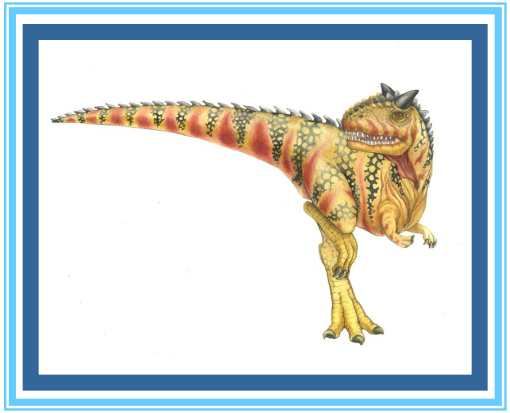
Spring 2022 COMP 3511
Review #4



Coverages

- Scheduling Examples
- Synchronization

Multilevel Feedback Queue (MLFQ) Scheduling

- Multilevel-feedback-queue or MLFQ scheduler defined by
  - The 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

- MLFQ is commonly used in many systems such as BSD Unix, Solaris, Window NT and subsequent Window operating systems
  - MLFQ has several distinctive advantages:
  - It does not require prior knowledge on CPU burst time
  - It handles interactive jobs well by delivering similar performance as that of SJF or SRTF
  - It is also “fair” by making progress on CPU-bound jobs

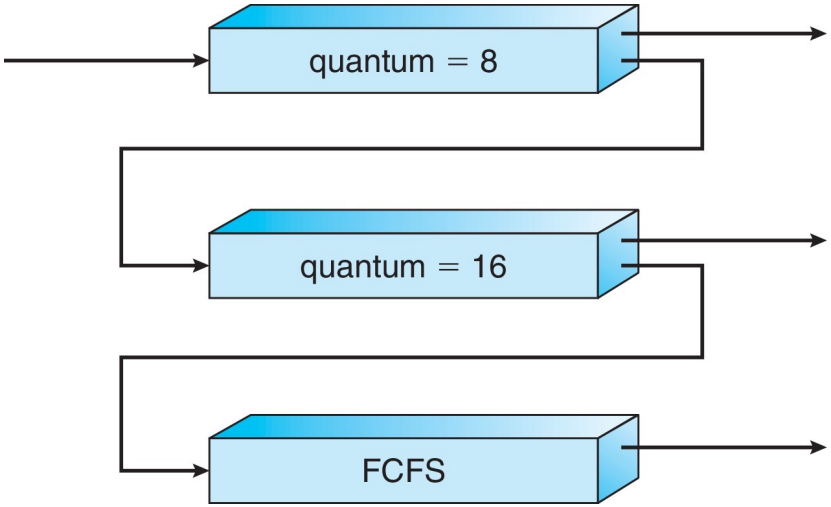
Example of Multilevel Feedback Queue

Three queues:

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

Scheduling

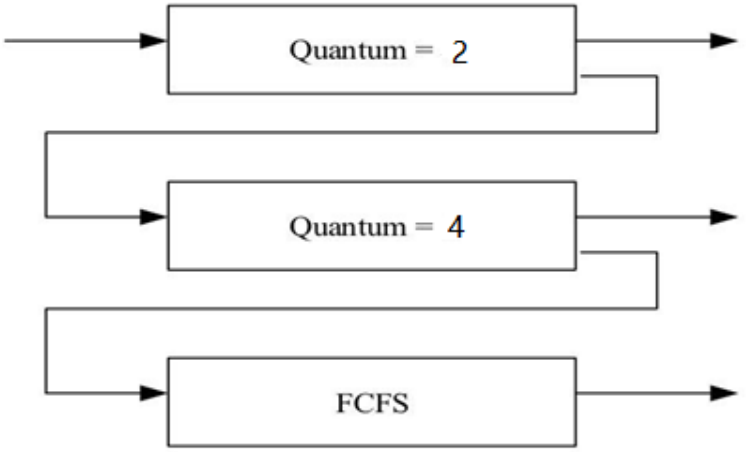
- A new job enters queue Q_0 which is served FCFS, also preempts jobs from Q1 or Q2 if currently running on CPU
  - When it gains CPU, job receives 8 ms
  - If it does not finish in 8 milliseconds, job is moved to the queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue Q_2
- If a job from Q_1 or Q_2 is preempted by a new job from Q_0, it joins the head of the queue Q_1 or Q_2 , respectively

