

Process Scheduling (cont.) and introduction to process synchronisation

Mohammad S. Hasan

Staffordshire University, UK

Lecture Outline

◆ CPU Scheduling

- Priority scheduling
- Round Robin scheduling
- Multi-level queue scheduling

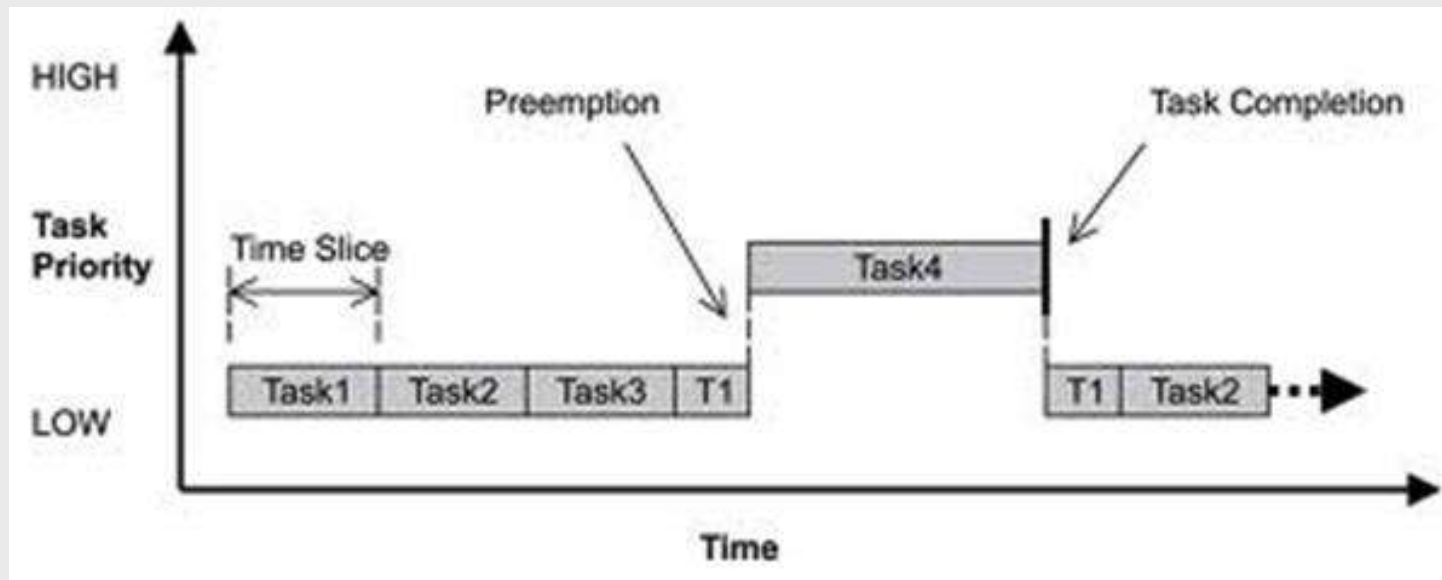
◆ Process synchronisation

- critical section problem & mutual exclusion
- hardware support for process synchronisation

Priority Scheduling

- ◆ A priority number (integer) is associated with each process
 - the CPU is allocated to the process with the highest priority
 - some schemes use smallest integer e.g. 0 as the highest priority
- ◆ Can be preemptive or nonpreemptive
- ◆ SJF is effectively priority scheduling where priority is the predicted next CPU burst time.

Preemptive Priority Scheduling



Priority Scheduling (cont.)

