Fall 2015 COMP 3511
Operating Systems

Lab #5 Review

Outline

- Operating system scheduling examples
- Scheduling algorithms

Linux Scheduling Through Version 2.5

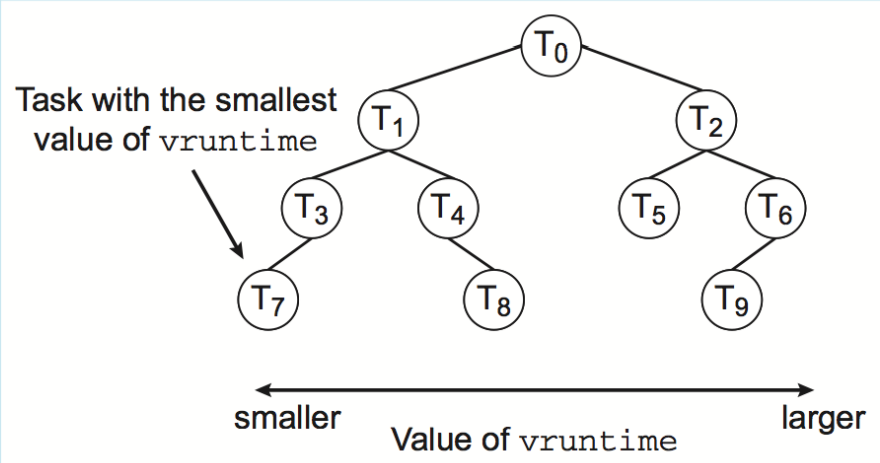
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

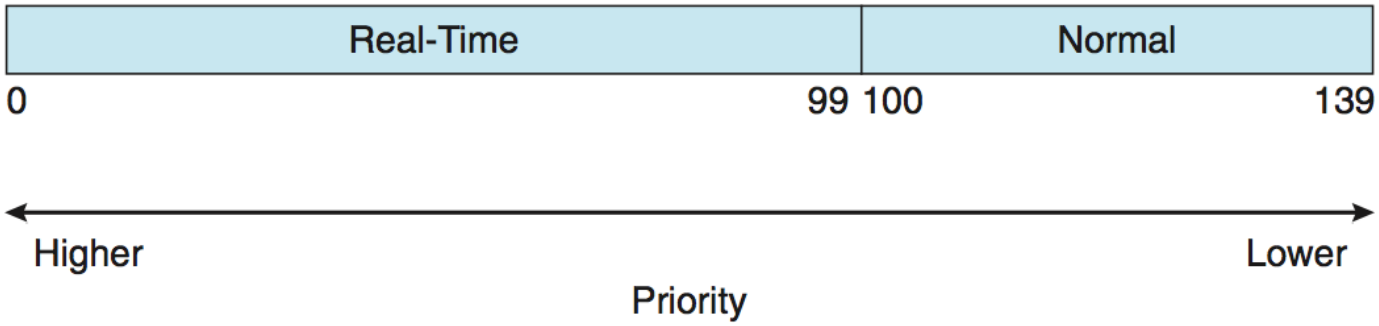
CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- 

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows Priorities

Highest priority

Priority Classes

Relative Priority	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Lowest priority

Solaris

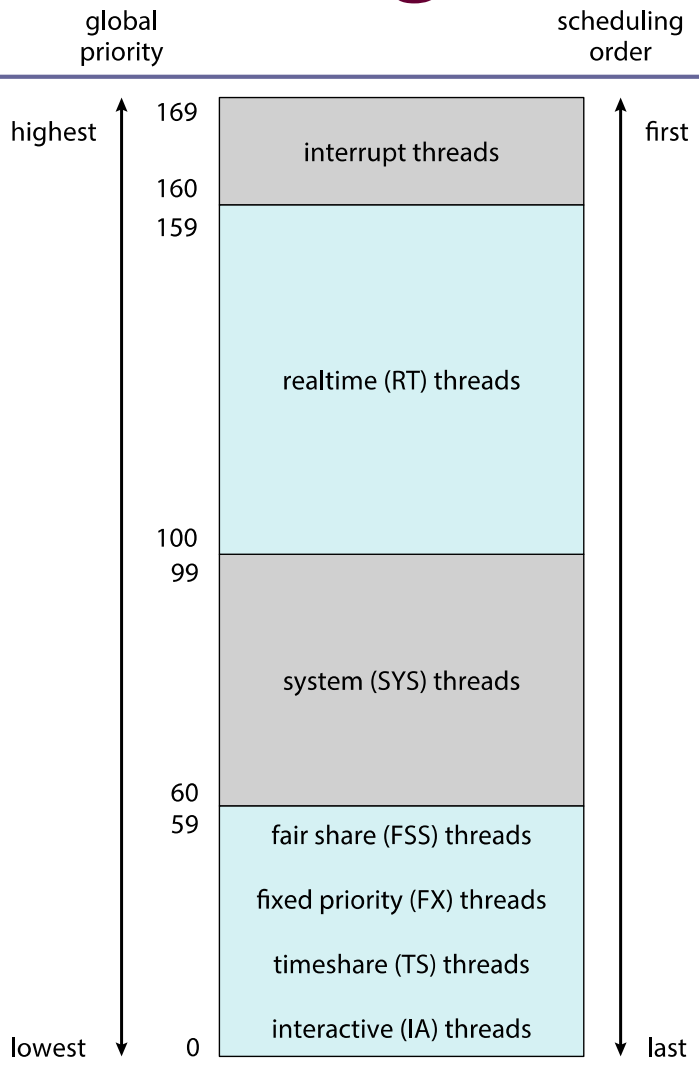
- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris Dispatch Table

