Operating Systems 2017/2018

Part III – Process Scheduling

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Slides based on the book

'Operating System Concepts, 9th Edition,

Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley'

Chapter 6

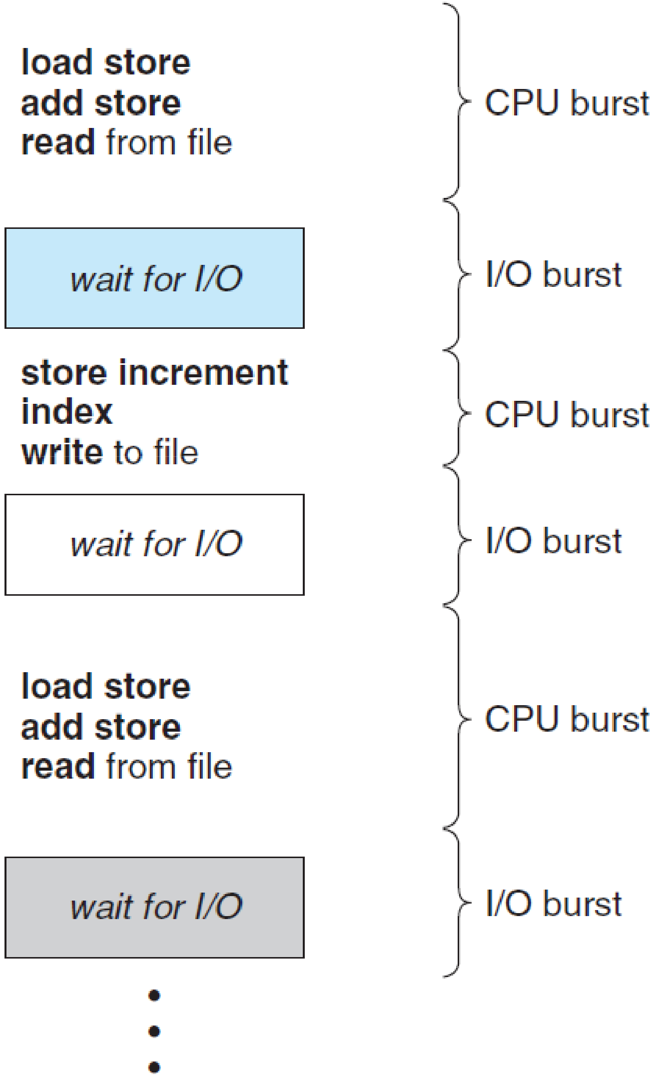
Motivation

- The goal of **multiprogramming** is to have some process running at all times, thus **maximizing CPU utilization**
  - When one process has to wait, the operating system takes that process away from the CPU and gives the CPU to another process

- A fundamental operating system function, which is also the basis of multiprogramming, is thus **process scheduling**
  - By efficiently scheduling the CPU among several processes, the operating system can serve more tasks and make the computer more productive

CPU-I/O Burst Cycle

- Process execution can be seem as a **cycle of CPU execution and I/O wait times**
  - Process execution begins with a CPU burst that is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on…
  - Eventually, the final CPU burst ends with a system request to terminate execution



Scheduling Decisions

- Scheduling decisions may take place when a process:
  - Switches from running to waiting state (as the result of a I/O request)
  - Switches from waiting to ready (as the result of I/O completion)
  - Switches from running to ready state (as the result of an interrupt)
  - Terminates
- The scheduler selects from among the **processes in the ready queue and allocates the CPU to one of them**
  - When the scheduling decisions takes place only under circumstances 1 and 4, we say that the scheduler is **nonpreemptive (or cooperative)**
  - Otherwise, the scheduler is **preemptive**
- Preemptive scheduling **requires special hardware such as a timer**

Preemptive Scheduling

- Preemptive scheduling can result in **race conditions** (i.e., the output depends on the execution sequence of other uncontrollable events)
  - While one process is updating data, it is preempted so that a second process can run. The second process then tries to read the same data, which can be in an inconsistent state.
  - The processing of a system call may involve changing important kernel data (for instance, I/O queues). If the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure, then chaos occurs.

- Because interrupts can occur at any time, these sections of code must be guarded from concurrent accesses by several processes and for that **interrupts are disabled at entering such sections and only reenabled at exit**

Scheduling Criteria

- Many criteria have been suggested for comparing scheduling algorithms. Some of the most well-know are:
  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – number of processes that complete execution per time unit
  - **Turnaround/Completion time** – amount of time required to execute a process (interval from the time of submission to the time of completion)
  - **Waiting time** – amount of time a process has been waiting in the ready queue
  - **Response time** – amount of time it takes from when a request was submitted until a first response (not output) is produced (for time-sharing environments)

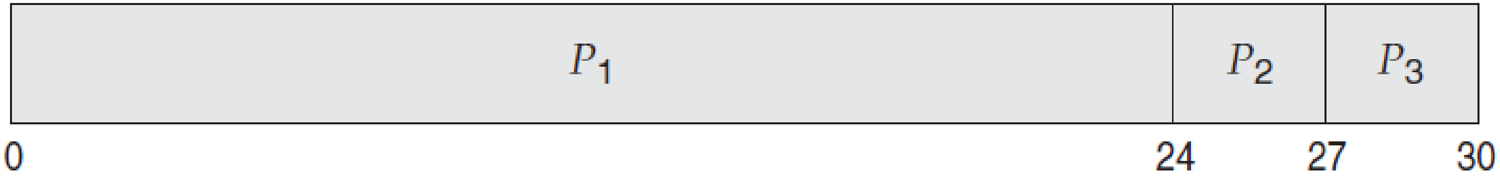
- Optimization criteria:
  - Maximize CPU utilization and throughput
  - Minimize turnaround time, waiting time and response time

First-Come First-Served (FCFS)

- The process that **requests the CPU first is allocated the CPU first**
  - Easily managed with a FIFO queue
  - When a process enters the ready queue, it is linked onto the tail of the queue
  - When the CPU is free, it is allocated to the process at the head of the ready queue (and the process is then removed from the queue)