◆ Problem – Starvation

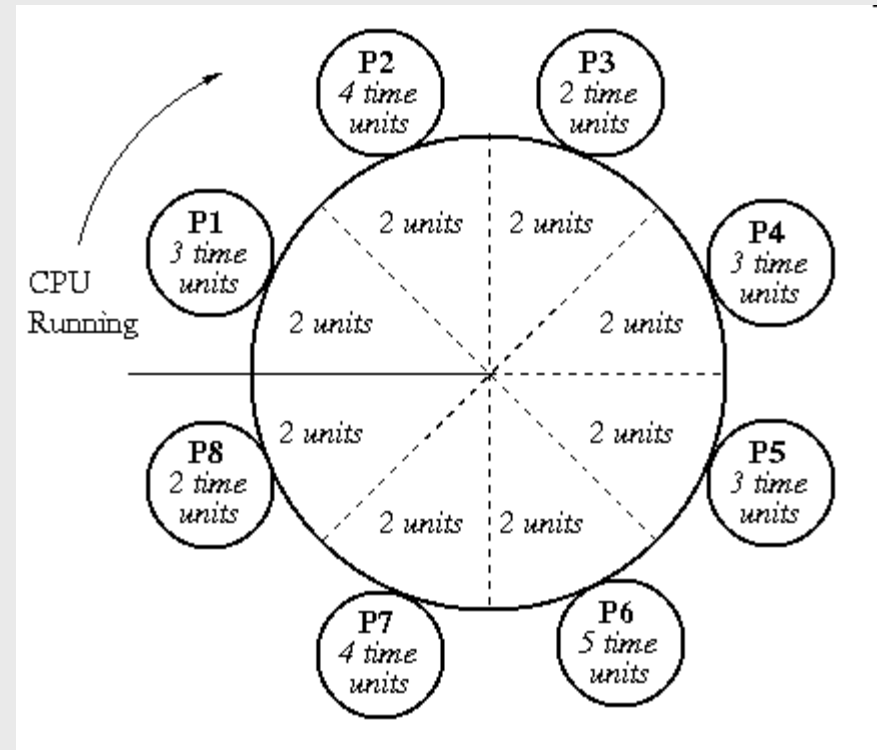
- Continuous arrival of high priority process
- low priority processes may never execute.

◆ Solution – Aging

- Increase the priority of processes with time

Round Robin (RR) Scheduling

- ◆ The CPU time is shared among the processes
- ◆ Each process gets a small unit of CPU time
 - Time quantum - usually 10-100 ms
 - After this time has elapsed, the process is preempted and added to the end of the ready queue.



Round Robin (RR) Scheduling (cont.)

Scenario

- Number of processes = n
- Time quantum = q
- Context switch time = s

Performance

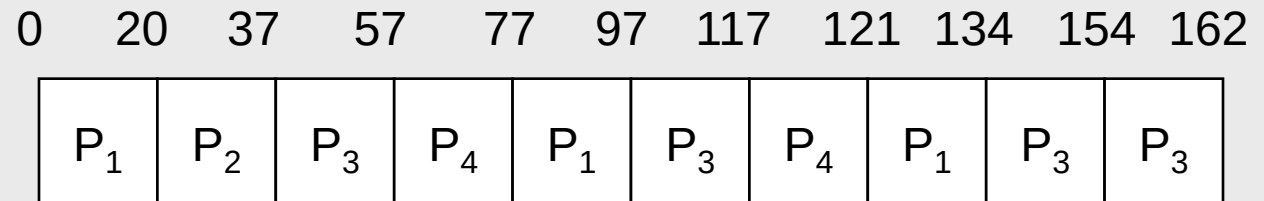
- No process waits more than $(n-1)q$ time units
- q very large $\rightarrow$ behaves like FCFS
- q small $\rightarrow$ context switch overhead becomes bigger
 - if $q = s$ then no work will get done - all context switching

