
Chap 7, 8: Scheduling

Introduction

- **Multiprogramming**

- Multiple processes in the system with one or more processors
- Increases processor utilization by organizing processes so that the processor always has one to execute
- Resource management
 - Resources for **time** sharing
 - Multiple processes use a resource in a time-shared manner
 - **Processor**
 - Process scheduling: Allocates processor time slots to processes
 - Resources for **space** sharing
 - Partition a resource and let each process use the partitions
 - **Memory**

Goals of Scheduling

- **Goals of process scheduling**

- Improving system performance

- **Typical performance indices**

- **Turnaround time**: amount of time to execute a particular process

- $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$

- **Response time**: amount of time it takes to start responding

- $T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$

- Throughput: number of processes completed per time unit

- Fairness

- Utilization: Percentage of time that the resource is busy during a given interval

- Predictability

- Etc

- Each system selects a scheduling policy with the consideration on the performance indices for its application domain

Scheduling Policies

- **Preemptive/non-preemptive scheduling**
 - Preemptive scheduling
 - CPU may be preempted to another process independent of the intention of the running process
 - Flexibility, adaptability, performance improvements
 - For time-sharing systems and real-time systems
 - Incurs a cost associated with access to shared data
 - [Process synchronization]
 - Affects the design of operating system kernel
 - Kernel data integrity and consistency
 - Preemptible kernel
 - High context switching overhead

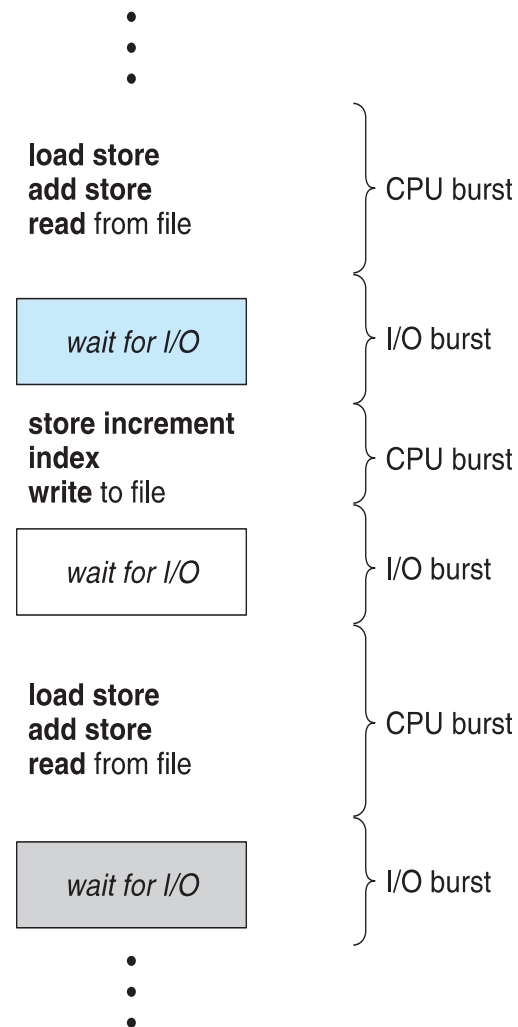
Scheduling Policies

- **Preemptive/non-preemptive scheduling**
 - Non-preemptive scheduling
 - Process uses the CPU until it voluntarily releases it (eg. for system call)
 - No preemption
 - Pros
 - Low context switch overhead
 - Cons
 - Frequent priority inversions
 - May result in longer mean response time

Terminologies

- **CPU burst vs. I/O burst**

- Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst
 - Each cycle of CPU execution
- I/O burst
 - Each cycle of I/O wait
- Burst time is an important factor(criteria) for scheduling algorithms



Scheduling Schemes

- **FIFO, FCFS (First-Come First Service)**
- **SJF (Shortest Job First)**
- **STCF (Shortest Time-to-Completion First)**
- **RR**
- **Priority**
- **MLFQ**

Scheduling Schemes

- **FCFS(First-Come-First-Service) scheduling**
 - Non-preemptive scheduling
 - Scheduling criteria
 - Arrival time (at the ready queue)
 - Faster arrival time process first
 - High resource utilization
 - Adequate for batch systems, not for interactive systems
 - Disadvantages
 - **Convoy effect**
 - short process behind long process
 - Consider one CPU-bound and many I/O-bound processes
 - Longer mean response time

First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
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:



- Turnaround time for $P_1 = 24$; $P_2 = 27$; $P_3 = 30$
- Average Turnaround time: $(24 + 27 + 30)/3 = 27$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2, P_3, P_1

- The Gantt chart for the schedule is:



- Turnaround time for $P_1 = 30; P_2 = 3; P_3 = 6$
- Average Turnaround time: $(30 + 3 + 6)/3 = 13$
- Much better than previous case

Scheduling Schemes

- **SJF (Shortest Job First) scheduling**
 - Non-preemptive scheduling
 - Scheduling criteria
 - Burst time
 - Shortest next CPU burst time first scheduling
 - Pros
 - Gives minimum average waiting time for a given set of processes
 - Minimizes the number of processes in the system
 - Reduces the size of the ready queue
 - Reduces the overall space requirements
 - Fast responses to many processes

Scheduling Schemes

- **SJF (Shortest Job First) scheduling**

- Cons

- Starvation, indefinite postponement(blocking)
 - Long burst-time processes
 - Can be solved by aging
- No way to know the length of the next CPU burst for each process
 - It is necessary to have a scheme for burst time estimation
 - Estimation by exponential average

