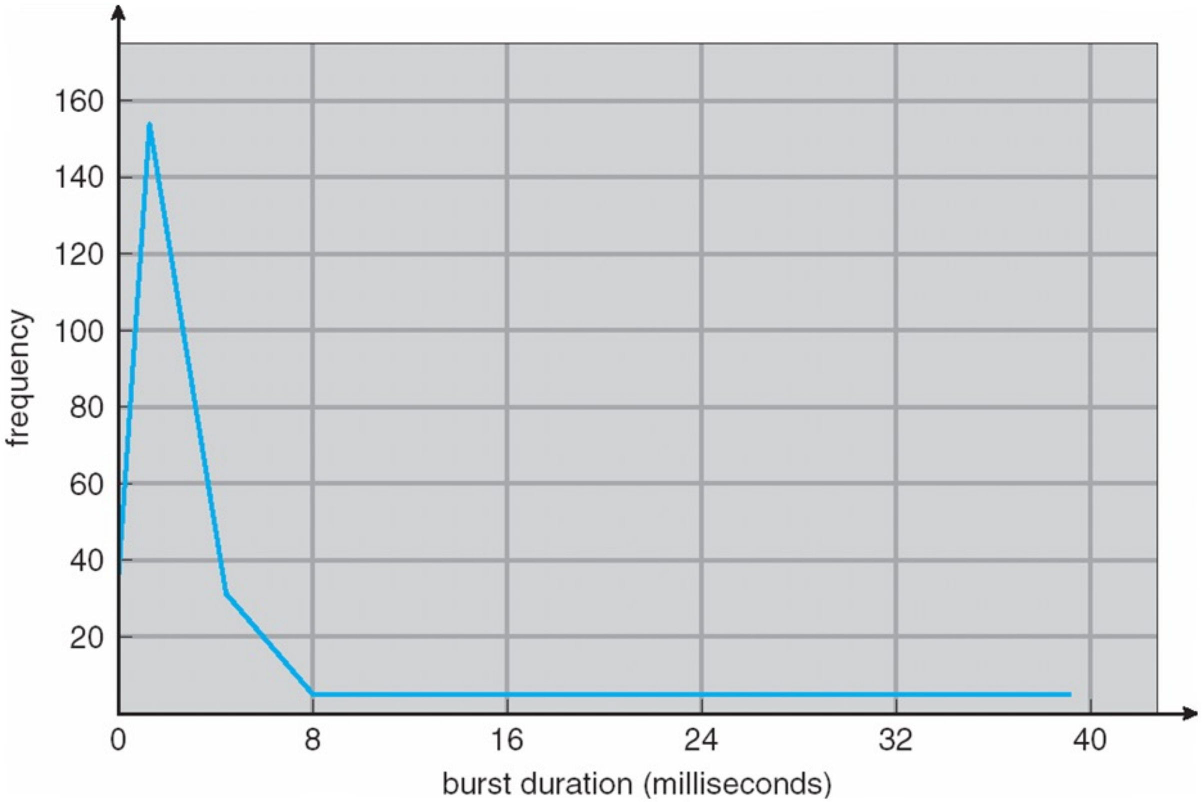
Basic Concepts

- 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



Colorado State University

Histogram of CPU-burst Times



Typical distribution of CPU bursts. Most CPU bursts are just a few ms.

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. These need to be considered**
  - access to shared data by multiple processes
  - preemption while in kernel mode
  - interrupts occurring during crucial OS activities

