# Deadlock

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 37

Nov 24, 2014

Reading: OSC Ch 7 (deadlock)

# Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource
- Hold and wait (incremental allocation)
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - e.g, There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads,
    - $T_1$ is waiting for a resource that is held by $T_2$
    - $T_2$ is waiting for a resource that is held by $T_3$, …
    - $T_n$ is waiting for a resource that is held by $T_1$

# Methods for Handling Deadlocks

- Deadlock prevention: design system to ensure that it will *never* enter a deadlock
  - E.g., monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
    - E.g., Java JMX findDeadlockedThreads()
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ignore the problem and hope that deadlocks never occur in the system
  - Used by most operating systems, including UNIX
  - Resort to manual version of recovery

# Techniques for Deadlock Prevention

- Eliminate the Shared Resources
  - E.g., give each Philosopher two chopsticks, open the other bridge lane, …
  - Or at least two virtual chopsticks
  - OK, if sharing was do to resource limitations
  - Not if sharing is due to true interactions
    - Must modify Directory Structure AND File Index AND the Block Free list
    - Must enter the intersection to turn left

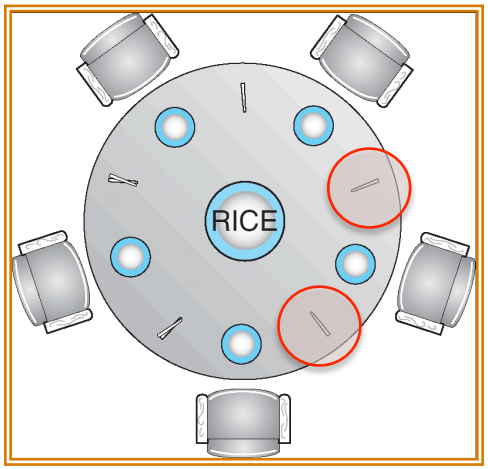
# Techniques for Deadlock Prevention

- Eliminate the Shared Resources
- Eliminate the Mutual Exclusion
  - E.g., many processes can have read-only access to file
  - But still need mutual-exclusion for writing

# Techniques for Deadlock Prevention

- Eliminate the Shared Resources
- Eliminate the Mutual Exclusion
- Eliminate Hold-and-Wait

# Acquire all resources up front

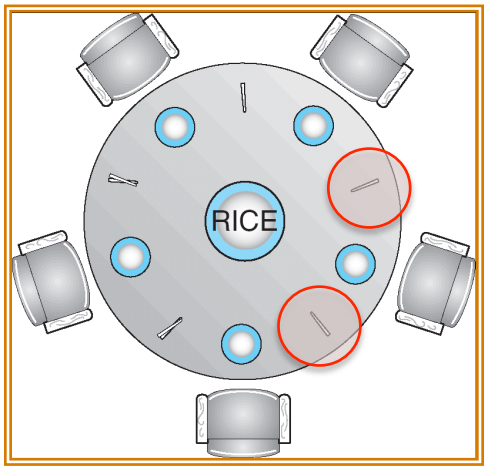



- Philosopher grabs for both chopsticks at once
  - If not both available, don't pickup either, try again later
- Phone call signaling attempts to acquire resources all along the path, "busy" if any point not available
- File Systems: lock {dir. Structure, file index, free list}
  - Or the piece of each in a common block group
- Databases: lock all tables touched by the query
- Hard in general, but often natural in specific cases

# Techniques for Deadlock Prevention

- Eliminate the Shared Resources
- Eliminate the Mutual Exclusion
- Eliminate Hold-and-Wait
- Permit pre-emption

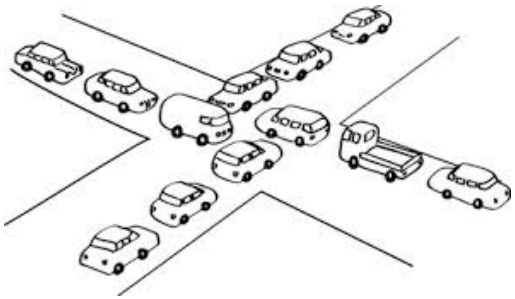
# Incremental Acquisition with Pre-emption

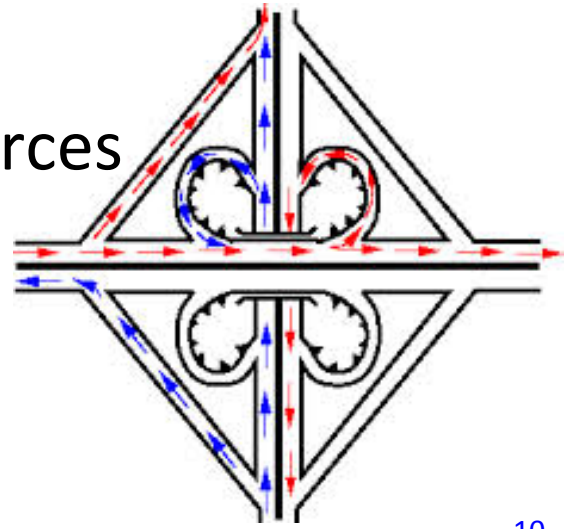



- Philosopher grabs one, goes for other, if not available, releases the first
  - Analogous for sequence of system resources
- Danger of turning deadlock into livelock
  - Everyone is grabbing and releasing, no one every gets two
- Works great at low utilization
  - Potential for thrashing (or failure) as utilization increases
- Similar to CSMA (carrier sense multiple access) in networks
- Randomize the back-off

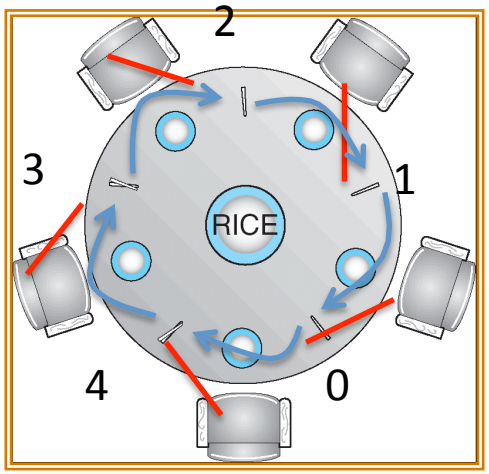
# Techniques for Deadlock Prevention

- Eliminate the Shared Resources
- Eliminate the Mutual Exclusion
- Eliminate Hold-and-Wait
- Permit pre-emption
- Eliminate the creation of circular wait
  - Dedicated resources to break cycles
  - Ordering on the acquisition of resources