1. t_n = actual length of n^{th} CPU burst

2. τ_{n+1} = predicted value for the next CPU burst

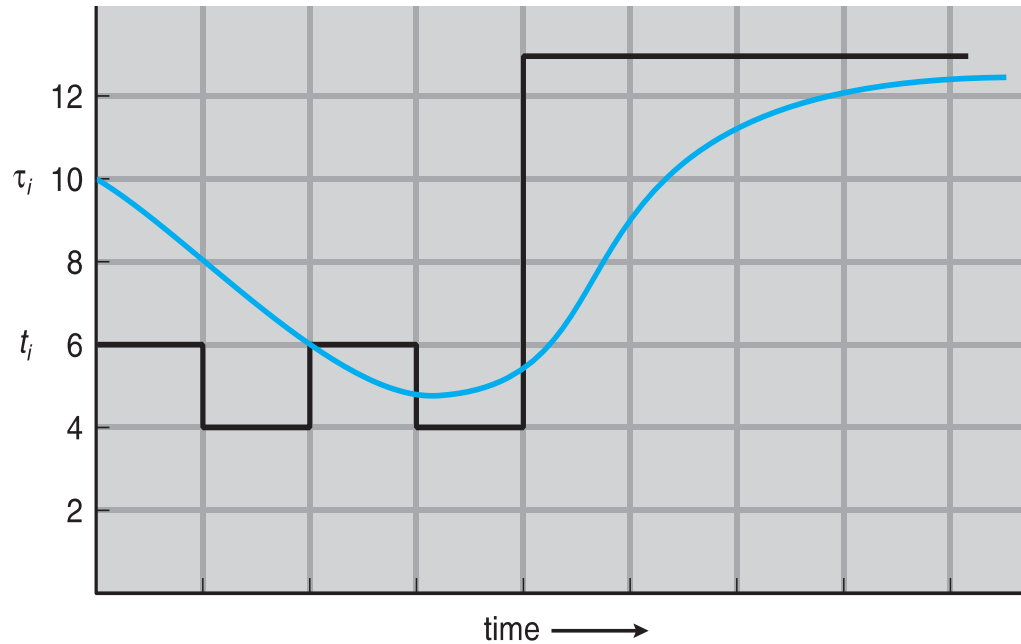
3. $\alpha, 0 \leq \alpha \leq 1$

4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$. Commonly, α set to $1/2$

Scheduling Schemes

- **SJF (Shortest Job First) scheduling**

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



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

Scheduling Schemes

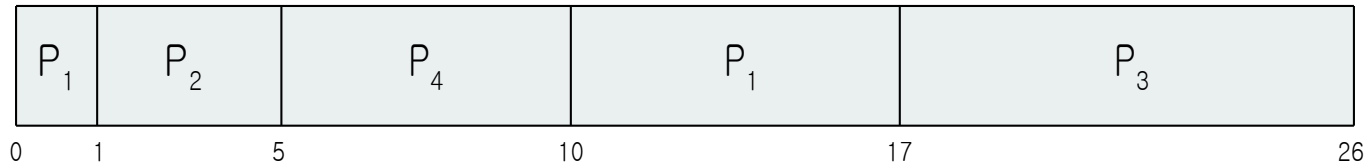
- **STCF (Shortest Time-to-Completion First) scheduling**
 - Variation of SJF scheduling (**preemptive** SJF)
 - Preemptive scheduling
 - Preempt current running process when another process with shorter remaining CPU burst time arrives at the ready queue
 - Cons
 - Burst time estimation overhead as in SJF
 - Overhead for tracing remaining burst time
 - High context switching overhead

Example of STCF

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive SJF* Gantt Chart



- Average Turnaround time = $[(17-0)+(5-1)+(26-2)+(10-3)]/4 = 52/4 = 13$

A New Metric: Response Time

- At time-shared machines, users would sit at a terminal and demand interactive performance from the system.
- **Response time:** the time from when the job arrives in a system to the first time it is scheduled
 - $T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$

A, B, and C arrive at the same time

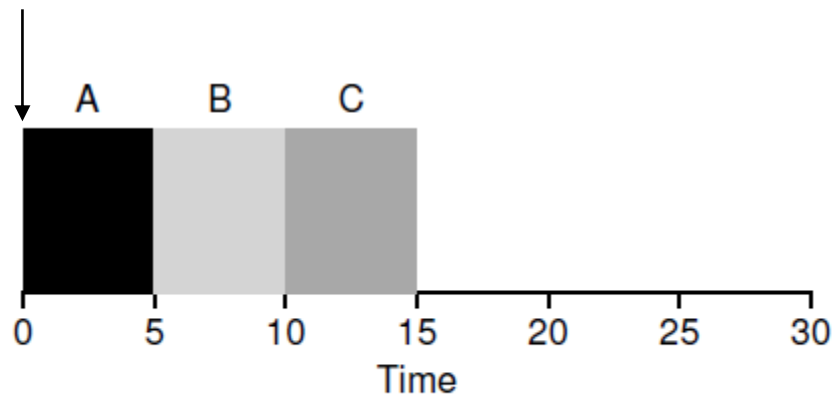


Figure 7.6: SJF Again (Bad for Response Time)

Scheduling Schemes

- **RR (Round-Robin) scheduling**

- Preemptive scheduling
- Scheduling criteria
 - Arrival time (at the ready queue)
 - Faster arrival time process first
- Time slice (scheduling quantum) for each process
 - System parameter
 - The (running) process that has exhausted his time slice releases the CPU and goes to the ready state (timer runout)
 - Prevents monopoly of the CPU by a process
- High context switching overhead due to preemptions
- Adequate for interactive/time-sharing system

