KOÇ UNIVERSITY

CPU Scheduling Algorithms

Didem Unat

Lecture 6

COMP304 - Operating Systems (OS)

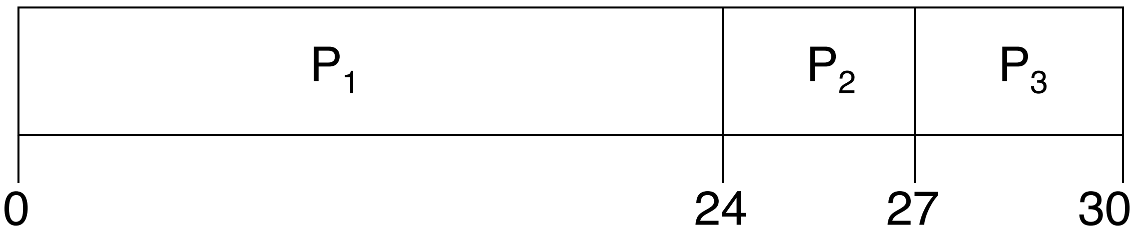
Terminology

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- 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)

1. First-Come, First Served (FCFS)

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
The Gantt Chart for the schedule is:



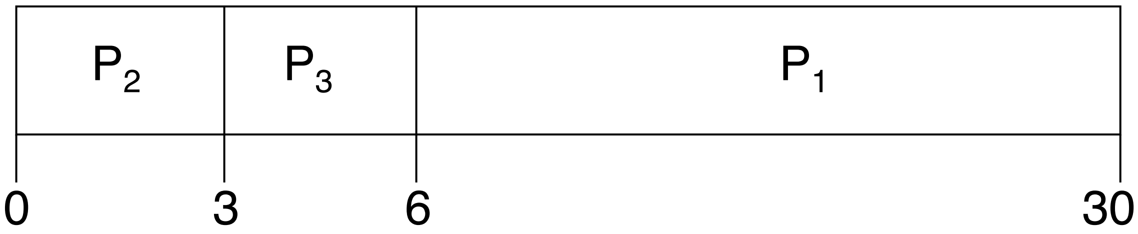
- Waiting times for: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

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.
- **Convoy effect:** short process behind long process

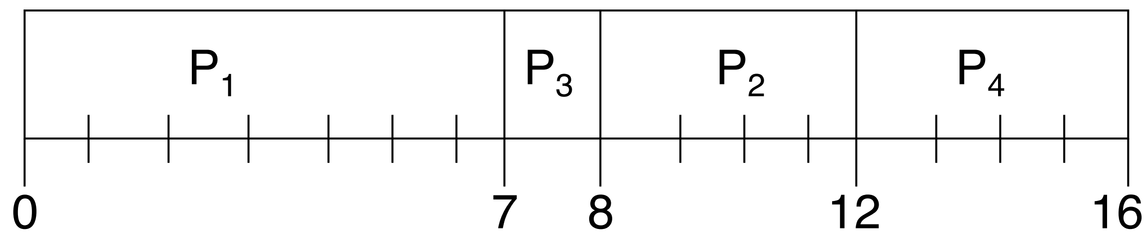
2. Shortest Job First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - **Non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - **preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This is known as the **Shortest-Remaining-Time-First (SRTF)**.
- SJF is **optimal** – gives minimum average waiting time for a given set of processes.

Non-preemptive (SJF) Scheduling

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

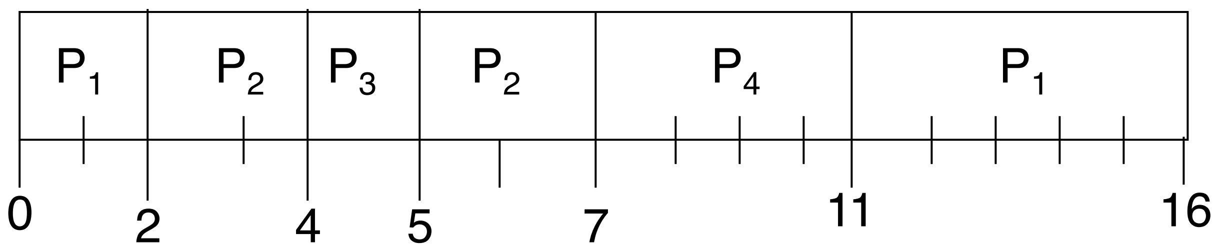


- Average waiting time ?
  $= (0 + 6 + 3 + 7)/4 = 4$

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time ?
  $= (9 + 1 + 0 + 2)/4 = 3$

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 $\equiv$ highest priority).
- Can be preemptive and nonpreemptive
- FCFS and SJF are special cases of priority scheduling
  - Why?
  - In FCFS priority is based on the arrival time
  - SJF is a priority scheduling where priority is the predicted next CPU burst time.

