Chapter 4

"Deadlocks"

1) *Deadlock Definition and Deadlock Modeling.*

Deadlock

- "A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause".

Deadlock Modeling

- Fig. 4-1 (a) represents that Resource R is given to Process A.
- Fig. 4-1 (b) represents that Process B requests for Resource S.
- Suppose process D holds resource T and process C holds resource U.
- Now process D requests resource U and process C requests resource T but none of process will get this resource because these resources are already hold by other process so both can be blocked, with neither one be able to proceed, this situation is called deadlock.

Example:

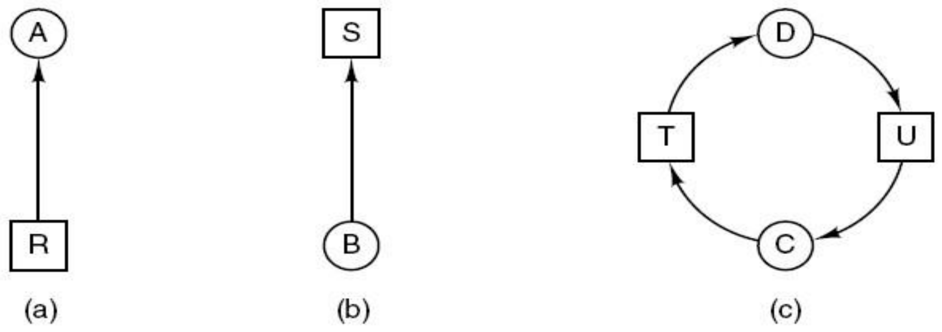


Figure 4-1 Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- As shown in figure 4-1, resource T assigned to process D and resource U is assigned to process C.
- Process D is requesting / waiting for resource U and process C is requesting / waiting for resource T.
- Processes C and D are in deadlock over resources T and U.

2) 4 necessary conditions for deadlock.

- There are four conditions that must hold for deadlock:

1) Mutual exclusion condition
    - Each resource is either currently assigned to exactly one process or is available.
2) Hold and wait condition
    - Process currently holding resources granted earlier can request more resources.

3) No preemption condition

- Previously granted resources cannot be forcibly taken away from process. They must be explicitly released by the process holding them.

4) Circular wait condition

- There must be a circular chain of 2 or more processes. Each process is waiting for resource that is held by next member of the chain.

NOTE: All four of these conditions must be present for a deadlock to occur.

3) Strategies to deal with the deadlocks

1. Just ignore the problem. Maybe if you ignore it, it will ignore you.
2. Detection and recovery. Let them occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions.

4) Deadlock ignorance. OR Ostrich Algorithm.

- Pretend (imagine) that there's no problem.
- This is the easiest way to deal with problem.
- This algorithm says that stick your head in the sand and pretend (imagine) that there is no problem at all.
- This strategy suggests to ignore the deadlock because deadlocks occur rarely, but system crashes due to hardware failures, compiler errors, and operating system bugs frequently, then not

to pay a large penalty in performance or convenience to eliminate deadlocks.

- This method is reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows take this approach
  - Resources (memory, CPU, disk space) are plentiful
  - Deadlocks over such resources rarely occur
  - Deadlocks typically handled by rebooting

5) Deadlock Detection and Recovery.

a) Deadlock Detection:

1. **Deadlock detection with single resource of each type**
2. **Deadlock detection with multiple resources of each type**

b) Deadlock Recovery

a) Deadlock Detection:

1. Deadlock detection with single resource of each type

- Let us begin with the simplest case: there is only one resource of each type. Such a system might have one scanner, one printer, one plotter, and one tape drive, but no more than one of each class of resource.
- we can construct a resource graph.
- If this graph contains one or more cycles, a deadlock exists.
- Any process that is part of a cycle is deadlocked.

- If no cycles exist, the system is not deadlocked.
- Example: consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process *A* holds *R* and wants *S*.
2. Process *B* holds nothing but wants *T*.
3. Process *C* holds nothing but wants *S*.
4. Process *D* holds *U* and wants *S* and *T*.
5. Process *E* holds *T* and wants *V*.
6. Process *F* holds *W* and wants *S*.
7. Process *G* holds *V* and wants *U*.