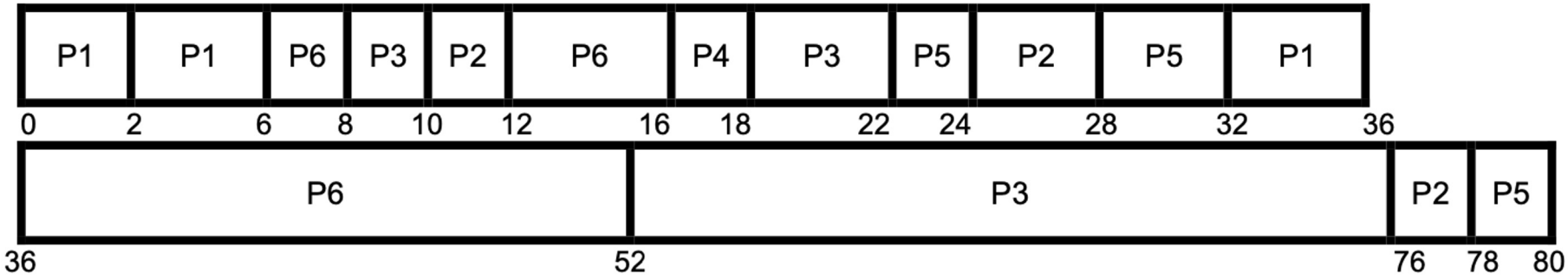


- This approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top

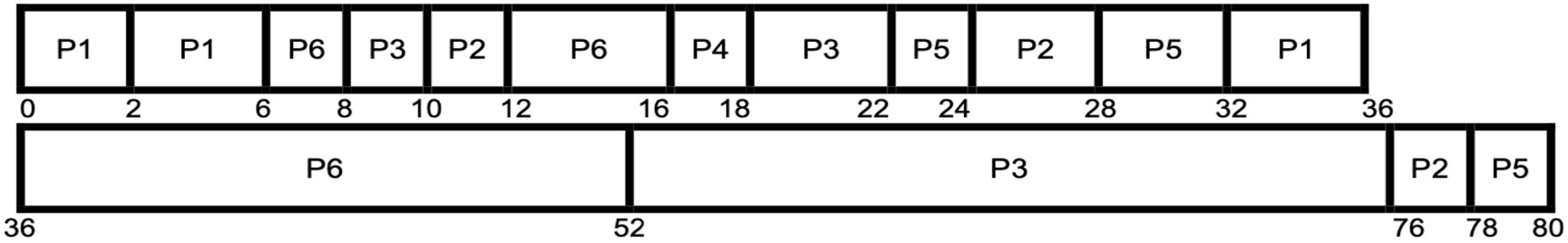
MLFQ Scheduling: Example

Process	Arrival Time	Burst Time
P1	0	10
P2	10	8
P3	8	30
P4	16	2
P5	22	8
P6	6	22





MLFQ Scheduling: Example



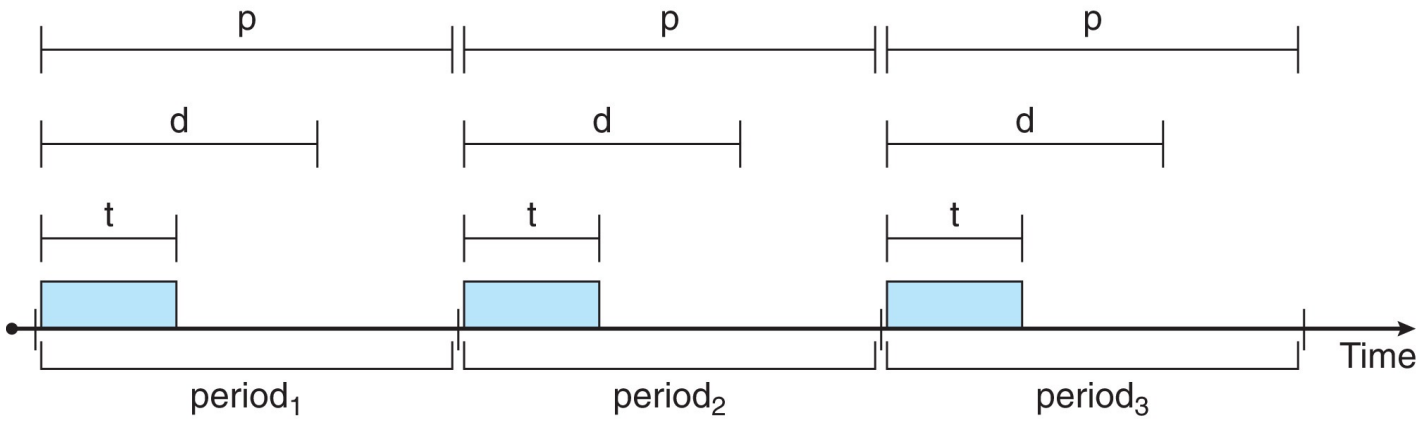
Averaging waiting time = turn-around time - CPU burst time

P1: 36-0-10 = 26
P2: 78-10-8 = 60
P3: 76-8-30 = 38
P4: 18-16-2 = 0
P5: 80-22-8 = 50
P6: 52-6-22 = 24

Average waiting time: (26+60+38+0+50+24)/6 =33

Priority-based Scheduling

- Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Processes have the characteristics: periodic ones require CPU at constant intervals (periods)
  - Has processing time *t*, deadline *d,* period *p* , in which $0 \le t \le d \le p$