priority	time quantum	New priority after time quantum expired	New priority after return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

E.g., a thread with priority 20 would be preempted after 120 ms. It is then suspended and assigned with a new priority 10, i.e. longer time in its next run.

Solaris Scheduling



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

CPU Scheduler & Dispatcher

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process
  1. Switches from running to waiting state
  2. Switches from running to ready state
  2. Switches from waiting to ready
  3. Terminates

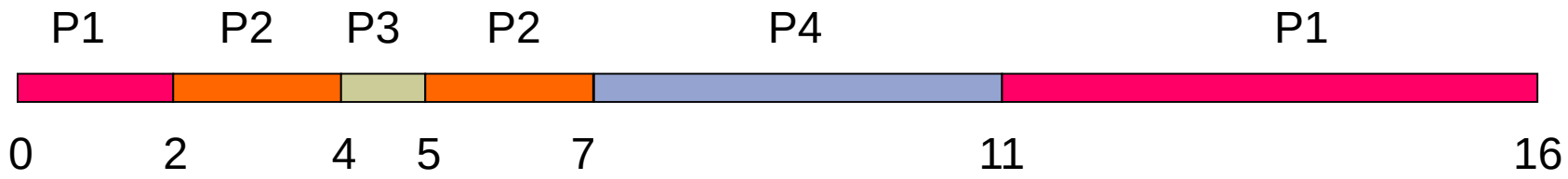
- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

Scheduling Criteria

- To Maximize:
  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – # of processes that complete their execution per time unit
- To Minimize:
  - **Turnaround time** – amount of time to execute a particular process
  - **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 the first response is produced, not output (for time-sharing environment)

Preemptive SJF: Example

Process	Burst Time	Arrival Time
P1	7	0
P2	4	2
P3	1	4
P4	4	5



P1 gets preempted at time 2

P2 gets preempted at time 4

The average waiting time
(9 + 1 + 0 +2)/4 = 3

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 a priority scheduling where priority is the predicted next CPU burst time
- Problem : **Starvation** – low priority processes may never execute
- Solution : **Aging** – as time progresses increase the priority of the process

Round Robin (RR) Scheduling

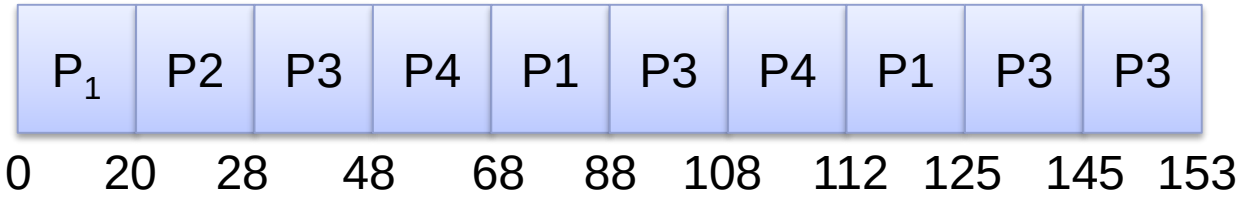
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After quantum expires, the process is preempted (OS timer interrupt) and added to the end of the ready queue.
- n processes in the ready queue and the time quantum is q
  - Each process gets 1/n of the CPU time
  - In chunks of at most q time units at once.
  - No process waits more than (n-1)q time units
- Performance
  - q large ➔ FIFO
  - q small ➔ Interleaved
  - q must be large with respect to context switch, otherwise overhead is too high

Example of RR in Notes

- Example:

Process	Burst Time
P1	53
P2	8
P3	68
P4	24

- The Gantt chart is:



- Waiting time for

P1=(68-20)+(112-88)=72
P2=(20-0)=20
P3=(28-0)+(88-48)+(125-108)=85
P4=(48-0)+(108-68)=88

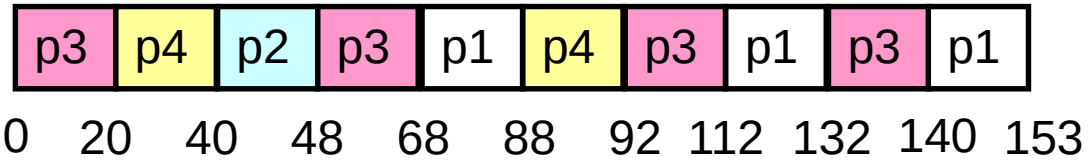
- Average waiting time = (72+20+85+88)/4=66¼
- Average completion time = (125+28+153+112)/4 = 104½

Consider Different Arrival Time

- Example:

Process	Burst Time	Arrival Time
P1	53	28
P2	8	16
P3	68	0
P4	24	10

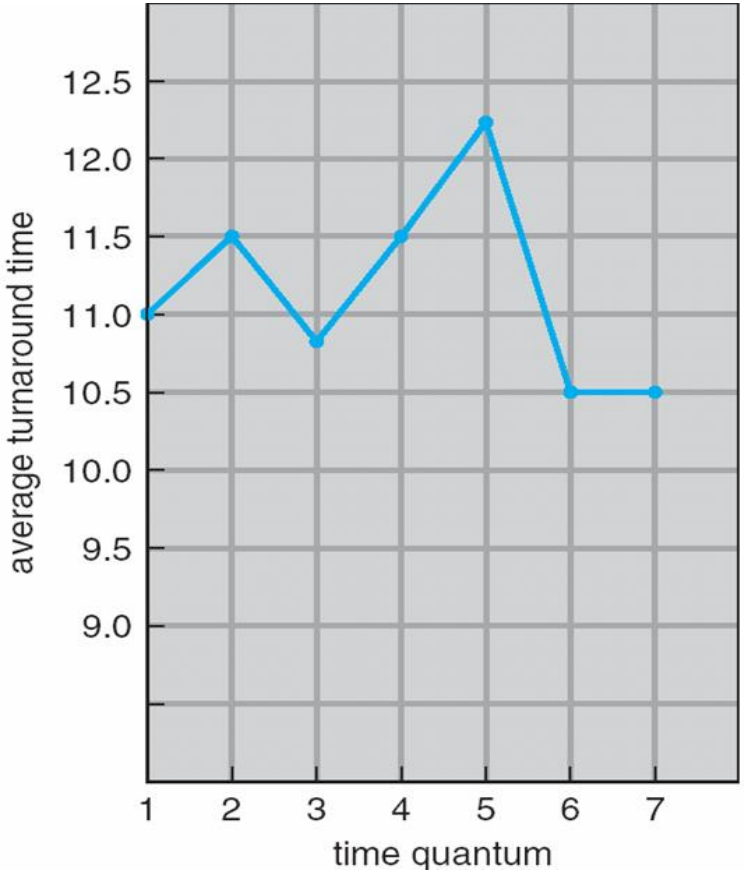
- The Gantt chart is: turnaround time = completion time-arrival time



- Waiting time for P1=(68-28)+(112-88)+(140-132)=72
  P2=(40-16)=24
  P3=(48-20)+(92-68)+(132-112)=72
  P4=(20-10)+(88-40)=58

- Average waiting time = (72+24+72+58)/4 = 56½
- Average completion time = (153+48+140+92)/4 = 108¼
- Average turnaround time = (125+32+140+82)/4 = 94¾

Turnaround Time Varies With The Time Quantum



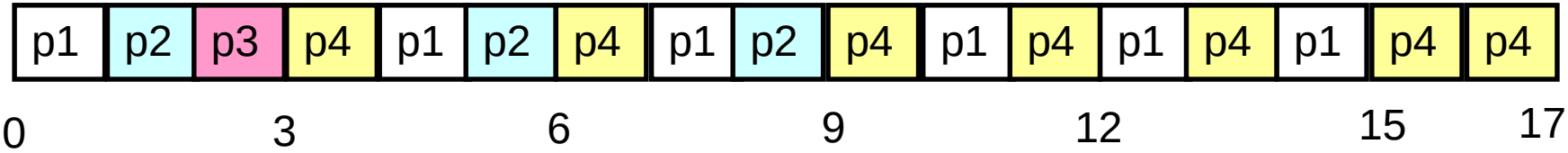
process	time
P_1	6
P_2	3
P_3	1
P_4	7

Turnaround time (waiting time + burst)

