



CONCURRENCY CONTROL

SPOS Unit IV

INTERPROCESS COMMUNICATION

- Three issues:-

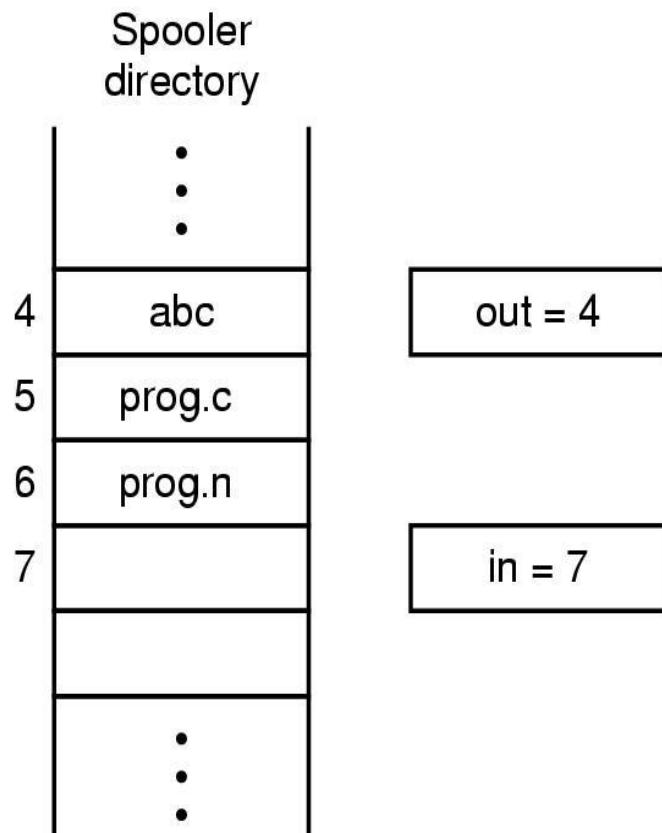
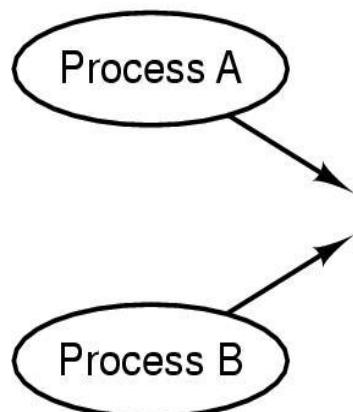
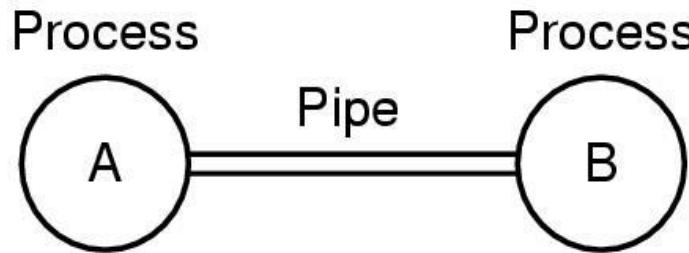
- Pass information between processes
- Avoid 2 processes get into each others way & so the deadlock
- Proper sequencing

- Race condition:- eg :- 2 processes want to access shared memory areas same time

- Result of most of the test runs are fine but once in rare while something weird & unexpected happens

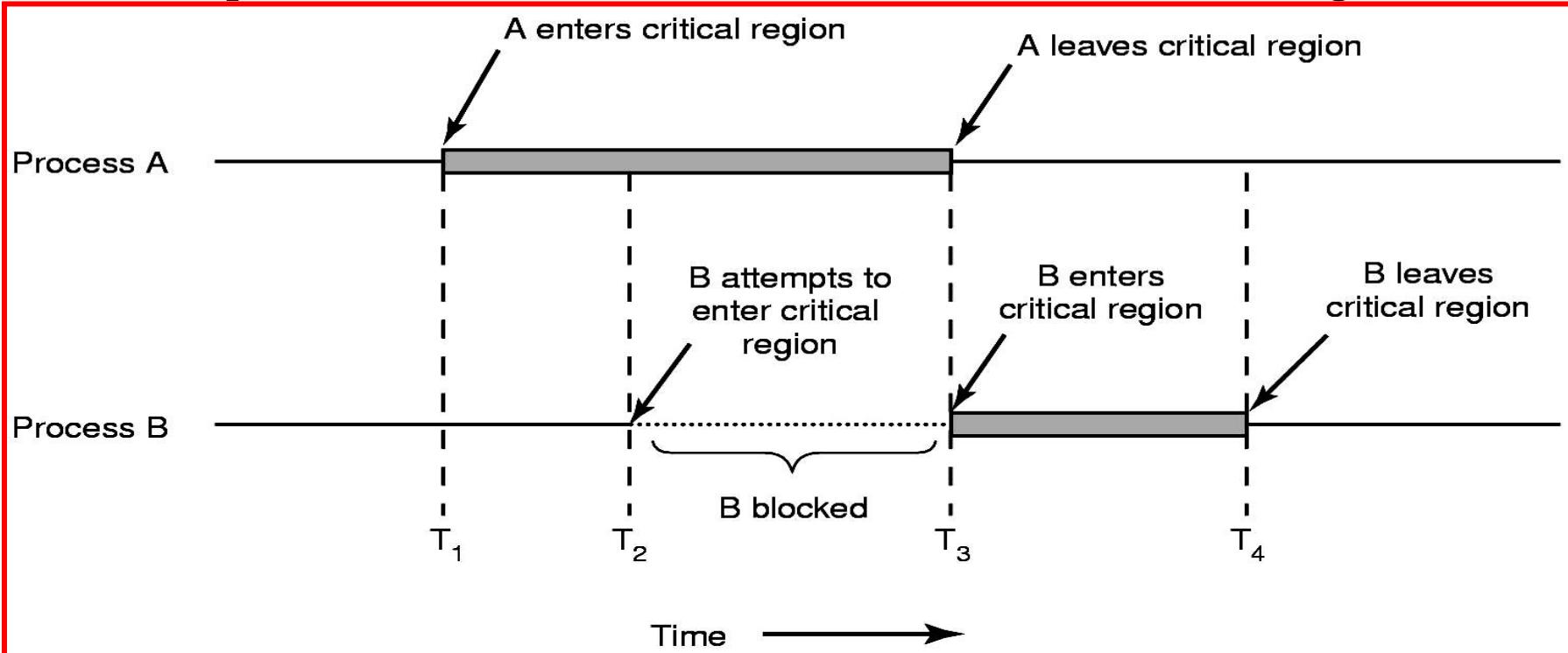
- Critical Regions:- sensitive part of code:-

