Chapter 4: Deadlocks

Sukomal Pal, CSE, IIT(BHU)

Chapter 4. Deadlocks

- *Deadlocks: Definition, Necessary and sufficient conditions for Deadlock, Deadlock Prevention*
- *Deadlock Avoidance: Banker’s algorithm, Deadlock detection and Recovery.*

INTRODUCTION

- *Deadlock* symbolizes a lock that is closed and whose key is, as if, lost.
- In OS it refers to a situation where a set of concurrent processes (or threads) perpetually block or starve for want of some resources held by some other processes (or threads) within the set.

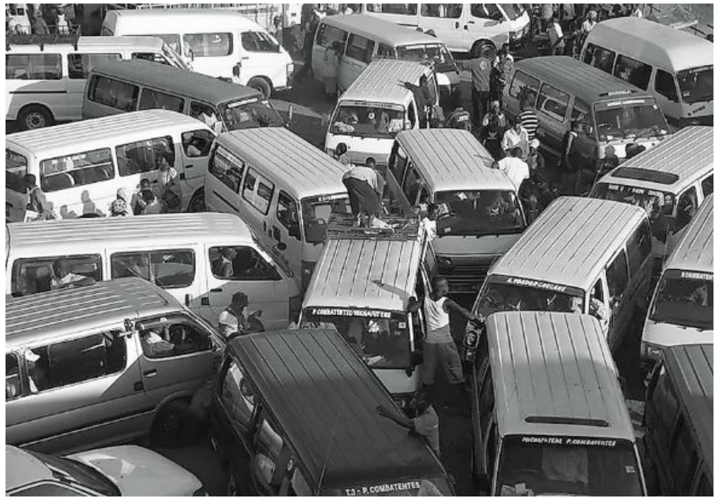


Fig 4.1: Deadlock on road

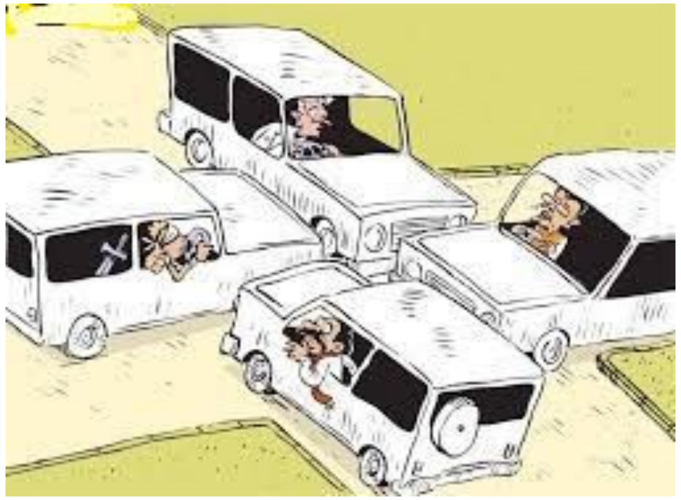


Fig 4.2: Not a Deadlock

INTRODUCTION

- Deadlock is characterized by the following:
  - it is caused for the want of *computing resources* (of any type).
  - nature of *starvation* is *perpetual.*
  - *starvation* occurs to *more than one* processes (or threads) simultaneously.
  - the set of processes (or threads) have dependencies on each other in such a manner that they cannot come out of the perpetual stalemate on their own.