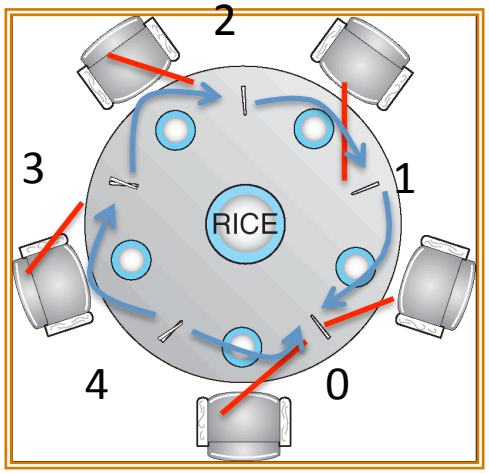



# Cyclic Dependence of resources




- Suppose everyone grabs left first
- Acquisition of the right chopstick depends on the acquisition of the left one
- A cycle of dependences forms

# Ordered Acquisition to prevent cycle from forming




- Suppose everyone grabs lowest first
- Dependence graph is acyclic
- Someone will fail to grab chopstick 0 !
- How do you modify the rule to retain fairness ?
- OS: define ordered set of resource classes
  - Acquire locks on resources in order
  - Page Table => Memory Blocks => …

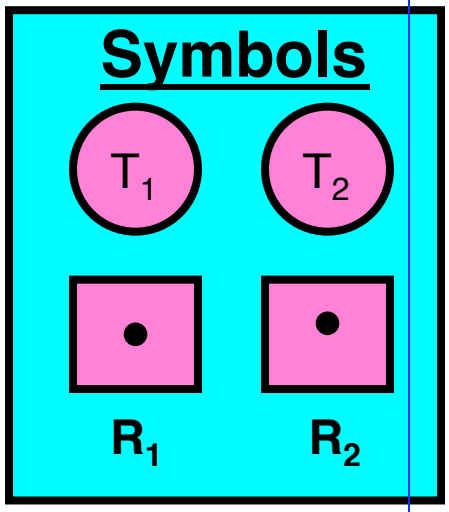
# Deadlock Detection

- There are threads that never become ready
- Are they deadlocked or just … ?

# A Simple Resource Graph

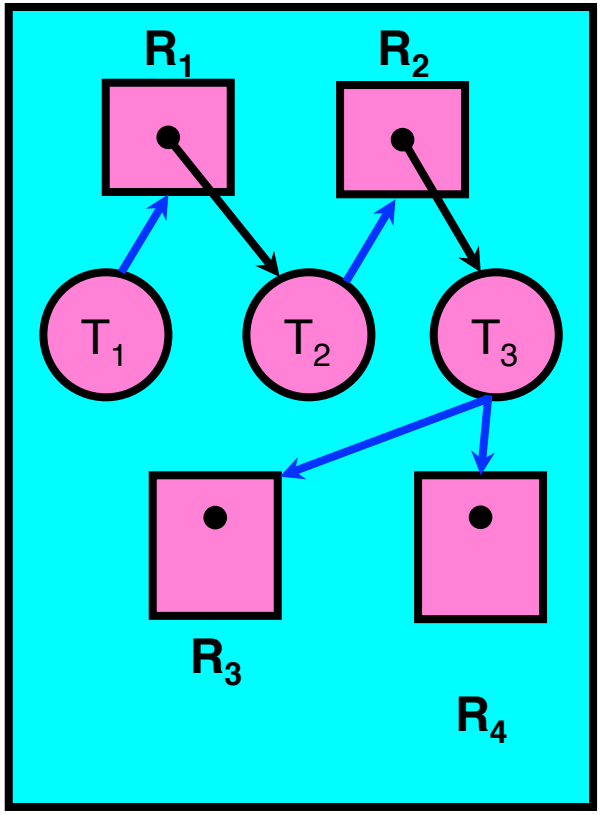
- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

    *locks in this case*
  - Each thread utilizes a resource as follows:
    - `Request() / Use() / Release()`

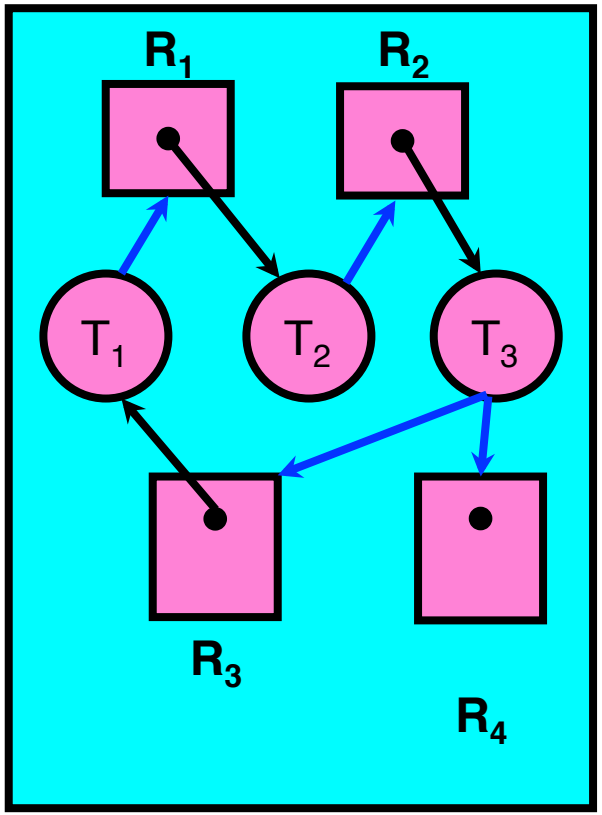



- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - owns edge – directed edge $T_i \rightarrow R_j$
  - waiter edge – directed edge $R_j \rightarrow T_i$