- Shared memory
- Shared resource
- Shared variable
- Global variable etc.



IPC 1:- MUTUAL EXCLUSION

- To ensure only 1 process is accessing critical area at a time
- Four conditions to provide mutual exclusion
 1. No two processes **simultaneously in critical region**
 2. No assumptions made about **speeds** or numbers of CPUs
 3. No process running **outside** its critical region may block another process
 4. No process must wait **forever** to enter its critical region



MUTUAL EXCLUSION WITH BUSY WAITING

- Various proposals:-

1. Disable interrupt:- hence indirectly context switching
 - Disadv:- its unwise to give user process to turn off interrupt.
2. Lock variable:- scheduling is still on but no checking
 - Process can try after some time
 - But this lock is again a shared variable.
3. Strict alternation[spin lock]:- i.e. allot in a turn
 - But it violate condition 3 i.e. blocked by noncritical process.

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

MUTUAL EXCLUSION WITH BUSY WAITING CONTD....

4. Petersons solution:- combination of Turn[4] i.e. busy wait & Lock[3]

```
#define FALSE 0
#define TRUE 1
#define N      2          /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;               /* number of the other process */

    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;          /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

IMPLEMENTATION OF PETERSONS

- TSL instruction[test & set lock]:-Require help from hardware
enter_region:

```
TSL REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET | return to caller; critical region entered
```

| copy lock to register and set lock to 1
| was lock zero?
| if it was non zero, lock was set, so loop

leave_region:

```
MOVE LOCK,#0  
RET | return to caller
```

| store a 0 in lock

○ In case if TSL not available as inbuilt feature.

enter_region:

```
MOVE REGISTER,#1  
XCHG REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET
```

| put a 1 in the register
| swap the contents of the register and lock variable
| was lock zero?
| if it was non zero, lock was set, so loop
| return to caller; critical region entered

leave_region:

```
MOVE LOCK,#0  
RET
```

| store a 0 in lock
| return to caller

SLIP AND WAKEUP

- Busy waiting is a wastage of time
- **Producer consumer problem**:- when ever buffer full producer will go in sleeping. Llly whenever buffer is empty consumer will be sleeping.
- BUT wakeup sent to a process that is not yet sleeping is lost & hence **wakeup waiting bit introduced(a combination)**
- also here counter is a shared resource managed by users
- All this instructions are not atomic.

Also mutual exclusion cannot be used for **synchronization**

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                                /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        item = produce_item();                /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(item);                  /* put item in buffer */
        count = count + 1;                  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        if (count == 0) sleep();              /* if buffer is empty, got to sleep */
        item = remove_item();                /* take item out of buffer */
        count = count - 1;                  /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                /* print item */
    }
}
```

SEMAPHORE

- Almost same as producer-consumer. Count will be decremented till 0 & then go for sleeping, all this are designed as **atomic action[disable interrupt]**.
- Operation performed:-
UP DOWN
- But for more than 1 CPU lock should be used. Like TSL.
- BINARY SEMAPHORE:-**
one having maximum count=1.
- Can be used for synchronization

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```

MUTEXES

- It's a Binary semaphore its good only for managing mutual exclusion.
- Operation performed:- Lock Unlocked
- Specially used with THREADS where no clock available for stopping.
- If unable to acquire call thread_yield() to give-up CPU. & hence no busy waiting. As used in single user space as that of thread, easy to implement.

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

mutex_lock:

```
TSL REGISTER,MUTEX  
CMP REGISTER,#0  
JZE ok  
CALL thread_yield  
JMP mutex_lock
```

```
| copy mutex to register and set mutex to 1  
| was mutex zero?  
| if it was zero, mutex was unlocked, so return  
| mutex is busy; schedule another thread  
| try again later
```

ok: RET | return to caller; critical region entered

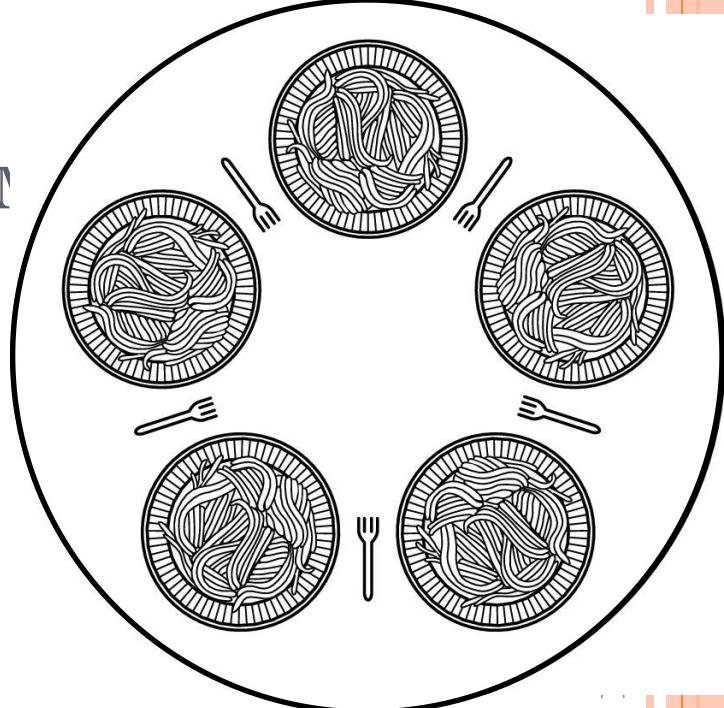
mutex_unlock:

```
MOVE MUXEX,#0  
RET | return to caller
```

```
| store a 0 in mutex
```