Priority Scheduling

Pi	$\tau(p_i)$	Priority
0	350	5
1	125	2
2	475	3
3	250	1
4	75	4

- <u>Nonpreemptive example</u>

0	250	375	850	925	1275
p_3	p_1	p_2	p_4	p_0	

$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_4)+\tau(p_2)+\tau(p_1)\ )+\tau(p_3) = 350+75+475+125+250$
$= 1275$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_3) = 125+250 = 375$

$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_1)+\tau(p_3) = 475+125+250 = 850$

$T_{TRnd}(p_3) = \tau(p_3) = 250$

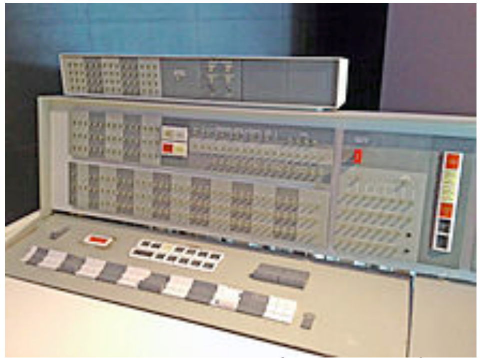
$T_{TRnd}(p_4) = \tau(p_4)+\ \tau(p_2)+\ \tau(p_1)+\tau(p_3) = 75+475+125+250 = 925$

$R(p_0) = 925$
$R(p_1) = 250$
$R(p_2) = 375$
$R(p_3) = 0$
$R(p_4) = 850$

Average response time $R_{avg} = (925+250+375+0+850)/5 = 2400/5 = 480$

Problem with Priority Scheduling

- Problem ≡ **Starvation** – low priority processes may never execute.
- Solution ≡ **Aging** – as time progresses increase the priority of the process.



IBM 7094 operator's console

- Rumor has it that,
  - when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.

Priority

- What if a high-priority process needs to access the data that is currently being held by a low- priority process?

- The high-priority process is blocked by the low-priority process. This is priority inversion.
- This can be solved with priority-inheritance protocol.
  - The low priority process accessing the data inherits the high priority until it is done with the resource.
  - When the low-priority process finishes, its priority reverts back to the original.

4. Round Robin (RR)

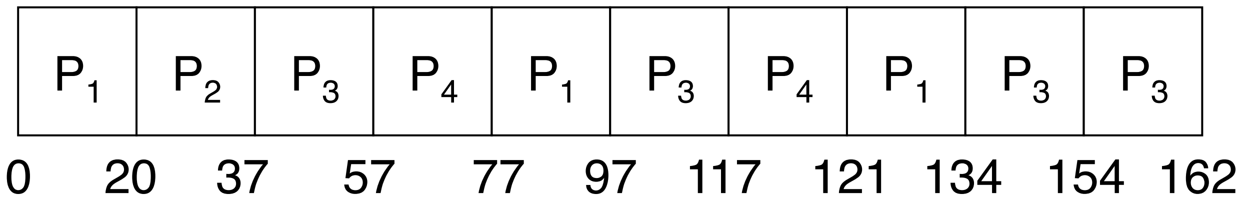
- Each process gets a small unit of CPU time (**time quantum**), usually 10-100 milliseconds.
- After this time has elapsed, the process is **preempted** and added to the end of the ready queue.