RR with time quantum 20ms

Processes: P1, P2, P3 and P4

Process	BurstTime
P1	53
P2	17
P3	68
P4	24

The Gantt chart

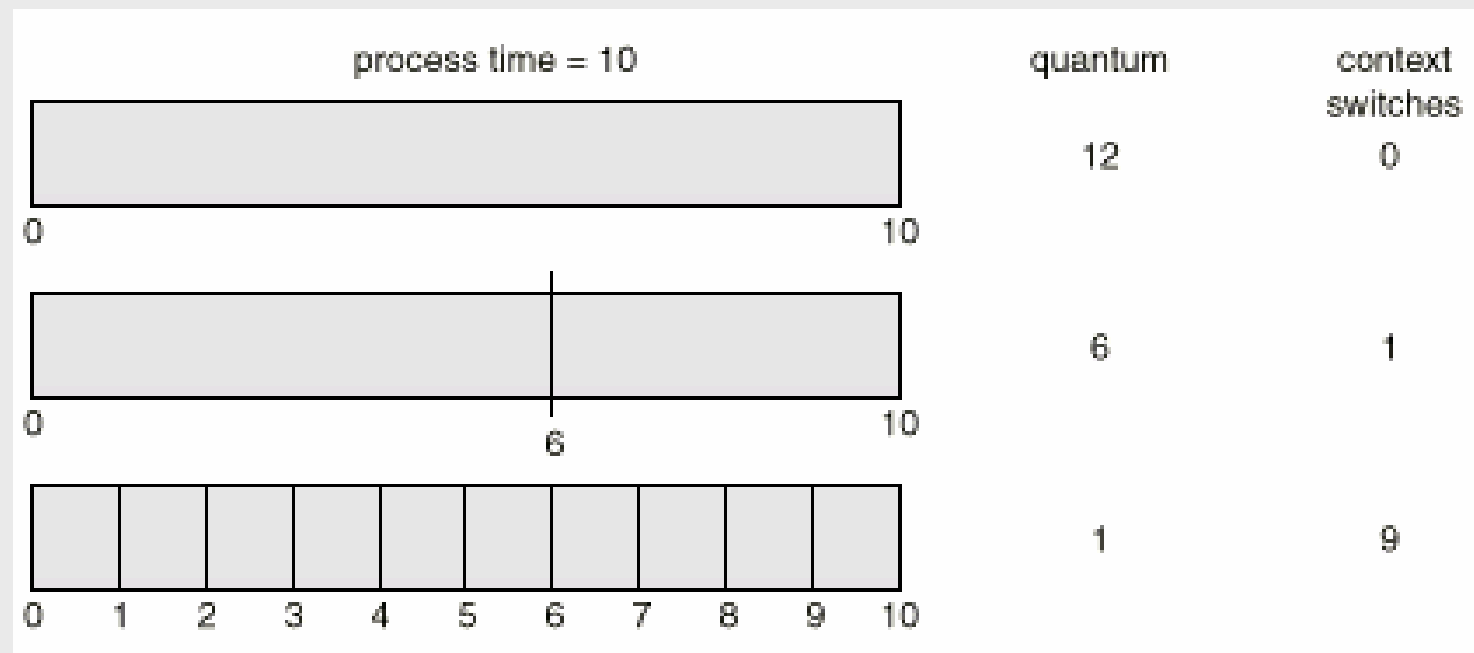


RR and SJF comparison

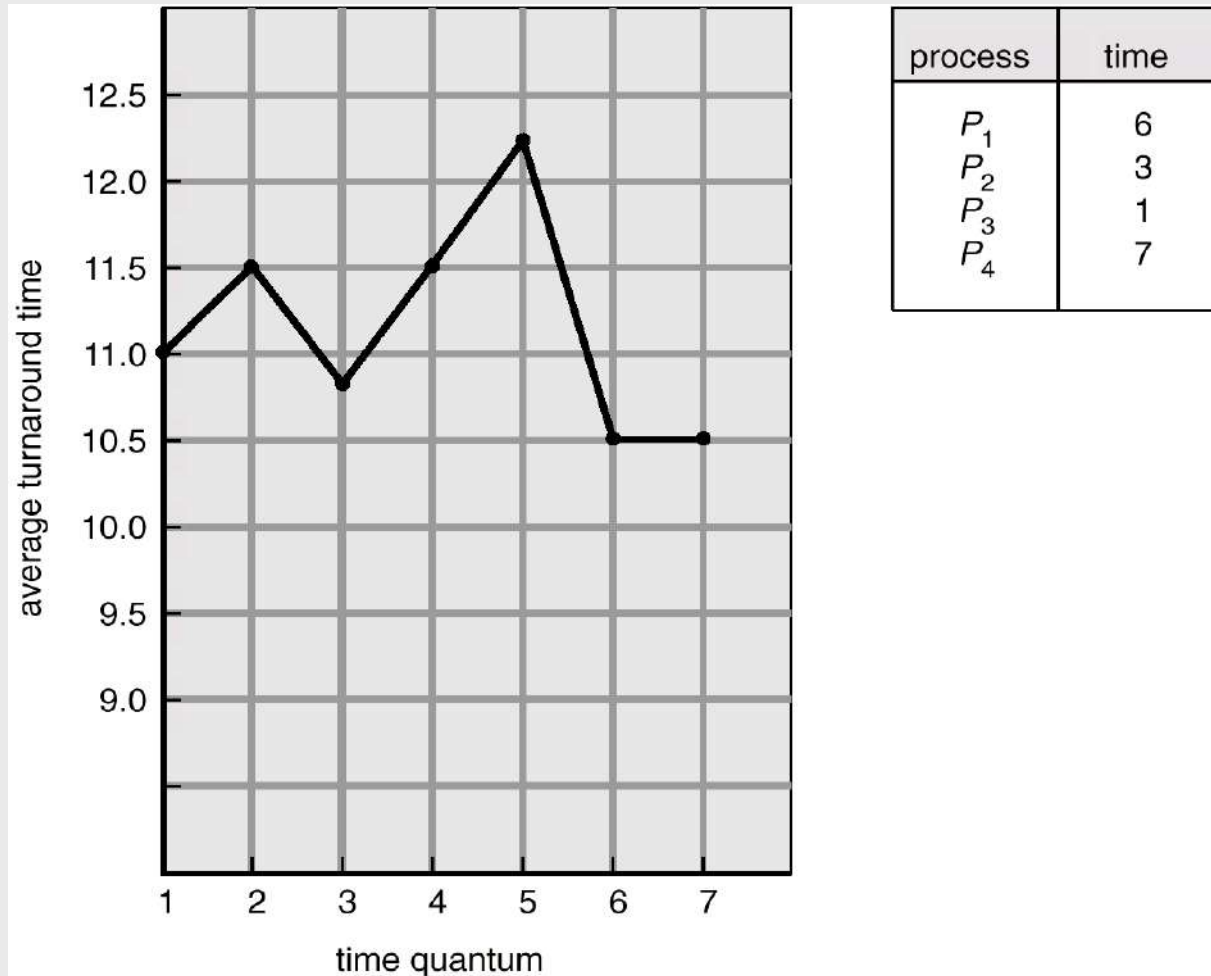
◆ Typically

- RR has higher average turnaround time than SJF
- RR has better response time than SJF

Time quantum and context switch

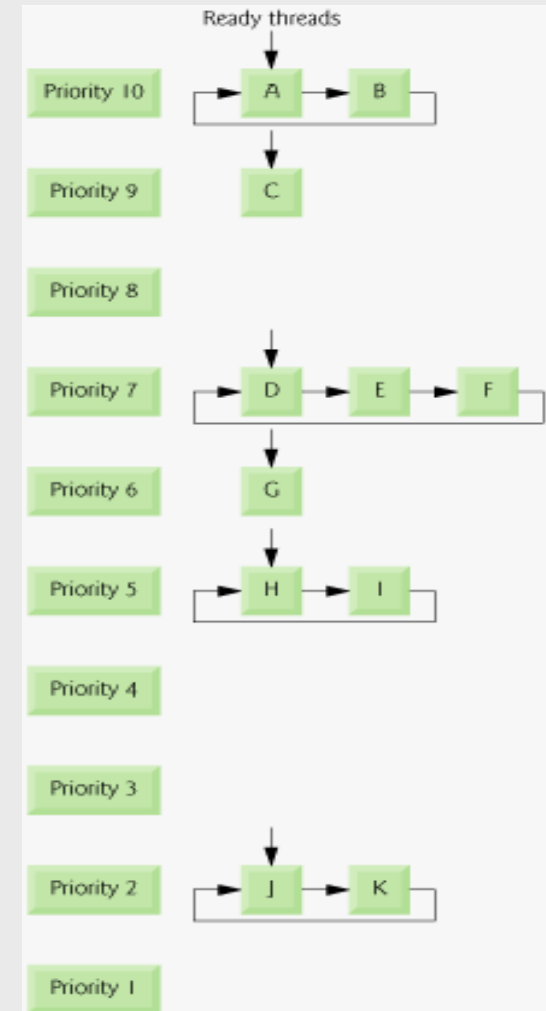


Time quantum and Turnaround Time



Multilevel Queue Scheduling

- ◆ Ready queue is partitioned into separate priority queues
 - foreground (interactive) and background (batch) queues
 - or system, interactive and batch queues
- ◆ Each queue may have its own scheduling algorithm
 - foreground – RR
 - background – FCFS



Multilevel Queue Scheduling (cont.)