CLASSICAL IPC PROBLEMS

- DINNING PHILOSOPHERS PROBLEM
- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



```
#define N 5                                     /* number of philosophers */\n\nvoid philosopher(int i)                      /* i: philosopher number, from 0 to 4 */{\n    while (TRUE) {\n        think();\n        take_fork(i);\n        take_fork((i+1) % N);\n        eat();\n        put_fork(i);\n        put_fork((i+1) % N);\n    }\n}
```

A nonsolution to the dining philosophers problem

DINING PHILOSOPHERS

```
#define N          5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N    /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

/* number of philosophers */
/* number of i's left neighbor */
/* number of i's right neighbor */
/* philosopher is thinking */
/* philosopher is trying to get forks */
/* philosopher is eating */
/* semaphores are a special kind of int */
/* array to keep track of everyone's state */
/* mutual exclusion for critical regions */
/* one semaphore per philosopher */
/* i: philosopher number, from 0 to N-1 */
/* repeat forever */
/* philosopher is thinking */
/* acquire two forks or block */
/* yum-yum, spaghetti */
/* put both forks back on table */

Solution to dining philosophers problem (part 1)

DINING PHILOSOPHERS

```
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */  
{  
    down(&mutex);  
    state[i] = HUNGRY;  
    test(i);  
    up(&mutex);  
    down(&s[i]);  
}  
  
void put_forks(i)                                     /* i: philosopher number, from 0 to N-1 */  
{  
    down(&mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    up(&mutex);  
}  
  
void test(i)                                         /* i: philosopher number, from 0 to N-1 */  
{  
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
        state[i] = EATING;  
        up(&s[i]);  
    }  
}
```

Solution to dining philosophers problem (part 2)

READER WRITER PROBLEM

- Eg:- airline reservation system.
- Number of readers can exist at a time. But at the max only one writer can be there[exclusive]
- Writer should have higher priority as STARVATION problem may occurred.
- Only “rc” is critical
- No. of readers can be very large.

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */
```

```
void reader(void)
{
    while (TRUE) {
        down(&mutex); /* repeat forever */
        /* get exclusive access to 'rc' */
        rc = rc + 1; /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        read_data_base(); /* access the data */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc - 1; /* one reader fewer now */
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}
```

```
void writer(void)
{
    while (TRUE) {
        think_up_data(); /* repeat forever */
        /* noncritical region */
        down(&db); /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db); /* release exclusive access */
    }
}
```



SLEEPING BARBERS PROBLEM

- A barber sleeps in barber chair if no customer present
- Solution 3 semaphores:
 1. customers[counting]
 2. barber[binary]
 3. barber chair[mutex]

SLEEPING BARBERS PROBLEM

```
#define CHAIRS 5          /* # chairs for waiting customers */

typedef int semaphore;    /* use your imagination */

semaphore customers = 0;  /* # of customers waiting for service */
semaphore barbers = 0;    /* # of barbers waiting for customers */
semaphore mutex = 1;      /* for mutual exclusion */
int waiting = 0;          /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);    /* go to sleep if # of customers is 0 */
        down(&mutex);         /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);         /* one barber is now ready to cut hair */
        up(&mutex);           /* release 'waiting' */
        cut_hair();            /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);           /* enter critical region */
    if (waiting < CHAIRS) {  /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);     /* wake up barber if necessary */
        up(&mutex);           /* release access to 'waiting' */
        down(&barbers);       /* go to sleep if # of free barbers is 0 */
        get_haircut();         /* be seated and be serviced */
    } else {
        up(&mutex);           /* shop is full; do not wait */
    }
}
```

- A process is deadlocked when it is waiting on an event which will never happen

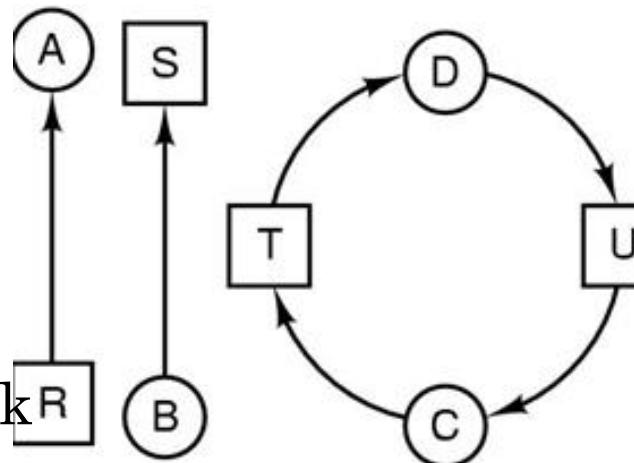
- No processes can
 - run
 - release resources
 - be awakened

