BITS Pilani
Pilani Campus

Computer Organization and Software Systems

Session 11

Prof. C R Sarma
WILP.BITS-Pilani

Last Session

Contact Hour	List of Topic Title	Text/Ref Book/external resource
19-20	• Scheduling Algorithms FCFS, SJF, SRTF, Priority and RR	T2

Today's Session

Contact Hour	List of Topic Title	Text/Ref Book/external resource
21-22	• Scheduling algorithms • Process Coordination	T2

Multilevel Queuing

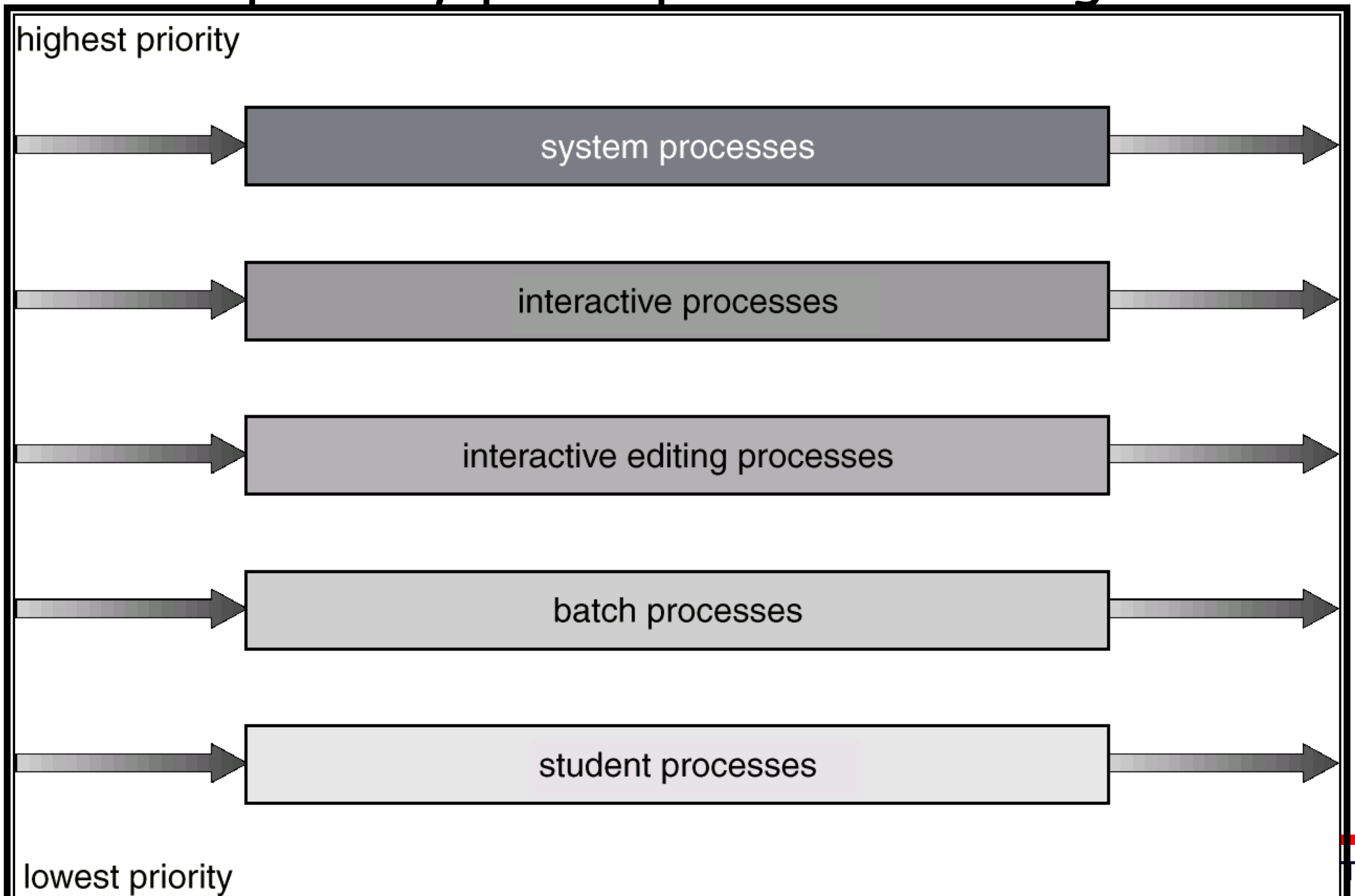
- Process classification based on response-time requirements and scheduling needs
  - Foreground (interactive) processes
  - background (batch) processes
- Foreground processes may have priority (externally defined) over background processes
- Multilevel queue scheduling algorithm
  - Partition the ready queue into several separate queues