◀ Scheduling between the queues

● Fixed priority scheduling

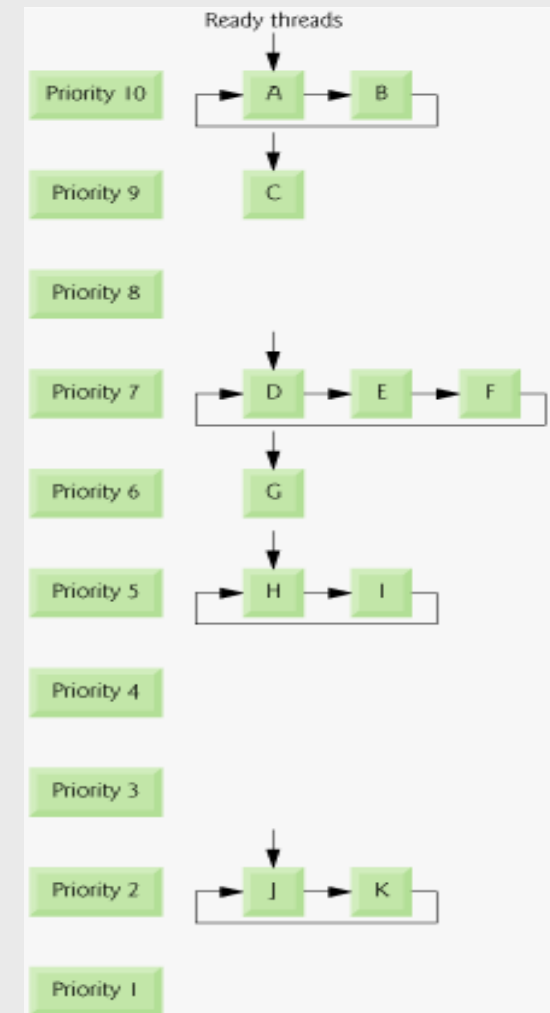
■ serve all from foreground then background.

■ But possibility of starvation.

● Time slice

■ each queue gets a certain amount of CPU time

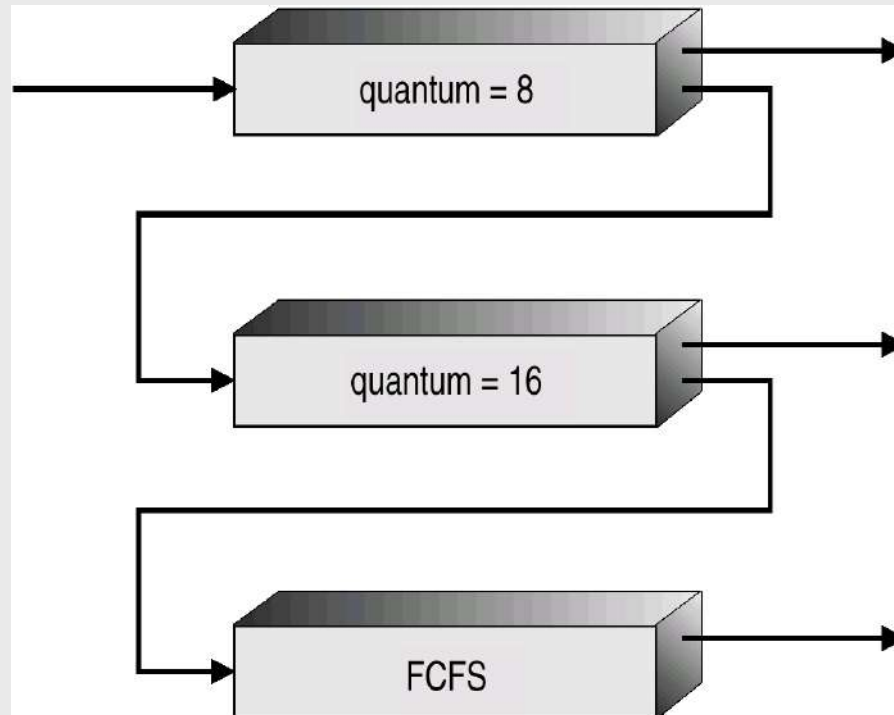
■ 80% to foreground in RR; 20% to background in FCFS



Multilevel Feedback Queue (cont.)