# Resource Allocation Graph Examples



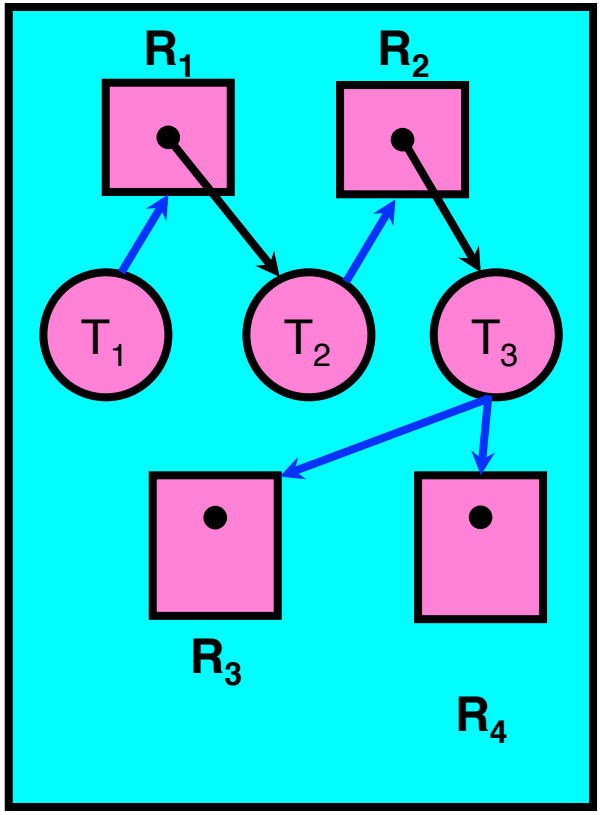

Simple Resource Allocation Graph



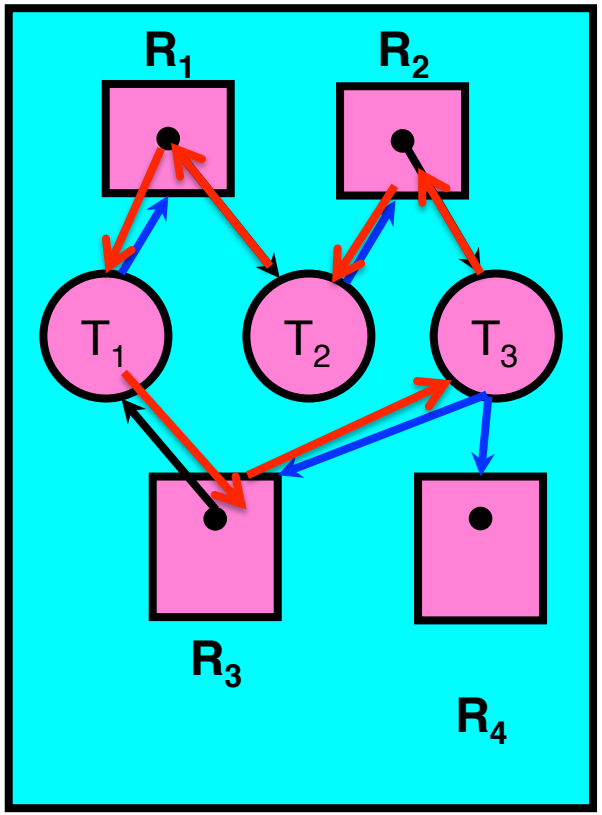

Deadlocked Resource Allocation Graph

# How would you look for cycles?

# Resource Allocation Graph Examples



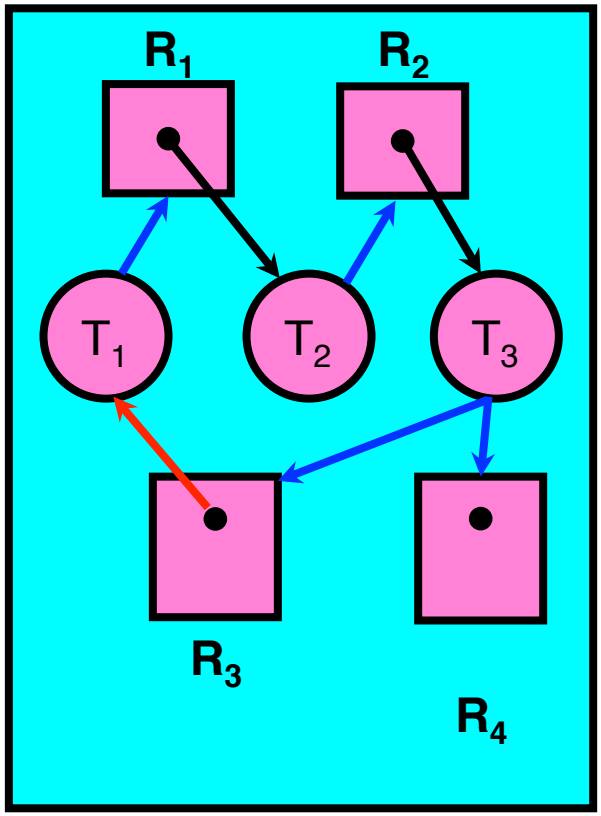

Simple Resource Allocation Graph




Deadlocked Resource Allocation Graph

# How would avoid cycle creation ?

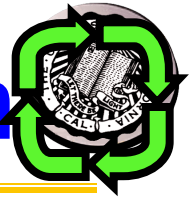
- On attempt to acquire an owned lock
  - Check to see if adding the request edge would create a cycle



# More General Case

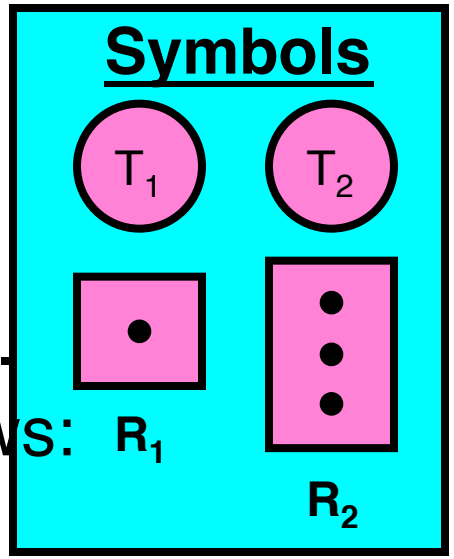
- Each resources has a capacity (# instances)
- Each thread requests a portion of each resource

# General Resource-Allocation Graph



- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$

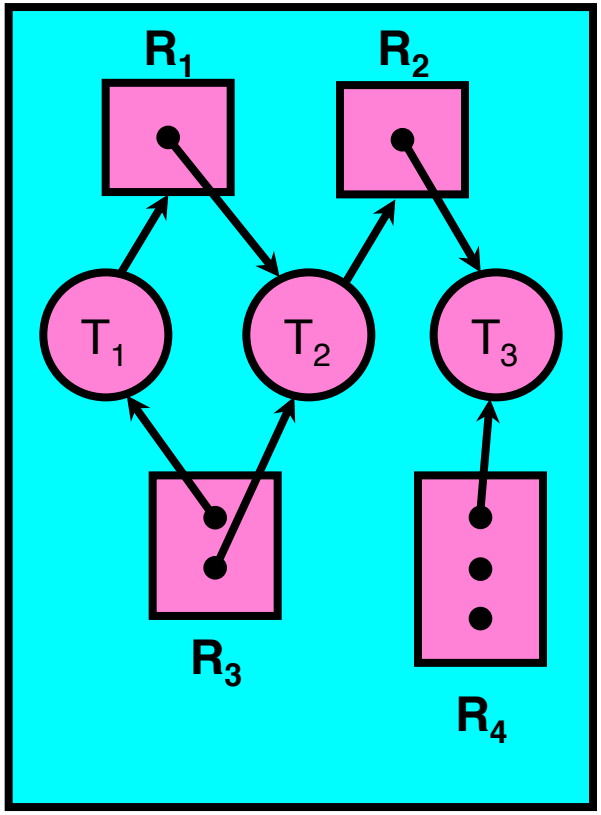
    *CPU cycles, memory space, I/O devices*
  - Each resource type $R_i$ has $W_i$ instances.
  - Each thread utilizes a resource as follows:
    - `Request() / Use() / Release()`