Not Controlled by the process

Dispatcher

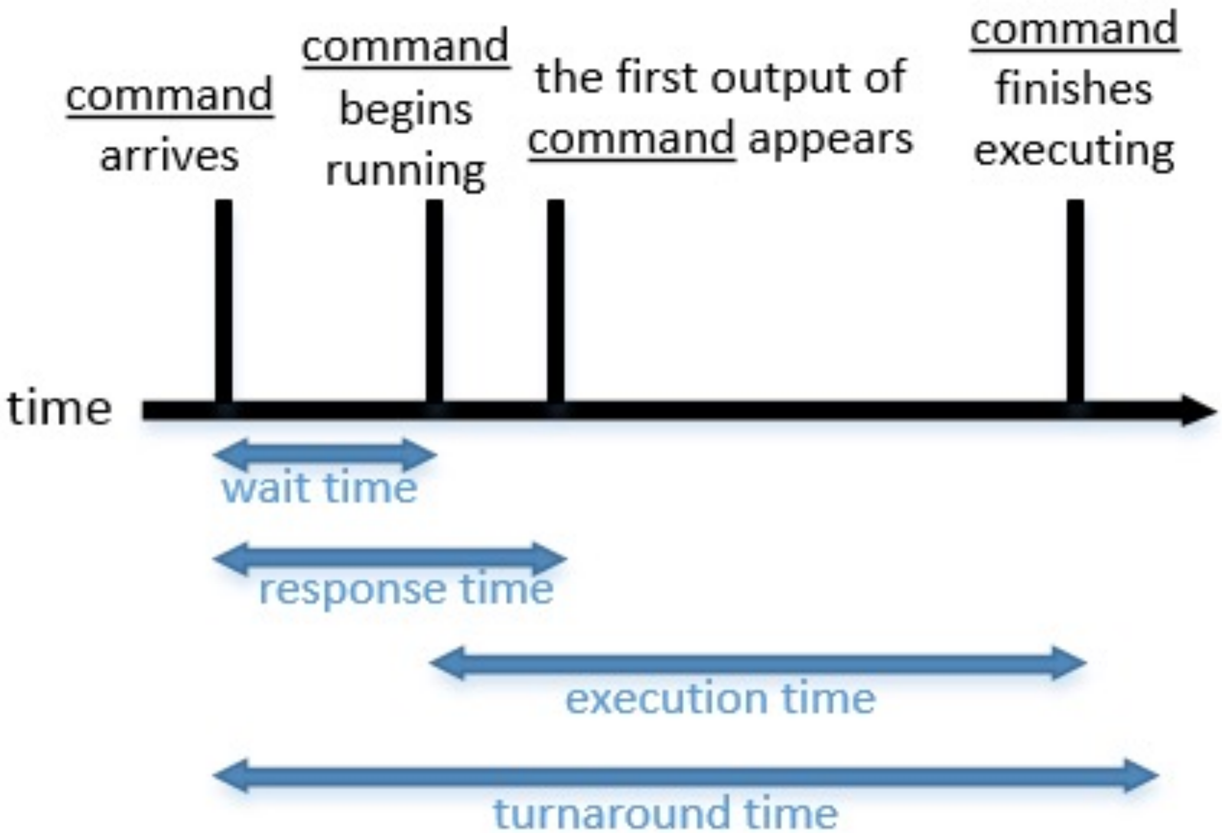
- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to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

The Dispatcher (dentist's office)

Scheduling Criteria

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

Terms *for a single process*



UCLA

Scheduling Algorithms

We will now examine several major scheduling approaches

- **Decide** which process in the ready queue is allocated the CPU
- Could be preemptive or nonpreemptive
  - preemptive: remove in middle of execution ("forced")
- Optimize ***measure*** of interest
  - We will use **Gantt charts** to illustrate *schedules*
  - Bar chart with start and finish times for processes

[Involuntary deboarding!]()

Colorado State University

Non-preemptive vs Preemptive sheduling

- **Non-preemptive:** Process keeps CPU until it relinquishes it when
  - **It terminates**
  - **It switches to the waiting state**
  - Used by initial versions of OSs like Windows 3.x
- **Preemptive** scheduling
  - Pick a process and let it run for a maximum of some fixed time
  - If it is still running at the end of time interval?
    - Suspend it and pick another process to run
- A **clock interrupt** at the end of the time interval to give control back of CPU back to scheduler

Scheduling Algorithms

- First- Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
  - Shortest-remaining-time-first
- Priority Scheduling
- Round Robin (RR) with time quantum
- Multilevel Queue
  - Multilevel Feedback Queue
- “Completely fair”

Comparing Performance

- Average waiting time etc.