  - The rate of a periodic task is $1/p$
  - A process may have to announce its deadline requirements to the scheduler. The scheduler decides whether to admit the process or not depending on whether it can guarantee that the process will complete on time (by its deadline)

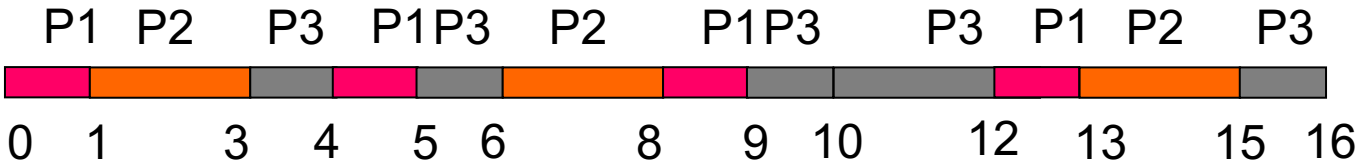


Rate Montonic Scheduling

- A static priority is assigned based on the inverse of its period
  - Shorter (longer) period = higher (lower) priority;
  - The rationale is to assign a higher priority to tasks requiring CPU more often

RM Scheduling: Example

Process	Processing Time	Deadline	Period
P1	1	2	4
P2	2	5	6
P3	3	7	9



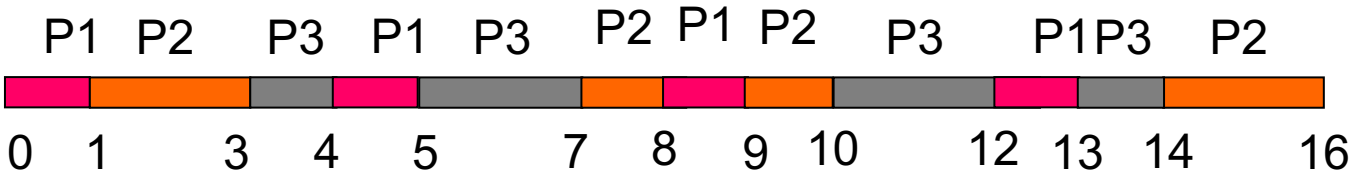
- Miss one deadline: P3 arrives at time 0 misses the deadline (7) by finishing at time 10