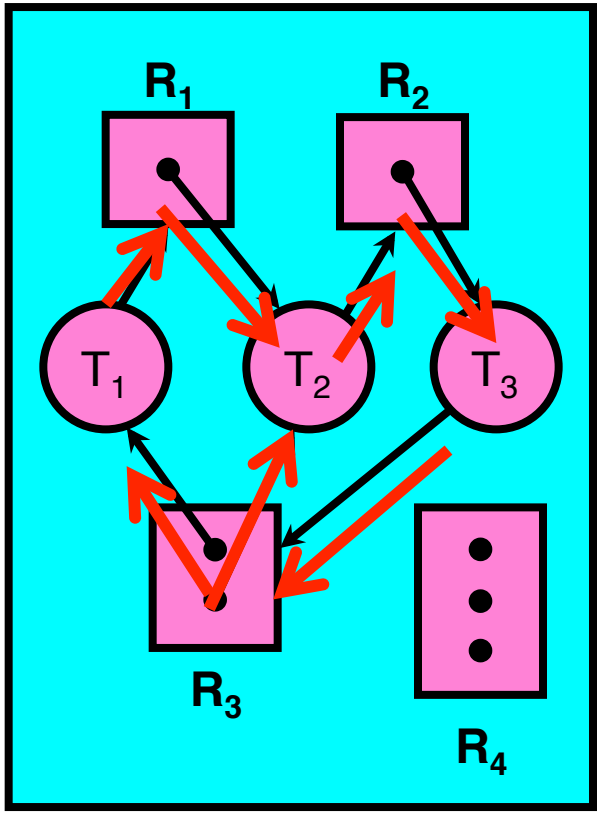

- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge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

# Resource Allocation Graph Examples

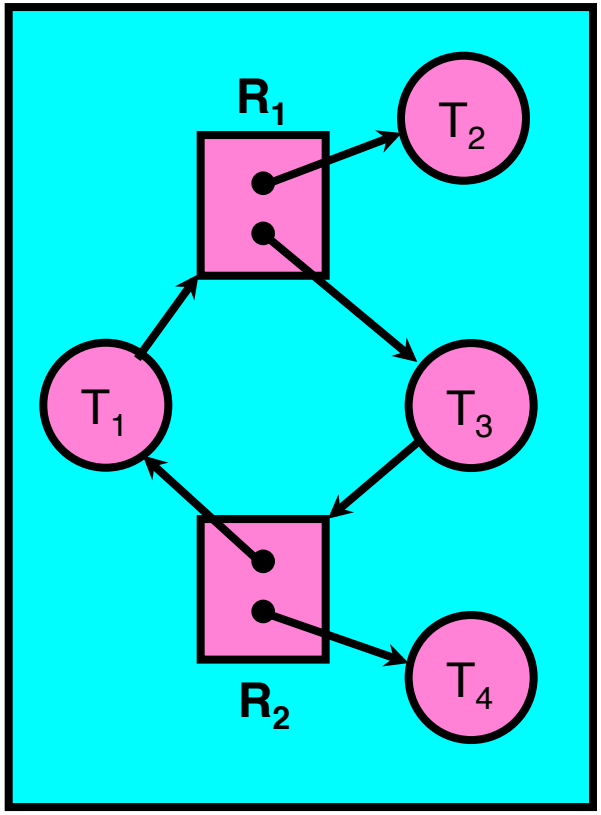
- Recall:
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$




**Simple Resource Allocation Graph**




**Allocation Graph With Deadlock**




**Allocation Graph With Cycle, but No Deadlock**

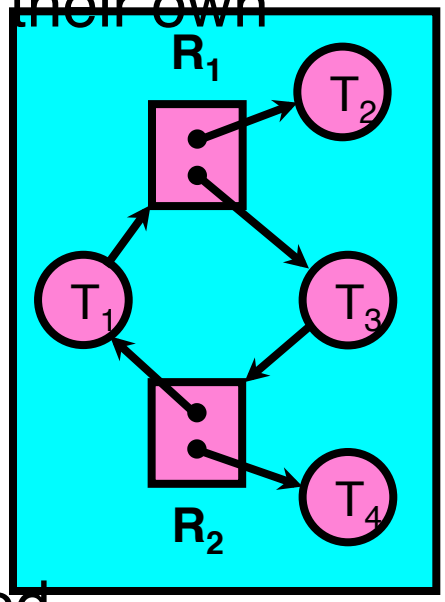
# Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

| | |
|---|---|
| [FreeResources]: | Current free resources each type |
| [$Request_X$]: | Current requests from thread X |
| [$Alloc_X$]: | Current resources held by thread X |

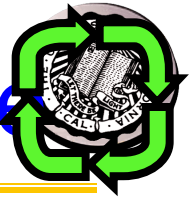
  - See if tasks can eventually terminate on ~~their own~~