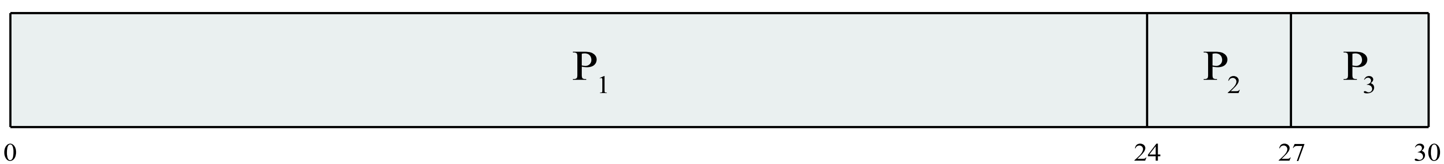
First- Come, First-Served (FCFS) Scheduling

- Process requesting CPU first, gets it first
- Managed with a FIFO queue
  - When process **enters** ready queue
    - PCB is tacked to the **tail** of the queue
  - When CPU is **free**
    - It is allocated to process at the **head** of the queue
- Simple to write and understand

Henry Gantt, 1910s

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 *but almost the same time*.
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = ; P_2 = ; P_3 =
- Average waiting time: (+ +)/ =
- Throughput: / = per unit time

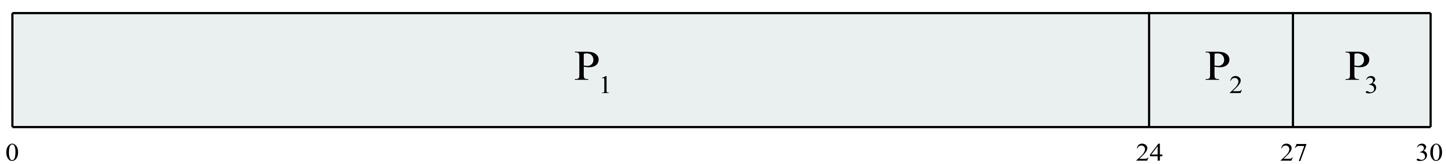
Pause for students to do the computation

First- Come, First-Served (FCFS) Scheduling

Henry Gantt, 1910s

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 *but almost the same time*.
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
- Throughput: 3/ 30 = 0.1 per unit time

Colorado State University

FCFS Scheduling (Cont.)

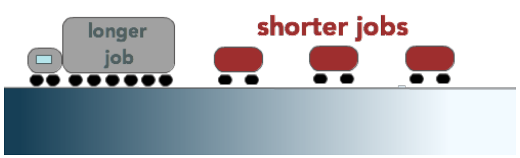
Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- 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
- But note -Throughput: 3/30 = 0.1 per unit same
- **Convoy effect** - short processes behind a long process
  - Consider one CPU-bound and many I/O-bound processes

The Convoy Effect, visualized



Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- Reduction in waiting time for short process *GREATER THAN* Increase in waiting time for long process
- 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
  - Estimate or could ask the user

Colorado State University

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- All arrive at time 0.
- SJF scheduling chart: Draw it here.