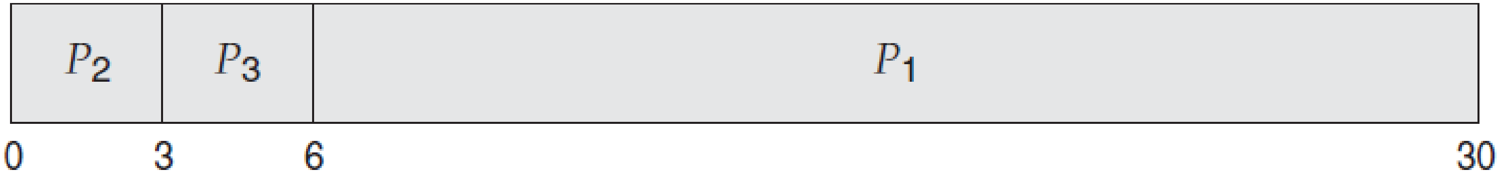
- FCFS is **nonpreemptive**, once the CPU has been allocated to a process, that process keeps the CPU until it either terminates or requests I/O
  - The **average turnaround and waiting time is often quite long**
  - **Troublesome for time-sharing systems**, where it is important that each user get a share of the CPU at regular intervals (it would be disastrous to allow one process to keep the CPU for an extended period)

First-Come First-Served (FCFS)



Process	Burst Time
P_1	24
P_2	3
P_3	3

- FCFS average waiting time
  - (0 + 24 + 27) / 3 = 51 / 3 = 17



Process	Burst Time
P_2	3
P_3	3
P_1	24

- FCFS average waiting time
  - (0 + 3 + 6) / 3 = 9 / 3 = 3

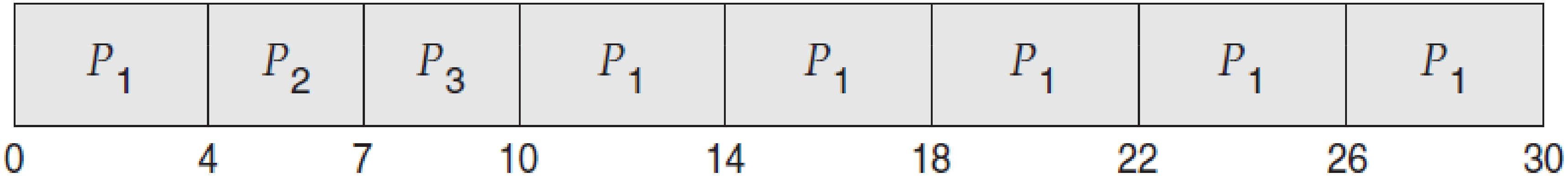
Round Robin (RR)

- Kind of **FCFS with preemption** specially designed for time-sharing systems:
  - Each process gets a **time quantum** or **time slice** (small unit of CPU time)
  - Timer interrupts every quantum to schedule next process, the current process is preempted and added to the end of the ready queue (ready queue works like a circular queue)

- If the time quantum is Q and there are N processes in the ready queue, 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)
  - Q large $\Rightarrow$ same as FCFS
  - Q small $\Rightarrow$ increases number of context switches, overhead can be too high