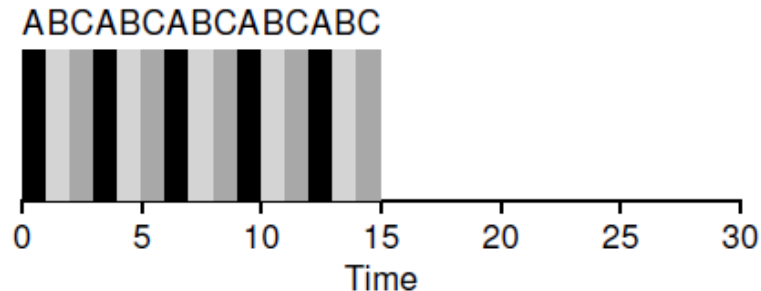


Figure 7.7: Round Robin (Good for Response Time)

Scheduling Schemes

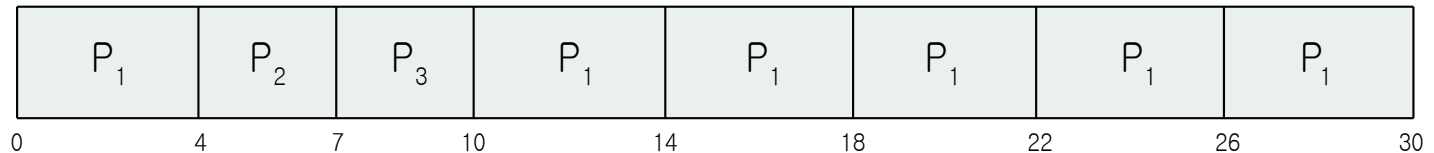
- **RR (Round-Robin) scheduling**

- Performance of the RR scheme depends heavily on the size of the time slice
 - Very large (infinite) time slice → FCFS
 - Very small time slice → processor sharing
 - Appears to the users as though each of the n processes has its own processor running at $1/n$ the speed of the real processor
 - Better response time
 - High context switching cost
 - OS actions of saving and restoring a few registers
 - H/W flush: Cache, TLB, branch predictor
- Deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to amortize the cost of switching without making it so long that the system is no longer responsive.

Example of RR with Time Quantum = 4

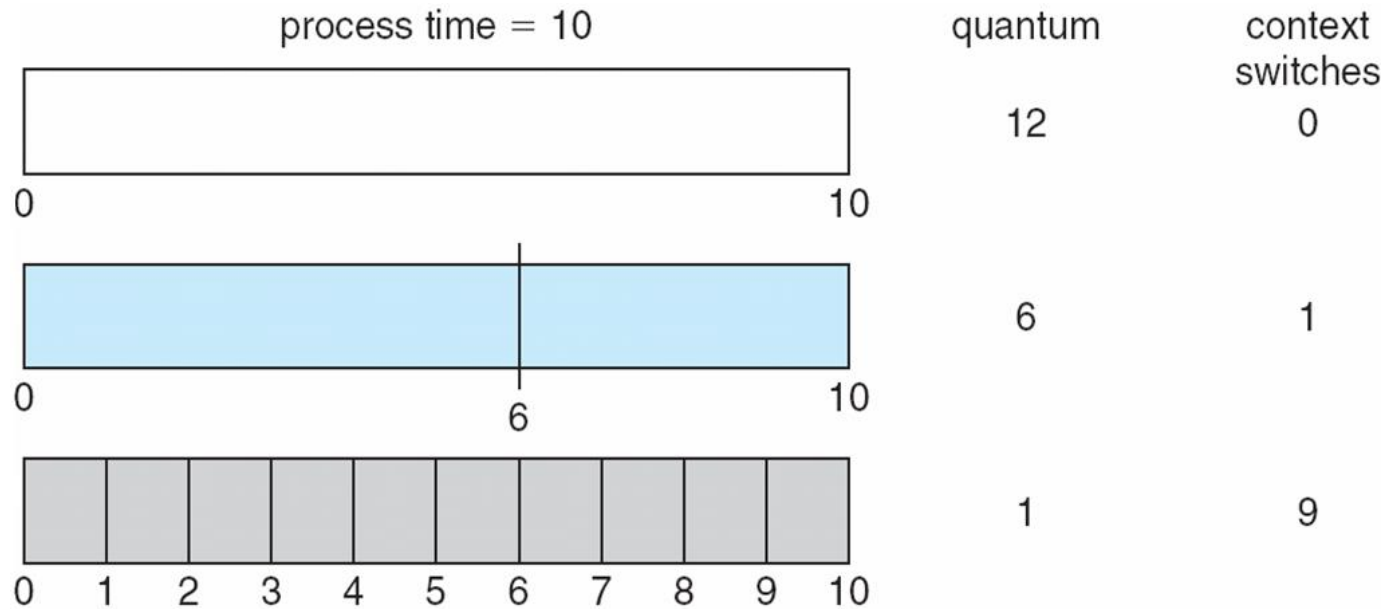
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

