BITS Pilani
Pilani Campus

Computer Organization and Software Systems

Session 11

Dr. Lucy J. Gudino
WILP & Department of CS & IS

Last Session

List of Topic Title	Text/Ref Book/external resource
• Scheduling algorithms (Priority, RR, Multilevel / Feedback)	T2

Today's Session

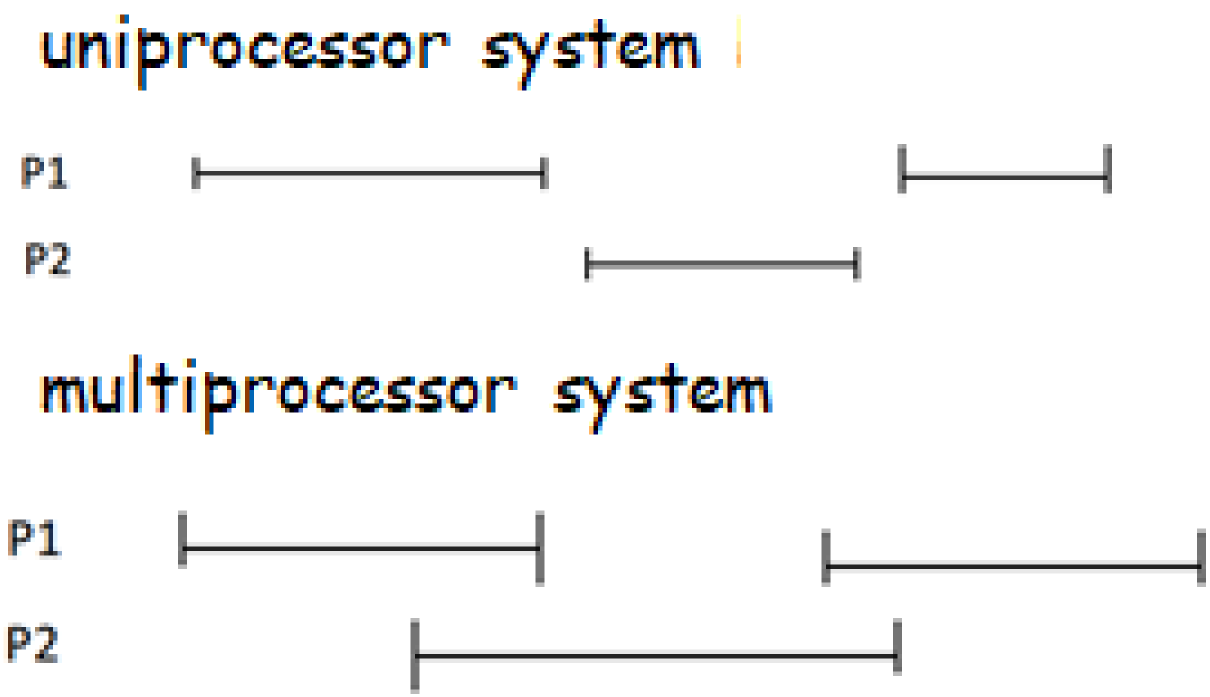
List of Topic Title	Text/Ref Book/external resource
• Process Coordination • Synchronization • Deadlock	T2

Process Synchronization

BITS Pilani
Pilani Campus

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 P0

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

Process P1

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" ➔ Decker'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

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" ➔ Decker'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
turn = 0 flag = {F, F}

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	Critical Section… flag[0] = false	

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

Algorithm

turn = 0 flag = {F, F}

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

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

6	Critical Section... flag[o] = 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
- 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
  - Also 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

***wait* (*S*):**
while *S*≤ 0 do *no-op*;
S--;
***signal* (*S*):**
S++;

Shared data:
semaphore S ; //initially *S* = 1

Process *P0:*
do {
remainder section

wait(S);
critical
signal(S);

remainder section
} while (1);

Process *P1:*
do {
remainder section

wait(S);
critical section
signal(S);

remainder section
} while (1);

Process synchronization

- Semaphore can be used to solve various synchronization problems
- Example : two concurrent processes : P1 (with statement S1) and P2 (with statement 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.