- Average waiting time for P_1,P_2,P_3,P_4 = (+ + +) / =

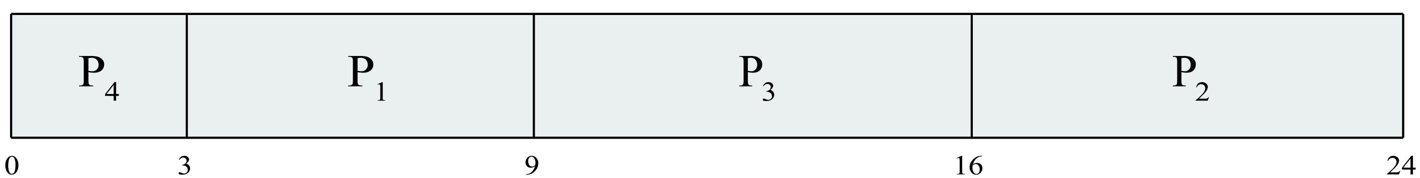
Pause for students to do the computation

Colorado State University

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- All arrive at time 0.
- SJF scheduling chart



- Average waiting time for P_1,P_2,P_3,P_4 = (3 + 16 + 9 + 0) / 4 = 7

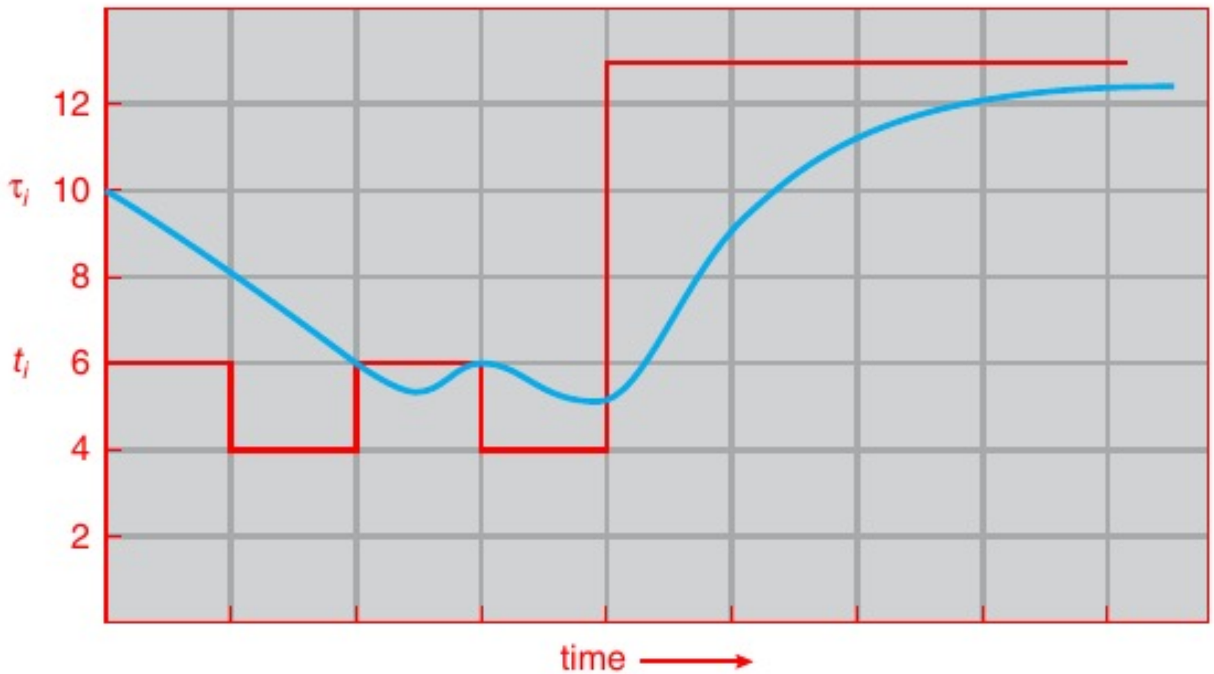
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the recent bursts
  - 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

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	…

Blue line: guess
Red line: actual

$\alpha = 0.5$

Ex:
$0.5 \times 6 + 0.5 \times 10 = 8$

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\ t_n$
  - Only the actual last CPU burst counts
- $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$
- If we expand the formula, substituting for τ_n , 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

Widely used for predicting stock-market etc

Colorado State University

Shortest-remaining-time-first (**preemptive** SJF)