- 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*-1)*q* time units.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



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

Round Robin (Time Quantum=50)

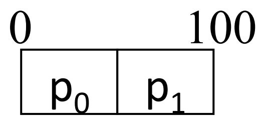
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0	50
p_0	

$R(p_0) = 0$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$R(p_0) = 0$
$R(p_1) = 50$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0 100

p_0	p_1	p_2

$R(p_0) = 0$
$R(p_1) = 50$
$R(p_2) = 100$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0		100		200		300

p_0	p_1	p_2	p_3	p_4	p_0

$R(p_0) = 0$
$R(p_1) = 50$
$R(p_2) = 100$
$R(p_3) = 150$
$R(p_4) = 200$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0 100 200 300 400 475 550

p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3	p_4	p_0	p_1

$T_{TRnd}(p_1) = 550$

$T_{TRnd}(p_4) = 475$

$R(p_0) = 0$
$R(p_1) = 50$
$R(p_2) = 100$
$R(p_3) = 150$
$R(p_4) = 200$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0 100 200 300 400 475 550 650

p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3

650 750 850 950

p_0	p_2	p_3	p_0	p_2	p_3

$T_{TRnd}(p_1) = 550$

$T_{TRnd}(p_3) = 950$

$T_{TRnd}(p_4) = 475$

$R(p_0) = 0$

$R(p_1) = 50$

$R(p_2) = 100$

$R(p_3) = 150$

$R(p_4) = 200$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0		100		200		300		400		475		550		650
p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3	

650		750		850		950		1050
p_0	p_2	p_3	p_0	p_2	p_3	p_0	p_2	p_0

$T_{TRnd}(p_0) = 1100$

$T_{TRnd}(p_1) = 550$

$T_{TRnd}(p_3) = 950$

$T_{TRnd}(p_4) = 475$

$R(p_0) = 0$

$R(p_1) = 50$

$R(p_2) = 100$

$R(p_3) = 150$

$R(p_4) = 200$

Round Robin (Time Quantum=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0 100 200 300 400 475 550 650

p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3

650 750 850 950 1050 1150 1250 1275

p_0	p_2	p_3	p_0	p_2	p_3	p_0	p_2	p_0	p_2	p_2	p_2	p_2

$T_{TRnd}(p_0) = 1100$

$T_{TRnd}(p_1) = 550$

$T_{TRnd}(p_2) = 1275$

$T_{TRnd}(p_3) = 950$

$T_{TRnd}(p_4) = 475$

$R(p_0) = 0$

$R(p_1) = 50$

$R(p_2) = 100$

$R(p_3) = 150$

$R(p_4) = 200$

Round Robin (Time Quantum=50)

- Equitable
- Most widely-used

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

0 100 200 300 400 475 550 650

p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3	p_4	p_0	p_1	p_2	p_3

650 750 850 950 1050 1150 1250 1275

p_0	p_2	p_3	p_0	p_2	p_3	p_0	p_2	p_0	p_2	p_2	p_2	p_2

$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$
$T_{TRnd}(p_2) = 1275$
$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

$R(p_0) = 0$
$R(p_1) = 50$
$R(p_2) = 100$
$R(p_3) = 150$
$R(p_4) = 200$

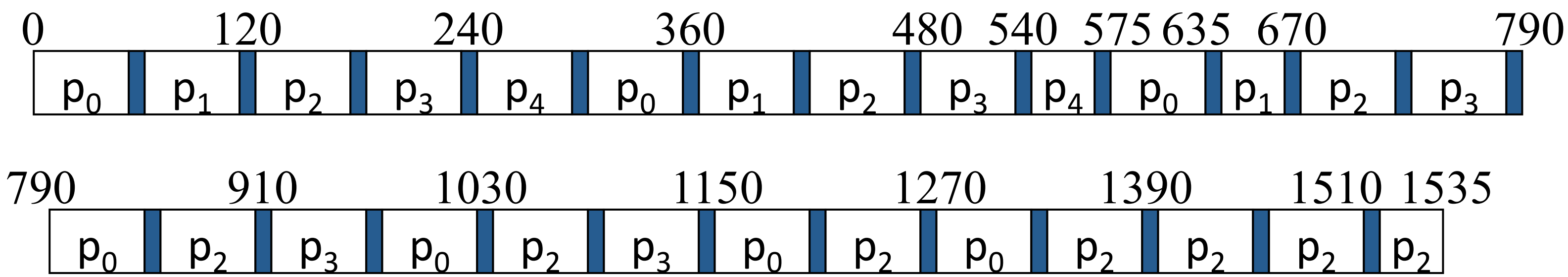
$T_{TRnd\_avg} = (1100+550+1275+950+475)/5 = 4350/5 = 870$