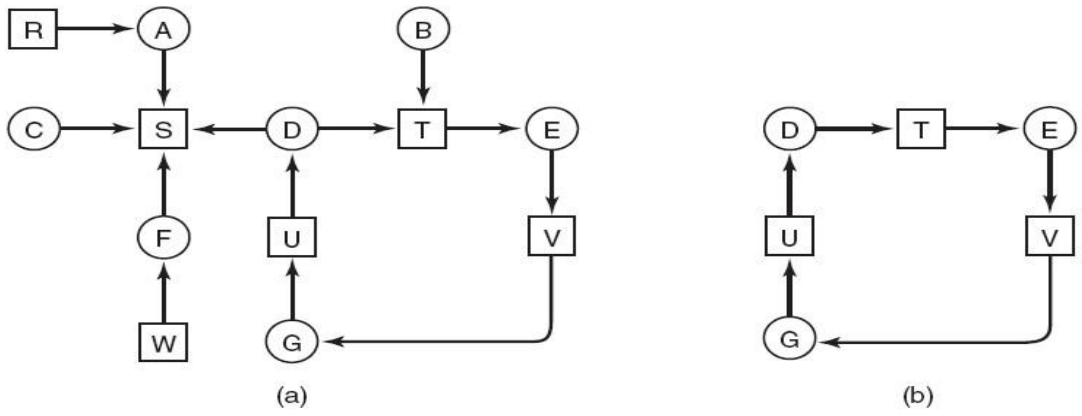


Figure 4-2 (a) A resource graph. (b) A cycle extracted from (a).

Algorithm: Deadlock detection with single resource of each type

1. For each node, N in the graph, perform the following five steps with N as the starting node.
2. Initialize L to the empty list, label all arcs as unmarked.
3. Add current node to end of L, check to see if node now

appears in L two times. If it does, graph contains a cycle (listed in L), algorithm terminates.

4. From given node, see if any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and 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 example as shown in figure 4-2 (a),
    - We are starting from node D.
    - Empty list L = ()
    - Add current node so Empty list = (D).
    - From this node there is one outgoing arc to T so add T to empty list.
    - So Empty list become L = (D, T).
    - Continue this step….so we get empty list as below

      L = (D, T, E)……….. L = (D, T, E, V, G, U, D)
    - In the above step in empty list the node D appears twice, so deadlock.

2. Deadlock detection with multiple resources of each type

- When multiple copies of some of the resources exist, a matrix-based algorithm can be used for detecting deadlock among n processes.
- Let the number of resource classes be *m with*
  - *E*1 resources of class 1
  - *E*2 resources of class 2, and generally,
  - *Ei* resources of class *i* (1 ≤ *i* ≤ *m*).
- *E* - **existing resource vector**
- *A* - **available resource vector**
- *C* - **current allocation matrix**
- *R* – **request matrix**.

- "Every resource is either allocated or is available".

- This observation means that:

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

C_{ij} - *Currently* allocated resources of **type *j*** to ***process i***

A_j - *Available* resources of **type *j***

E_j - *Existing* resources of **type *j***

Rij – Requested resources of **type *j*** by ***Process i***

- Let A be the available resource vector, with Ai giving the number of instances of resource i that are currently available.

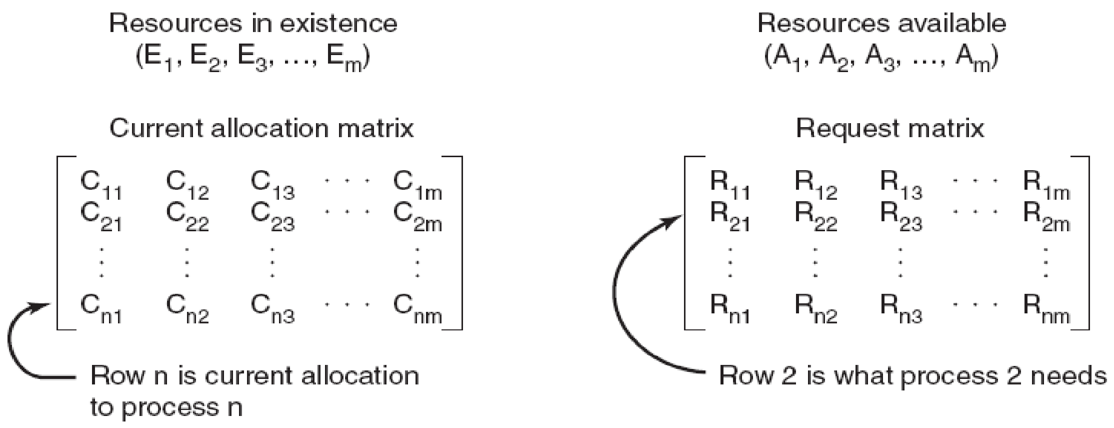

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$
Resources available
$(A_1, A_2, A_3, \ldots, A_m)$
Current allocation matrix
$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$
Request matrix
$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$
Row n is current allocation to process n
Row 2 is what process 2 needs

Figure 4-3. The four data structures needed by the deadlock detection algorithm.