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request_node] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
        }
    }
} until(done)
```



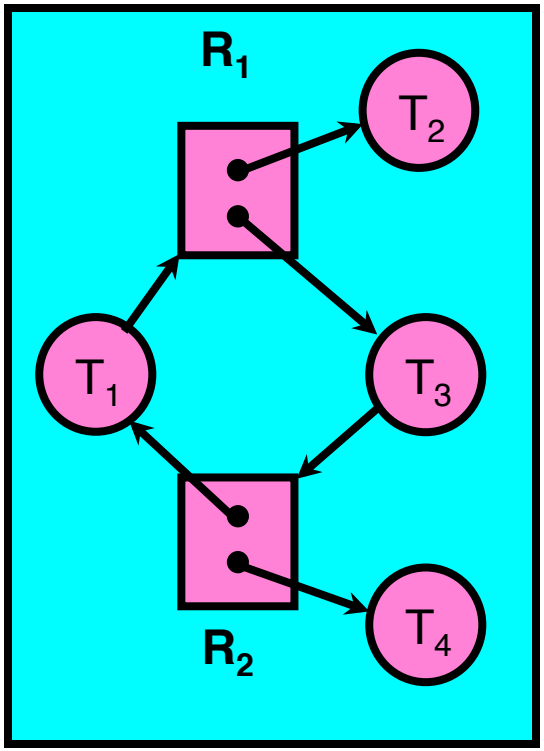

  - Nodes left in UNFINISHED ⇒ deadlocked

# Deadlock Detection Algorithm Example

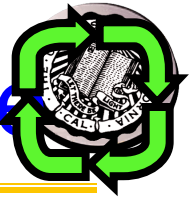


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Requestnode] <= [Avail]) {
      remove node from UNFINSHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```



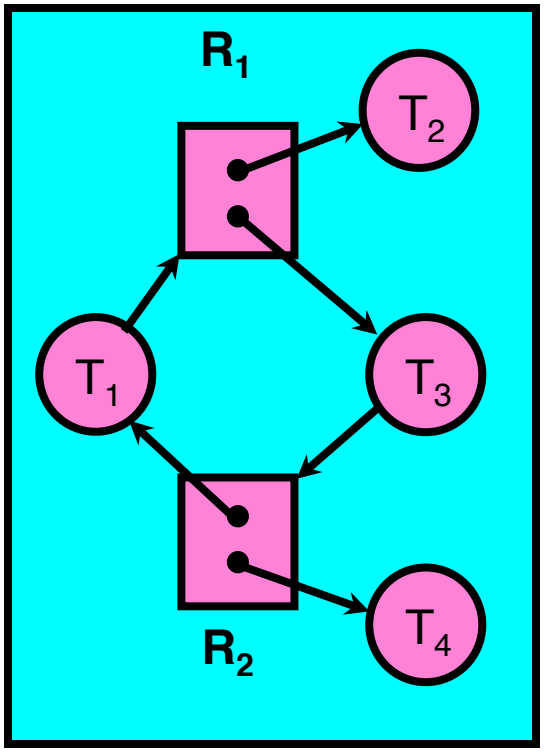
# Deadlock Detection Algorithm Example



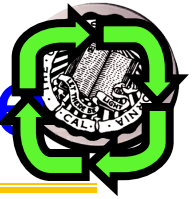
```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT1] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT1]
         done = false
      }
   }
} until(done)
```

False

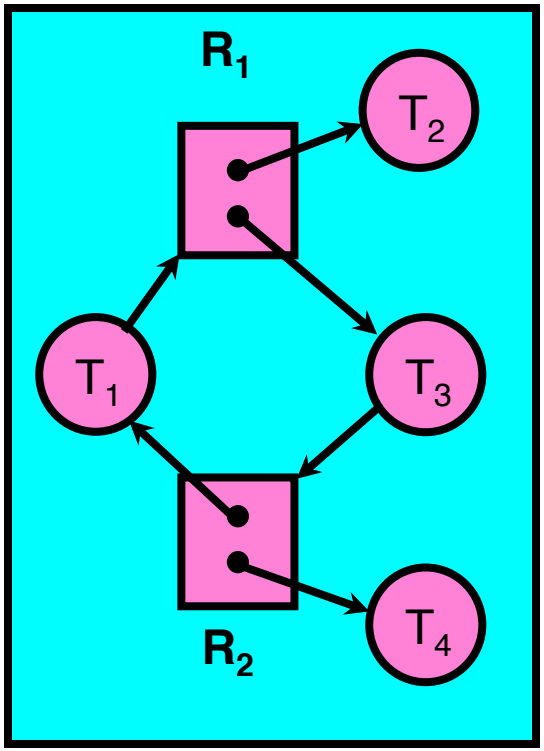


# Deadlock Detection Algorithm Example

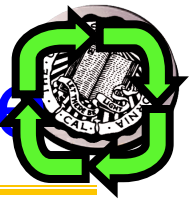


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Requestnode] <= [Avail]) {
      remove node from UNFINSHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```

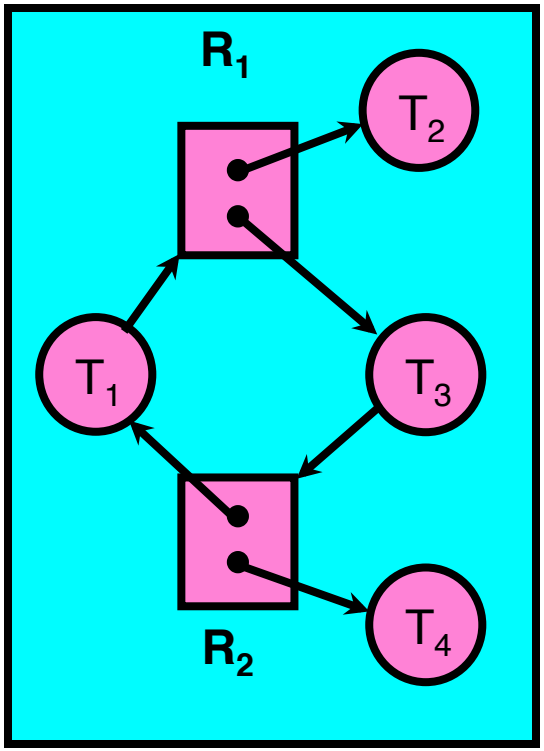


# Deadlock Detection Algorithm Example

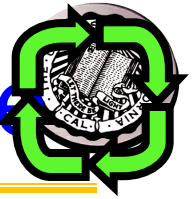


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT2] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT2]
         done = false
      }
   }
} until(done)
```

