Process synchronization

Prof J P Misra
BITS, Pilani

Till now we have seen

- Types of computing systems
- What is OS ?
- What is Process ?
- Process scheduling (Single CPU)
  - Maximize CPU utilization
  - Minimize response time/Average wait time
  - Increase throughput
  - Fairness to all processes

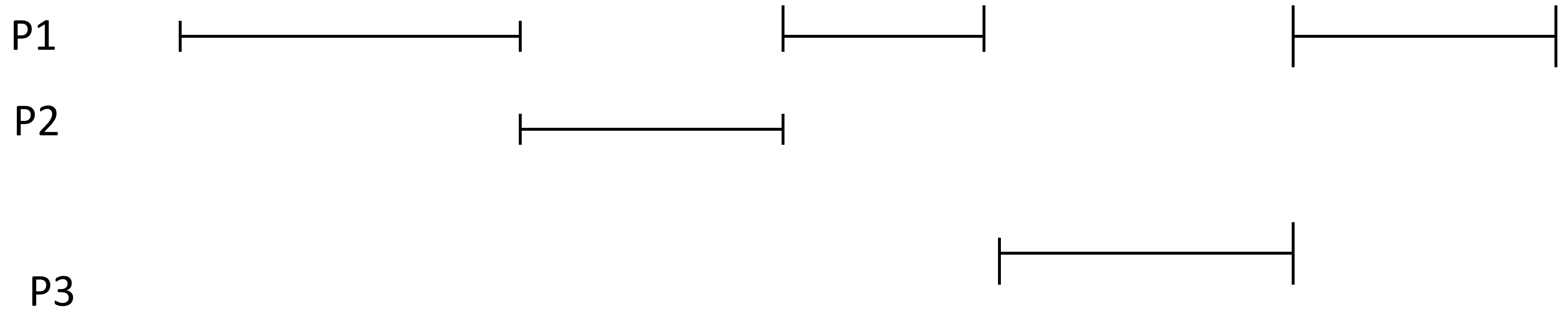
Today's Agenda

- How do we maximize CPU utilization / improve efficiency?
  - Multiprogramming
  - Multiprocessing

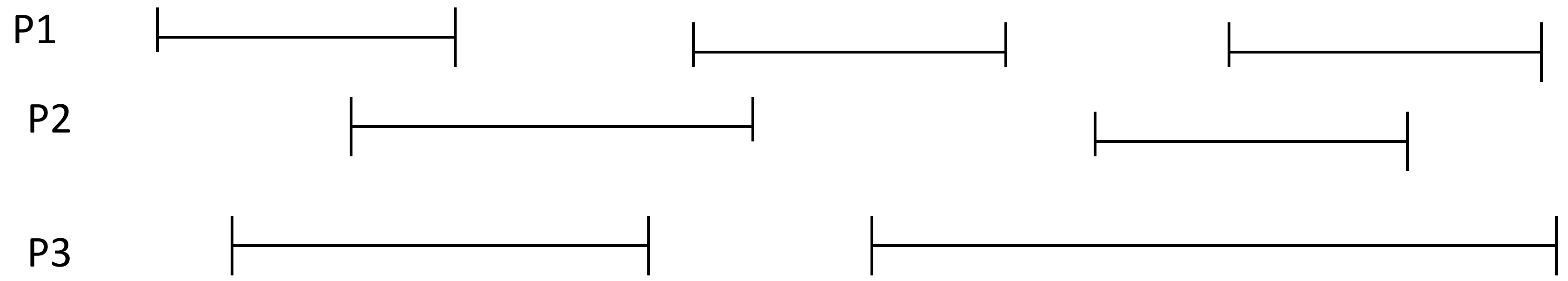
Concurrent Operation

- In uniprocessor system Processes are interleaved in time
- In multiprocessor system processes are overlapped
- Interleaving and overlapping improves processing efficiency
- Interleaved /overlapped process may produce unpredictable results if not controlled properly

Interleaving



Overlapping



Why problem arises with Interleaving & Overlapping

- Finite resources
- Relative speed of execution of processes can not be predicted
- Sharing of resources(non shareable) among processes
  - Sharing of memory is required for Inter process communication

Example

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

Example

<u>Process P1</u>

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

<u>Process P2</u>

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

Operating System Concern

- The result of a process must be independent of the speed at which the execution is carried out

We need to understand: How processes interact?

- The problem is faced as system has several resources which are to be shared among the processes
- In system we have processes which are
  - unaware of existence of other processes and such processes compete for resources
  - Processes indirectly aware of each other such processes exhibit cooperation
  - Processes directly aware of each other, show cooperation

We need to understand: How processes interact?