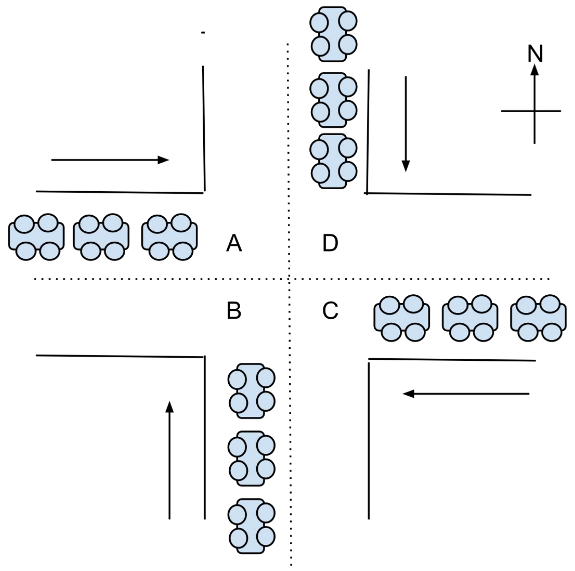


Fig 4.3a: Possibility of deadlock

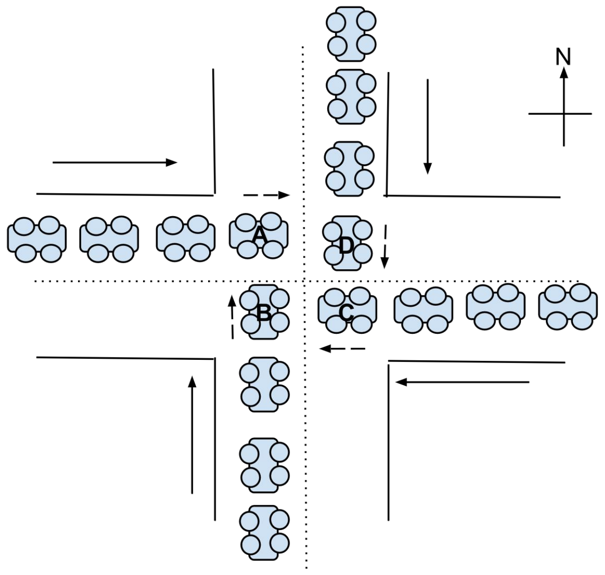


Fig 4.3b: Deadlock

Example of Deadlock

- **Livelock:** Continuous state of change without progress (e.g., two people trying to pass each other in a narrow hallway).
- Deadlock needs external intervention,livelock may resolve naturally.

1. **Dining Philosophers:**

- Each philosopher picks up their left fork first.
- All philosophers wait for the right fork indefinitely.
- Leads to **deadlock** as no philosopher can proceed.

2. **Mutex Locks**

- Two threads (Thread 1 & Thread 2) holding mutexes in different order.
- Each waits for the other's lock -> Deadlock occurs.

Example of Deadlock

```
/****** main process **************/
…
pthread_mutex_t mutex1;
pthread_mutex_t mutex2;
…
pthread_mutex_init(&mutex1, NULL);
pthread mutex init(&mutex2, NULL);
…
```

```
       /****** thread1 ***********/
void *function1(void *param)
{
   pthread_mutex_lock(&mutex1);
  pthread_mutex_lock(&mutex2);
   …
  /** * some work****/
  …

   pthread_mutex_unlock(&mutex2);
   pthread_mutex_unlock(&mutex1);

   pthread_exit(0);
}
```

```
       /******  thread2 ***********/
void *function2(void *param)
{
   pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex1);
   …
  /** * some work****/
  …

   pthread_mutex_unlock(&mutex1);
   pthread_mutex_unlock(&mutex2);

   pthread_exit(0);
}
```

Fig 4.4: Possibility of a deadlock in a multi-threaded program

Example of Deadlock

```
       /****** thread1 ***********/
void *function1(void *param)
{
   while (1){
     pthread_mutex_lock(&mutex1);
     if (pthread_mutex_trylock(&mutex2)) {
              …
             /** * some work****/
           …
           pthread_mutex_unlock(&mutex2);
           pthread_mutex_unlock(&mutex1);
           break;
     }
    else
           pthread_mutex_unlock(&mutex1);
   } /* end of while */
   pthread_exit(0);
}
```

```
       /******  thread2 ***********/
void *function2(void *param)
{
   while (1){
     pthread_mutex_lock(&mutex2);
     if (pthread_mutex_trylock(&mutex1)) {
              …
             /** * some work****/
           …
           pthread_mutex_unlock(&mutex1);
           pthread_mutex_unlock(&mutex2);
           break;
     }
    else
           pthread_mutex_unlock(&mutex2);
   } /* end of while */
   pthread_exit(0);
}
```