Average response (wait) time $R_{avg} = (0+50+100+150+200)/5 = 500/5 = 100$

RR (TQ=50, Overhead =10)

•Context Switch Overhead must be considered

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



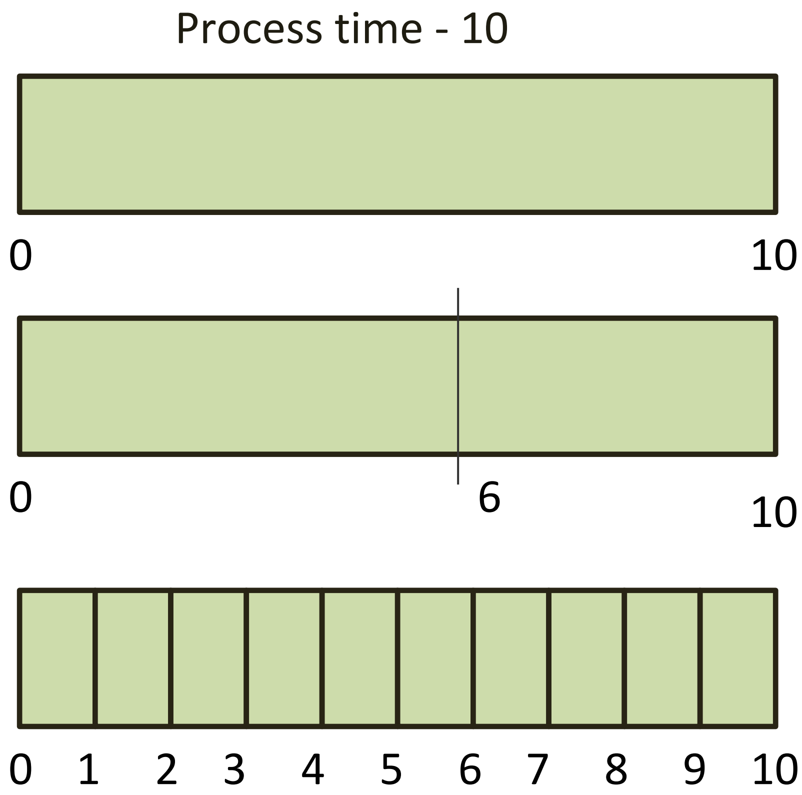
$T_{TRnd}(p_0) = 1320$
$T_{TRnd}(p_1) = 660$
$T_{TRnd}(p_2) = 1535$
$T_{TRnd}(p_3) = 1140$
$T_{TRnd}(p_4) = 565$

$R(p_0) = 0$
$R(p_1) = 60$
$R(p_2) = 120$
$R(p_3) = 180$
$R(p_4) = 240$

$T_{TRnd\_avg} = (1320+660+1535+1140+565)/5 = 5220/5 = 1044$

Average response (wait) time $R_{avg} = (0+60+120+180+240)/5 = 600/5 = 120$

Time Quantum and Context Switch Time



quantum	Context switches
12	0
6	1
1	9

- *q* large ⇒ FIFO
- *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.