- A system is in deadlock when 1 or more processes are deadlocked

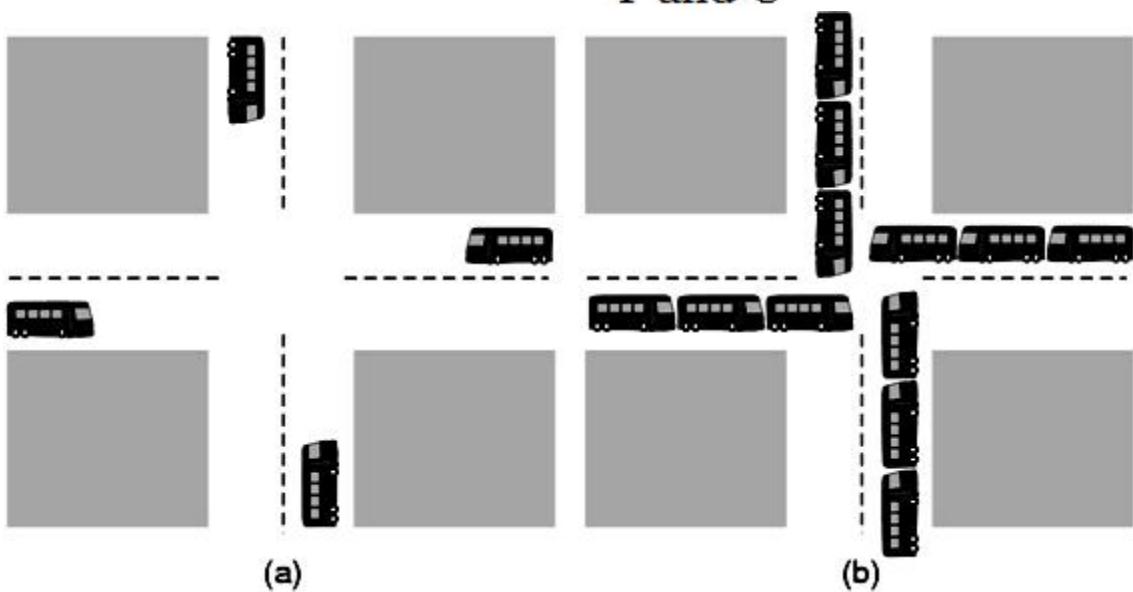
- Following points are already discussed in Expt 6:- deadlock, 4 conditions, Bankers algorithm for single resource, Deadlock detection & avoidance

DEADLOCK

Deadlock modeling:-



- Resource R assigned to process A
- Process B is requesting/waiting for resource S
- Process C and D are in deadlock over resources T and U



a) A potential deadlock b) An actual deadlock.

DEADLOCK MODELING

A

Request R
Request S
Release R
Release S

B

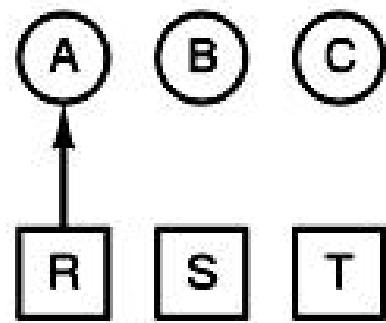
Request S
Request T
Release S
Release T

C

Request T
Request R
Release T
Release R

1. A requests R
 2. B requests S
 3. C requests T
 4. A requests S
 5. B requests T
 6. C requests R
- deadlock

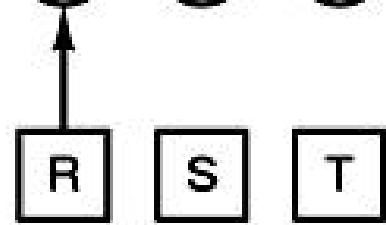
(d)



(a)



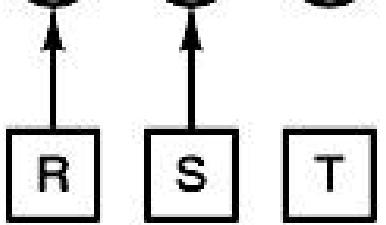
(e)



(b)



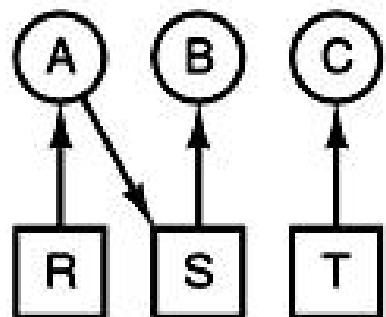
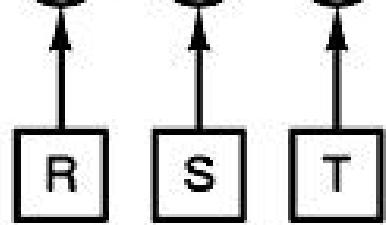
(f)



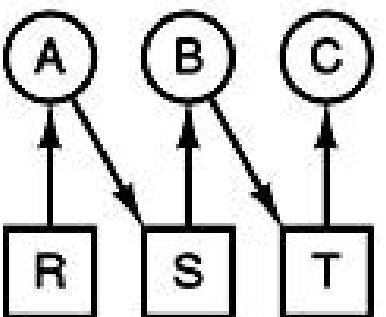
(c)



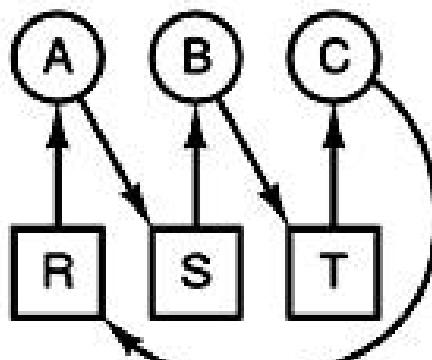
(g)



(h)