- When Processes compete for resources, the problem of deadlock , mutual exclusion and starvation may occur
- When processes are directly aware of other processes, the problem of deadlock and starvation might exist

Concurrency Control Problem

- Mutual exclusion
- Starvation
- Deadlock

Cooperating & competing processes can cause problem when executed concurrently

How Process cooperation is achieved ?

- Shared Memory
  - Mutual exclusion
- Message passing

Solution to critical section problem

- Successful use of concurrency requires
  - Ability to define critical section
  - Enforce mutual exclusion
- Any Solution to critical section problem must satisfy
  - Mutual exclusion
  - Progress : when no process in critical section, any process that makes a request is allowed to enter critical section without any delay
  - 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
  - Special Instructions
- OS support
  - Semaphore
  - Monitor

Software Approach (Solution 1)
Var turn :0..1;

```
Process 0                           Process 1
------                              ------
------                              ------
While turn < >0 do { nothing};      While turn < >1 do { nothing};

< Critical Section code >;          < Critical Section code >;
Turn := 1;                          Turn := 0;
```

Solution 1

- This solution guarantees mutual exclusion
- 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

This problem arises due to fact that it stores name of the process that may enter critical section rather than the process state

Second Approach

Var flag:Array[0..1] of Boolean; initially flag is initialized to false

```
While flag[1] do {nop};
Flag[0]:= true;
< critical section>;
Flag[0]:= false;
```

- --
- --

```
While flag[0] do {nop};
Flag[1]:= true;
< critical section>;
Flag[1]:= false;
```

- --
- --

P1	P2
While flag[1] do {nop};	
	While flag[0] do {nop};
Flag[0]:= true;	
	Flag[1]:= true;
	< critical section>;
< critical section>;	
	Flag[1]:= false;
Flag[0]:= false;	• --
• --	• --
• --	

Second approach

- If one process fails outside its critical section including the flag setting code then the other process is not blocked
- It does not satisfy the Mutual exclusion
- It is not independent of relative speed of process execution
- Mutual exclusion is not satisfied as processes can change their state after it is checked by other process

Third approach

```
Flag[0]:= true;
While flag[1] do {nop};
< critical section>;
Flag[0]:= false;
```

- --
- --

```
Flag[1]:= true;
While flag[0] do {nop};
< critical section>;
Flag[1]:= false;
```

- --
- --

- This approach satisfy mutual exclusion
- This approach may lead to dead lock

What is wrong with this implementation ?

- A process sets its state without knowing the state of other. Dead lock occurs because each process can insist on its right to enter critical section
- There is no opportunity to back off from this situation (discourteous processes)

Fourth approach

```
Flag[0]:= true;
While flag[1] do
Begin
Flag [0]:=false;
<delay for short time>
Flag[0]:=true
End;
< critical section>;
Flag[0]:= false;
• --
• --
```

```
Flag[1]:= true;
While flag[0] do
Begin
Flag [1]:=false;
<delay for short time>
Flag[1]:=true
End;
< critical section>;
Flag[1]:= false;
• --
• --
```

Fifth Approach

```
Var flag:Array[0..1] of Boolean;
Turn: 0..1;
Procedure p0
Begin
    Repeat
            flag[0]:= true;
            while flag[1] do if turn = 1then
                                    begin
                                    flag[0]:=false;
                                    while turn=1 do {nothing};
                                    flag[0]:=true
                                    end;
< critical section >
Turn:=1;
Flag[0]:=false;
Forever
End;
```

Thank You

Process synchronization

Prof J P Misra
BITS, Pilani

We have seen

- Process types (competing , non competing)
- Resource types (sharable , Non sharable)
- Process synchronization Issues & requirement
  - Mutual exclusion
  - Starvation
  - Deadlock
- Software approach to handle process synchronization

Mutual Exclusion (Hardware Approach)

- Process interleaving is mainly due to interrupts / system calls in the system.
- Because of interrupts or system call, a running processes gets suspended and another process starts running which results into interleaved code execution.
- Interleaving of processes is main cause due to which mutual exclusion is required

How do we prevent Interleaving ?

- Interleaving can be prevented by disabling the interrupt in the uniprocessor system

Mutual exclusion by disabling interrupt

```
Repeat
 < disable interrupt >;
 < Critical Section >;
 <enable Interrupt >;
 < remainder section >
Forever.
```

Problem with Hardware Approach

- Interrupt Disabling can degrade the system performance as it will loose ability to handle critical events which occur in the system
- This approach is not suited for multiprocessor system

Special Machine Instruction

- We find entry into critical section requires eligibility check and this check consists of several operation eg.