- The deadlock detection algorithm is based on comparing vectors.
- Let us define the relation *A* ≤ *B* on two vectors *A* and *B* to mean that each element of *A* is less than or equal to the corresponding element of *B*.
- Mathematically, *A* ≤ *B* holds if and only if *Ai* ≤ *Bi* for 1 ≤ *i* ≤ *m*.
- Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked.
- When the algorithm terminates, any unmarked processes are known to be deadlocked.

Algorithm: Deadlock detection with multiple resources of each type

1. Look for an unmarked process, Pi, for which the 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.

Note : When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

Example: Deadlock detection with multiple resources

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1\ ) \end{array} \qquad \begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 2 & 1 & 0 & 0\ ) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Figure 4-4. An example for the deadlock detection algorithm.

- At this moment, we run the algorithm:

1) P3 can be satisfied, so P3 completes, returns resources A = (2 2 2 0)
2) P2 can be satisfied, so P2 completes, returns resources A = (4 2 2 1)
3) P1 can be satisfied, so P1 completes.

    - Now all processes are marked, so no deadlock.

b) Deadlock Recovery

- **Recovery through preemption**
  - In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
  - The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource.
  - Recovering this way is frequently difficult or impossible.
  - Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

- **Recovery through rollback**
  - Checkpoints are placed periodically.
  - Checkpointing a process means that 'its state is written to a file' so that it can be restarted later.
  - The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process.
  - When a deadlock is detected, it is easy to see which resources are needed.
  - To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints.
  - In effect, the process is reset to an earlier moment when it

did not have the resource, which is now assigned to one of the deadlocked processes.

- If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

- **Recovery through killing processes**
    - The crudest, but simplest way to break a deadlock is to kill one or more processes.
    - One possibility is to <u>kill a process in the cycle</u>. With a little luck, the other processes will be able to continue.
    - If this does not help, <u>it can be repeated until the cycle is broken</u>.
    - Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources.
    - In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs.

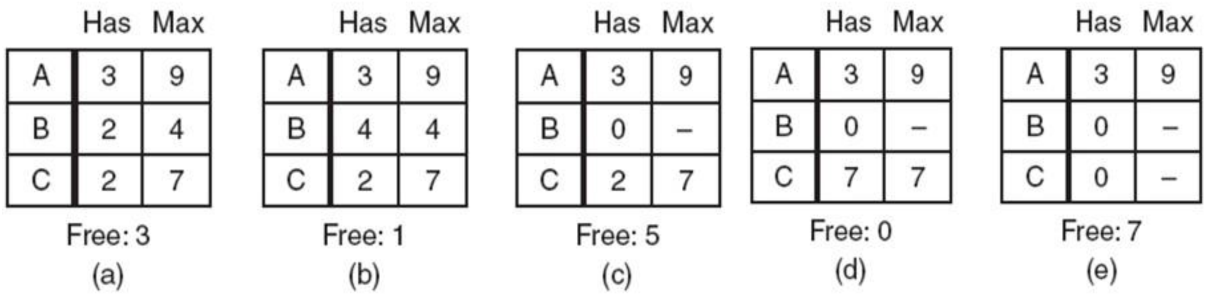
6) Safe and unsafe states with example.

- **Safe State:**

  "A state is said to be safe if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately"
- It is easiest to illustrate this concept by an example using one resource.
- In Fig. 4-6(a) we have a state in which process *A* has 3 instances

of the resource but may need as many as 9 eventually.

- Process *B* currently has 2 and may need 4 altogether, later.
- Similarly, process *C* also has 2 but may need an additional 5.