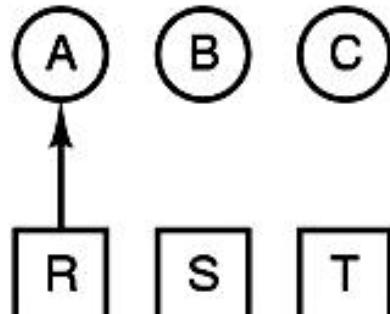
(i)



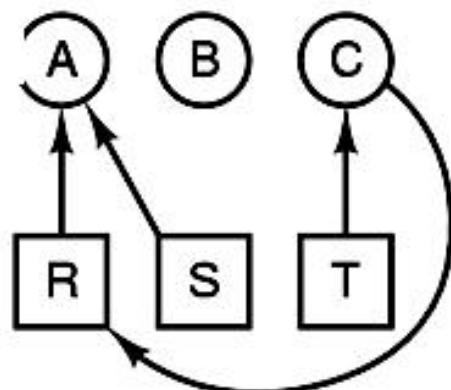
(j)

DEADLOCK MODELING

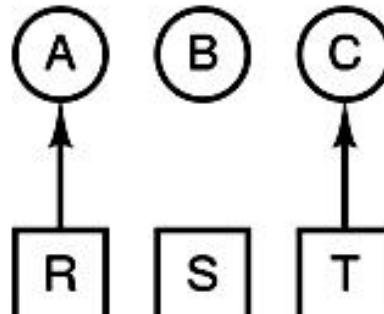
1. A requests R
 2. C requests T
 3. A requests S
 4. C requests R
 5. A releases R
 6. A releases S
- no deadlock



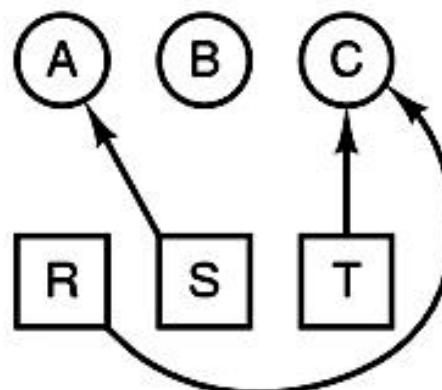
(k)



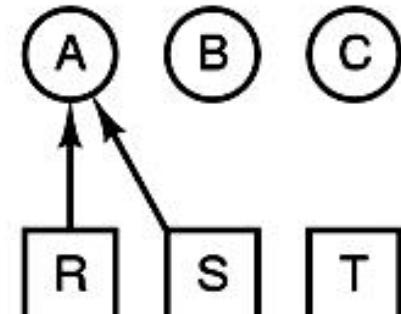
(o)



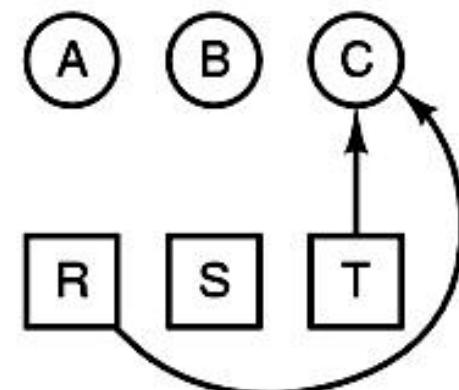
(m)



(p)



(n)



(q)

WAYS OF HANDLING DEADLOCK

- Deadlock Prevention
- Deadlock Detection
- Deadlock Avoidance
- Deadlock Recovery

RESOURCE ACQUISITION

- Capturing a resource
- Using semaphore or mutex
- Ordering of resource acquisition is very important.

OSTRICH ALGORITHM

- Stick your head in sand & pretend there is no problem at all
 - deadlocks occur very rarely
 - cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
 - convenience
 - correctness
- Deadlocks have Symptoms & hence detection possible.

```
typedef int semaphore;  
semaphore resource_1;
```

```
void process_A(void) {  
    down(&resource_1);  
    use_resource_1();  
    up(&resource_1);  
}
```

(a)

Using a semaphore to protect resources.
(a) One resource. (b) Two resources

```
semaphore resource_1;  
semaphore resource_2;  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources();  
    up(&resource_2);  
    up(&resource_1);  
}
```

```
void process_B(void) {  
    down(&resource_2);  
    down(&resource_1);  
    use_both_resources();  
    up(&resource_1);  
    up(&resource_2);  
}
```

Code with a potential deadlock.

```
typedef int semaphore;  
semaphore resource_1;  
semaphore resource_2;
```

```
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources();  
    up(&resource_2);  
    up(&resource_1);  
}
```

(b)

```
typedef int semaphore;  
semaphore resource_1;  
semaphore resource_2;  
void process_A(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources();  
    up(&resource_2);  
    up(&resource_1);  
}
```

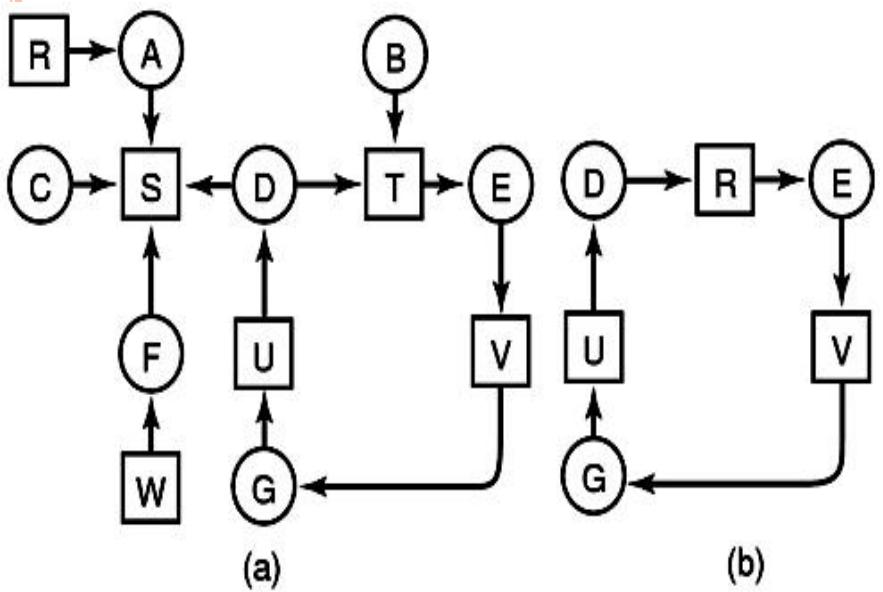
```
void process_B(void) {  
    down(&resource_1);  
    down(&resource_2);  
    use_both_resources();  
    up(&resource_2);  
    up(&resource_1);  
}
```