Fig 4.5: Possibility of a livelock in a multi-threaded program

Resources

- A computing system can have one or more instances of each resource type, but only a finite number of instances.
- **Types of Resources:**
  - Hardware: Processors, memory, I/O devices.
  - Software: Files, semaphores, sockets.
- **Shareable vs. Non-Shareable Resources:**
  - Shareable: Read-only files (no deadlock risk).
  - Non-Shareable: Mutex locks, I/O devices (deadlock possible).
- **Reusable vs. Consumable Resources:**
  - Reusable: Can be used multiple times (e.g., CPU, memory, semaphores).
  - Consumable: Disappears after use (e.g., messages, signals).

Resources

Resource access

During execution, a thread needs and uses several resources.

The uses always obey the following sequence.

i. *Request:* A thread makes a request to the OS kernel for one or more instances of a resource. If an instance of the resource is not available, The thread waits (or blocks) till it acquires an instance of the resource.

ii. *Use:* Once acquired, it uses the instance of the resource non-shareably.

iii. *Release:* After the use, the thread returns the resource back to the kernel.

Resource Allocation Graph

- **Resource Allocation Process:**
  - Request: Thread requests a resource.
  - Use: Thread holds and uses the resource.
  - Release: Thread returns the resource after use.
- **Resource Allocation Graph (RAG):**
  - Nodes: Threads and Resources.
  - Edges: Request edges (T → R) and Allocation edges (R → T).
  - **Cycle in RAG = Deadlock.**
- **Multiple Instances of Resources:**
  - If multiple instances exist, a resource can be assigned to multiple threads.
  - No cycle = No deadlock.

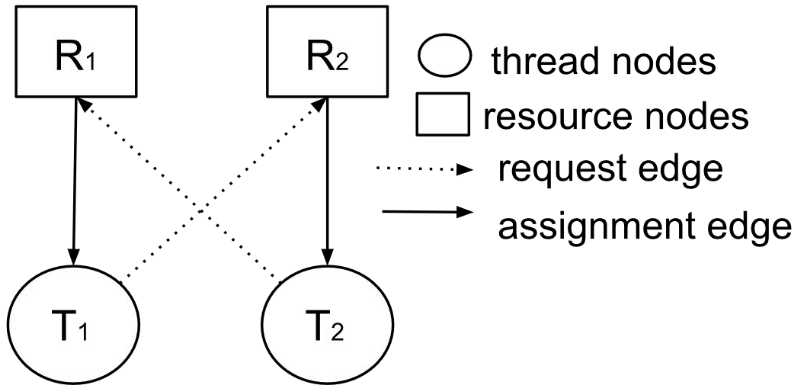


Fig 4.4: Resource Allocation Graph

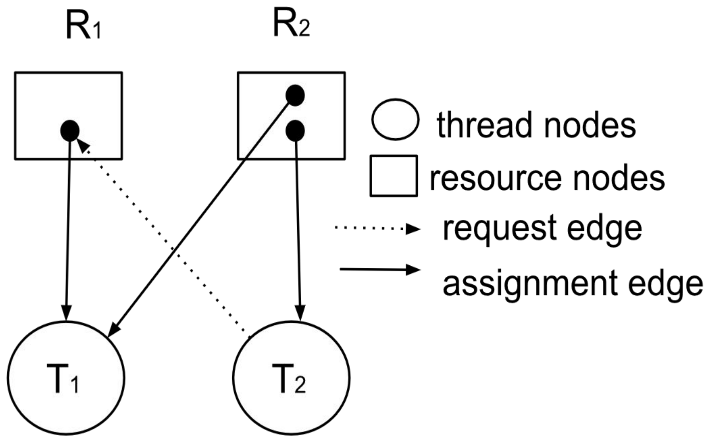


Fig 4.5: RAG With Multiple instances

CONDITIONS of A DEADLOCK

i.● **Mutual Exclusion:** When resources are used by threads non-sharably, then only there may emerge a possibility of deadlock.