Contd…

- Process assignment to queue based on
  - Memory size, priority of process or process type
- Each queue has its own scheduling algorithm
- Example: foreground and background processes.
  - The foreground queue scheduled by an RR algorithm
  - background queue is scheduled by an FCFS algorithm.

Multilevel Queue Scheduling

- There must be scheduling among the queues
  - fixed priority preemptive scheduling



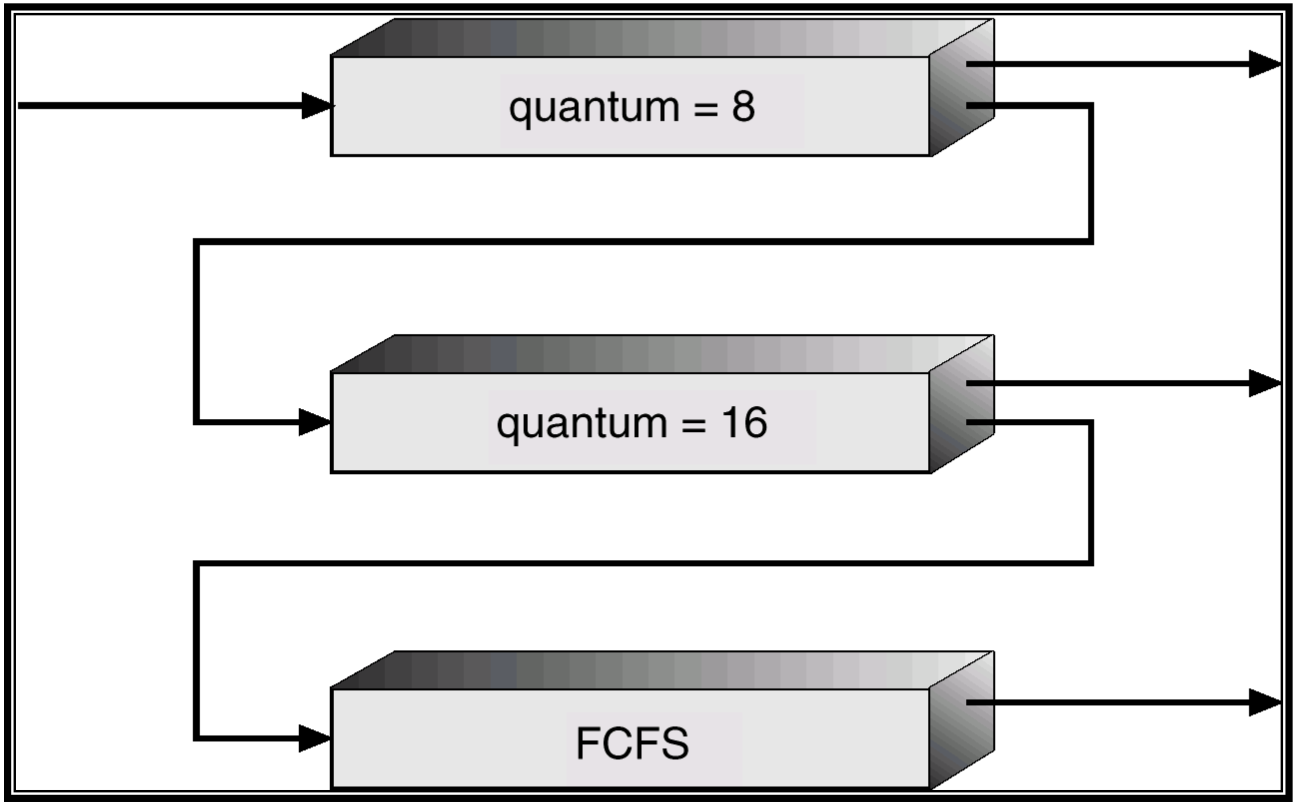
Multilevel Feedback Queue

- Disadvantage of Multilevel queue : Inflexible
- 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 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
- 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 receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Multi-level Q