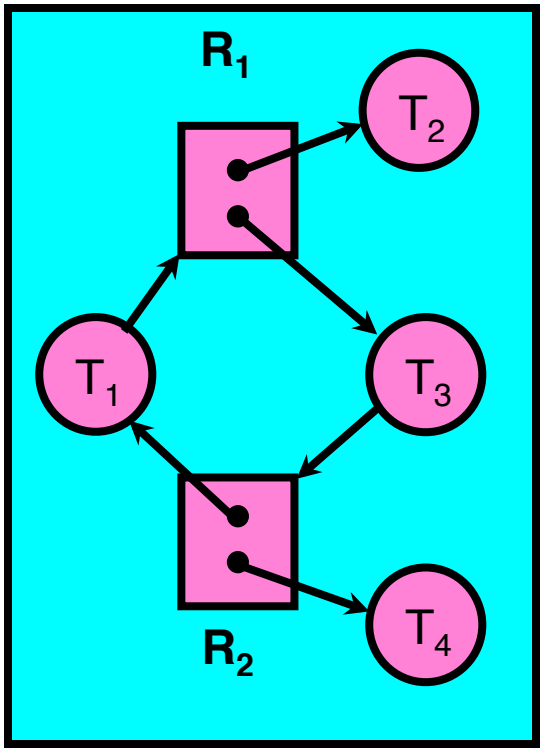


# Deadlock Detection Algorithm Example

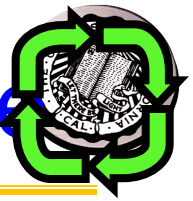


$[Request_{T1}] = [1,0]$; $Alloc_{T1} = [0,1]$
$[Request_{T2}] = [0,0]$; $Alloc_{T2} = [1,0]$
$[Request_{T3}] = [0,1]$; $Alloc_{T3} = [1,0]$
$[Request_{T4}] = [0,0]$; $Alloc_{T4} = [0,1]$
$[Avail] = [0,0]$
UNFINISHED = {T1,T3,T4}

```
do {
   done = true
   Foreach node in UNFINISHED {
     if ([Request_T2] <= [Avail]) {
       remove node from UNFINSHED
       [Avail] = [Avail] + [Alloc_T2]
       done = false
     }
   }
} until(done)
```

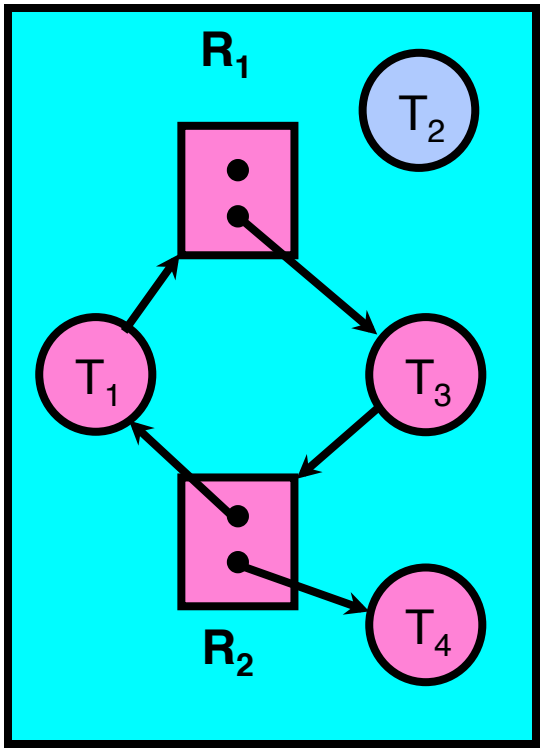


# Deadlock Detection Algorithm Example

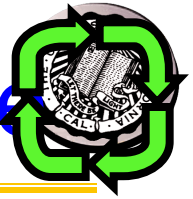


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT2] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT2]
         done = false
      }
   }
} until(done)
```

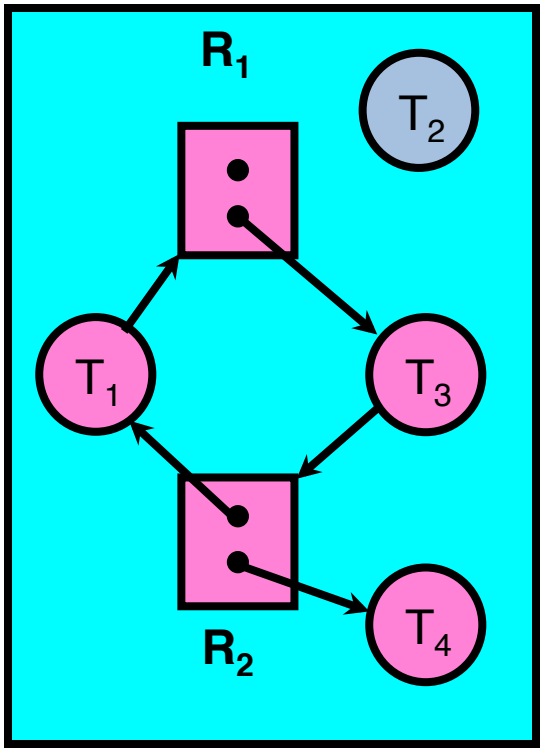


# Deadlock Detection Algorithm Example

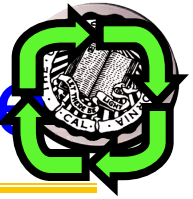


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT2] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT2]
         done = false
      }
   }
} until(done)
```

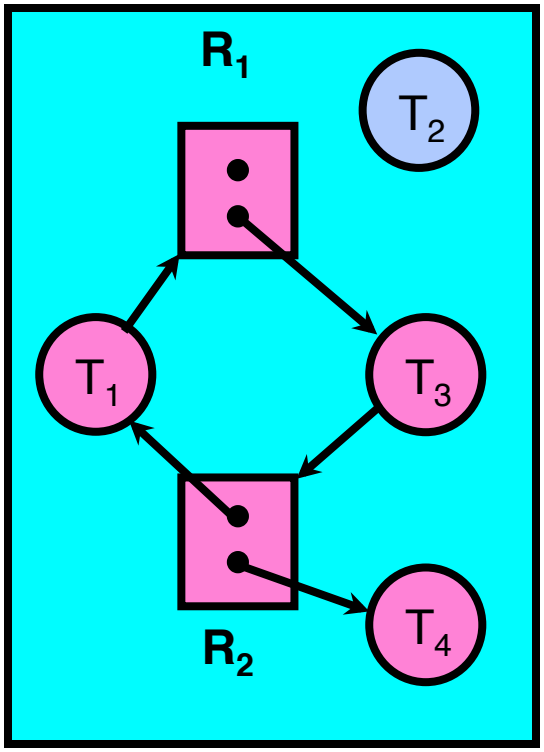


# Deadlock Detection Algorithm Example

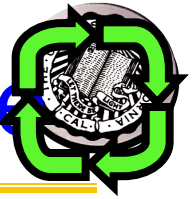


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Requestnode] <= [Avail]) {
      remove node from UNFINSHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```