Earliest Deadline First Scheduling (EDF)

- Earliest-deadline-first (EDF) scheduling assigns priorities dynamically according to the deadline
  - the earlier (later) the deadline, the higher (lower) the priority

EDF Scheduling: Example

Process	Processing Time	Deadline	Period
P1	1	2	4
P2	2	5	6
P3	3	7	9



- Misses no deadlines

Critical Section Problem

- The race condition - the results depend on the timing execution of the code. With some bad luck, i.e., context switches that occur at untimely points during the execution, the result could be wrong. In fact, we may get an *indeterminate result* , where it is not known what the output will be and it is indeed likely to be different across runs
- A Critical Section is a piece of code that accesses a shared variable or a shared resource, thus race condition might occur and needs to be avoided
- A critical section must **NOT** be concurrently executed by more than one thread – mutual exclusion. This guarantees that if one thread is executing within a critical section, the others will be prevented from doing so
- The idea behind making a series atomic actions - “all or nothing”; it should either appear as if all of the actions be executed at once, or that none of them occurred. This prevents race condition
- However, it is not of advantage to design “atomic update” of complex data structure, a matrix, for instance. Instead, a general set of synchronization primitives are designed, some with hardware support, in combination the operating system assistance

Solution to Critical-Section Problem

1. **Mutual exclusion** - If process P_i is executing in its critical section, no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, the selection of a process that will enter the critical section next *cannot* be postponed *indefinitely* – selection of one process entering

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted – any waiting process

Peterson's Solution

- A classical software-based solution- this provides a good algorithmic description of solving the critical-section (CS) problem

- Two-process solution
- Assume that the **`load`** and **`store`** instructions are atomic; that is, cannot be interrupted

- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**

- The variable **`turn`** indicates whose turn (which process) it is to enter the critical section

- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready and requests to enter the CS

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
          critical section
    flag[i] = false;
          remainder section
} while (true);
```

P_0

```
do {
        flag[0] = true;
        turn = 1;
        while (flag[1] && turn == 1);
                  critical section
        flag[0] = false;
                  remainder section
        } while (true);
```

P_1

```
do {
        flag[1] = true;
        turn = 0;
        while (flag[0] && turn == 0);
                  critical section
        flag[1] = false;
                  remainder section
        } while (true);
```

Peterson's Solution – Proof

- **Mutual exclusion**: P_i enters its critical section only if either `flag[j] == false` or `turn ==i`. If both processes are trying to enter the critical section `flag[0]==flag[1] == true`, the value of `turn` can be either `0` or `1` but not both

- P_i can be prevented from entering its critical section only if it is stuck in the `while loop` with the condition `flag[j]==true` and `turn==j;`

- If P_j is not ready to enter the critical section, then `flag[j]==false` and P_i can enter its critical section.

- If P_j is inside the critical section, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i can to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus since P_i does not change the value of the variable `turn` while executing the `while` statement, P_i can will enter its critical section (**progress**) after at most one entry (**bounded waiting**)

Peterson's Solution – Discussion

- The solution works to protect "Critical Section" part of the codes, but
  - It is really complex even for a simple example – two processes
  - Codes are different for different threads, what if there are many
  - It involves "busy waiting", when one thread is inside "Critical Section", the other has to wait, wasting CPU cycles

- OS provides better solutions (high level primitives) beyond **`load`** and **`store`**


Q
A