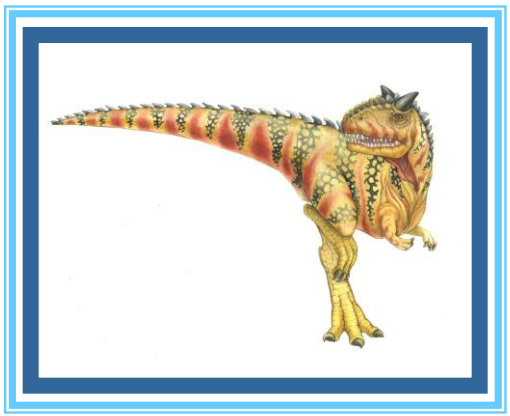
Chapter 8: Deadlocks

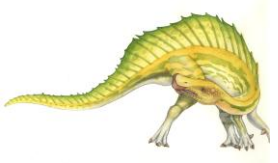


Silberschatz, Galvin and Gagne ©2018



Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock





Chapter Objectives

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock

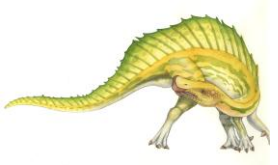


Troy University Resources



System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**





Deadlock with Semaphores

- Data:
  - A semaphore S_1 initialized to 1
  - A semaphore S_2 initialized to 1
- Two threads T_1 and T_2
- T_1:

```
wait(S1)
wait(S2)
```

- T_2:

```
wait(S2)
wait(S1)
```



Troy University Resources



Deadlock with Mutex Locks

If thread 1 holds a while thread 2 holds b, each can wait forever for the other lock.

```
/* Shared mutexes a and b are initialized before use. */
/* thread 1 */                          /* thread 2 */
pthread_mutex_lock(&a);                  pthread_mutex_lock(&b);
pthread_mutex_lock(&b);                  pthread_mutex_lock(&a);
/* use both resources */                /* use both resources */
pthread_mutex_unlock(&b);                pthread_mutex_unlock(&a);
pthread_mutex_unlock(&a);                pthread_mutex_unlock(&b);
```

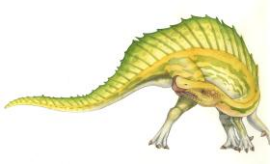
This example demonstrates a possible deadlock. The two threads do not deadlock on every possible schedule.





Four Necessary Deadlock Conditions

- Mutual exclusion: at least one needed resource permits only one user at a time.
- Hold and wait: a participant holds resources while requesting more.
- No preemption: the resources cannot simply be taken from their holders.
- Circular wait: a cycle exists in which each participant waits for a resource held by the next.
- All four are necessary for resource deadlock in this model. With multiple instances, a graph cycle alone is insufficient to prove deadlock.



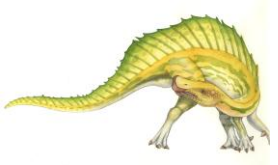
Troy University Resources



Resource-Allocation Graph

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

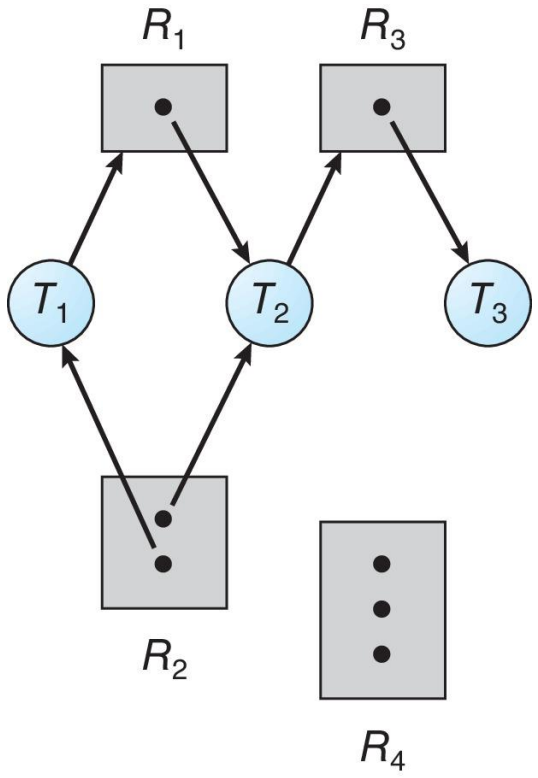
- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$