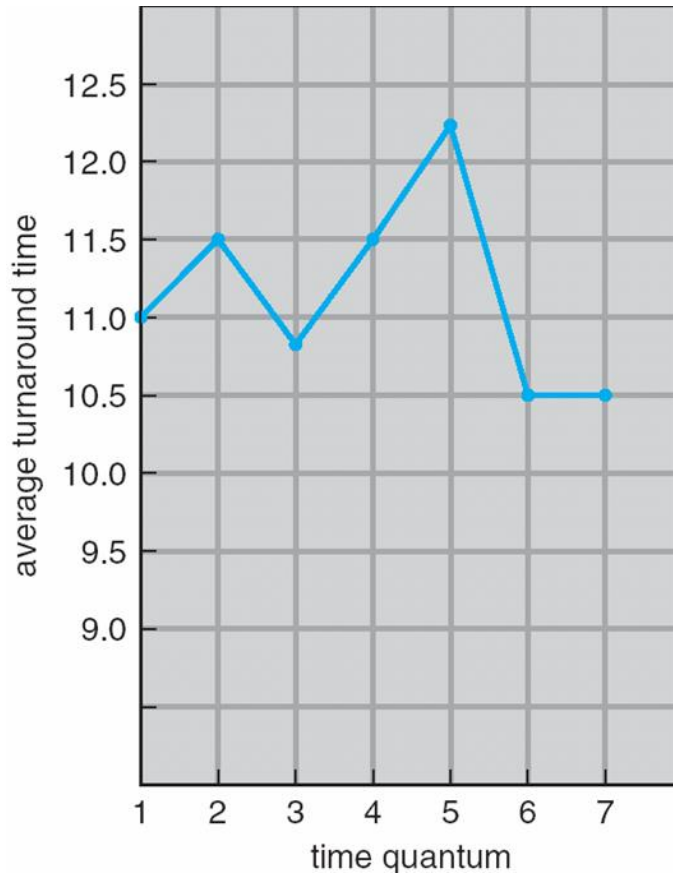


- Typically, **higher average turnaround** than SJF, but **better response**
 - *RR is indeed one of the worst policies if turnaround time is our metric*
- **q** should be large compared to context switch time
- **q** usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Incorporating I/O

- When a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is blocked waiting for I/O completion
- When the I/O completes, an interrupt is raised, and the OS runs and moves the blocked process back to the ready state.

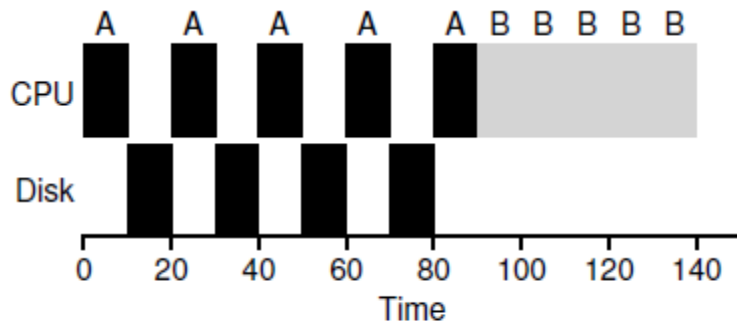


Figure 7.8: Poor Use of Resources

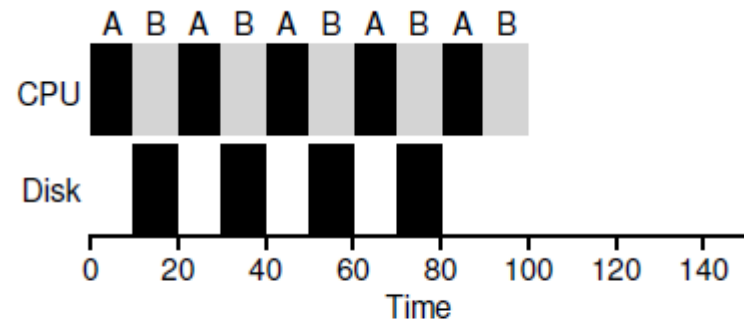


Figure 7.9: Overlap Allows Better Use of Resources

Treat each CPU burst as a job

Scheduling Schemes

- **Priority scheduling**

- Scheduling criteria
 - Process priority
 - Tie breaking: FCFS
- Priority range is different for each system
- Mapping from the numerical value of the priority to the priority level is different for each system

- Can be either preemptive or non-preemptive
- Major problem
 - Starvation
 - Solution
 - **Aging** – as time progresses increase the priority of the process

Scheduling Policies

- **Priority**
 - Classification
 - Static priority (external priority)
 - Decided at process creation time and fixed during execution of the process
 - Not adaptable to system environments
 - Simple, low-overhead
 - Dynamic priority (internal priority)
 - Initial priority at process creation time
 - May vary as the state of the system and processes changes
 - Adaptable to system environments
 - Complex, high overhead due to priority adjustment

Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

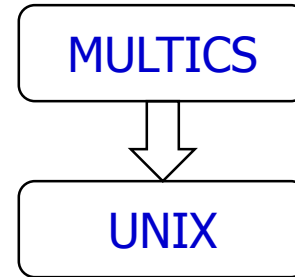
- **Priority scheduling Gantt Chart**



- **Average waiting time = 8.2 msec**

MLFQ (Multi-Level Feedback Queue)

- First described by Corbato et al. in 1962 in Compatible Time-Sharing System (CTSS) and Multics



Corbato, MIT 1965
(Turing Award 1990)

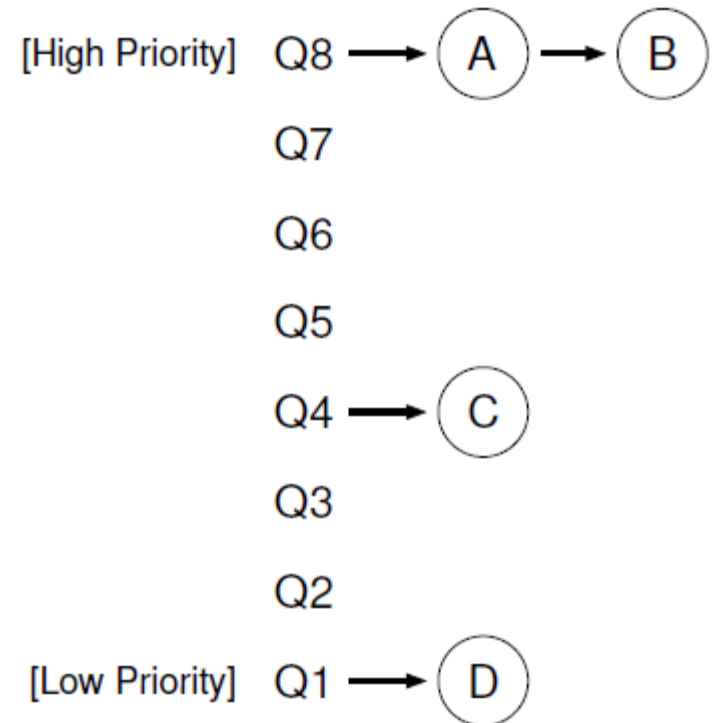
Ken Thompson
Dennis Ritchie
Bell Lab 1973
(Turing Award 1983)

- To optimize turnaround time
 - running shorter jobs first
 - Problem: SJF/STCF cannot know how long a job will run for
- To be responsive to interactive users
 - Round Robin
 - Problem: RR is terrible for turnaround time.
- Our problem
 - Given that we in general do not know anything about a process, how can we build a scheduler to achieve these goals?
 - **learn from the past to predict the future**

MLFQ: Basic Rules

- Multiple separate ready queues, each assigned a different priority.
- **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
- **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.