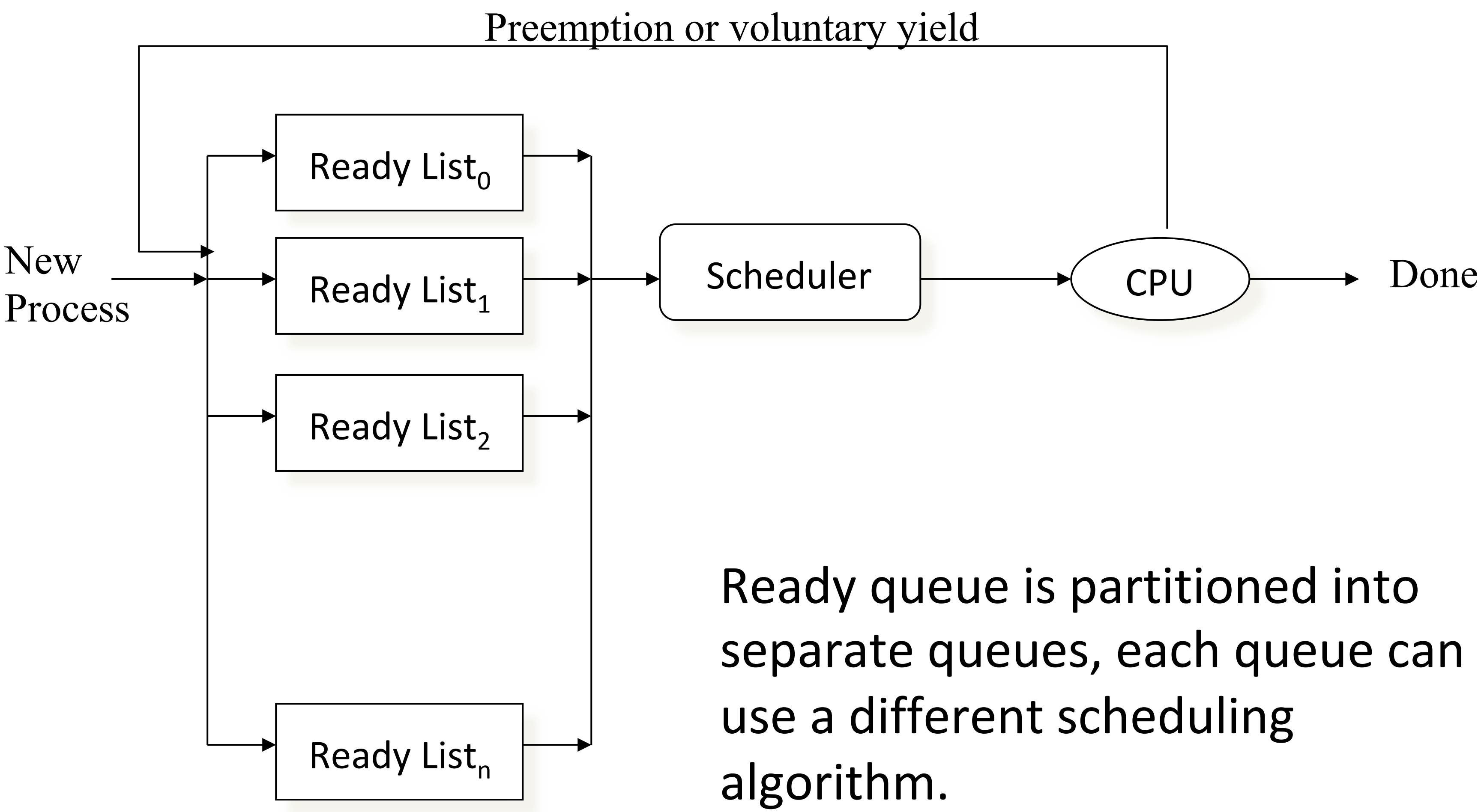
RR algorithm

- Consider a variant of the RR scheduling algorithm where the entries in the ready queue are pointers to the PCBs.
- What would be the effect of putting two pointers to the same process in the ready queue?
- How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