- ➢ There should be at least one resource that is used by threads in a mutually exclusive way.

ii. **Hold and Wait:** During execution, threads are allowed to hold one or more resources and, at the same time, request to acquire a few more resources held by other thread(s).

iii. **No Preemption:** None of the resources are preempted from the threads that hold them.

iv. **Unresolvable Circular Wait:** A set of threads T = $\{T_1,\ T_2,\ \ldots\ T_n\}$ hold and wait for resources from a set R = $\{\{R_1, R_2 ... R_n\}$ in such a way that $R_1 \rightarrow T_1, T_1\ \rightarrow R_2,\ R_2 \rightarrow T_2$, ..., $R_n \rightarrow T_n$ and $T_n\ \rightarrow R_1$.

- ➢ Threads and resources make a cycle in the resource allocation graph with assignment and request edges.

Circular Wait

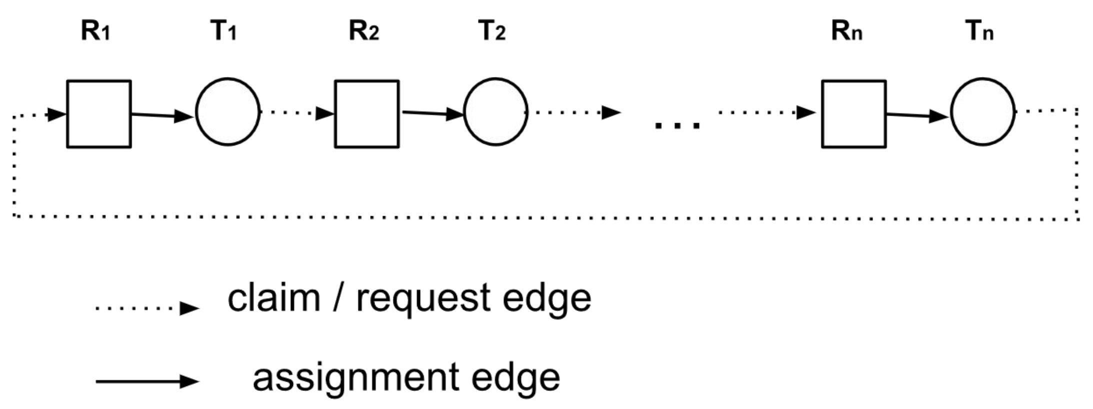


Fig 4.6: Circular Wait(RAG containing Cycle)