Resource Allocation Graph Example

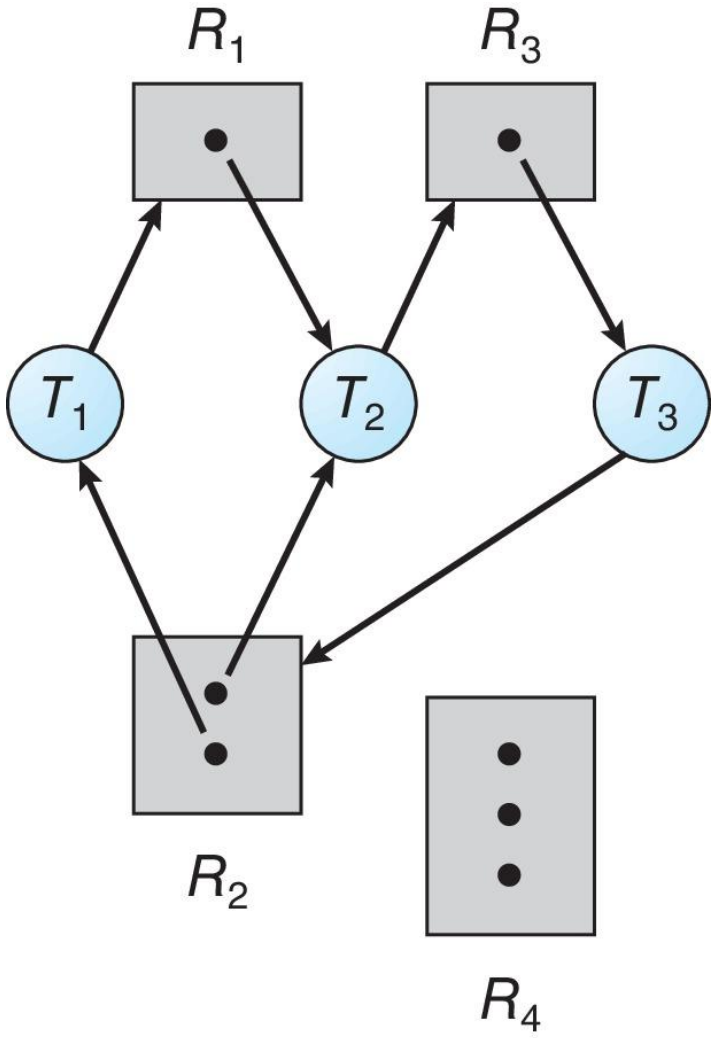
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3