5. Multilevel Queue

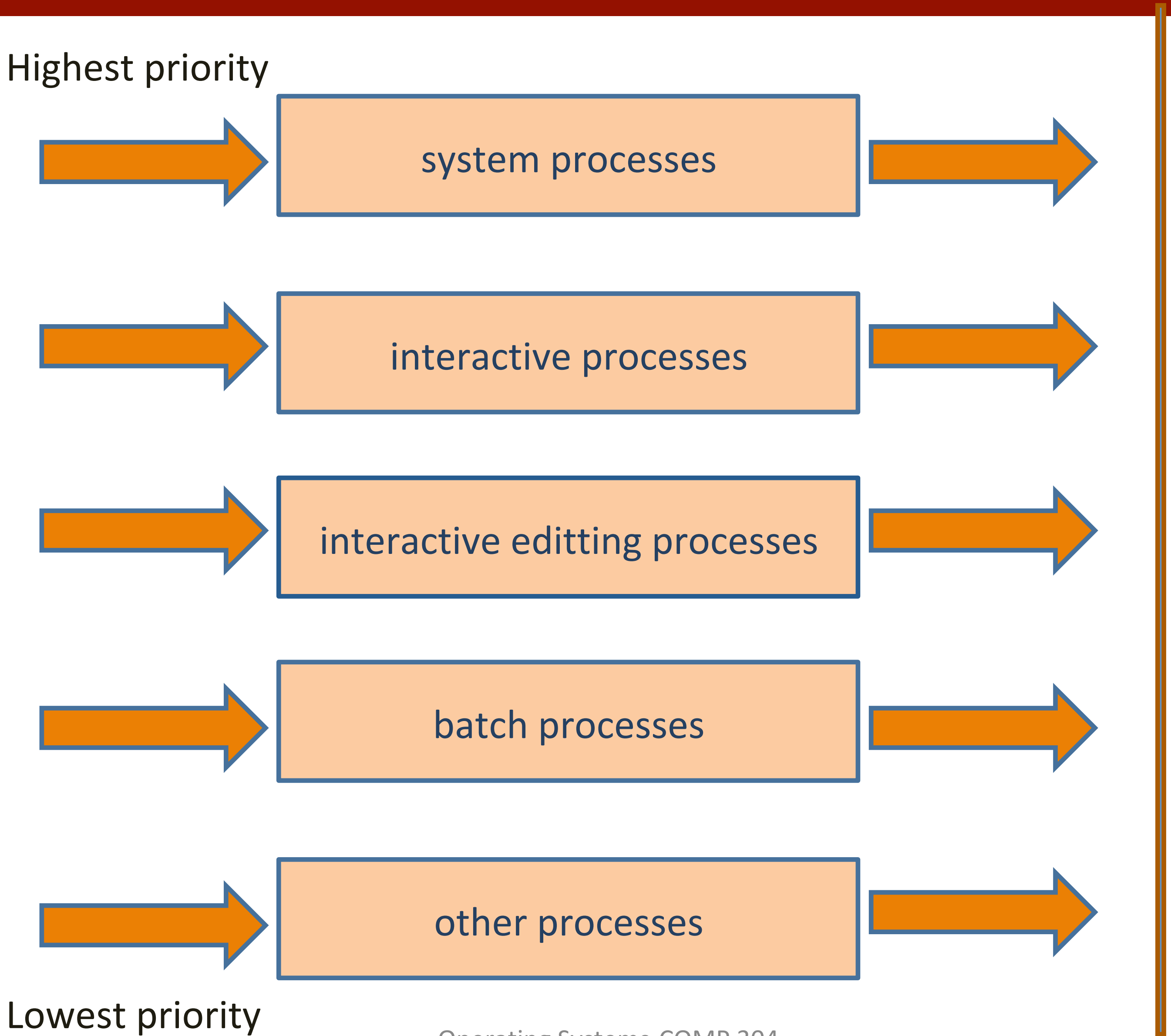
- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm,
  foreground – RR
  background – FCFS
- Scheduling must be done between the queues.
  - Fixed priority scheduling – (i.e., serve all from foreground then from background). Possibility of **starvation**.
  - 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

Multi-Level Queues



Ready queue is partitioned into separate queues, each queue can use a different scheduling algorithm.

Example- Multi-level Queue

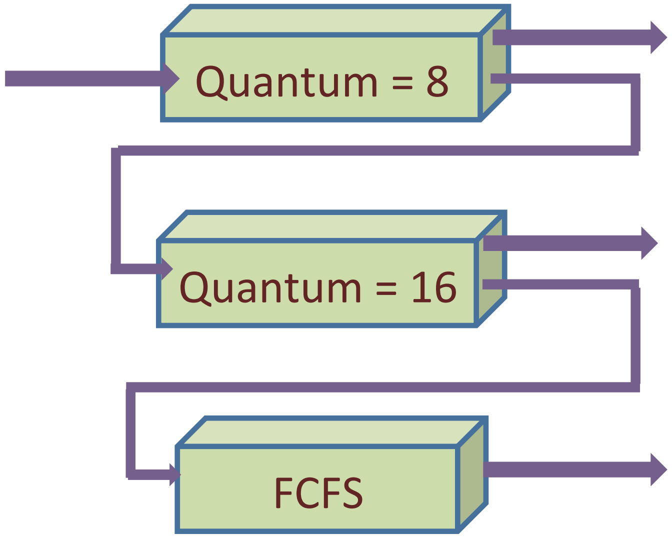


6. Multilevel Feedback Queue

- Multilevel queue with feedback scheduling is similar to multilevel queue; however, it allows processes to move between 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