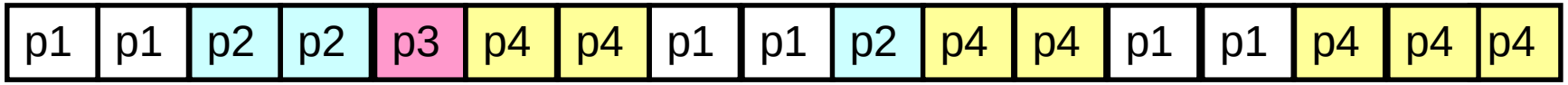
the total amount of time to execute a particular process

Turnaround Time Varies With The Time Quantum: Example

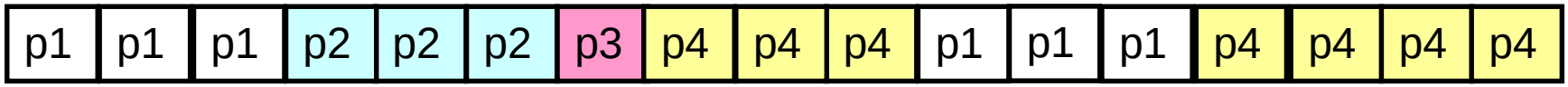
P1: 6; P2: 3
P3: 1; P4: 7



Q=1 Avg Turnaround Time = (15+9+3+17)/4 = 11



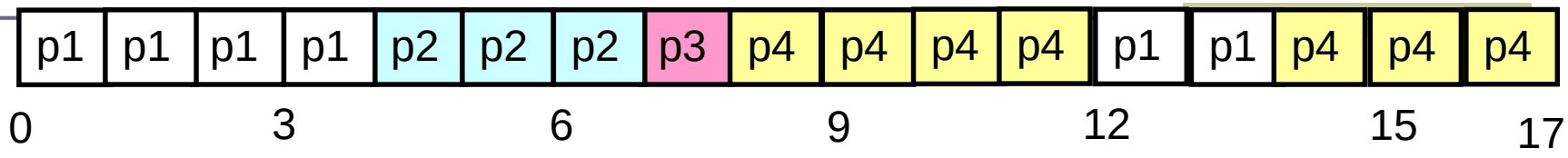
Q=2 Avg Turnaround Time = (14+10+5+17)/4 = 11.5



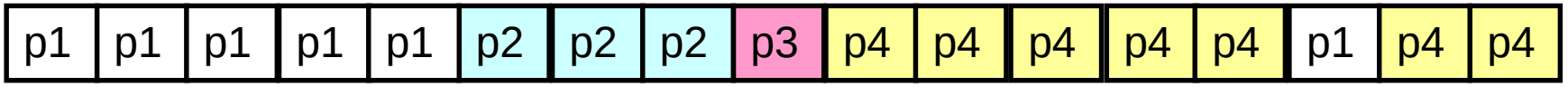
Q=3 Avg Turnaround Time = (13+6+7+17)/4 = 10.75

P1: 6; P2: 3
P3: 1; P4: 7

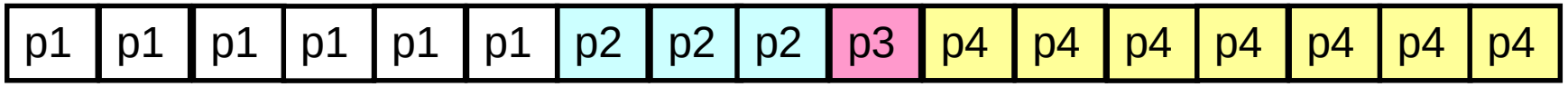
Q=4 Avg Turnaround Time = (14+7+8+17)/4 = 11.5



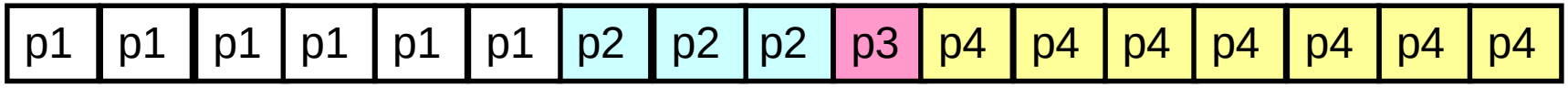
Q=5 Avg Turnaround Time = (15+8+9+17)/4 = 12.25