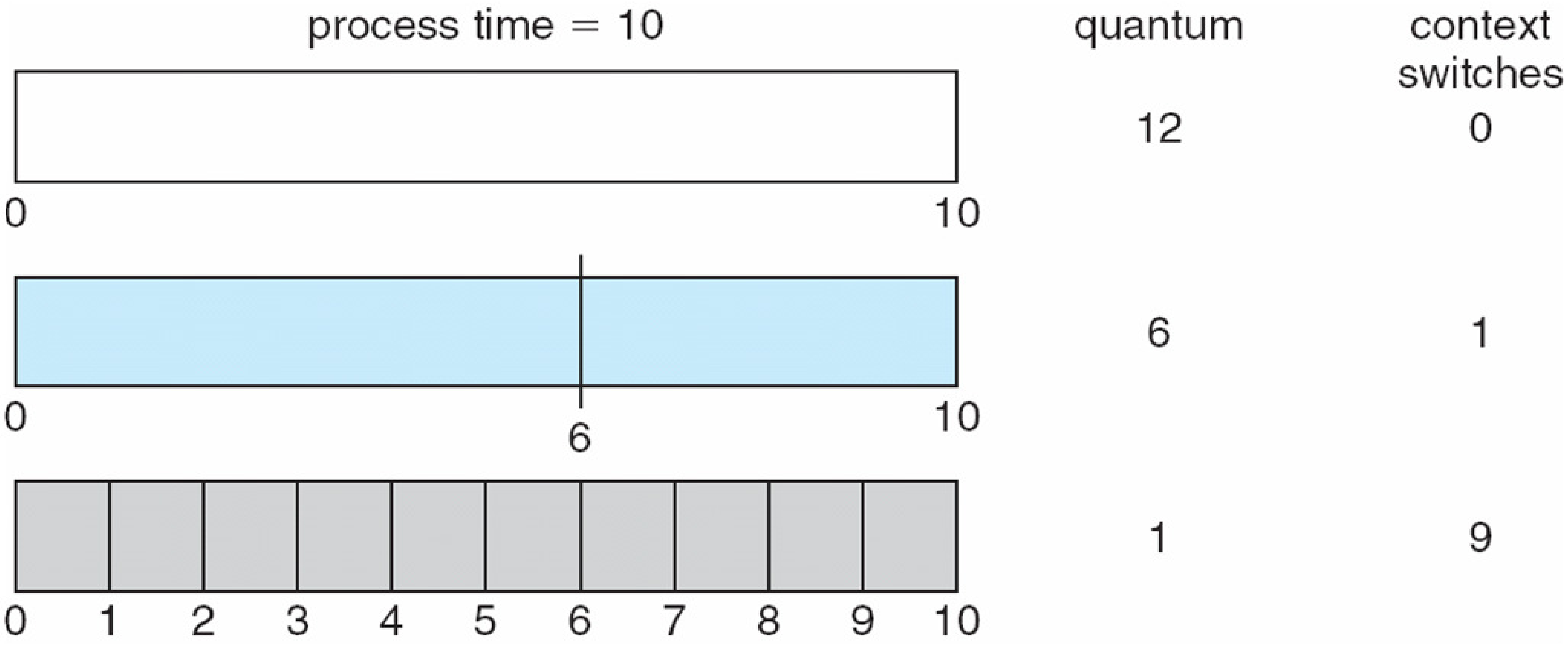
Round Robin (RR) with Time Quantum 4

Process	Burst Time
P_1	24
P_2	3
P_3	3



- RR (time quantum 4) average waiting time
  - (6 + 4 + 7) / 3 = 17 / 3 = 5.66

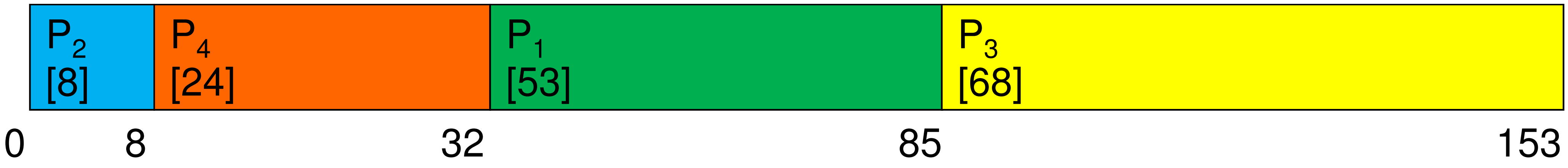
Time Quantum x Context Switch Time



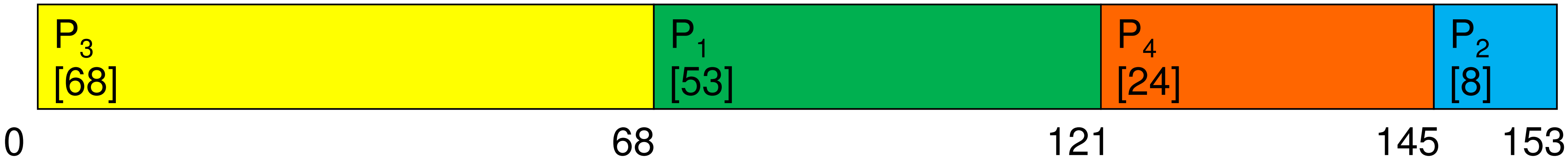
- Context switch time should be a small fraction of the time quantum:
  - The time required for a context switch is typically less than 10 microseconds
  - Most modern systems have time quantum ranging from 10-100 milliseconds

FCFS x RR

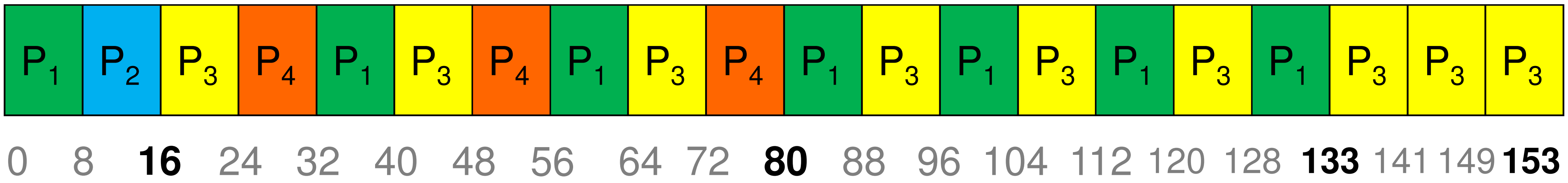
Best FCFS



Worst FCFS



Best RR – Time Quantum 8



FCFS x RR

	Quantum	P_1	P_2	P_3	P_4	Average
Average Waiting 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
Average 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

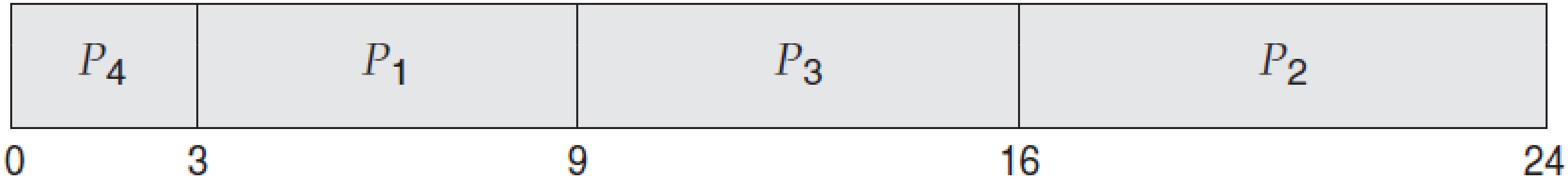
Shortest-Job-First (SJF)

- Associate each process with the length of its next CPU burst and use these lengths to **schedule the process with the shortest CPU burst**
  - If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie
  - Also called **shortest-time-to-completion-first (STCF)** but a more appropriate name would be **shortest-next-CPU-burst** since scheduling depends on the length of the next CPU burst of a process, rather than its total length

- **SJF is optimal** because it always gives the minimum average waiting time for a given set of processes
  - Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process
  - The **difficulty is knowing the length of the next CPU burst**

Shortest-Job-First (SJF)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- SJF average waiting time
  - (0 + 3 + 9 + 16) / 4 = 28 / 4 = 7
- FCFS average waiting time
  - (0 + 6 + 14 + 21) / 4 = 41 / 4 = 10.25

Predicting Next CPU Burst Length

- We may not know the length of the next CPU burst, but we may be able to predict its value using the length of the previous CPU bursts
- Generally predicted as an **exponential average** of the measured lengths of previous CPU bursts with the formula:

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

where

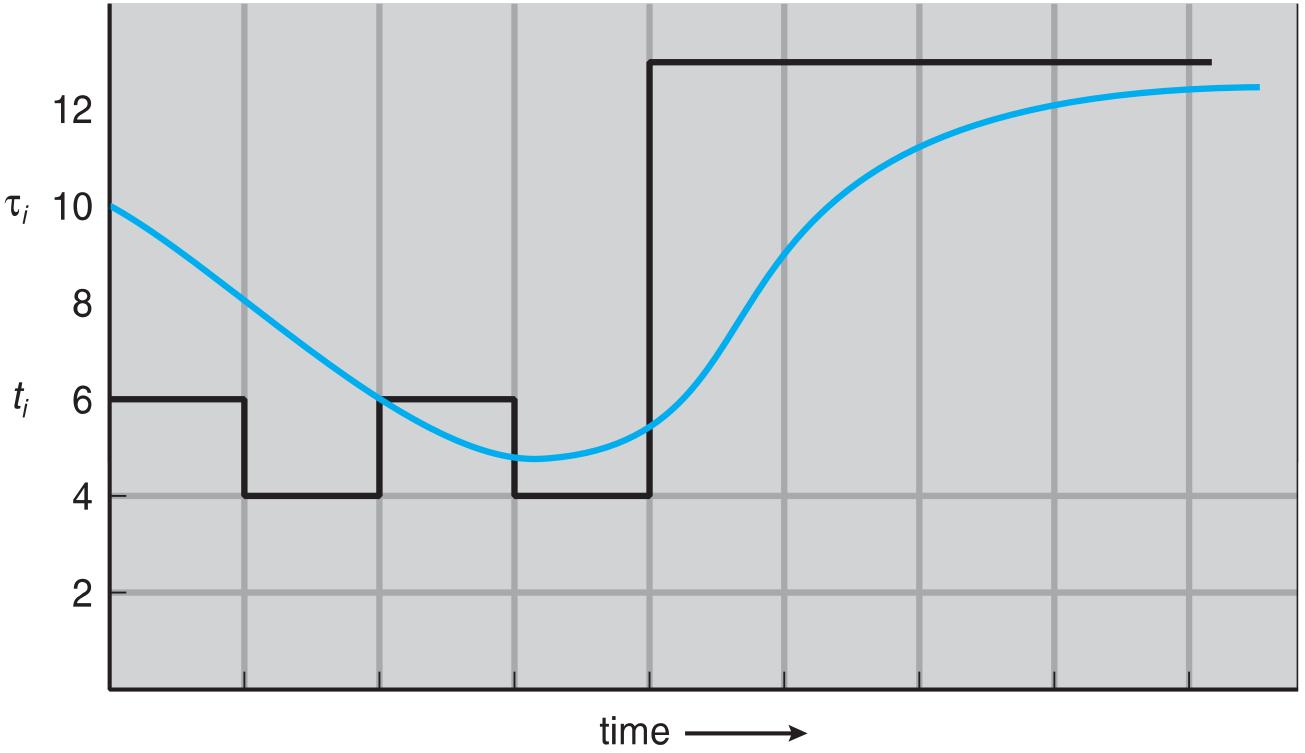
$\tau_{n+1} =$ predicted value for the next CPU burst

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

$0 \le \alpha \le 1$, commonly $\alpha = 1/2$

- Considering $\alpha = 1/2$ we thus have $\tau_{n+1} = (t_n + \tau_n)/2$

Predicting Next CPU Burst Length



CPU burst (t_i) 6 4 6 4 13 13 13 …

"guess" (τ_i) 10 8 6 6 5 9 11 12 …

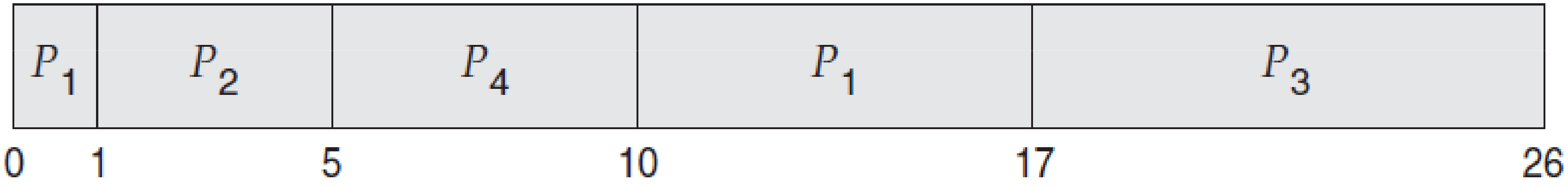
$$\tau_{n+1} = (t_n + \tau_n)/2$$

Shortest-Remaining-Time-First (SRTF)

- SJF scheduling can be **either nonpreemptive or preemptive**
  - Preemptive SJF is usually called SRFT scheduling
- The choice of being preemptive or not occurs when a new process arrives at the ready queue and the next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process
  - **SJF (nonpreemptive)** scheduling will allow the currently running process to finish its CPU burst
  - **SRTF (preemptive)** scheduling will preempt the currently executing process and schedule the newly arrived process

Shortest-Remaining-Time-First (SRTF)

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

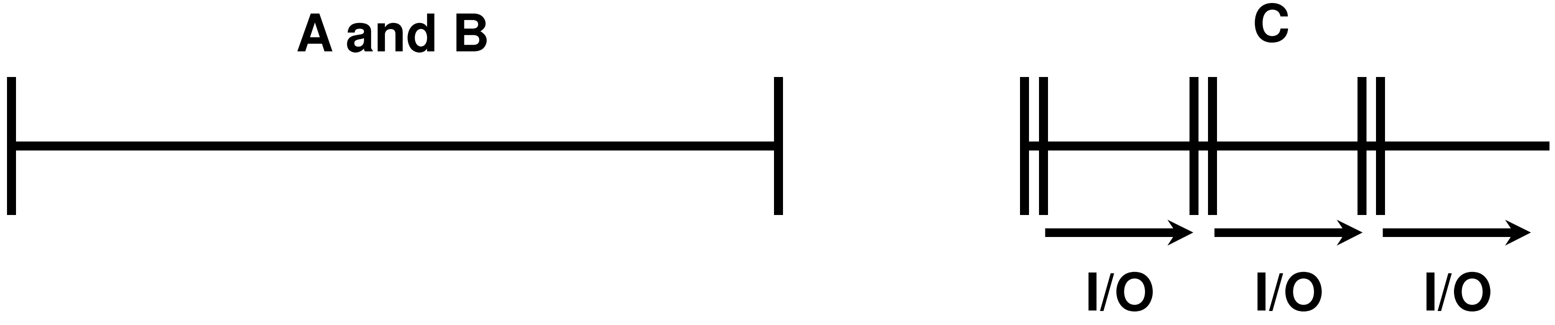


- SRTF (preemptive) average waiting time
  - [(17-8-0) + (5-4-1) + (26-9-2) + (10-5-3)] / 4 = 26 / 4 = 6.5
- SJF (nonpreemptive) average waiting time
  - [0 + (8-1) + (12-3) + (17-2)] / 4 = 31 / 4 = 7.75