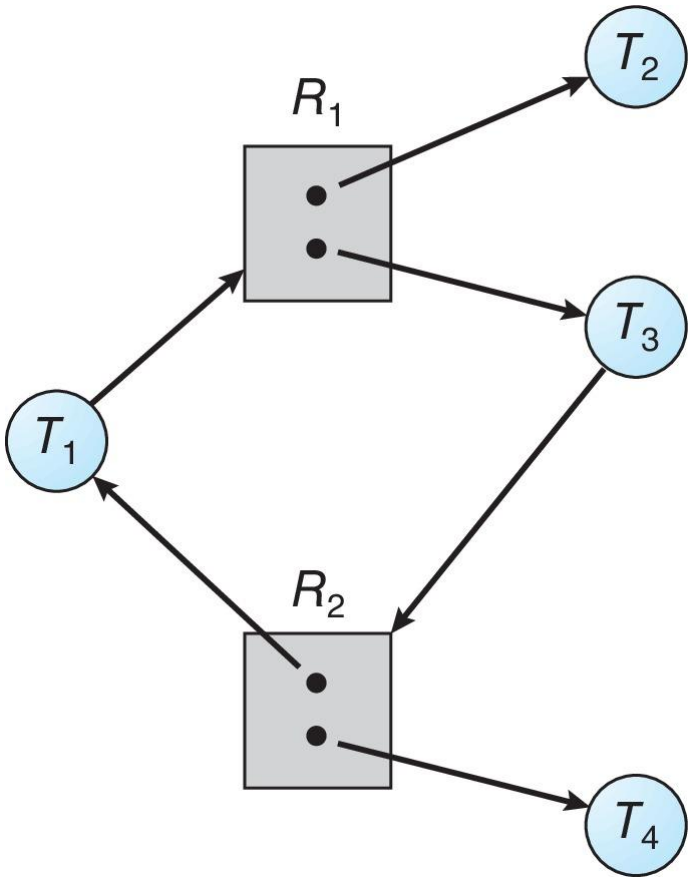
Troy University Resources

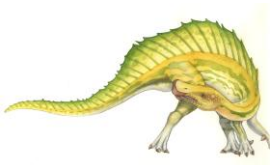
Resource Allocation Graph with a Deadlock





Graph with a Cycle But no Deadlock



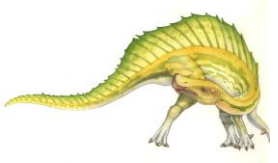


Troy University Resources



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock





Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.



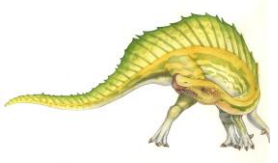
Troy University Resources



Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a thread requests a resource, it does not hold any other resources
  - Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it.
  - Low resource utilization; starvation possible





Deadlock Prevention

- Resource preemption can work when resource state can be saved and restored safely.
- If a request cannot be granted, such a policy may reclaim held resources and retry later.
- A mutex protecting partly updated data generally cannot be taken away safely. Recovery requires restoring the protected state.
- A total order on lock acquisition breaks circular wait. Every participating path must follow the same order.



Troy University Resources



Circular Wait

- Give every mutex a stable, unique order.
- All threads acquire multiple locks in increasing order.
- If first_mutex has ID 1 and second_mutex has ID 5, acquire 1 before 5.
- The illustrated acquisition of 5 before 1 violates that order.

A consistent global order prevents a cycle in the lock-acquisition graph.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```





Transfers Between Accounts

Pseudocode: both A-to-B and B-to-A transfers use the same lock order, regardless of transfer direction.

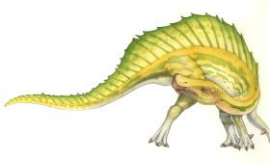
```
transfer(from, to, amount):
    if from == to:
        return
    first  = account with smaller stable unique ID
    second = account with larger stable unique ID
    lock(first.mutex)
    lock(second.mutex)
    withdraw(from, amount)
    deposit(to, amount)
    unlock(second.mutex)
    unlock(first.mutex)
```





Deadlock Avoidance

- An avoidance algorithm evaluates whether granting a request preserves a safe allocation state.
- Banker's algorithm requires each participant's declared maximum demand for each resource type.
- Available, Allocation, and Max describe the current state. Need = Max - Allocation.
- Safe-state reasoning applies to declared resource demands and the assumption that a participant can finish once its remaining needs are met.