- Interactive process
 - Repeatedly relinquishes the CPU while waiting for input
 - High priority
- Batch process (CPU-bound)
 - Uses the CPU intensively for long periods of time
 - Low Priority



Attempt #1: How To Change Priority

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level.

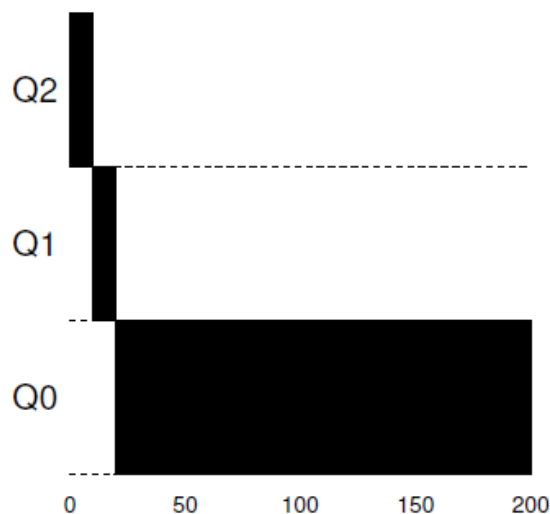


Figure 8.2: Long-running Job Over Time

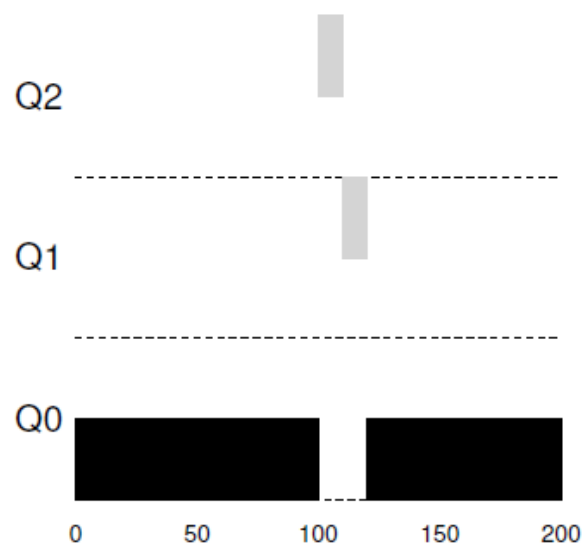


Figure 8.3: Along Came An Interactive Job

Problems With Our Current MLFQ

- **Starvation**
 - if there are “too many” interactive jobs in the system, long-running jobs will never receive any CPU time (they starve).
 - Need Priority Boost
- **Gaming the scheduler**
 - a smart user could rewrite their program
 - before the time slice is over, issue an I/O operation (to some file you don’t care about) and thus relinquish the CPU
- Program may **change its behavior** over time

Attempt #2: The Priority Boost

- **Rule 5:** After some time period S , move all the jobs in the system to the topmost queue.
 - Prevent starvation and detect the change of behavior
- **Aging** is also a choice
 - Processes that have long waiting time moves up in the queue hierarchy

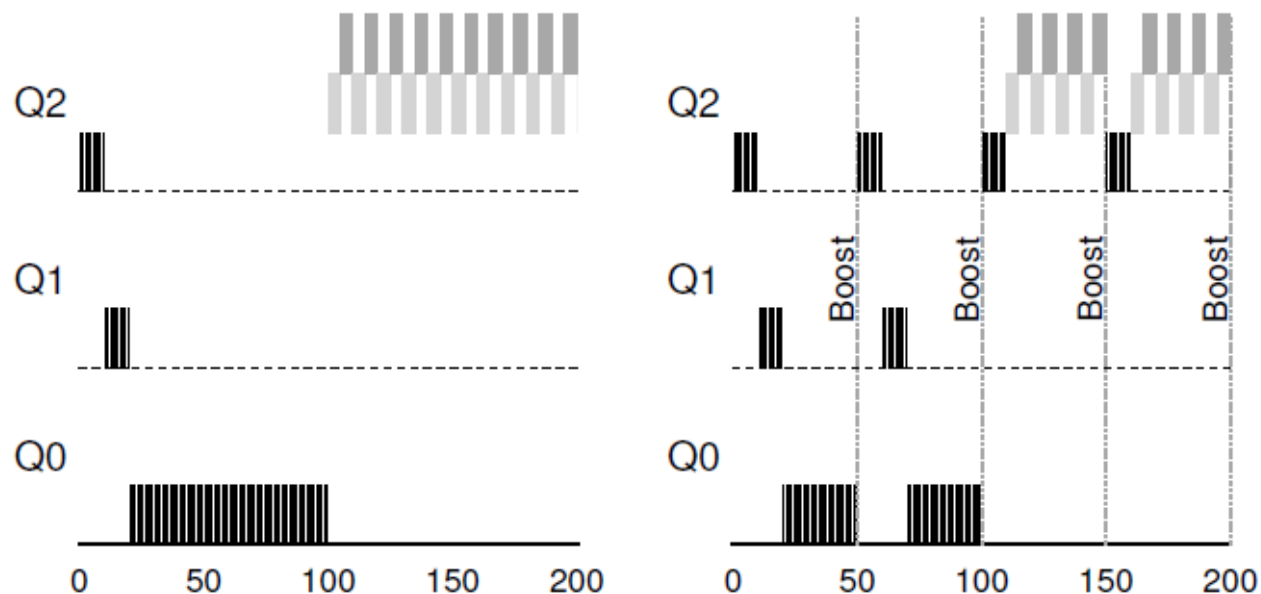


Figure 8.5: Without (Left) and With (Right) Priority Boost

Attempt #3: Better Accounting

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
 - Instead of forgetting how much of a time slice a process used at a given level, the scheduler should keep track