- ◆ **A process can move between the various 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 to determine when to upgrade a process
 - method to determine when to demote a process
 - method to determine which queue a process will enter initially

Multilevel Feedback Queue Example



Three queues

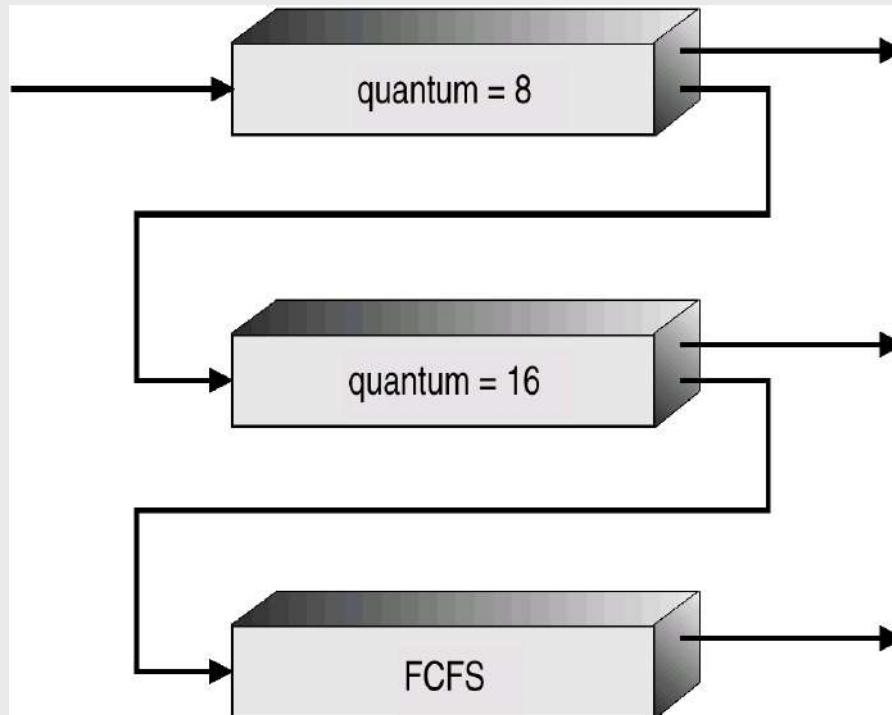
- Q0 – time quantum 8 ms and FCFS
- Q1 – time quantum 16 ms and FCFS
- Q2 – FCFS

Scheduling

- A new job enters queue Q0
- receives 8 ms time quantum
- If it NOT finish in 8 ms, job is moved to queue Q1.

Multilevel Feedback Queue Example (cont.)

◀ Scheduling (cont.)