Safe State

- When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<T_1, T_2, \ldots, T_n>$ of ALL the threads in the systems such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources + resources held by all the T_j, with $j < I$
- That is:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on

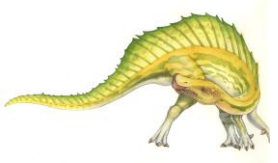


Troy University Resources



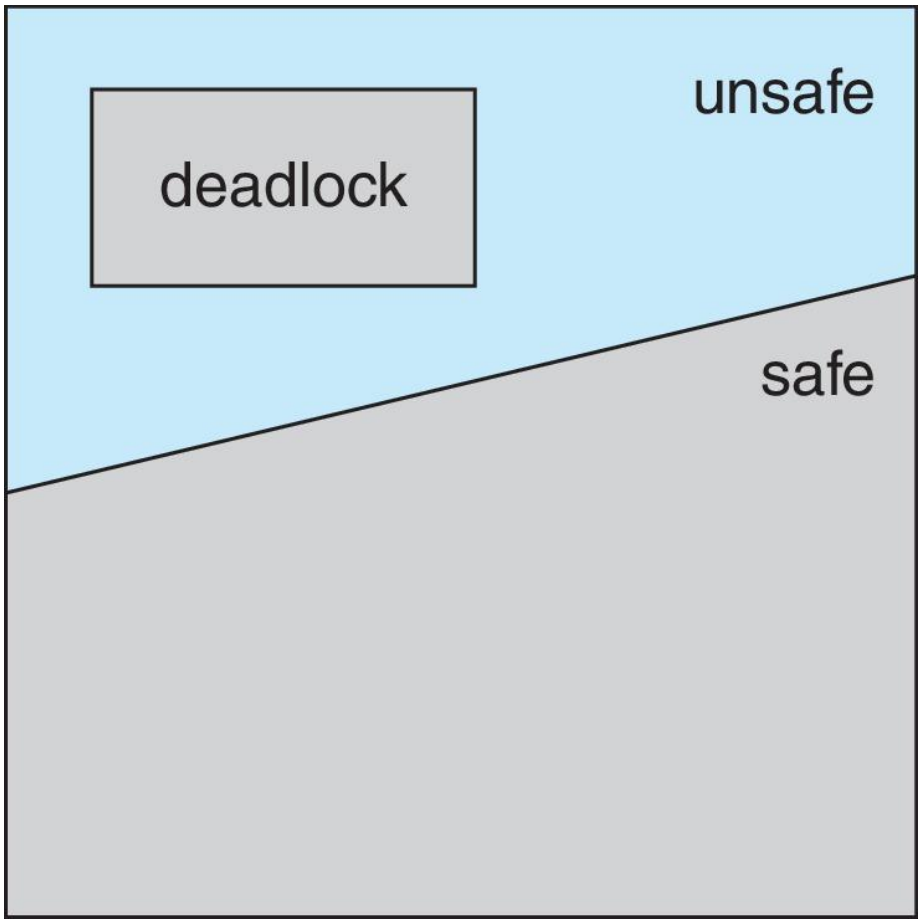
Basic Facts

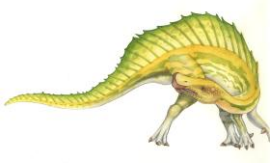
- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.





Safe, Unsafe, Deadlock State



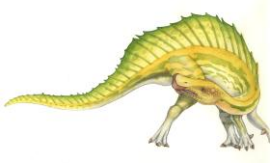


Troy University Resources



Avoidance Algorithms

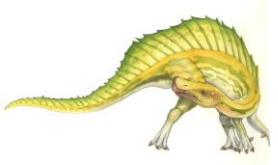
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm





Resource-Allocation Graph Scheme

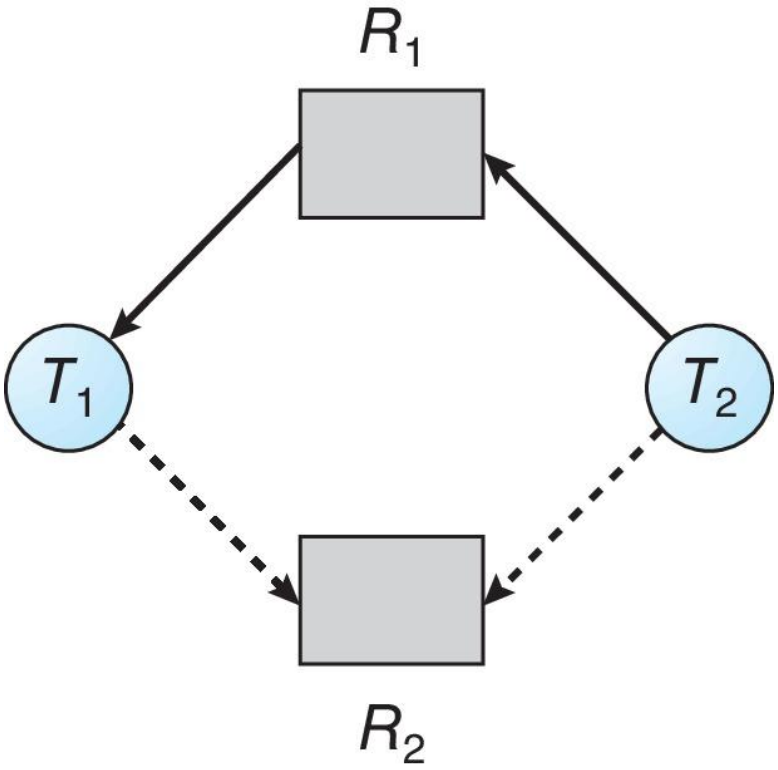
- **Claim edge Ti → Rj means thread Ti may request resource Rj; it is represented by a dashed line**
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource is released by a thread, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

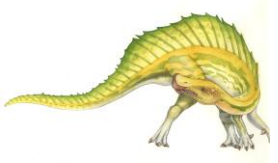


Troy University Resources



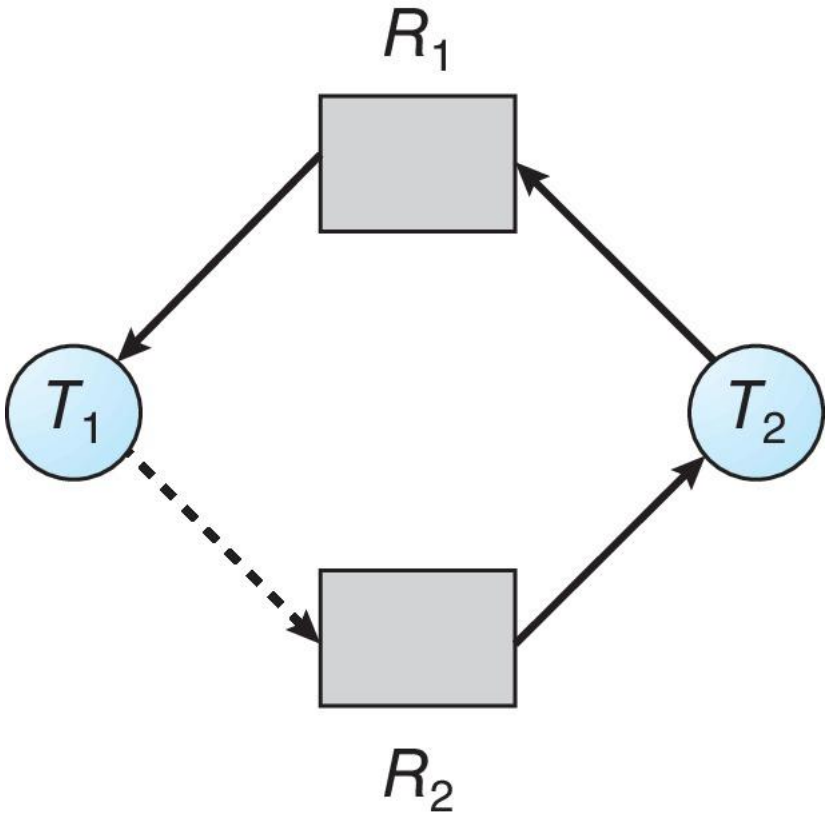
Resource-Allocation Graph

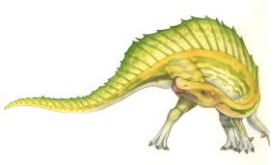






Unsafe State In Resource-Allocation Graph







Resource-Allocation Graph Algorithm

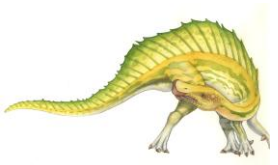
- Suppose that thread T_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph





Banker's Algorithm

- Multiple instances of resources
- Each thread must a priori claim maximum use
- When a thread requests a resource, it may have to wait
- When a thread gets all its resources it must return them in a finite amount of time



Troy University Resources



Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If Max [i,j] = k, then process T_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then T_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need$[i,j] = k, then T_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$





Safety Algorithm

- Initialize Work = Available and Finish[i] = false for every thread.
- Find an unfinished thread i whose Need[i] is componentwise at most Work.
- Simulate its completion: Work += Allocation[i], set Finish[i] = true, and repeat.
- If every thread finishes, the discovered order is a safe sequence.
- If unfinished threads remain and none can proceed, this state is unsafe. Unsafe does not mean already deadlocked.

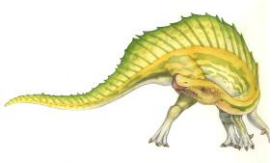


Troy University Resources



Resource-Request Algorithm

- Check Request[i] ≤ Need[i]. Exceeding the maximum claim is an error.
- Check Request[i] ≤ Available. Otherwise the thread waits for resources.
- Temporarily update Available -= Request[i], Allocation[i] += Request[i], and Need[i] -= Request[i].
- Run the safety algorithm. Commit the allocation if the state is safe.
- If unsafe, restore all three original values and make the requester wait.



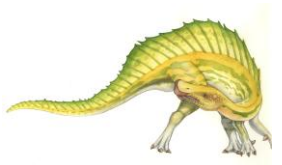


Banker's Algorithm: Initial State

Total resources = (10, 5, 7). Available = (3, 3, 2). All comparisons are componentwise.

Thread	Allocation (A B C)	Max (A B C)
T0	0 1 0	7 5 3
T1	2 0 0	3 2 2
T2	3 0 2	9 0 2
T3	2 1 1	2 2 2
T4	0 0 2	4 3 3

Allocated totals = (7, 2, 5). Allocated + Available equals the total resources.





Banker's Algorithm: Need

One safe sequence is ⟨T1, T3, T4, T2, T0⟩.

Thread	Need = Max - Allocation
T0	7 4 3
T1	1 2 2
T2	6 0 0
T3	0 1 1
T4	4 3 1

Work evolves: (3,3,2), (5,3,2), (7,4,3), (7,4,5), (10,4,7), (10,5,7).





T1 Requests (1, 0, 2)

Fits both Need and Available. After the grant, Available = (2, 3, 0).

Thread	Allocation (A B C)	Need (A B C)
T0	0 1 0	7 4 3
T1	3 0 2	0 2 0
T2	3 0 2	6 0 0
T3	2 1 1	0 1 1
T4	0 0 2	4 3 1

A safe sequence is ⟨T1, T3, T4, T0, T2⟩, so the request can be granted.



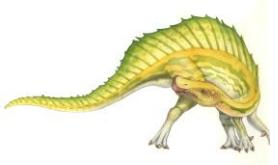


Further Requests After the T1 Grant

Start each check separately from the state on the preceding slide. No other thread has finished.

Independent request	Available check	Decision
T4 requests (3, 3, 0)	3 units of A exceed the 2 available.	Wait; do not allocate.
T0 requests (0, 2, 0)	Fits Available, but tentative Work = (2, 1, 0).	Unsafe: no thread can finish. Roll back and wait.

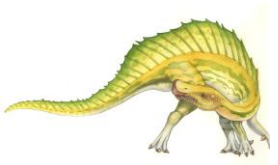
Resource availability is necessary for a grant. The safety test is also required.





Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

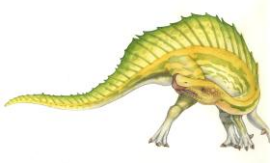


Troy University Resources



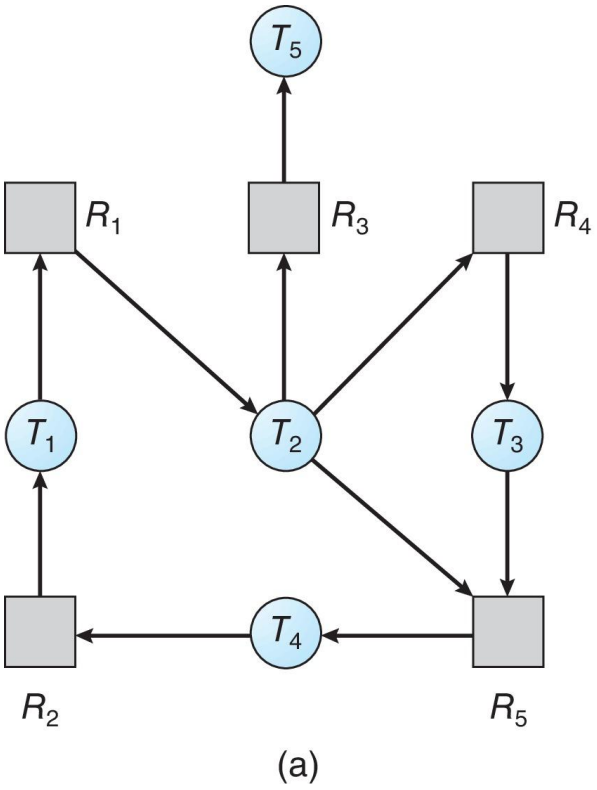
Wait-for Graph: Single Instances

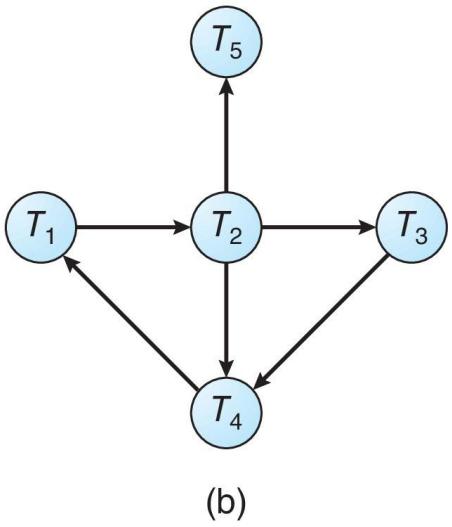
- For one instance of every resource type, collapse resource vertices into thread-to-thread wait edges.
- Ti → Tj means Ti waits for a resource held by Tj.
- In this model, a cycle in the wait-for graph identifies a deadlock.
- A depth-first graph search takes O(V + E) with adjacency lists. A dense graph can have $O(n^2)$ edges.





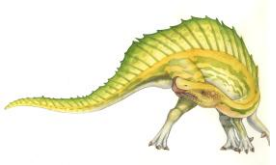
Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph



Troy University Resources



Several Instances of a Resource Type

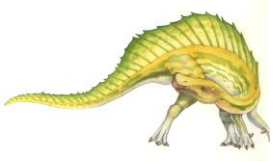
- **Available**: A vector of length ***m*** indicates the number of available resources of each type
- **Allocation**: An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each thread.
- **Request**: An ***n*** **x** ***m*** matrix indicates the current request of each thread. If ***Request*** **[*i*][*j*] = *k***, then thread T_i is requesting ***k*** more instances of resource type R_j.





Detection Algorithm: Initialization

- Work = Available.
- If Allocation[i] is the all-zero vector, set Finish[i] = true. Otherwise set Finish[i] = false.
- Find an unfinished thread i with Request[i] ≤ Work, using its current outstanding request.
- If one exists, continue with the completion step on the next slide. Otherwise stop the search.