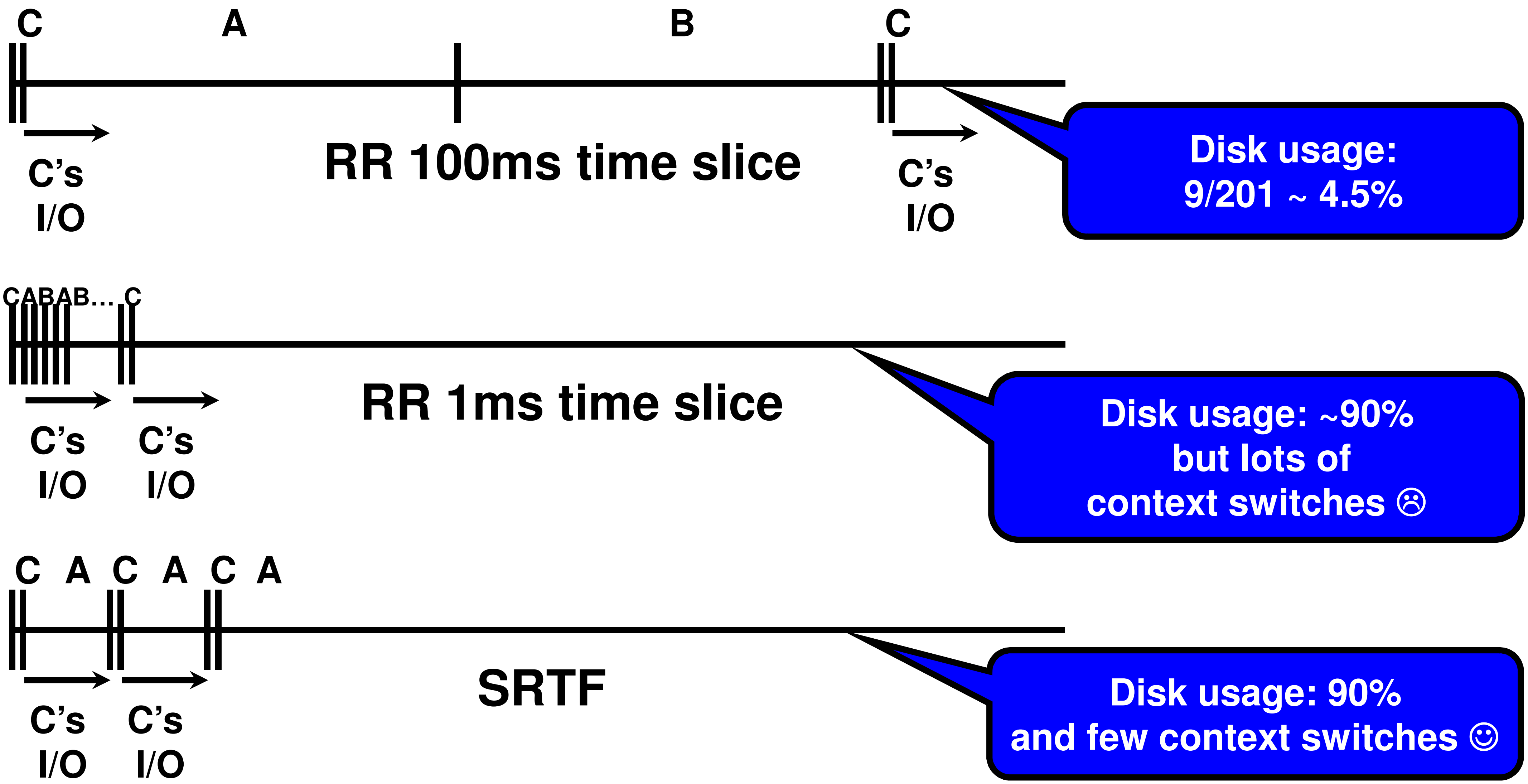
RR x SRTF

- Consider three processes:
  - Processes A and B: CPU-bound, each run for a hour
  - Process C: I/O-bound, loop 1ms CPU, 9ms disk I/O

- If only one at a time:
  - Processes A or B use 100% of the CPU
  - Process C uses 10% of the CPU (90% accessing the disk)



RR x SRTF



C A B C

C's I/O

RR 100ms time slice

C's I/O

Disk usage:
9/201 ~ 4.5%

CABAB… C

C's I/O C's I/O

RR 1ms time slice

Disk usage: ~90%
but lots of
context switches ☹

C A C A C A

C's I/O C's I/O

SRTF

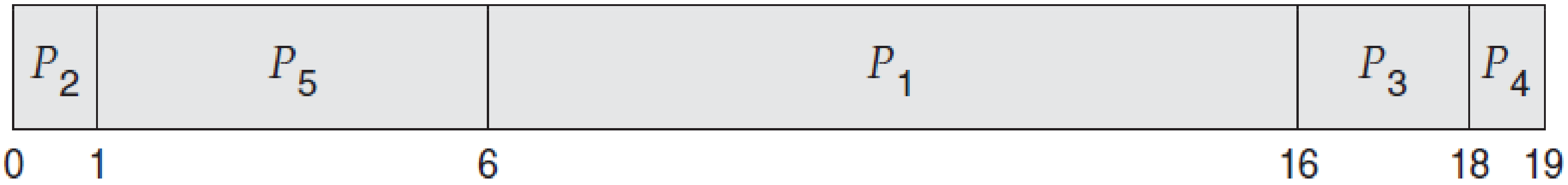
Disk usage: 90%
and few context switches ☺

Priority Scheduling

- Priority scheduling associates a **priority number** with each process and the **CPU is allocated to the process with the highest priority**
  - Equal-priority processes are scheduled in FCFS order
  - SJF and SRTF can be seen as priority algorithms
- Priority scheduling can be either:
  - **Preemptive**, preempts the CPU if the priority of the newly arrived process is higher than the priority of the currently running process
  - **Nonpreemptive**, allows the currently running process to finish its CPU burst
- A major problem is **indefinite blocking or starvation**
  - Low priority processes may never execute and wait indefinitely
  - A common solution is **aging**, which involves **gradually increasing the priority of processes that wait in the system for a long time**