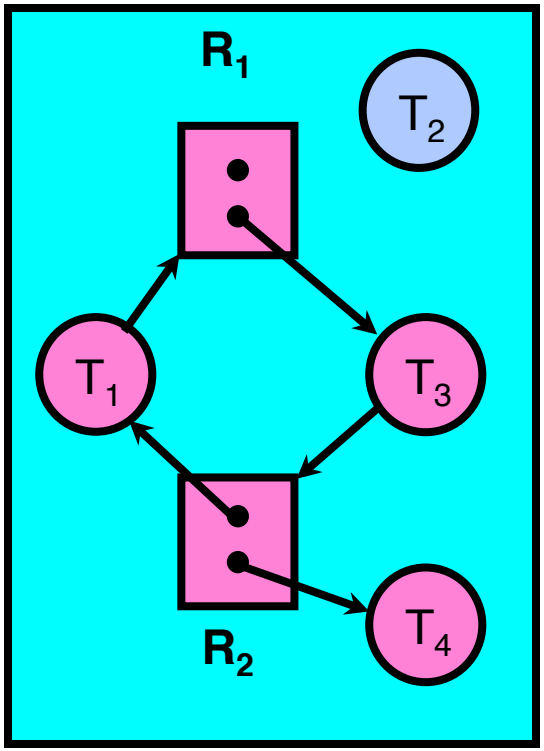


# Deadlock Detection Algorithm Example

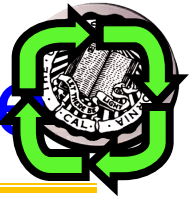


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT3] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT3]
         done = false
      }
   }
} until(done)
```

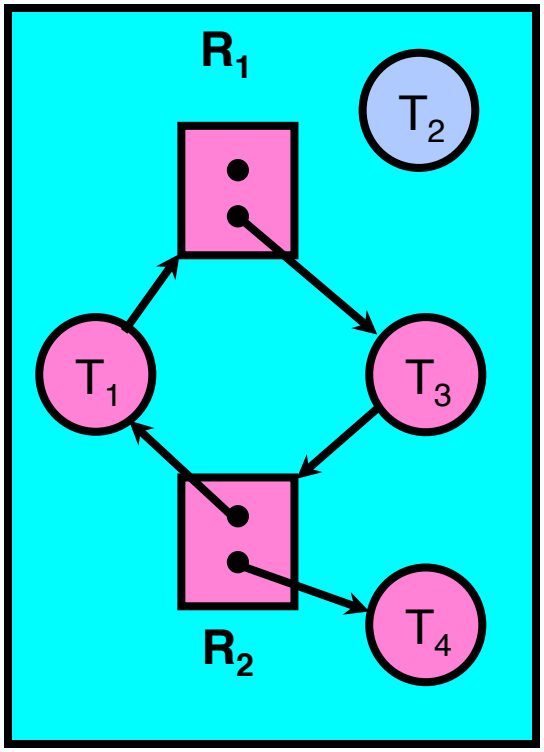


# Deadlock Detection Algorithm Example

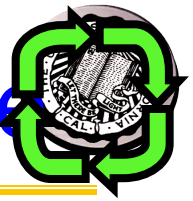


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
  done = true
  Foreach node in UNFINISHED {
    if ([Requestnode] <= [Avail]) {
      remove node from UNFINSHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```

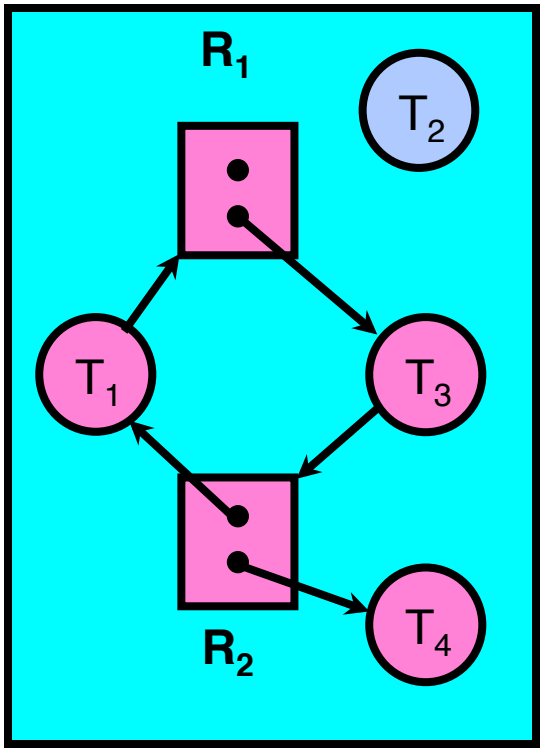


# Deadlock Detection Algorithm Example

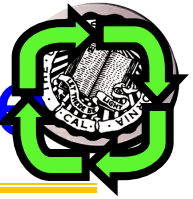


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3,T4}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT4] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT4]
         done = false
      }
   }
} until(done)
```

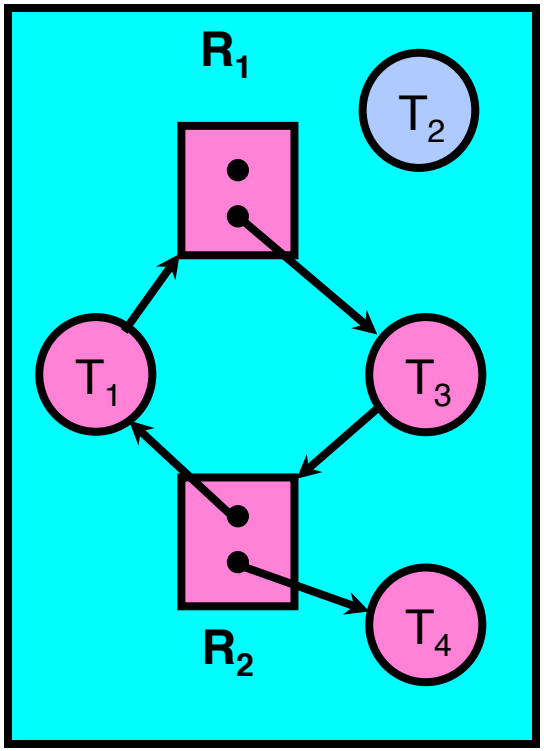


# Deadlock Detection Algorithm Example

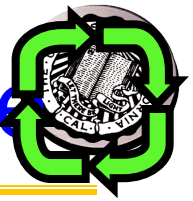


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,0]
UNFINISHED = {T1,T3}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT4] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT4]
         done = false
      }
   }
} until(done)
```

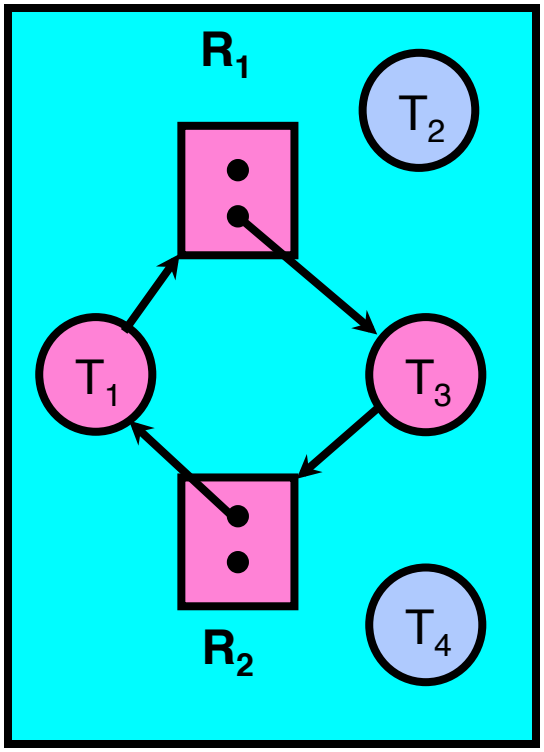


# Deadlock Detection Algorithm Example

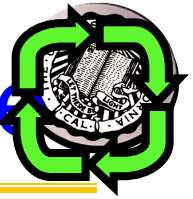


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT4] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT4]
         done = false
      }
   }
} until(done)
```