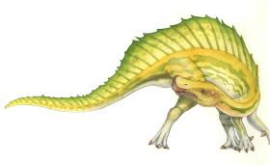


Troy University Resources



Detection Algorithm: Completion

- For the selected thread, set Work += Allocation[i] and Finish[i] = true. Search again.
- When no further completion is possible, unfinished resource holders form the detected deadlocked set in this model.
- Threads holding no resources can still be blocked behind that set, even though initialization marked them finished.
- The straightforward matrix algorithm takes $O(m \times n^2)$ time for n threads and m resource types.



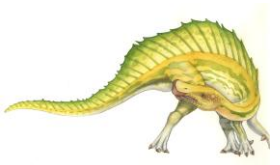


Deadlock Detection: Initial State

Available = (0, 0, 0). Total resources = (7, 2, 6).

Thread	Allocation (A B C)	Request (A B C)
T0	0 1 0	0 0 0
T1	2 0 0	2 0 2
T2	3 0 3	0 0 0
T3	2 1 1	1 0 0
T4	0 0 2	0 0 2

Completion order ⟨T0, T2, T3, T1, T4⟩ finishes all threads, so no deadlock is detected.



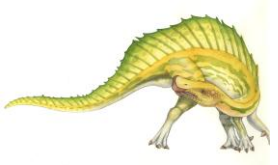


Detection After T2 Requests One C

Keep the same allocations and Available = (0, 0, 0), but replace T2's request.

Thread	Current Request (A B C)
T0	0 0 0
T1	2 0 2
T2	0 0 1
T3	1 0 0
T4	0 0 2

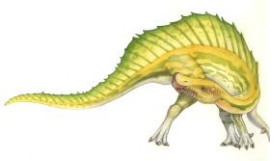
Only T0 can finish, giving Work = (0, 1, 0). T1, T2, T3, and T4 remain deadlocked.





When to Run Deadlock Detection

- Detection can run periodically, when a request cannot be satisfied, or when progress falls unexpectedly.
- Frequent detection finds problems sooner but consumes more computation.
- Recovery cost depends on the actual dependency graph, held resources, and work that must be undone.
- Removing one victim per disjoint cycle is a simplified case. Overlapping cycles and multiple resource instances need analysis of the whole state.

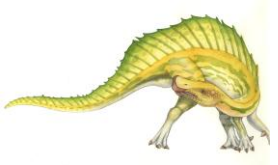


Troy University Resources



Recovery by Termination

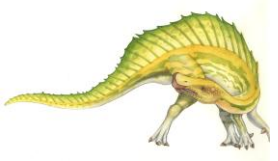
- Abort all deadlocked participants, or choose victims one at a time and detect again.
- Choose victims using priority, work already done, resources held, and the cost of restarting.
- Terminating an arbitrary thread can leave locks held or shared data inconsistent.
- A practical recovery mechanism needs cleanup, transaction rollback, process isolation, or another way to restore a consistent state.





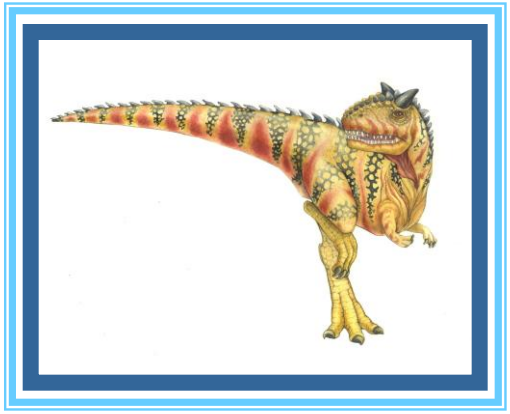
Recovery by Resource Preemption

- Select a victim whose recoverable resources can be reclaimed at an acceptable cost.
- Restore a consistent checkpoint or roll back a transaction before restarting.
- A resource must support safe preemption. A mutex alone cannot undo a half-completed update.
- Track previous victim choices so the same participant is not repeatedly selected and starved.



Troy University Resources

End of Chapter 8



Silberschatz, Galvin and Gagne ©2018