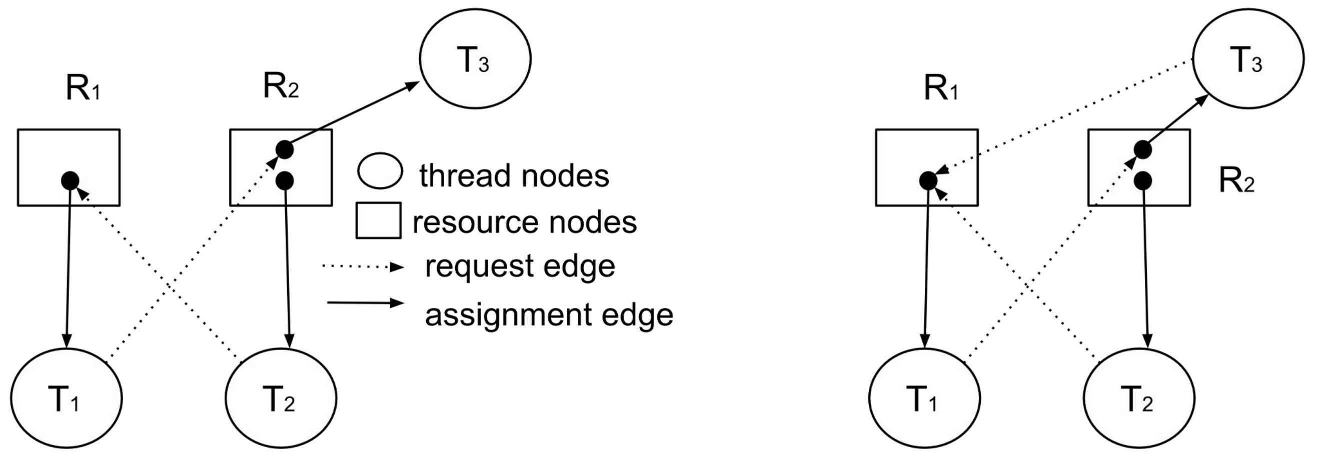


Fig 4.7: RAGs with multiple instances of resources

HANDLING DEADLOCKS

- To stop occurrences of deadlocks, we must make sure that not all the four conditions are true at any point of time
- At least one of the four conditions must be negated

The strategies are clubbed into the following three categories.

1. *Deadlock Prevention*: Requests to resources are monitored and allowed to be made only if all the four conditions are not satisfied simultaneously.
2. *Deadlock Avoidance*: The threads notify their overall need of resources in advance and the resources are allocated only if the allocation is *safe* (it does not lead to the possibility of a deadlock).
3. *Deadlock Detection & Recovery*: Deadlocks are allowed to happen. But they are detected, and appropriate recovery actions are taken.

Deadlock Prevention

1. **Preventing Mutual Exclusion**
2. **Preventing Hold & Wait :** Ensure a process requests all resources at once.
3. **Preventing No Preemption :** Allow resource reallocation from waiting processes.
4. **Preventing Circular Wait :** Order resources numerically and request in ascending order.

Deadlock Avoidance

- They allow the first three conditions (mutual exclusion, hold & wait, and no preemption) to continue.
- Safety of a system of threads is checked before any new allocation of resources.

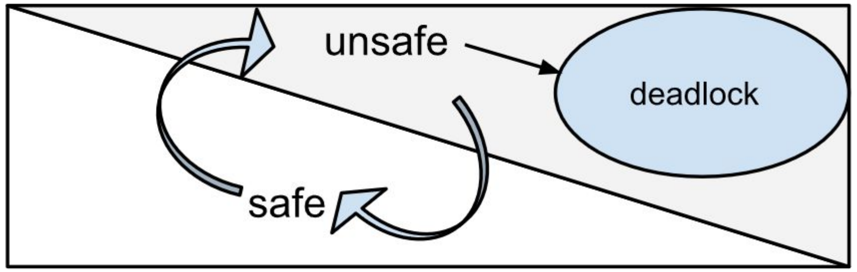


Fig 4.8: Different states for a set of threads

Banker’s Algorithm

- **Resources**: Total available resources are represented by a m-dimensional vector,

Maximum resource needs: Total requirement of all resource types by different threads is represented by a $(n \times m)$

matrix

Resource allocation: Similarly, current allocation of resources at a given time, is also represented by another $(n \times m)$ matrix,

Available resources: As resources are allocated to threads, free and available instances of resources reduce. The current number of available instances of resources is represented by an m-dimensional vector.

Outstanding needs: Once the threads are allocated resources, remaining resource needs of the threads are also represented by an $n \times m$ matrix

Resource requests: Another matrix of $(n \times m)$ dimension represents new (incremental) need of all the threads,

Banker’s Algorithm

- The following relationships and constraints always hold true.