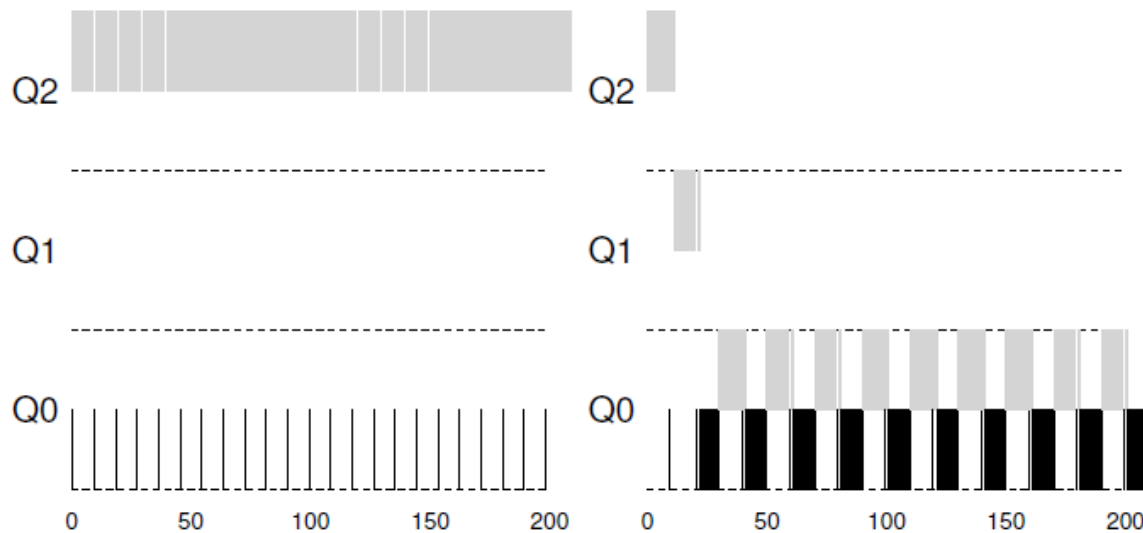


Figure 8.6: Without (Left) and With (Right) Gaming Tolerance

Attempt #4: Different Time Slice

- **Three queues:**

- Q_2 – RR with time quantum 8ms
- Q_1 – RR with time quantum 16ms
- Q_0 – FCFS

- **Scheduling**

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

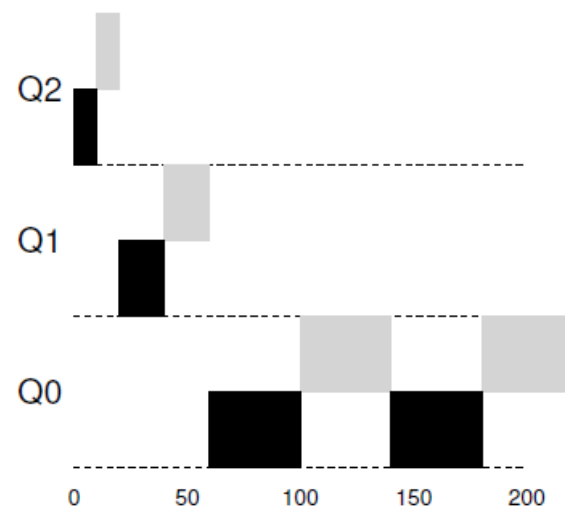
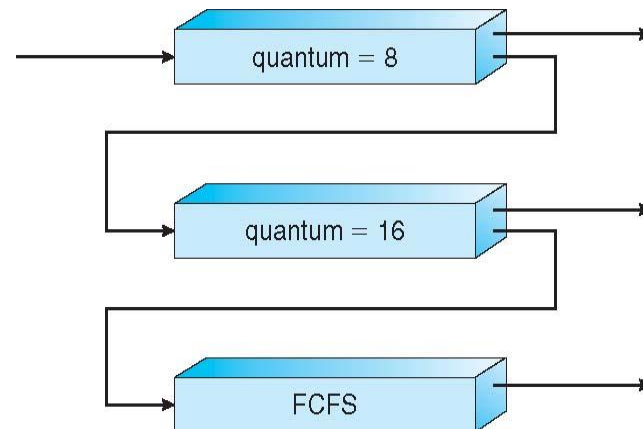


Figure 8.7: Lower Priority, Longer Quanta

Parameters for MLFQ scheduling

- The number of queues
- The scheduling algorithm for each queue
- The time slice of each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

- Easy Configuration
 - Provides a set of tables that determine exactly how the priority of a process is altered, how long each time slice is, and how often to boost the priority of a job (Solaris)
 - Uses a formula to calculate the current priority level of a job (FreeBSD)