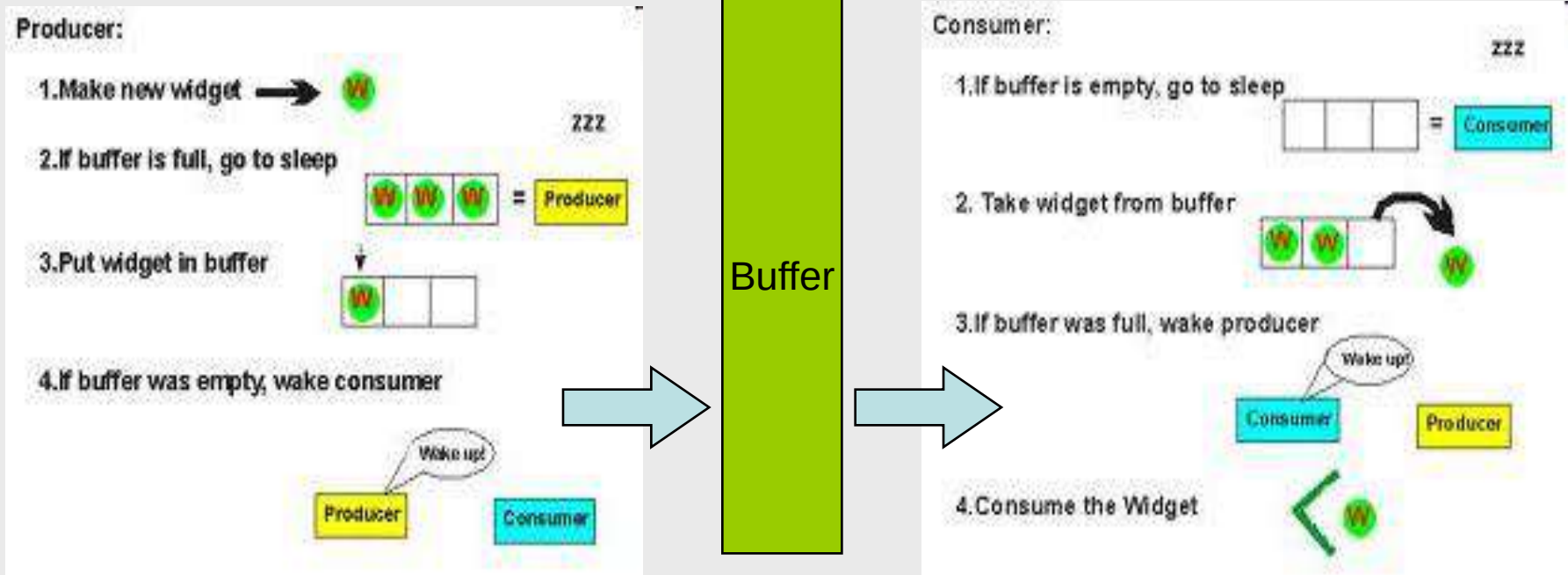


- At Q1 job receives 16 additional ms
- If it still does not complete, it is preempted and moved to queue Q2.

Process Synchronisation

- ◆ **Concurrent access to shared data may result in data inconsistency.**
- ◆ **Ensuring the orderly execution of cooperating processes.**
- ◆ **Example**
 - **The producer-consumer problem**
 - **Previously discussed**

The Producer-Consumer Synchronisation Problem



The Producer-Consumer Synchronisation Problem (cont.)

- ◆ **Counter shared variable**
 - Stores the number of items in the buffer
 - Initially set to 0
- ◆ **Producer**
 - Produce an item
 - $\text{Counter} = \text{counter} + 1$
- ◆ **Consumer**
 - Consume an item
 - $\text{Counter} = \text{counter} - 1$



The Producer-Consumer Synchronisation Problem (cont.)

➤ Producer

- Counter = counter + 1

- MOV AX counter

- ADD AX 1

- MOV counter AX

➤ Consumer

- Counter = counter -1

- MOV BX counter

- SUB BX 1

- MOV counter BX

➤ Two items in the buffer

- Counter = 2

➤ Produce 1 and consume 1

- MOV AX counter (AX=2)

- MOV BX counter (BX=2)

- ADD AX 1 (AX=3)

- SUB BX 1 (BX=1)

- MOV counter AX (counter=3)

- MOV counter BX (counter=1)



The Producer-Consumer Synchronisation Problem (cont.)

- ◆ **Parallel execution of process on shared data might cause invalid result**

- **The outcome of the execution (value of data) depends on order in which accesses occur**

- ◆ **Solution**

- **Ensure that only one process can access shared data at a time**
- **Counter = counter +1 or counter = counter -1 must be executed atomically**

The Critical Section Problem

- ◆ Assume N processes have a shared data X
- ◆ Critical Section
 - The code segment where each process access the shared data
- ◆ Problem
 - ensure that when one process is executing in its critical section
 - no other process is allowed to execute in its critical section
 - NOT allowed to access the shared data X

The Critical Section Problem (cont.)

❖ Execution must be mutually exclusive

- requires synchronisation.

❖ General algorithm

● Repeat

■ entry section // controls entry to critical section

■ critical section

■ exit section // manages exit from critical section

■ remainder section // other code

● until false;

Solution to Critical Section Problem

◆ 3 conditions need to be met to provide a solution

● Mutual Exclusion

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

● Progress

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, the selection of the next process can NOT be postponed indefinitely

Solution to Critical Section Problem (cont.)

◆ 3 conditions need to be met to provide a solution

● 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

Hardware Support for Synchronisation

- ◆ Many processors provide atomic instruction to test and set the value of a variable.
- ◆ Test-and-set instruction
 - Can be defined as a method / function
 - `boolean method TestAndSet (boolean word_X) {
 return word_X; // return value of word
 word_X = true; // set value to true`
 - `}`

Reference

◆ **Operating System Concepts - Chapter 5 & 6**