Deadlock-free code.

DEADLOCK DETECTION

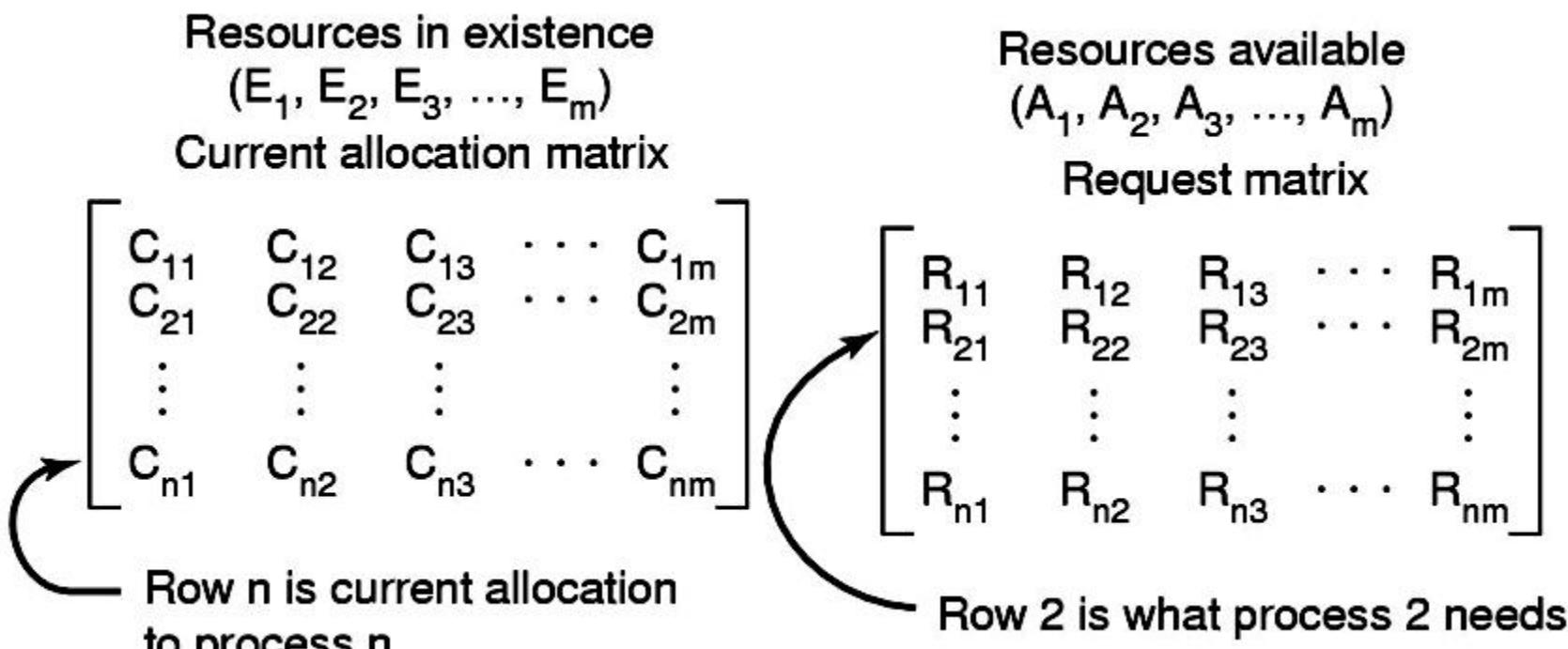
- Deadlock detection with 1 resource of each type:-

- Note resource ownership & requests (a)
 - A cycle can be found within graph, denoting deadlock(b)
 - List of nodes been marked to prevent repeated inspection
 - Depth 1st search mechanism widely used



- Algorithm for detecting deadlock:
 1. For each node [N] perform following 5 steps with N as starting node
 2. Initialize L to empty list, designate all arcs as unmarked.
 3. Add current node to end of L, check to see if node now appears in L 2 times. If it does, graph contains a cycle (listed in L), algorithm terminates.
 4. From given node, see if any unmarked outgoing arcs. yes go to step 5; if not, go to step 6.
 5. Pick unmarked outgoing arc at random & mark it. Then follow it to new current node & go to step 3
 6. If this is initial node, graph does not contain any cycles, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 3.