(Q1 &Q2 use RR, P(Q1)>P(Q2), tq1=4,tq2=3)

Process	AT	BT	Priority	CT	TAT	WT
P1	0	10	2			
P2	3	7	1 (H)			
P3	4	6	2 (L)			
P4	12	5	1			
P5	18	8	1			

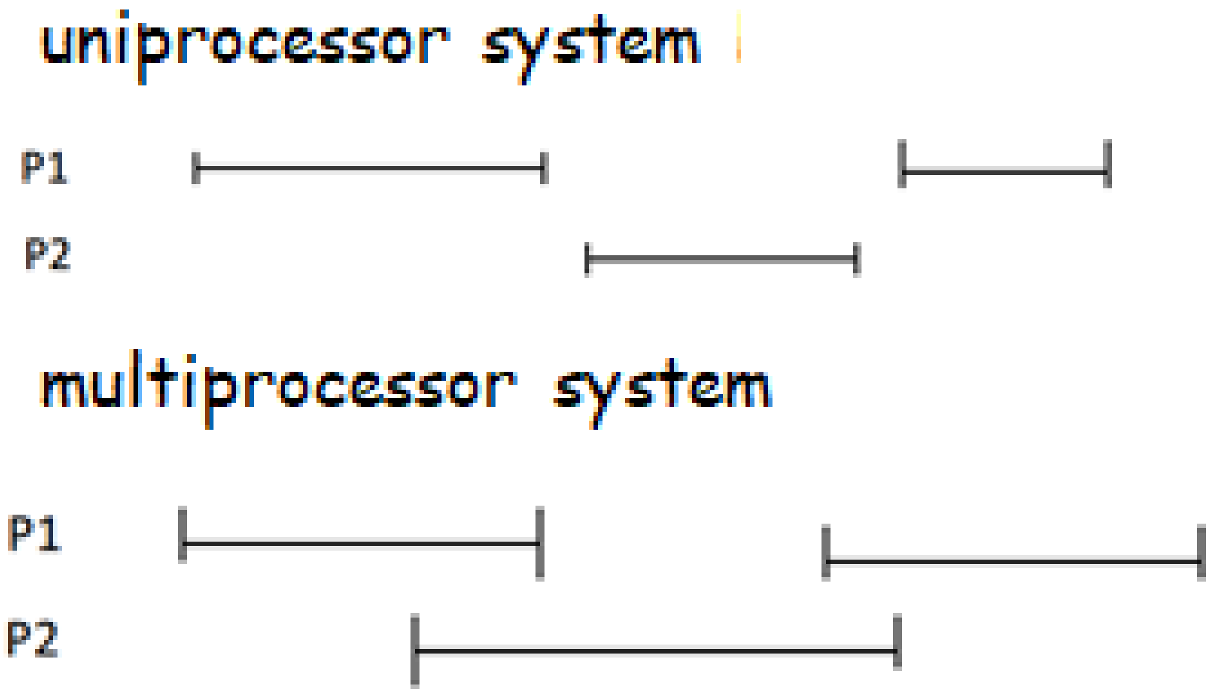
Q1									
Q2									

BITS Pilani
Pilani Campus

Process Synchronization

Introduction

- How do we maximize CPU utilization / improve efficiency?
  - Multiprogramming Vs Multiprocessing
- How to achieve concurrent operation in
  - Uniprocessor System Vs Multiprocessor system



Contd…

- Interleaving and overlapping improves processing efficiency, but may produce unpredictable results if not controlled properly
- Example:

```
Procedure echo;
Var out,in:Character;
Begin
    input (in, keyboard);
    out:=in;
    output(out,Display)
End
```

Example (Contd...)