	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

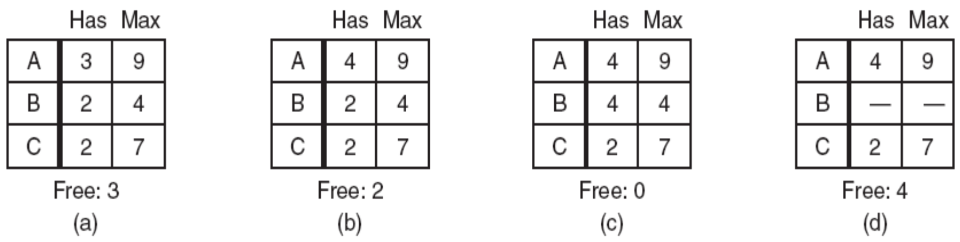
Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Figure 4-6. Demonstration that the state in (a) is safe.

- A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free.
- The state of Fig. 4-6 (a) is safe because there exists a sequence of allocations that allows all processes to complete.
- Namely, the scheduler could simply run *B* exclusively, until it asked for and got two more instances of the resource, leading to the state of Fig. 4-6 (b).
- When *B* completes, we get the state of Fig. 4-6 (c).
- Then the scheduler can run *C* leading eventually to Fig. 4-6 (d). When *C* completes, we get Fig. 4-6 (e).
- Now *A* can get the six instances of the resource it needs and also complete.
- Thus the state of Fig. 4-6 (a) is safe because the system, by careful scheduling, can avoid deadlock.



	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)

Figure 4-7. Demonstration that the state in (b) is not safe.

- Now suppose we have the initial state shown in Fig. 4-7 (a), but this time *A* requests and gets another resource, giving Fig. 4-7 (b).
- Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run *B* until it asked for all its resources, as shown in Fig. 4-7 (c).
- Eventually, *B* completes and we get the situation of Fig. 4-7 (d).
- At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion.
- Thus the allocation decision that moved the system from Fig. 4-7 (a) to Fig. 4-7 (b) results into an unsafe state.
- It is worth noting that an unsafe state is not a deadlocked state.
- The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

7) Banker’s Algorithm for Deadlock Avoidance with illustration.

- Deadlock can be avoided by allocating resources carefully.
- Carefully analyze each resource request to see if it can be safely granted.
- Need an algorithm that can always avoid deadlock by making right choice all the time.

a) Banker’s algorithm for single resource

- A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965); it is known as the banker's algorithm and is an extension of the deadlock detection algorithm.
- It is modeled on the way a small town banker might deal with a group of customers to whom he has granted lines of credit.
- What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied.
- If granting the request leads to a safe state, it is carried out.
- In fig. 4-8 we see four customers, A, B, C, and D, each of whom has been granted a certain number of credit units.
- The banker knows that not all customers will need their maximum credit at a time, so he has reserved only 10 units rather than 22 to service them.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Figure 4-8. Three resource allocation states :(a) Safe. (b) Safe. (c) Unsafe.

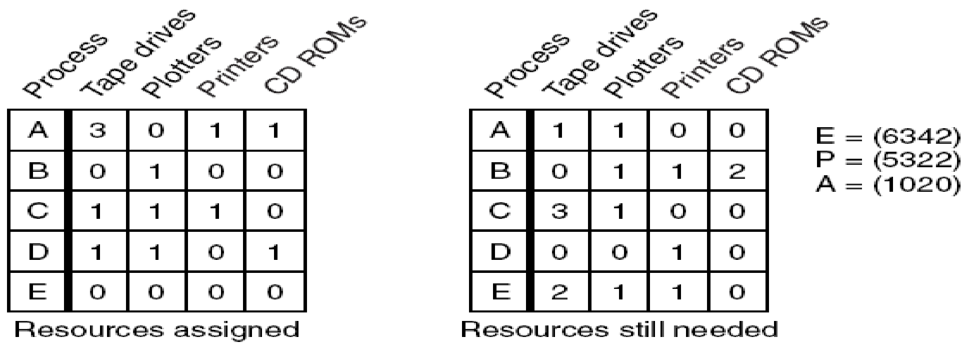
- First if we have situation as per fig 4-8 (a) then it is safe state because with 10 free units one by one all customers can be served.
- Second situation is as shown in fig. 4-8 (b) This state is safe because with two units left (free units), the banker can allocate units to C, thus letting C finish and release all four of his resources.
- With those four free units, the banker can let either D or B have the necessary units, and so on.
- Consider the third situation, what would happen if a request from B for one more unit were granted as shown in fig. 4-8 (c) then it becomes unsafe state.
- In this situation we have only one free unit and minimum 2 units are required by C. No one can get all resources to complete their work so it is unsafe state.

b) Banker's algorithm for multiple resource

- The banker's algorithm can be generalized to handle multiple resources.
- In fig. 4-9 we see two matrices. The one on the left shows how many of each resource are currently assigned to each of the five processes.
- The matrix on the right shows how many resource each

process still needs in order to complete the operation.