Case Studies - Scheduling in Linux

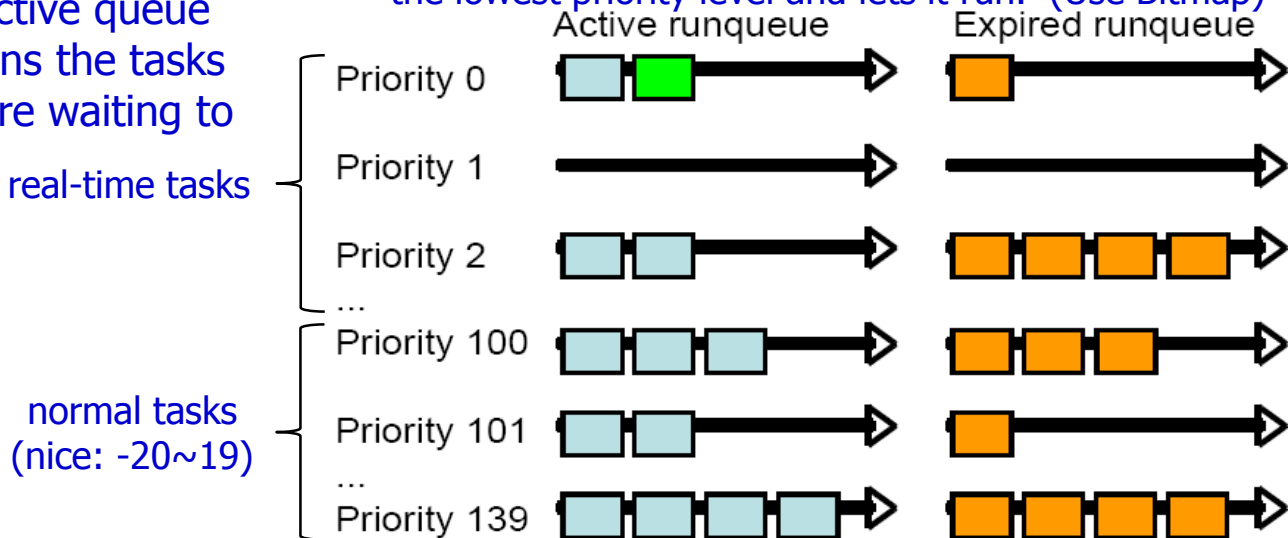
- **Version 2.5**
 - O(1) scheduling
 - Preemptive, priority based
 - Two priority ranges: time-sharing and real-time
 - Each processor maintains its own runqueue data structure and schedules itself independently
 - When all tasks at the lowest priority level have run, the expired and active queues at that priority level are exchanged.
 - SMP support
 - Processor affinity
 - Keep a task running on the same processor
 - Avoid the cost of invalidating and repopulating caches
 - Load balancing at every 200 ms
 - Fairness
 - Ideal for large server workloads
 - Worked well, but poor response times for interactive processes

O(1) scheduling

- As soon as the allocated execution time is used up the task is placed in the expired queue.
 - Time slice is allocated according to priority
 - If $(\text{priority} < 120) (140 - \text{priority}) * 20$, else $(140 - \text{priority}) * 5$
 - Ex. Priority: 100 \rightarrow 800ms, Priority: 120 \rightarrow 100ms, Priority: 139 \rightarrow 5ms
- Dynamic priority
 - $\max(100, \min(\text{static priority} - \text{bonus}, 139))$
 - bonus(-5~5) is given based on the average sleep time

The active queue contains the tasks that are waiting to run.

The scheduler picks the first task in the active queue of the lowest priority level and lets it run. (Use Bitmap) (Use Bitmap)



The expired queue contains the tasks that have recently run and therefore must wait for the other queue to be empty.

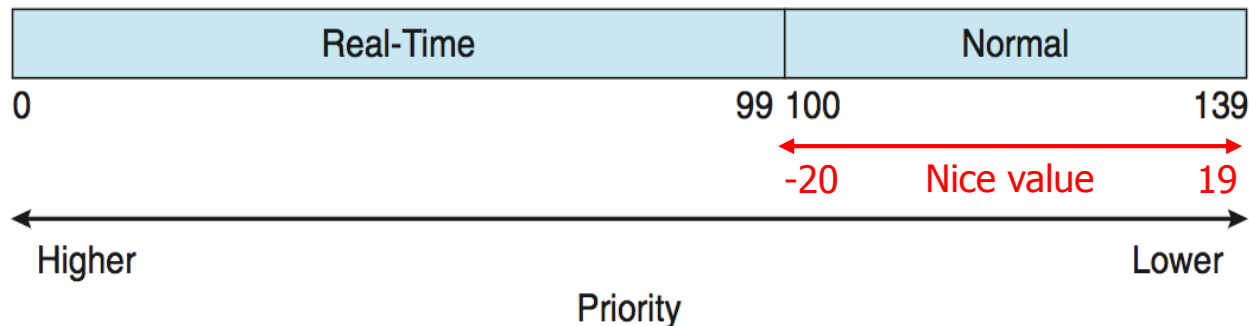
140 priority levels (lower numbers indicate higher priority);
The top 40: normal user tasks, The lower 100: real-time tasks.

Scheduling in Linux

- **Version 2.6.23+**

- CFS(Completely Fair Scheduler)

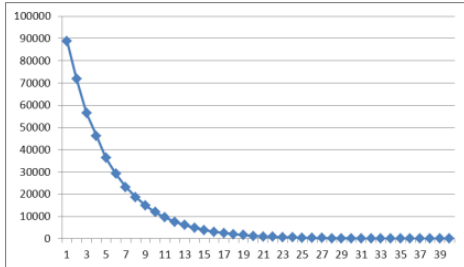
- Implementation of a well-studied, classic scheduling algorithm called fair queuing (fair scheduling)
 - Uses red-black trees
 - Not based on run queues
- Improvement in interactive performance of the scheduler
- $O(\log N)$ scheduling
 - N: number of tasks in the runqueue (red-black tree)



CFS (completely fair scheduler)