Process P1

1. input (in, keyboard);
2. -----------------
3. -----------------
4. -----------------
5. out:=in;
6. output(out, Display)

Process P2

1. --------
2. input (in, keyboard);
3. out:=in;
4. output(out, Display)

Contd…

- Reasons unpredictable results :
  - Finite resources
  - Relative speed of execution of processes can not be predicted
  - Sharing of resources (non shareable) among processes
- Processes compete for resources
  - Deadlock
  - Mutual exclusion
  - Starvation
- Cooperating & competing processes can cause problem when executed concurrently → **Synchronization**

Process Synchronization

- Process **Synchronization** means sharing system resources by processes in a such a way that, concurrent access to shared data is handled thereby minimizing the chance of inconsistent data
- A **Critical Section** is a code segment that accesses shared variables or resources and has to be executed as an atomic action.
- Successful use of concurrency requires –
  - Ability to define critical section and
  - Enforce mutual exclusion

Process structure

Process P_i

do {

:

ENTRY SECTION

critical section

Exit SECTION

remainder section

} while (TRUE);

Main requirements

Three requirements

- A **mutual exclusion** :When one process is using a shared modifiable data, the other processes will be excluded from doing the same thing
- **Progress** : when no process in critical section, any process that makes a request is allowed to enter critical section without any delay
- **Bounded Waiting** : Processes requesting critical section should not be delayed indefinitely (no deadlock, starvation)

❖ *No assumption should be made about relative execution speed of processes or number of processes*

❖ *A process remains inside critical section for a finite amount of time*

Approach to handle Mutual Exclusion

- Software Approach (User is responsible for enforcing Mutual exclusion)
- Hardware Support – Disabling of Interrupt and using Special Instructions
- OS support – Semaphore

Critical Section (Solution1)

```
Procedure echo;
Var out,in:Character;
Begin
   input (in, keyboard);
   out:=in;
   output(out,Display)
End
```

Process 0	Process 1
----	----
----	----
while turn == 1 do {nothing }	while turn == 0 do {nothing }
<Critical Section>	<Critical Section>
turn = 1	turn = 0

"I finished with it, now you have it" → Dekker's Algorithm

- Drawback 1: processes must strictly alternate – Pace of execution of one process is determined by pace of execution of other processes
- Drawback 2: if one processes fails other process is permanently blocked

Critical Section (Solution 2) Peterson's solution

- Good algorithmic description of solving the problem
- Two process solution
- The two processes share two variables:
  - **int turn;**
  - **Boolean flag[2]**
- The variable **turn** indicates whose turn 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!

Algorithm

Process P_i

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);

                critical section

        flag[i] = false;
        -------
} while (true);
```

Algorithm

t	P0	P1
1	flag[0] = true; turn = 1;	
2		flag[1] = true; turn = 0;
3	while (flag[1] && turn = = 1); critical section	
4		while (flag[0] && turn = = 0); critical section
5	flag[o] = false	

6		while (flag[0] && turn = = 0);
		critical section
		flag[1] = false;

Synchronization – Hardware Approach

– Protecting critical regions via locks

```
do {
      acquire lock
            critical section
      release lock
            remainder section
} while (TRUE);
```

Synchronization – Hardware Approach…

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Example : TestAndSet instruction, Swap instruction

Hardware approach – TestAndSet

- **TestAndSet** instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.

- **Definition of the TestAndSet () instruction:**

```
boolean TestAndSet(boolean *target) {
        boolean rv = *target;
        *target = true;
        return rv;
}
```

Mutual Exclusion with TestAndSet instruction

Shared data: Boolean lock = FALSE;

Process P_0

```
do {
 while (TestAndSet( &lock))
            ; //wait
  critical section
  lock = FALSE;
  remainder section
} while (TRUE);
```

Process P_1