Flag[0]= true;

While flag[1] do {nothing}

- We need instruction which can execute these operations in atomic manner.

Test & Set Instruction

```
Function testset (var i:integer):boolean;
Begin
   if i=0 then
        begin
                i:=1;
                testset:=true
        end
Else testset:=false
End.
```

Mutual exclusion using testset

```
Const n;
Var lock; (Initialized to 0 in the beginning)
Procedure p(i:integer);
Begin
    Repeat
          Repeat { nothing } untill testset(lock);
          < critical section >
          lock:=0;
          <remainder section >
    forever
end;
```

Properties of Machine Instruction Approach.

- It is applicable to any number of processes
- It can be used to support multiple critical section. Each critical section can be defined by its own variable
- Busy waiting is employed
- Starvation is possible
- Dead lock is possible

Semaphore (OS Approach)

- We can view semaphore as integer variable on which three operations are defined
  - Can be initialized to a non negative value
  - Decrement operation (*wait*)if the value becomes negative then process executing wait is blocked
  - Increment operation (*Signal*) if the value is not positive then a process blocked by wait operation is unblocked
- Other than these three operations, there is no way to inspect or manipulate semaphore

Mutual Exclusion

```
Var s:semaphore; initialized to 1
 Begin
 wait (s);
        < critical Section>;
 signal (s)
End.
```

Var s,r: semaphore; s is initialized to 1 & r is initialized to zero

Process P0

- -
- ---
- ------

```
Begin
 wait (s);
          < critical Section>;
 signal (r)
End.
```

Process P1

- -
- ---
- ------

```
Begin
 wait (r);
          < critical Section>;
 signal (s)
End.
```

Semaphore

- Semaphore can be viewed as integer variable on which three operations are defined
  - Initialization (non negative value)
  - Wait Operation: it decrements the variable and if the value is negative then the executing process is blocked
  - Signal Operation: increments the variable, if the value is not positive then a process blocked by wait operation is unblocked

Two Types of Semaphores

- **Counting Semaphore**
  - Integer value can range over an unrestricted domain.
- **Binary Semaphore**
  - Integer value can range only between 0 and 1; can be simpler to implement.

Wait Operation

```
Type semaphore =record
    count: integer;
    queue: List of processes
End;
Var s: semaphore;
Wait(s)
Begin
        s.count:=s.count-1;
        if s.count <0 then
                begin
                        place the process in s.queue;
                        block this process
                end;
end;
```

Signal Operation

```
Signal(s):
    s.count:=s.count+1
    if s.count<= 0 then
            Begin
                remove a process from s.queue;
                place this process on ready list
            end;
```

Note:

- S.count>= 0, s.count is number of processes that can execute wait(s) without blocking
- S.count<=0, the magnitude of s.count is number of processes blocked waiting in s.queue

Binary Semaphore

```
Type binarysemaphore =record
    value: (0,1);
    queue: List of processes
End;
S: binarysemaphore;
Waitb(s):
    If s.value=1 then s.value=0
                   else begin
                          place this process in s.queue;
                          block this process
                          end;
```

Binary Semaphore

```
signalb(s):
    If s.queue is empty then s.value=1
                    else begin
                            remove a  process from s.queue;
                            place this process in ready queue
                            end;
```

Mutual Exclusion Example

```
Var s:semaphore;(Initialized to 1)
        Begin
        repeat
                wait (s);
                <critical section>
                signal(s);
                <remainder section>
        forever
```

Dead lock

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	
	wait(Q);
wait(Q);	*wait*(S);
signal(S);	*signal*(Q);
signal(Q)	*signal*(S);

Producer Consumer Problem

- One or more producer are producing some items (data) and a single consumer is consuming these items one by one.
- Consumer can not consume until producer has produced
- While producer is producing, consumer can not consume and vice versa
- We assume producer can produce as many items it wants (infinite buffer)

```
Var n:semaphore (:=0)
    s:semaphore (:=1)
```

Producer:

```
Begin
    repeat
        produce;
        wait(s)
        append;
        Signal(s);
        signal (n);
    forever
End;
```

Consumer:

```
Begin
    repeat
        wait(n);
        wait(s);
        take;
        signal(s);
        consume;
    forever
End;
```

- What happens if signal(s) and signal(n) in producer process is interchanged ?
  - This will have no effect as consumer must wait for both semaphore before proceeding
- What if wait(n) and wait(s) are interchanged ?
  - If consumer enters the critical section when buffer is empty (n.count=0) then no producer can append to buffer and system is in deadlock.