Priority Scheduling

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



- Priority scheduling average waiting time
  - (0 + 1 + 6 + 16 + 18) / 5 = 41 / 5 = 8.2

FCFS x RR x SJF & SRTF: Pros and Cons

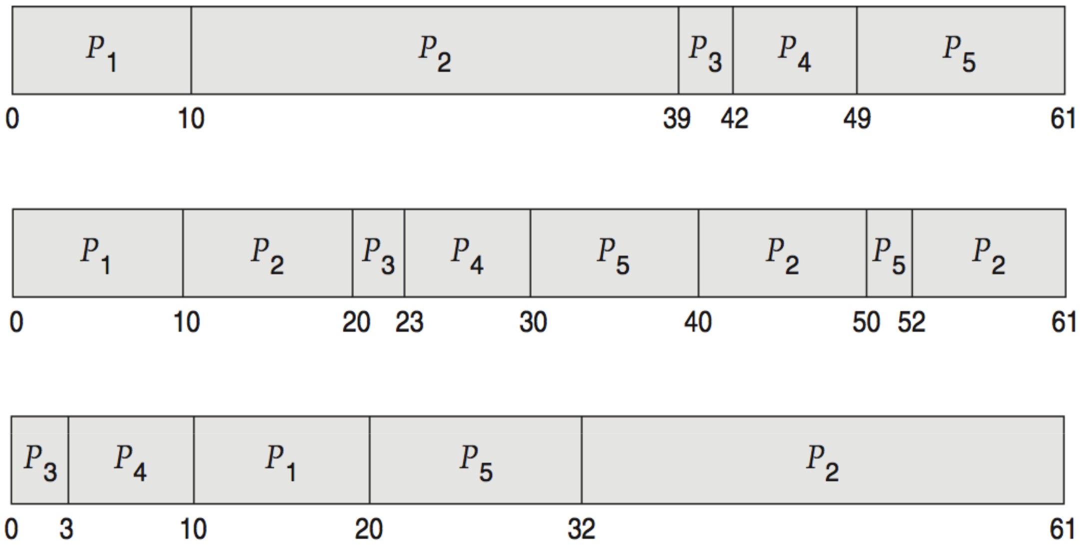
- FCFS
  - (+) Simple
  - (–) Short jobs get stuck behind long ones

- RR
  - (+) Better for short jobs
  - (–) Context switching time adds up for long jobs

- SJF & SRTF
  - (+) Optimal average waiting time
  - (+) Big effect on short jobs
  - (–) Hard to predict future
  - (–) Starvation

FCFS X RR x SJF: One Last Example

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

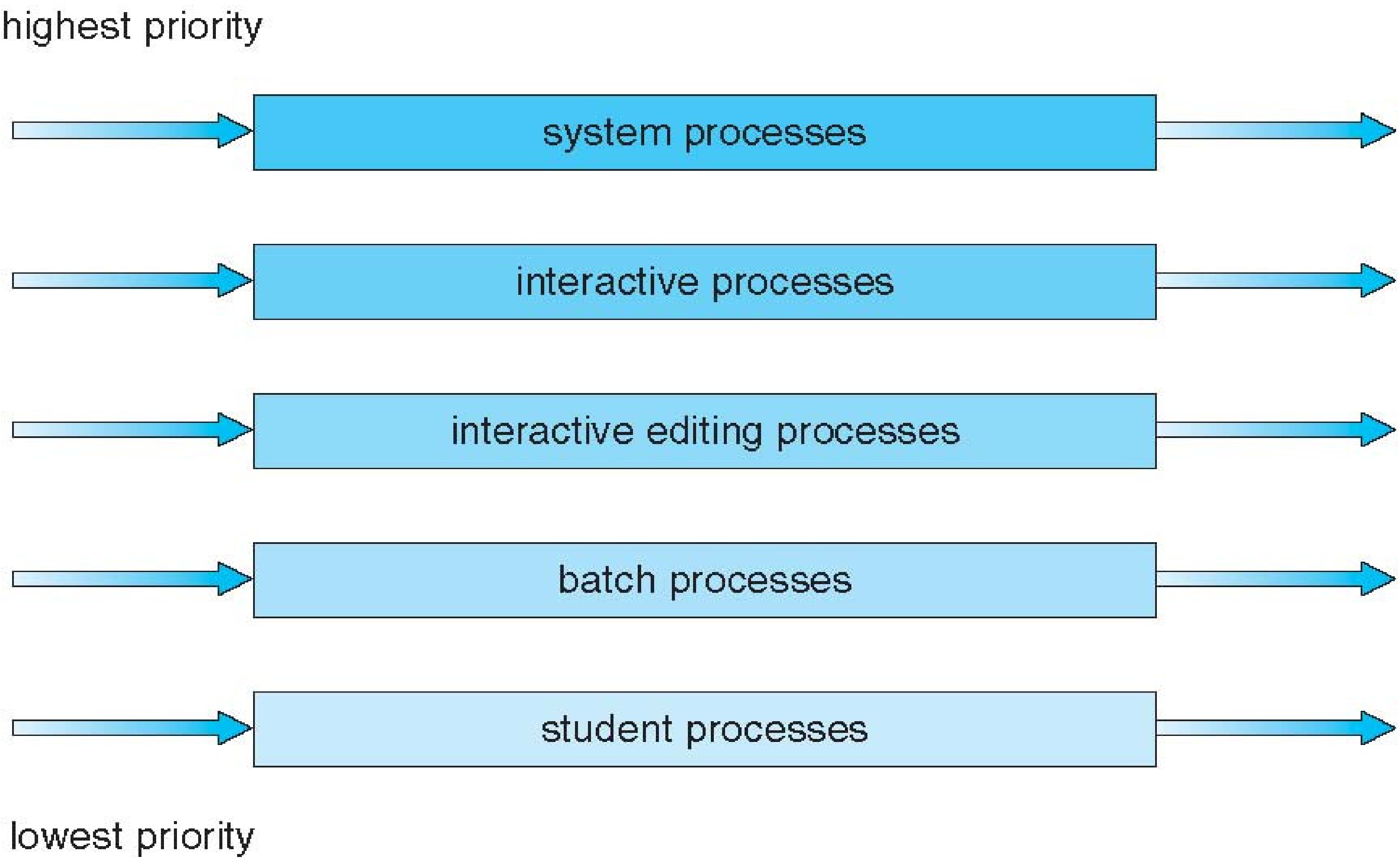


- FCFS average waiting time
  - (0 + 10 + 39 + 42 + 49) / 5 = 28
- RR (time quantum 10) average waiting time
  - [0 + (61-29) + 20 + 23 + (52-12)] / 5 = 115 / 5 = 23
- SJF (nonpreemptive) average waiting time
  - (0 + 3 + 10 + 20 + 32) / 5 = 65 / 5 = 13