1. $RES[j] \geq MAX[i][j]$ for all i, j (*maximum need of any thread for any resource-type cannot be more than the available number of instances in the system*)
2. $RES[j] \geq \sum_i ALLOC[i][j]$ for all i, j (*sum of allocated instances of any resource-type cannot be more than total number of instances at any moment*)
3. $AVAIL[j] = RES[j] - \sum_i ALLOC[i][j]$ for all i, j (*available number of resource-instances is whatever remains after allocation to all threads*)
4. $NEED[i][j] = MAX[i][j] - ALLOC[i][j] \geq 0$ for all i, j

Banker’s Algorithm

```
bool check_safety (AVAIL, ALLOC, NEED) {

  0. INITIALISATION:
       bool finish_possible[n] = {0, 0, …, 0}; /*flag*/
       bool safe = 1, unsafe = 0;
       int WORK[m];
         /* n = #threads, m=#types of resources */

        WORK = AVAIL;

  1.  Find an index i such that
        (NEED[i] < WORK)&&(finish_possible[i] == 0)
        if not found, goto Step 3.

  2.  WORK = WORK + ALLOC[i];
       finish_possible[i] = 1;
       goto Step 1.

  3. if (finish_possible[i] == 1 for all i) return (safe);
     else return (unsafe);
}
```

```
bool grant_request (AVAIL, ALLOC, REQ[i], MAX) {

  0. bool grant_possible = 1, grant_not_possible = 0;

  1.    NEED[i] = MAX[i] - ALLOC[i];

  2.   if (REQ[i] > NEED[i])
          return (error); /*request is more than max-need */

  3.   if (REQ[i] > AVAIL[i])
          return (grant_not_possible); /* Ti must wait */

  4. /* as if Ti is allocated the resources */
        AVAIL  = AVAIL – REQ[i];
        ALLOC[i]  = ALLOC[i] + REQ[i];
       NEED[i] = NEED[i] –REQ[i];

   5.  if( check_safety(AVAIL, ALLOC, NEED) == safe))
         return (grant_possible);
     else return (grant_not_possible); /* Ti must wait */
}
```

Fig 4.9: Banker’s Algorithm

Banker's Algorithm

- **Example:** *Consider the following snapshot of a system:*

	Allocation				**Max**				**Available**			
	A	B	C	D	A	B	C	D	A	B	C	D
T0	0	0	1	2	0	0	1	2	1	5	2	0
T1	1	0	0	0	1	7	5	0				
T2	1	3	5	4	2	3	5	6				
T3	0	6	3	2	0	6	5	2				
T4	0	0	1	4	0	6	5	6				

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix ***Need****?*

b. Is the system in a safe state?

c. If a request from thread T1 arrives for (0,4,2,0), can the request be granted immediately?

Deadlock Detection & Recovery

- OS detects deadlock and recovers from it, Detection based on **circular wait condition**

Case 1: Each resource has a single instance

- Use a **Wait-For Graph (WFG)**
- A cycle in WFG **confirms deadlock**
- Cycle detection time complexity: **$O(n^2)$**

Case 2: Resources have multiple instances

- Uses a **modified Banker's algorithm**
- Steps:
  - Threads with no allocated resources are ignored ($O(n)$)
  - Find a thread that can complete ($O(mn)$)
  - Simulate its completion and update availability ($O(mn^2)$)
  - If all threads finish, no deadlock; otherwise, deadlock exists
  - Time complexity: **$O(mn^2)$**