- **Weight-based time slice**

- Weight: based on priority



- Time slice depends on weight, not priority

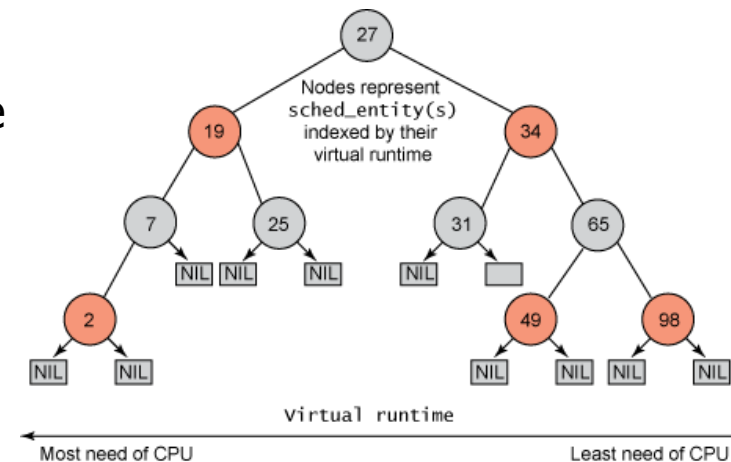
```
807 static const int prio_to_weight[40] = {
808 /* -20 */ 88761, 71755, 56483, 46273, 36291,
809 /* -15 */ 29154, 23254, 18705, 14949, 11916,
810 /* -10 */ 9548, 7620, 6100, 4904, 3906,
811 /* -5 */ 3121, 2501, 1991, 1586, 1277,
812 /* 0 */ 1024, 820, 655, 526, 423,
813 /* 5 */ 335, 272, 215, 172, 137,
814 /* 10 */ 110, 87, 70, 56, 45,
815 /* 15 */ 36, 29, 23, 18, 15,
816 };
kernel/sched/sched.h
```

- **time slice = period * (se→load.weight / cfs_rq→load.weight)**

- **period**: the time slice the scheduler tries to execute all tasks (default 20ms)
- **se→load.weight**: the weight of the schedulable entity. (prio_to_weight[])
- **cfs_rq→load.weight**: the total weight of all entities under CFS runqueue.
- Ex. Period = 100ms, task1 (prio=120), task2 (prio=121)
 - task1: $100 * (1024 / (1024 + 820)) = 56\text{ms}$
 - task2 : $100 * (820 / (1024 + 820)) = 44\text{ms}$

CFS (completely fair scheduler)

- Uses **virtual runtime** to track the progress of each entity
- Virtual runtime is the weighted time slice given to every schedulable entity
 - $\text{vruntime} = \text{NICE}_0_LOAD * (\text{delta_exec}/\text{se} \rightarrow \text{load.weight})$
 - `delta_exec`: the amount of execution time of that task (equals to slice if there is no preemption from higher priority task).
 - `NICE_0_LOAD`: the unity value of the weight
 - increase as a task consumes time-slice
 - decrease as a task has a higher priority
- Uses a vruntime-based **red-black tree** for each CPU
 - Red-black tree is a self-balancing binary tree
 - All runnable tasks are sorted in the RB tree by `se` \rightarrow `vruntime`
 - The leaf at the leftmost inside RB tree has the smallest value and most entitled to run at any given time.
- For interactivity optimization, CFS tunes the vruntime value that sleep for a period of time



Multiple Processor Scheduling

- **Classification-1**

- Homogeneous

- The processors in the system are identical in terms of their functionality
 - Can use any available processor to run any process in the queue
 - Considerations on devices that are attached to a specific processor are still necessary

- Heterogeneous

- See “Distributed computing”

Multiple Processor Scheduling

- **Classification-2**

- Asymmetric multiprocessing

- A single special processor, called master server, handles all scheduling decisions, I/O processing, and other system activities
 - The other processors execute only user codes
 - Simple

- Symmetric multiprocessing (SMP)

- Each processor: self-scheduling
 - Two types of ready queue
 - Global (common) ready queue
 - Private queue for each processor

Multiple Processor Scheduling

- Processor affinity
 - Avoid migration of processes from one processor to another and attempt to keep a process on the same processor
 - To reduce the cost of invalidating and repopulating caches
 - Soft affinity
 - Attempts to keep a process on the same processor, but does not guarantee it
 - Hard affinity
 - No migration of processes
 - Linux has system calls and commands that support hard affinity
 - # taskset -pc 0-2,4-7 35324
 - # taskset -p 0x000000F7 35324

```
#include <stdio.h>
#include <stdlib.h>
#define __USE_GNU
#include <sched.h>
#define MAX_CORES 8
int main()
{
    cpu_set_t mask;
    int core;

    /** Write Affinity **/
    CPU_ZERO(&mask);
    for(core=0; i < MAX_CORES; core++)
        CPU_SET(core, &mask);
    CPU_CLR(3,&mask);
    sched_setaffinity(0, sizeof(mask), &mask);

    /** Read Affinity */
    sched_getaffinity(0, sizeof(mask), &mask);
    printf("Process %d is allowed to run on core:
",getpid());
    for(core=0; i < MAX_CORES; core++){
        if(CPU_ISSET(core, &mask) == 1)
            printf("%d ",core);
    }
    return 0;
}
```

Multiple Processor Scheduling

- **Issues**

- Load balancing

- Attempts to keep the workload evenly distributed across all processors in the system

- Push migration

- A specific task that monitors the load on each processor

- When it finds imbalance

- Evenly distributes the load by moving (pushing) processes from overloaded to underloaded processors

- Pull migration

- Idle processor pulls a waiting processes from a busy processors

Real-time Scheduling

- **Priority-based scheduling**
- **RM(Rate-Monotonic) scheduling**
 - Static priority
- **EDF(Earliest-Deadline-First) scheduling**
 - Dynamic priority

Homework

- Submit Report-2 on Chap 10 (Multiprocessor Scheduling)