Multilevel Queue (MLQ)

- MLQ scheduling **partitions the ready queue into several separate queues**
  - The processes are **permanently assigned to one queue**, generally based on some property of the process, such as memory size, process priority, …
  - **Each queue has its own scheduling algorithm** (one queue might be scheduled using RR while other is scheduled by FCFS)
- In addition, there must be **scheduling among the queues**:
  - **Fixed priority scheduling** – each queue has absolute priority over lower-priority queues (**preemptive scheduling with possibility of starvation**)
  - **Time slice** – each queue gets a certain amount of CPU time which is then schedule amongst its processes (for example, 80% to the queue using RR and 20% to the queue using FCFS)

Multilevel Queue (MLQ)



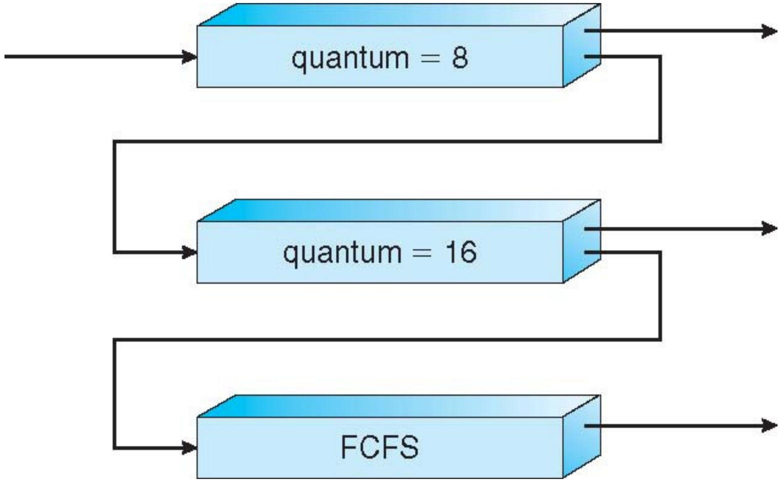
Multilevel Feedback Queue (MLFQ)

- Both setups for MLQ (fixed priority and time slice) have **low scheduling overhead, but are inflexible**

- MLFQ scheduling is more flexible as it **allows processes to move between queues**
  - Processes that use too much CPU time are moved to lower-priority queues
  - I/O-bound and interactive processes stay in the higher-priority queues
  - **Implement the concept of aging** by moving a process that waits too long in a lower-priority queue to a higher-priority queue, thus **preventing starvation**

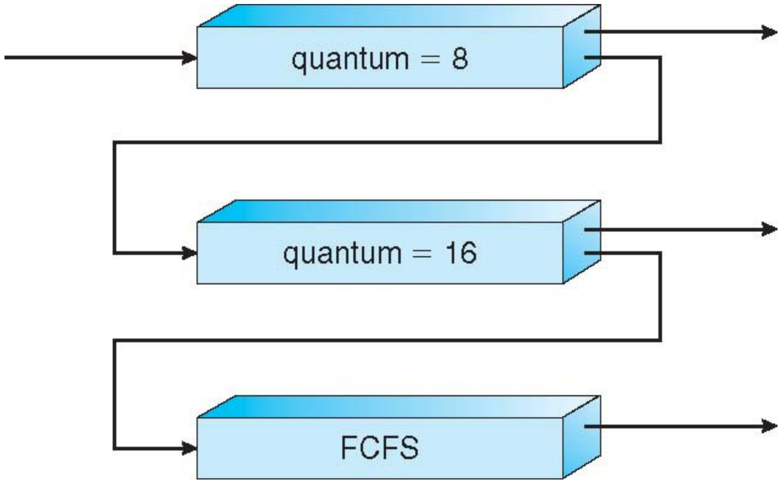
- BSD UNIX derivatives, Solaris, Windows NT and subsequent Windows operating systems use a form of MLFQ as their base scheduler

Multilevel Feedback Queue (MLFQ)



- Consider three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR with time quantum 16 milliseconds
  - Q_2 – FCFS
- Possible scheduling algorithm (I):
  - New processes enter the ready queue at the tail of Q_0
  - A process in the head of Q_0 is given a time quantum of 8ms, in the head of Q_1 is given a time quantum of 16ms, and in the head of Q_2 runs in an FCFS basis
  - Processes in Q_1 only run when Q_0 is empty and processes in Q_2 only run when both Q_0 and Q_1 are empty, but if a queue is not run for a certain amount of time, processes are moved to the next higher level (or topmost) queue
  - A process entering a higher level queue will preempt any process running in a lower level queue

Multilevel Feedback Queue (MLFQ)



- Consider three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR with time quantum 16 milliseconds
  - Q_2 – FCFS
- Possible scheduling algorithm (II):
  - If a process completes within its time quantum, it leaves the system
  - If a process uses all the time quantum, it is preempted and moved to the next lower level queue (thus penalizing CPU-bound processes)
  - If a process blocks for I/O, it leaves the current queue and when the process becomes ready again it is inserted at the tail of the same queue
  - Alternatively, once a process uses its total time quantum at a given level (regardless of how many times it has blocked for I/O), it is preempted and moved to the next lower level queue (thus preventing gaming the scheduler)

MLQ x MLFQ

- MLQ scheduling involves defining 4 parameters:
  - Number of queues
  - Scheduling algorithm for each queue
  - Scheduling algorithm among the queues (fixed priority or time slice)
  - Method to determine which queue a process will be assigned to