Deadlock Detection & Recovery

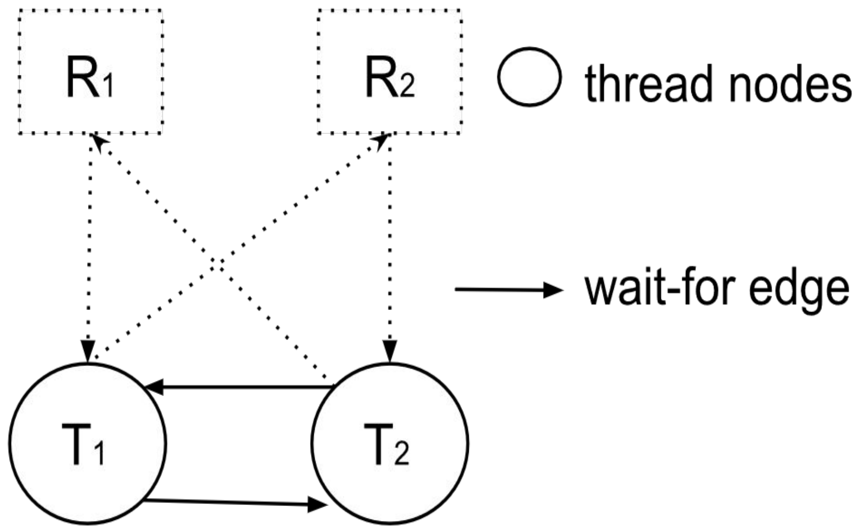


Fig 4.10: RAG**(Fig 4.4)** to Wait for Graph

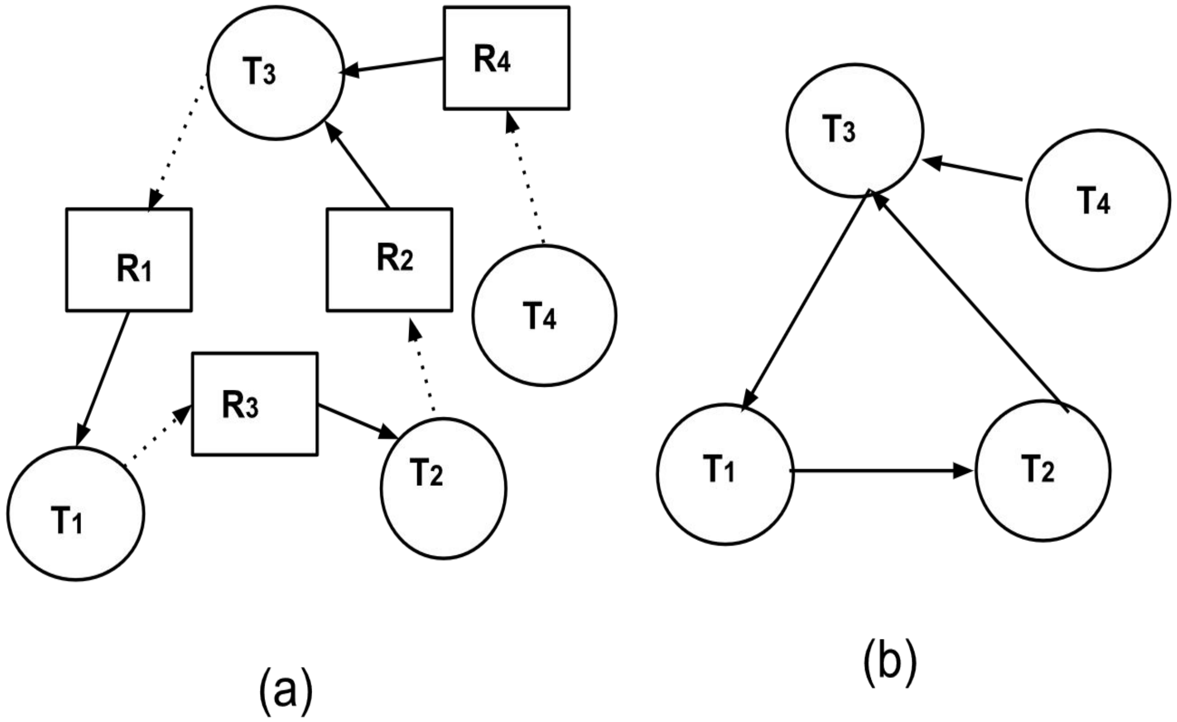


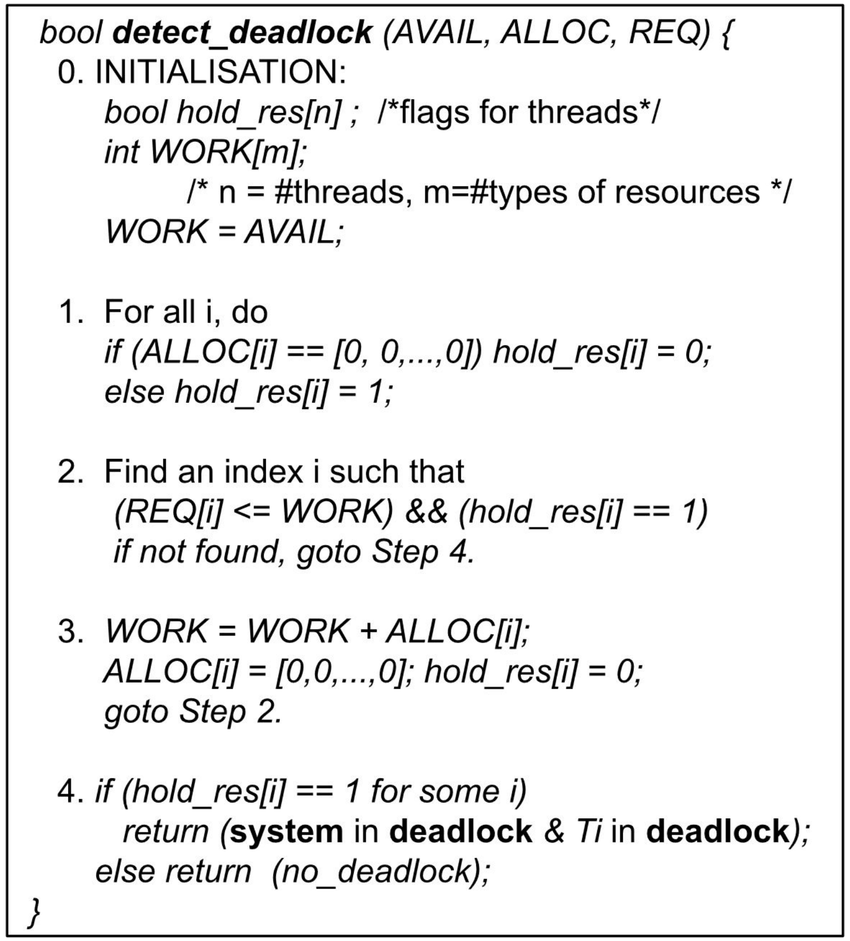
Fig 4.11: (a) RAG to (b) wait for graph

Deadlock Detection & Recovery

Resources with one or more instances:

- When resources have multiple instances, mere presence of a cycle is not a confirmation of a deadlock.
- Use a deadlock detection algorithm

Deadlock Detection & Recovery



```
bool detect_deadlock (AVAIL, ALLOC, REQ) {
  0. INITIALISATION:
       bool hold_res[n] ;  /*flags for threads*/
       int WORK[m];
             /* n = #threads, m=#types of resources */
       WORK = AVAIL;

  1.  For all i, do
       if (ALLOC[i] == [0, 0,...,0]) hold_res[i] = 0;
       else hold_res[i] = 1;

  2.  Find an index i such that
        (REQ[i] <= WORK) && (hold_res[i] == 1)
        if not found, goto Step 4.

  3.  WORK = WORK + ALLOC[i];
       ALLOC[i] = [0,0,...,0]; hold_res[i] = 0;
       goto Step 2.

  4. if (hold_res[i] == 1 for some i)
         return (system in deadlock & Ti in deadlock);
      else return  (no_deadlock);
}
```

Fig 4.12: Deadlock detection Algorithm

Recovery from Deadlock

Process Termination

- **Terminate all processes** in deadlock (brute-force, costly)
- **Terminate one process at a time** until deadlock is resolved
- Selection criteria:
    - Priority level (high, moderate, low)
    - Execution time (how long it has run)
    - Resource holding status (number of resources held)

Recovery from Deadlock

Resource Preemption

- **Forcibly reassign resources** from blocked threads to others
  - **Victim Selection**: which resources are to be chosen and from which threads?
  - **Rollback**: Restore the victim process to a safe state for future execution.
  - **Starvation Prevention**: Avoid repeated selection of the same victims.

Reference

[1] "OPERATING SYSTEMS", Author: Dr. Sukomal Pal Associate Professor Department of Computer Science & Engineering IIT (BHU), Varanasi, UP