```
do {
 while (TestAndSet( &lock))
            ; //wait
  critical section
  lock = FALSE;
  remainder section
} while (TRUE);
```

```
boolean TestAndSet(boolean *lock)
{ boolean rv = *lock; *lock =TRUE;  return rv; }
```

Semaphores

- is a variable which is treated in a special way
- allow processes to make use of critical section in exclusion to other processes
- process wanting to access critical section locks semaphore and releases lock on exit.
- is a synchronization tool

Contd…

- Basic properties of semaphore:
  - semaphore *S* – integer variable with non negative values
  - can only be accessed via two operations

    wait (S):

    while $S \leq 0$ do *no-op;*
    S--;

    signal (S):

    S++;
  - Semaphore operation is atomic and indivisible
    - wait and signal operations are carried out without interruption

Types of semaphore

- Binary semaphore : can have two values 0 and 1
  - On some systems, binary semaphores are known as mutex locks, as they are locks that provide *mutual exclusion.*
- *Counting Semaphore* : integer value can range over an unrestricted domain.

Critical Section of *n* Processes

Shared data:

semaphore S ; //initially *S* = 1

Process *Pi*:

```
do {
        remainder section
    wait(S);
        critical section
    signal(S);
        remainder section
} while (1);
```

wait **(*S*):**

while *S* ≤ 0 do *no-op*;

S--;

signal **(*S*):**

S++;

Process synchronization

- Semaphore can be used to solve various synchronization problems
- Example : two concurrent processes : P1 (with S1) and P2 (with S2)
  - requirement : **s2 should be executed only after s1 is executed**
  - semaphore variable : synch initialized to zero.

P1:

S1;

signal (synch);

P2:

wait (synch);

S2;

wait (synch):

while *synch* ≤ 0 do *no-op*;

synch--;

signal (synch):

synch++;

Contd…

- Main disadvantage : Busy Waiting → waiting wastes CPU cycles
- semaphore is also called a **spinlock** because the process "spins" while waiting for the lock.
- Solution: on finding zero semaphore value (binary semaphore), the process can block itself instead of busy waiting
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.

Contd…

- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue.
- We need to modify the definition of the wait () and signal () semaphore operations.

Consumer - Producer Problem

- Also known as bounded buffer problem
- Two processes - consumer and producer
- Consumer and Producer processes share a common, fixed size buffer
- Producer process : generates data and puts it in the buffer
- Consumer process: consumes data from the buffer
- *Problem statement : When a producer is placing an item in the buffer, then at the same time consumer should not consume any item.*

mutex = 1

Full = 0 ➔ Initially, all slots are empty.

Empty = n ➔All slots are empty

Producer

```
do{
//produce an item
wait(empty);
wait(mutex);
        //place in buffer
signal(mutex);
signal(full);

}while(true)
```

Consumer

```
do{
wait(full);
wait(mutex);
  // remove item from buffer
signal(mutex);
signal(empty);
// consumes item

}while(true)
```

That’s All for CS11

See Ya in

CS# 12

Multi-level Q

(Q1 &Q2 use RR, P(Q1)>P(Q2), tq1=4,tq2=3)

Process	AT	BT	Priority	CT	TAT	WT	RT
P1	0	~~10~~ ~~7~~ ~~4~~ 1	2	36	36	26	0
P2	3	~~7~~ ~~3~~ 0	1 (H)	10	7	0	0
P3	4	~~6~~ ~~3~~	2 (L)	35	31	25	27
P4	12	~~5~~ ~~1~~ 0	1	19	6	1	1
P5	18	~~8~~ ~~4~~ 0	1	26	8	0	0

TQ=4 Q1	~~P2~~	~~P2~~	~~P4~~	~~P4~~	~~P5~~	~~P5~~			
TQ=3 Q2	~~P1~~	~~P1~~	~~P3~~	~~P1~~	P3	P1			

P1	P2	P2	P1	P4	P4	P5	P5	P3	P1	P3	P1
0	3	7	10	13	17	18	21	26	29	32	35

36