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)
```

Deadlock

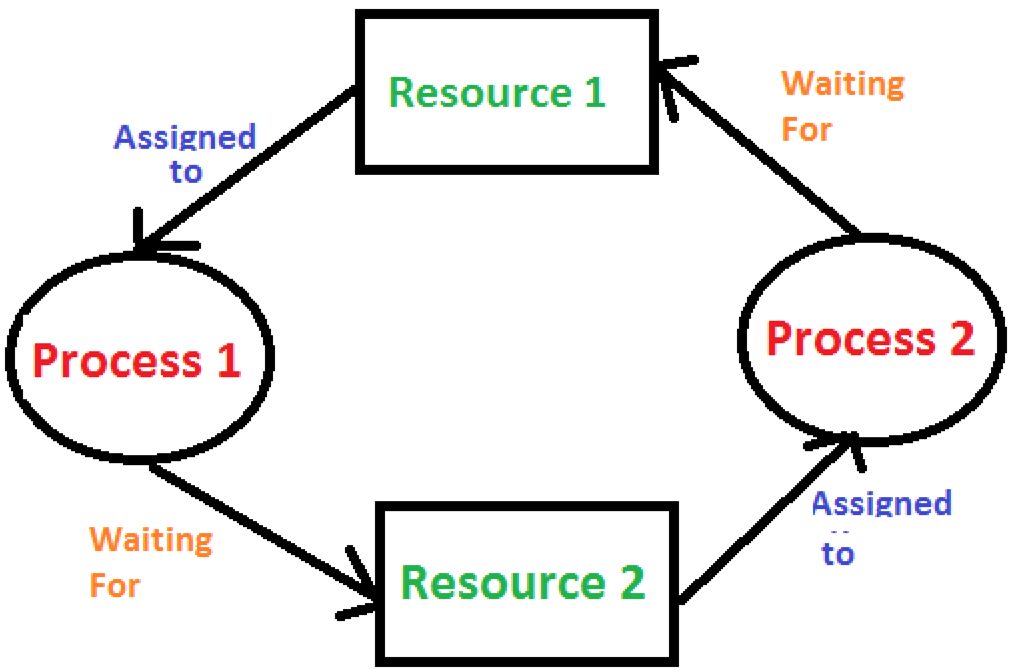
EXAMPLES:

"It takes money to make money".

"You can't get a job without experience; you can't get experience without a job."

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.



Contd…

- Example
  - System has 2 disk drives D1 and D2
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Finite number of resources
  - Example : Memory space, CPU, files, I/O devices – printers, monitor, DVD drives
- Resource types and instances
  - system with 3 printers ➔ Resource Type: printer and Number of instances : 3
- Process must request a resource before using it and must release the resource after using it

Contd…

- Each process utilizes a resource as follows:
  - request
  - use
  - release
- Device : request () and release()
- File: open() and close ()
- Memory: allocate () and free ()

Deadlock Characterization

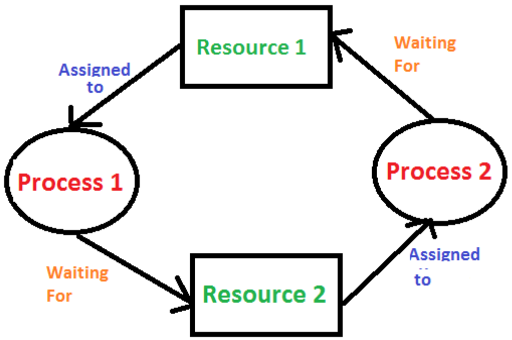
Mutual exclusion: only one process at a time can use a resource.

Hold and wait: a process holding at least one resource and is waiting to acquire additional resources held by other processes.

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait: there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

$P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



NECESSARY CONDITIONS

ALL of these four **must** happen simultaneously for a deadlock to occur

Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

V is partitioned into two types:

- $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

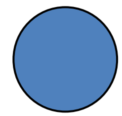
- $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

request edge – directed edge $P_{\mathrm{i}} \rightarrow R_j$

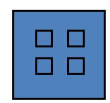
assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

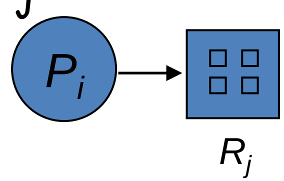
Process



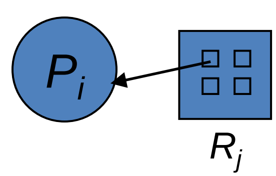
Resource Type with 4 instances



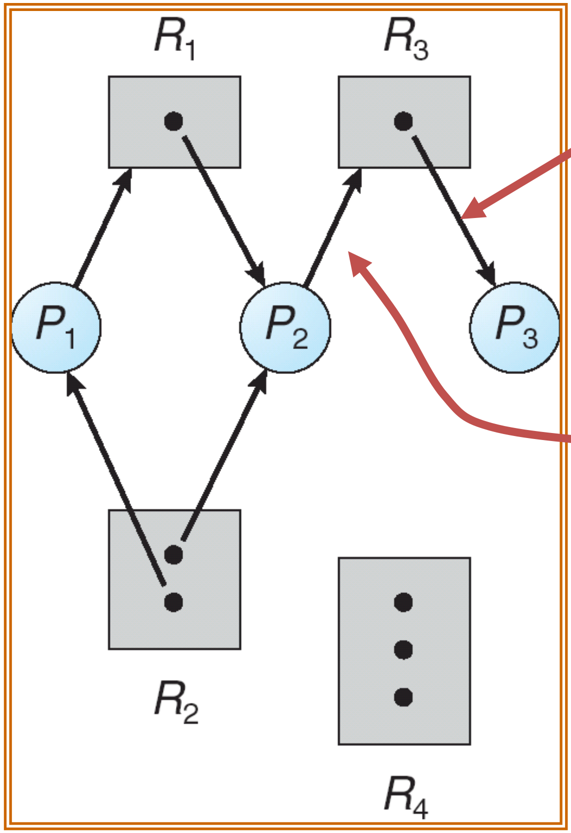
P_i requests instance of R_j



P_i is holding an instance of R_j



Example 1



R3 Assigned to P3

P2 Requests R3

Example 2

Cycle 1:
P1→R1→P2→R3→P3→R2→P1
Cycle 2:
P2→R3→P3→R2→P2