- Preemptive version called **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 (will preempt because 4<7)
P_3	2	9 (will not preempt)
P_4	3	5

- *Preemptive* SJF Gantt Chart

P_1	P_2	P_4	P_1	P_3
0–1	1–5	5–10	10–17	17–26

- Average waiting time for P1,P2,P3,P4

  = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

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

MIT had a low priority job waiting from 1967 to 1973 on IBM 7094! ☺

Colorado State University

Ex Priority Scheduling non-preemptive

Process	Burst Time	Priority
P_1	10	3
P_2	1	1 (highest)
P_3	2	4
P_4	1	5
P_5	5	2

- P1,P2, P3, P4,P5 all arrive at time 0.
- Priority scheduling Gantt Chart

- Average waiting time for P1, .. P5: (6+0+16+18+1)/5 = 8.2 msec

Colorado State University

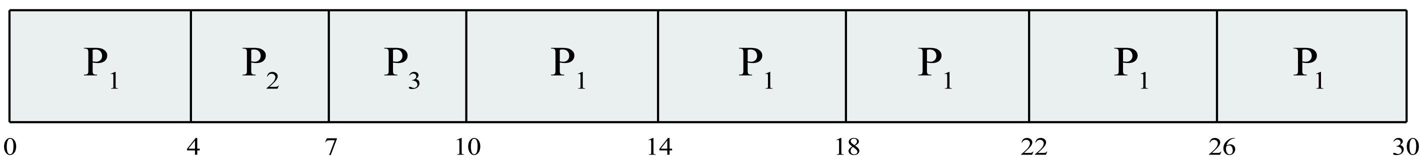
Round Robin (RR) with time quantum

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this, the process is preempted, 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 $(n\text{-}1)q$ 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 (overhead *typically in 0.5% range*)

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

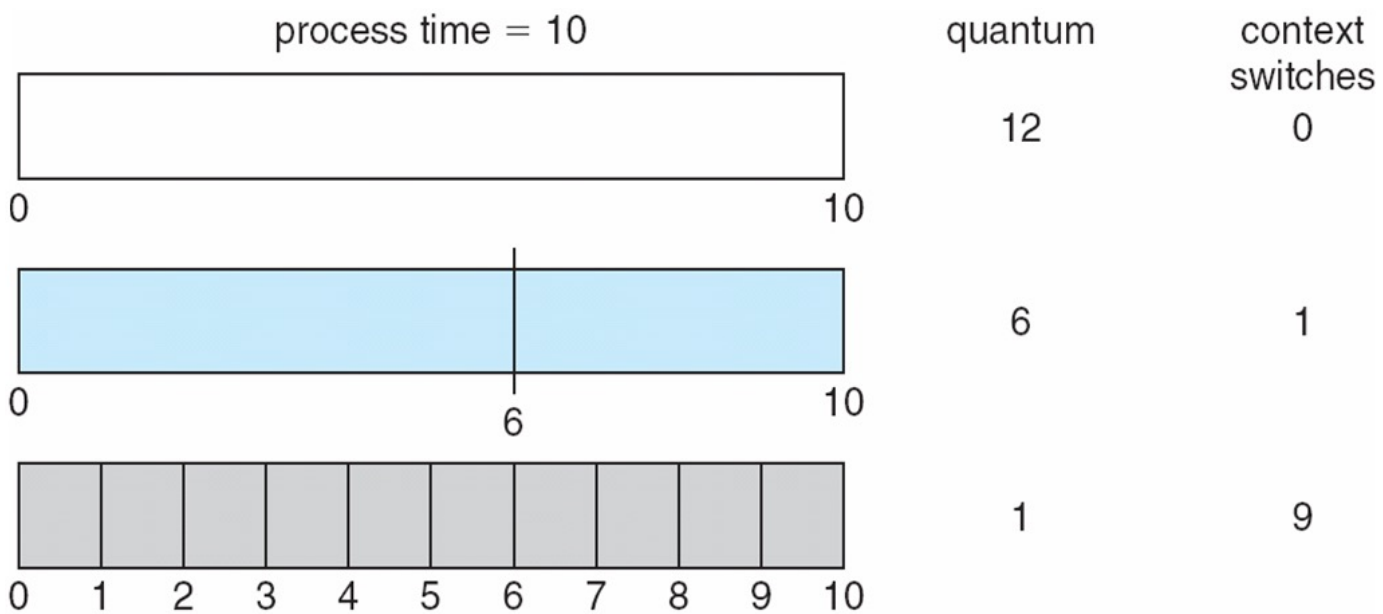
- Arrive a time 0 in order P1, P2, P3: The Gantt chart is:



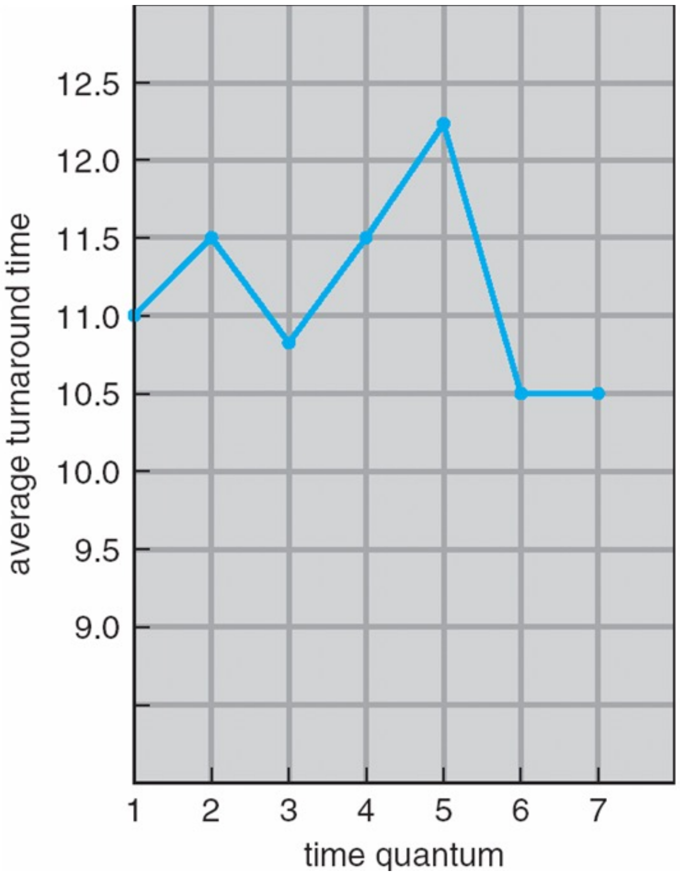
- Waiting times: P1:10-4 =6, P2:4, P3:7, average 17/3 = 5.66 units
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 μsec

Response time: Arrival to beginning of execution
Turnaround time: Arrival to finish of execution

Time Quantum and Context Switch Time



Much smaller quantum compared to burst: many switches



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Rule of thumb: 80% of CPU bursts should be shorter than q