FOR MULTIRESOURCE OF EACH TYPE



Algorithm:

1. Look for an unmarked process, P_i , for which i -th row of R is less than or equal to A .
2. If such a process is found, add the i -th row of C to A , mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

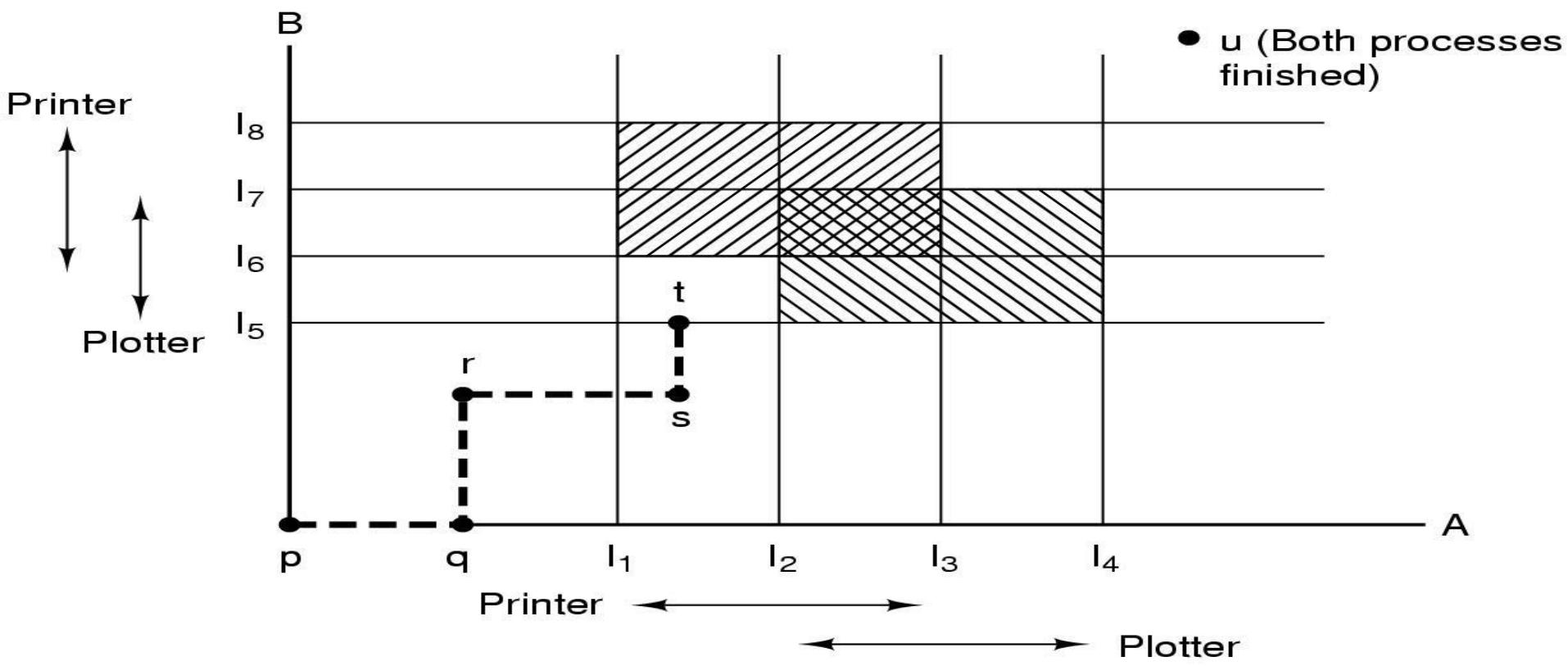
				Tape drives	Plotters	Scanners	CD Roms	
				A = (2 1 0 0)				
				Request matrix				
$E = (4 2 3 1)$	Current allocation matrix							
$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$								
$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$								

RECOVERY FROM DEADLOCK

- Recovery through preemption
 - Take a resource from some other process
 - Depends on nature of the resource
- Recovery through rollback
 - Checkpoint a process periodically
 - Use this saved state
 - Restart the process if it is found deadlocked
- Recovery through killing processes
 - Crudest but simplest way to break a deadlock
 - Kill one of the processes in the deadlock cycle
 - The other processes get its resources
 - Choose process that can be rerun from the beginning

DEADLOCK AVOIDANCE

- Three methods → (1) Resource trajectories (2) Safe & unsafe states (3) Banker's algorithm[refer expt 6]
- Figure 2 process deadlock resource trajectory At point t system is requesting a resource which can lead to potential deadlock → hence B should be suspended till A releases plotter.



SAFE & UNSAFE STATES

- If it is not deadlocked & there is some scheduling order in which every process can run to completion even if all of them suddenly request for their maximum resources immediately.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4

Free: 1

(b)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	-
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7

(e)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	-	-
C	2	7

Free: 4

(d)

BANKER'S ALGORITHM FOR MULTIPLE RESOURCES

- Algorithm for checking to see if a state is safe:

- Look for row[R], whose unmet resource needs all $\leq A$. If no such row exists, system may deadlock.
- Assume process of row chosen requests all resources it needs & finishes. Mark process as terminated, withdraw resource allotted
- Repeat steps 1 & 2 until either all processes marked terminated (initial state was safe) or no process left whose resource needs can be met (there is a deadlock).

Process
Tape drives
Plotters
Scanners
CD ROMs

A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process
Tape drives
Plotters
Scanners
CD ROMs

A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

$$\begin{aligned}E &= (6342) \\P &= (5322) \\A &= (1020)\end{aligned}$$

DEADLOCK PREVENTION 4 WAYS

- Attacking the mutual exclusion condition:→
 - Some devices (such as printer) can be spooled
 - Only the printer daemon uses printer resource not processes
 - Thus deadlock for printer eliminated
 - Not all devices can be spooled like disk
 - Principle:
 - Avoid assigning resource when not absolutely necessary
 - As few processes as possible actually claim resource
- Attacking the Hold and Wait Condition→
 - A process is given its resources on a "ALL or NONE" basis in starting
 - A process never has to wait for what it needs
 - Problems
 - May not know required resources at start of run practically
 - Also ties up resources other processes could be using, so not optimal & hence wastage
 - Possibility of starvation
 - Variation:
 - Process must give up all resources
 - Then request all immediately needed