- MLFQ scheduling involves defining 5 parameters:
  - Number of queues
  - Scheduling algorithm for each queue
  - Method to determine which queue a process will initially enter
  - Method to determine when to upgrade a process to a higher-priority queue
  - Method to determine when to demote a process to a lower-priority queue

MLQ x MLFQ: Pros and Cons

- MLQ
  - (+) Low scheduling overhead
  - (–) Fixed priority scheduling is unfair, inflexible and can lead to starvation
  - (–) Time slice can hurt the average waiting time

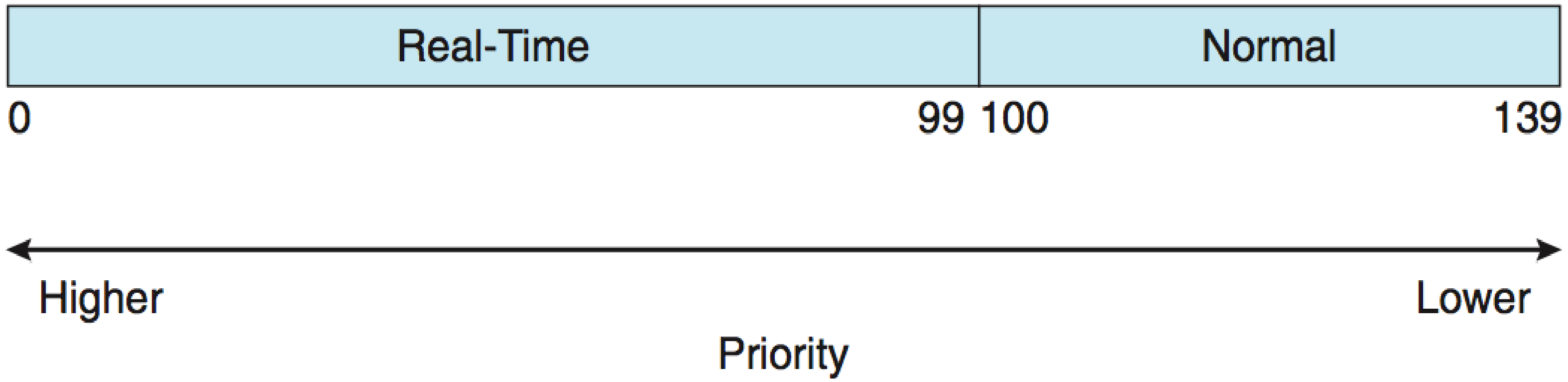
- MLFQ
  - (+) Excellent overall performance for short-running I/O bound processes and fair enough for long-running CPU-bound processes
  - (+) Results approximate SRTF
  - (+) Avoids starvation
  - (–) Requires some means by which to tune/select values for all 5 parameters

Completely Fair Scheduler (CFS)

- CFS is the scheduling algorithm adopted by the Linux kernel since release 2.6.23
- CFS tries to **divide CPU time fairly among all tasks** (processes or threads) **by taking into account their priorities and CPU usage history**
- CFS is based on **scheduling classes** where each class has a specific **priority range**
  - Scheduler picks the highest priority task from the highest priority class
  - Lower-priority tasks are preempted when higher-priority tasks are ready to run
- Typically, standard Linux kernels implement two scheduling classes:
  - **Real-time class**
  - **Normal (default) class**

Completely Fair Scheduler (CFS)

- The real-time class plus the normal class map into a global priority range:
  - **Real-time tasks** are assigned **static priorities** within the range [0,99]
  - **Normal tasks** have **nice values** and are assigned **dynamic priorities** within the range [100,139] based on their nice values
  - Nice values range from [-20,+19] and map to priorities [100,139] (the default nice value is 0)
  - A lower/higher nice value means higher/lower priority (the idea is that if a task increases its nice value, it is being nice to the other tasks in the system)



Completely Fair Scheduler (CFS)

- Real-time tasks are scheduled by priority and before tasks in other classes
- Normal tasks are scheduled accordingly to the **lowest virtual runtime value**
  - CFS maintains a per task virtual runtime value which **measures CPU time by associating a decay factor based on the nice value of the task**
  - Nice values of 0 yields a virtual runtime identical to the real runtime (if a task runs for 100 milliseconds, its virtual runtime will also be 100 milliseconds)
  - Lower-priority tasks have higher factors of decay, where higher-priority tasks have lower factors of delay (if a task runs for 100 milliseconds, its virtual runtime will be proportionally higher/lower than 100 milliseconds accordingly to its lower/higher-priority)
  - When a new task is created, it is assigned a virtual runtime equal to the current minimum virtual runtime

Completely Fair Scheduler (CFS)

- With CFS, tasks have **no fixed time slices** but rather **run until they are no longer the most unfairly treated task**

- CFS identifies a **target latency**, which is an interval of time during which every runnable task should run at least once
  - Target latency has default and minimum values but can increase if the number of active tasks in the system grows beyond a certain threshold

- Tasks get **proportions of CPU time** from the target latency value **accordingly to their relative priorities**

- When a task is awakened, the difference from its virtual runtime to the current minimum virtual runtime cannot exceed the target latency, otherwise its virtual runtime is adjusted to such limit
  - This prevents a task that has waiting too long from monopolizing the CPU

Completely Fair Scheduler (CFS)

- Consider the following scenario:
  - A target latency of 10ms and a decay factor of 2x
  - Process P_0 with virtual runtime 100ms and nice value 0 (min proportion: 2ms)
  - Process P_1 with virtual runtime 101ms and nice value -1 (min proportion: 4ms)
  - Process P_2 with virtual runtime 97ms and nice value -1 (min proportion: 4ms)

CFS x Virtual Runtimes	P_0	P_1	P_2
Initial virtual runtimes	100	101	97
Schedule P_2 for 6ms			100
Schedule P_0 for 2ms (I/O system call after 1ms)	101		
Schedule P_2 for 4ms	🟥		102
Schedule P_1 for 4ms	🟥	103	
Schedule P_2 for 4ms (hardware interrupt after 2ms)			103
Schedule P_0 for 2ms	103		