- These matrices are labeled as C and R respectively.



Process	Tape drives	Plotters	Printers	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	Printers	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

E = (6342)
P = (5322)
A = (1020)

Figure 4-9. The banker's algorithm with multiple resources.

- The three vectors at the right of the figure show the existing (total) resources E, the hold resources P, and the available resources A, respectively.
- From E we see that the system has six tape drives, three plotters, four printers, and two CD ROM drives. Of these, five tape drives, three plotters, two printers, and two CD ROM drives are currently assigned.
- This fact can be seen by adding up the four resource columns in the left hand matrix.
- The available resource vector is simply the difference between what the system has and what is currently in use, i.e. E – P = A

- The algorithm for checking to see if a state is safe can now be stated as follows:
  1) Look for each row in R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion (assuming processes keep all resources until they exit).
  2) Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the vector A.
  3) Repeat step 1 and 2 until either all processes are marked terminated (in which case the initial state was safe) or no process is left whose resources needs can be met (in which case there is a deadlock).

a. Now let us get back to the example of figure 4-9. The current state is safe. Suppose that process B now makes a request for the printer. This request can be granted because the resulting state is still safe (process D can finish, and then processes A or E, followed by the rest).
b. Now imagine that if process D request for 1 printer and 1 CD ROM then there is deadlock.

8) Deadlock Prevention

- Deadlock can be prevented by attacking the one of the four conditions that leads to deadlock.

1) **Attacking the Mutual Exclusion Condition**

- No deadlock if no resource is ever assigned exclusively to a single process.
- Some devices can be spooled such as printer, by spooling printer output; several processes can generate output at the same time.
- Only the printer daemon process uses physical printer.
- Thus deadlock for printer can be eliminated.
- Not all devices can be spooled.
- Principle:
  - Avoid assigning a resource when that is not absolutely necessary.
  - Try to make sure that as few processes as possible actually claim the resource.

2) **Attacking the Hold and Wait Condition**

- Require processes to request all their resources before starting execution.
- A process is allowed to run if all resources it needed is available. Otherwise nothing will be allocated and it will just wait.

- Problems:
  - A process may not know required resources at start of run.
  - Resource will <u>not</u> be used <u>optimally</u>.
  - It also <u>ties up resources</u> other processes could be using.
- Variation:
  - A process must give up all resources before making a new request. Process is then granted all prior resources as well as the new ones only if all required resources are available.

3) Attacking the No Preemption Condition

- To attack this condition, forcibly take a resource away from a process.
- When a process P0 request some resource R which is held by another process P1 then resource R is forcibly taken away from the process P1 and allocated to P0.
- *Problem*: Consider a process holds the printer, halfway through its job (half page printed); taking the printer away from this process without having any ill effect is not possible.

4) Attacking the Circular Wait Condition

- Provide a global numbering of all the resources.
- Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order.
- A process need not acquire them all at once.

- Circular wait is prevented if a process holding resource n cannot wait for resource m, if m > n.
- No way to complete a cycle.

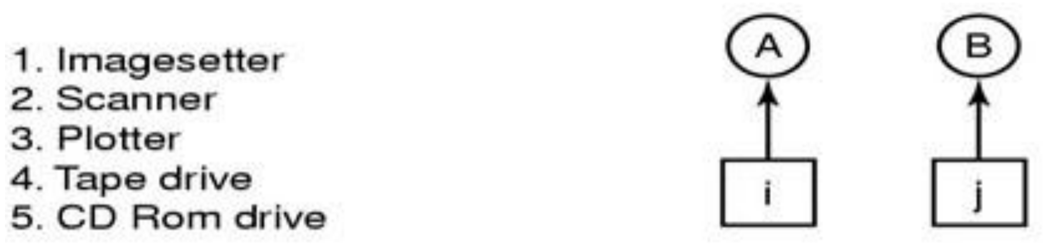

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

Figure 4-10 (a) Numerically ordered resources. (b) A resource graph.

- Assuming *i* and *j* are distinct resources, they will have different numbers.
- If *i* < *j*, then *B* is not allowed to request *i* because that is lower than what it already has.
- If *i* > *j*, then *A* is not allowed to request *j* because that is lower than what it already has.
- Either way, deadlock is impossible.