- Semaphore provide a primitive yet powerful tool for enforcing Mutual exclusion and process coordination but it may be difficult to produce correct program by using semaphore
- As wait and signal operation are scattered throughout a program, it is difficult to see the overall effect of these operations on semaphores they affect.

Bounded buffer

Var f,e,s :semaphore;(In the beginning s=1,f=0,e=n)

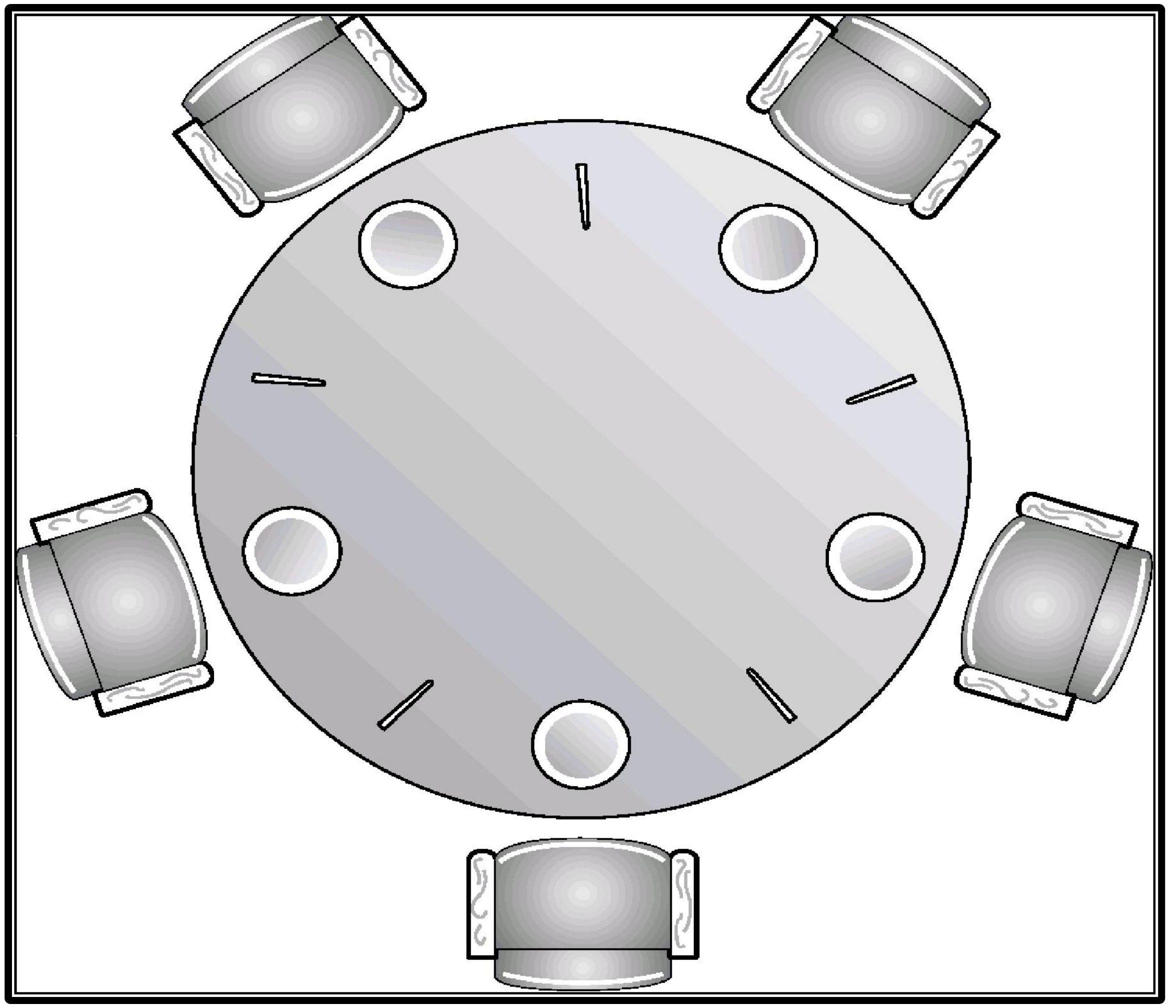
Producer

```
Begin
    repeat
        produce;
        wait (e)
        wait(s)
        append;
        Signal(s);
        signal (f);
    forever
End;
```

Consumer

```
Begin
    repeat
        wait(f);
        wait(s);
        take;
        signal(s);
        signal (e);
        consume;
    forever
End;
```

Dining-Philosophers Problem



Shared data **chopstick[5]: semaphore; initially it is initialized to 1**

Dining-Philosophers Problem

```
do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
          …
          eat
          …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
          …
          think
          …
  } while (1);
```

Thanks