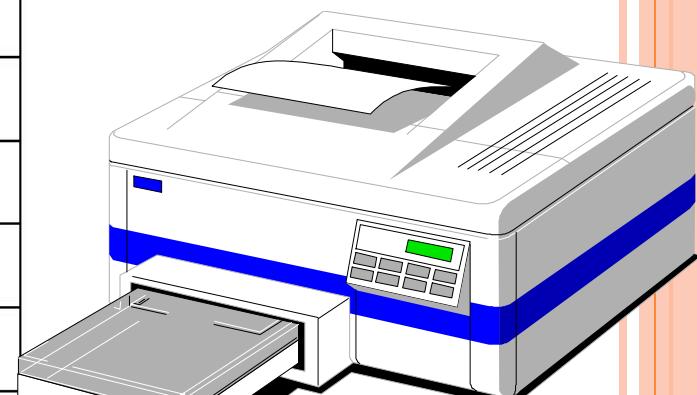
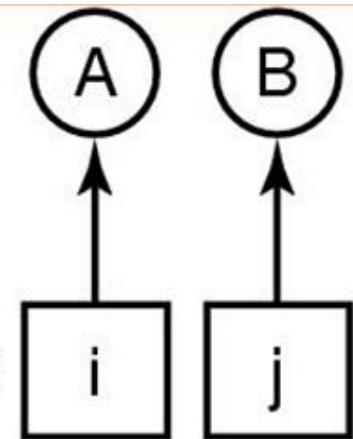
PREVENTION

• Attacking No Preemption Condition

- When request refused, it MUST release all held resources
- Also can be removed forcefully before finish
- Not a viable option always:-
Consider a process given printer halfway through its job now forcibly take away printer
- Very costlier
- Needs context switching
- Possibility of starvation

- Attacking Circular Wait Condition
- Resources are uniquely numbered
- Processes can only request resources in linear ascending order
- **Complicated design as NO. log, sequence need to be stored.**

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive



Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

THREE OTHER MISCELLANEOUS ISSUES

○ Two-phase locking

- Phase one
 - Process tries to lock all need records, one at a time
 - If needed record found locked, start over
 - (No real work done in phase one)
- If phase one succeeds, it starts second phase,
 - Performing updates
 - Releasing locks
- Similarity to requesting all resources at once
- Algorithm works where programmer can arrange
 - Program can be stopped, restarted

○ Nonresource Deadlocks

- Possible for 3 processes to deadlock
 - each is waiting for the other to do some task
- Can happen with semaphores
 - each process required to do a *down()* on two semaphores (*mutex* and another)
 - if done in wrong order, deadlock results

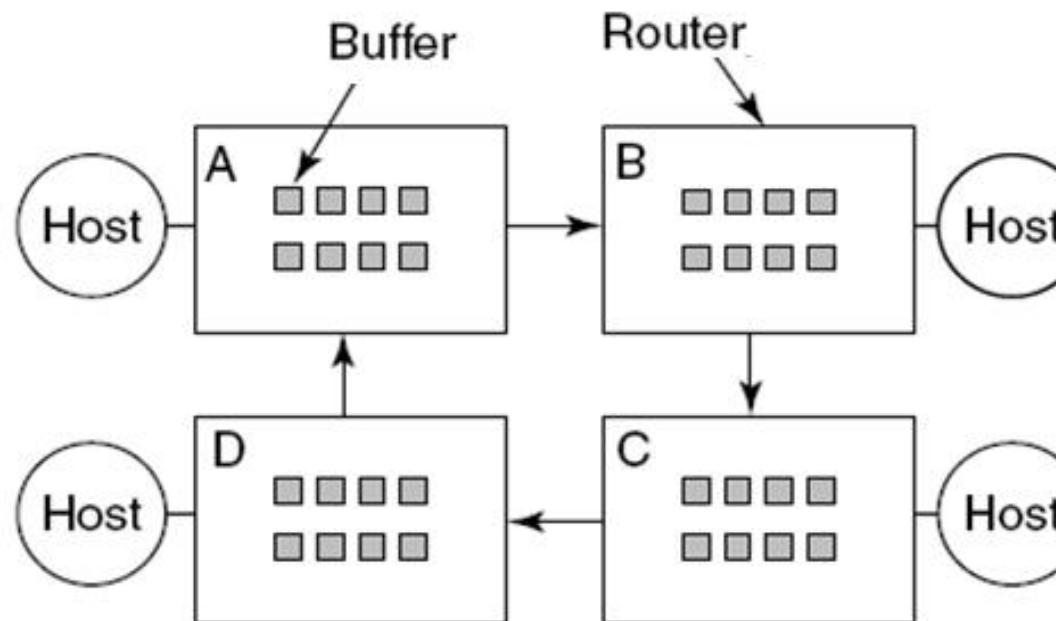
THREE OTHER MISCELLANEOUS ISSUES

○ Starvation:-

- Algorithm to allocate a resource
 - may be to give to shortest job first
- Works great for multiple short jobs in a system
- May cause long job to be postponed indefinitely
 - even though not blocked
- Solution: FIFO

○ Communication Deadlocks:-fig (a)

○ Livelock:- program(A) & (B)→



```
void process_A(void) {  
    enter_region(&resource_1);  
    enter_region(&resource_2);  
    use_both_resources( );  
    leave_region(&resource_2);  
    leave_region(&resource_1);  
}
```

```
void process_B(void) {  
    enter_region(&resource_2);  
    enter_region(&resource_1);  
    use_both_resources( );  
    leave_region(&resource_1);  
    leave_region(&resource_2);  
}
```

QUESTION : WHAT IS THE SIMPLEST AND
MOST USED METHOD TO RECOVER FROM A
DEADLOCK?

RE-BOOT



THANK YOU