Q=6 Avg Turnaround Time = (6+9+10+17)/4 = 10.5



Q=7 Avg Turnaround Time = (6+9+10+17)/4 = 10.5

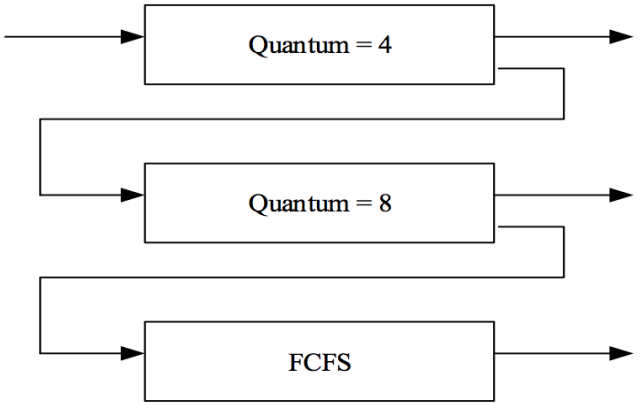


Multi-level Feedback Queue

Given the arrival time and CPU-burst of 5 processes shown in the following diagram:

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	15
P2	6	20
P3	10	2
P4	16	16
P5	32	6

Suppose the OS uses a 3-level feedback queue to schedule the above 5 processes. Round-Robin scheduling strategy is used for the queue with the highest priority and the queue with the second highest priority, but the time quantum used in these two queues is different. First-come-first serve scheduling strategy is used for the queue with the lowest priority. The scheduling is **preemptive**.

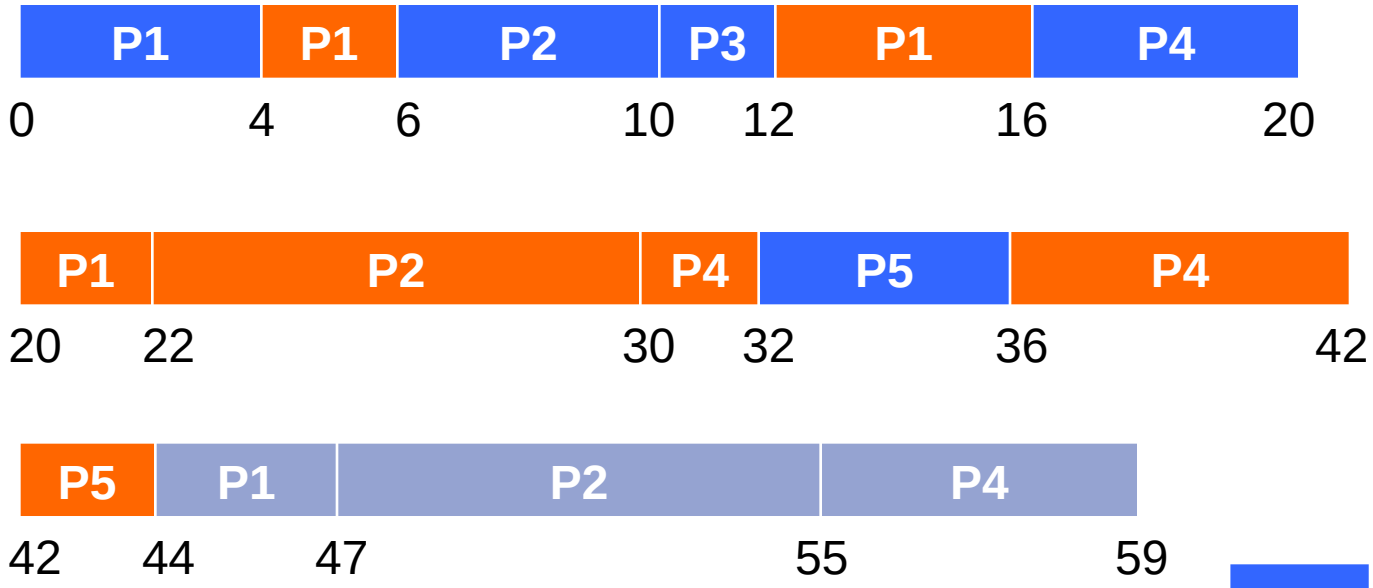


a) Construct a Gantt chart depicting the scheduling for the set of processes specified in the above diagram using this 3-level feedback queue.

b) Calculate the average waiting time for the schedule constructed in a).

Multi-level Feedback Queue

a) Gantt Chart:



b) Waiting time: (Finish time – Arrive time – Burst time)

P1: 47– 0 – 15 = 32;
P2: 55 – 6 – 20 = 29;
P3: 12 – 10 – 2 = 0;
P4: 59 – 16 – 16 = 27;
P5: 44 – 32 – 6 = 6;

Average waiting time:
18.8