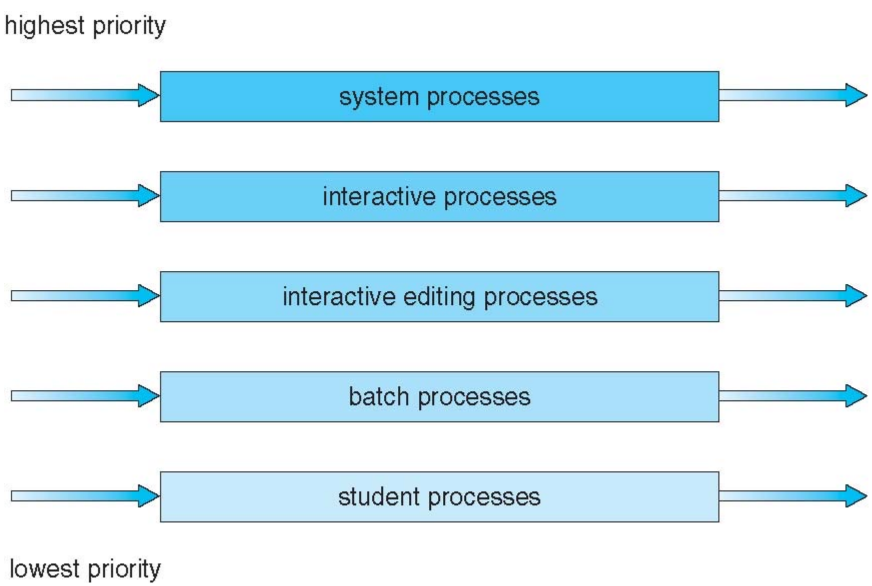
Illustration
Consider q=7:
Turnaround times for P1,P2,P3,P4:
6,9,10,17 av = 10.5
Similarly for q =1, ..6 (verify yourself)

Students: Repeat for q = 1, ..6 at home to verify the plot.

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm, e.g.:
  - 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. Or
  - 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



Real-time processes may have the highest priority.

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
  - Details at ARPACI-DUSSEAU

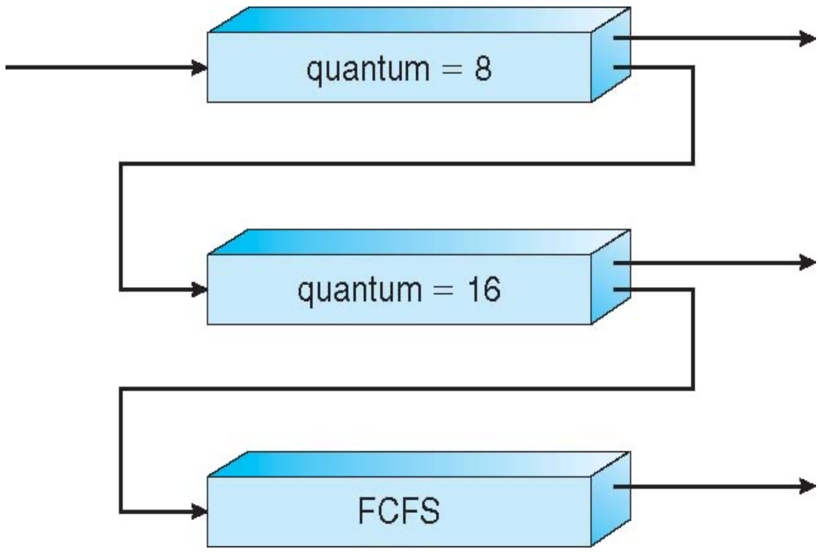
Inventor FJ Corbató won the Touring award!

Colorado State University

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 (no time quantum limit)

- 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



Upgrading may be based on aging. Periodically processes may be moved to the top level.

Variations of the scheme were used in earlier versions of Linux.

Colorado State University

Completely fair scheduler Linux 2.6.23

Goal: fairness in dividing processor time to tasks ([Con Kolivas, Anaesthetist](#))

- Variable time-slice based on number and priority of the tasks in the queue.
  - Maximum execution time based on waiting processes (Q/n).
  - Fewer processes waiting, they get more time each
- Queue ordered in terms of "virtual run time"
    - execution time on CPU added to value
  - smallest value picked for using CPU
  - small values: tasks have received less time on CPU
  - I/O bound tasks (shorter CPU bursts) will have smaller values
- *Balanced (red-black) tree* to implement a ready queue;
  - Efficient. O(log n) insert or delete time
- Priorities (*niceness*) cause different decays of values: higher priority processes get to run for longer time
  - virtual run time is the weighted run-time

Scheduling schemes have continued to evolve with continuing research. A comparison.

Colorado State University