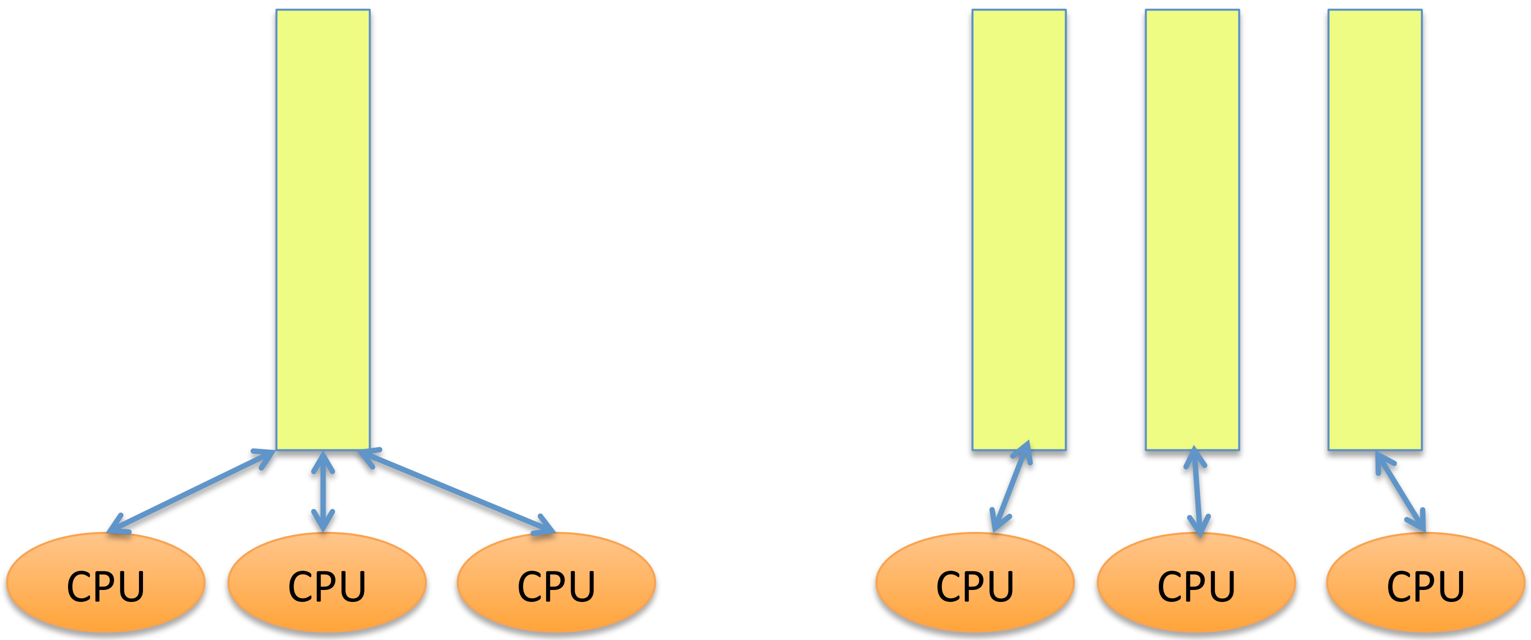
- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS non-preemption
- Example Scheduling
  - A new job enters queue Q_0. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 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.



Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
  - **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
  - **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity** (can be changed at a later time)
  - **hard affinity** (process doesn't move to another processor)
  - Variations including **processor sets**

Multicore Scheduling



- In SMP, each CPU has its own private ready queue for the processes waiting to be run on that CPU

Process Migration

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodically checks load on each processor, and if found any of them overloaded, pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pull waiting tasks from busy processor

Starvation

- Which of the following scheduling algorithms could result in starvation?
  - First-come, first serve
  - Shortest job first
  - Round robin
  - Priority

Reading

- Read Chapter 5 (Textbook)
- Read Chapter 4 (Linux Kernel Development)

- Acknowledgments
  - These slides are adapted from
    - Öznur Özkasap (Koç University)
    - Operating System and Concepts (9th edition) Wiley