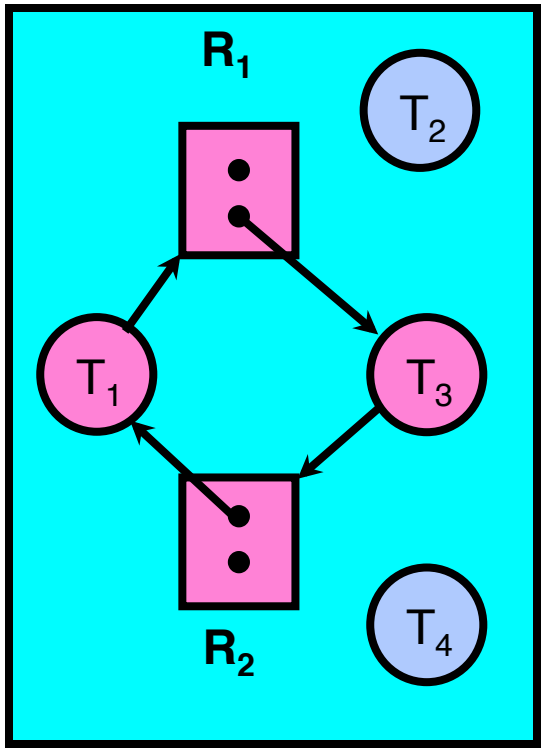


# Deadlock Detection Algorithm Example

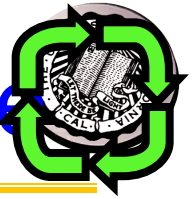


$[Request_{T1}] = [1,0]$; $Alloc_{T1} = [0,1]$
$[Request_{T2}] = [0,0]$; $Alloc_{T2} = [1,0]$
$[Request_{T3}] = [0,1]$; $Alloc_{T3} = [1,0]$
$[Request_{T4}] = [0,0]$; $Alloc_{T4} = [0,1]$
[Avail] = [1,1]
UNFINISHED = {T1,T3}

```
do {
   done = true
   Foreach node in UNFINISHED {
     if ([Request_T4] <= [Avail]) {
       remove node from UNFINSHED
       [Avail] = [Avail] + [Alloc_T4]
       done = false
     }
   }
} until(done)
```



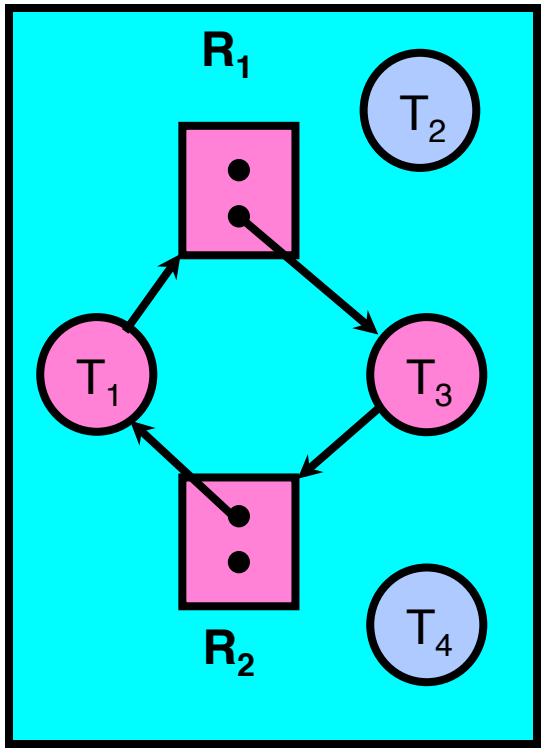
# Deadlock Detection Algorithm Example



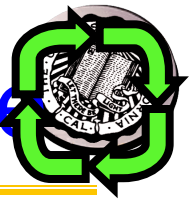
```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT4] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT4]
         done = false
      }
   }
} until(done)
```

False

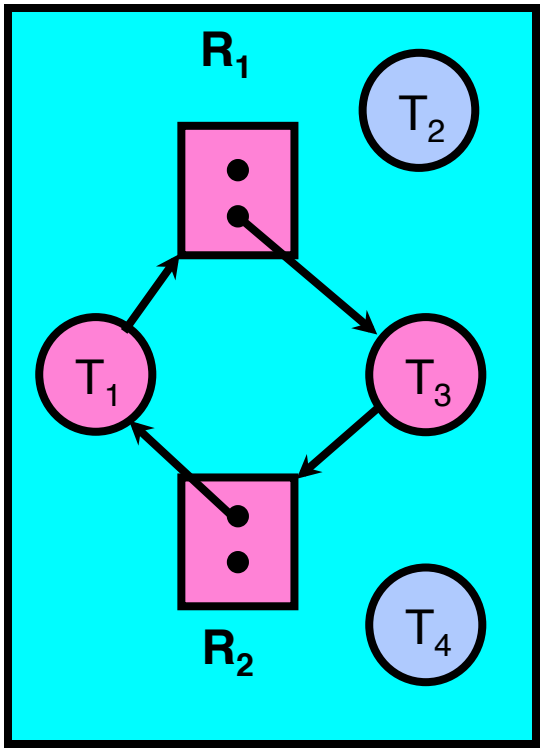


# Deadlock Detection Algorithm Example

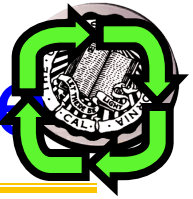


$[Request_{T1}] = [1,0]$; $Alloc_{T1} = [0,1]$
$[Request_{T2}] = [0,0]$; $Alloc_{T2} = [1,0]$
$[Request_{T3}] = [0,1]$; $Alloc_{T3} = [1,0]$
$[Request_{T4}] = [0,0]$; $Alloc_{T4} = [0,1]$
$[Avail] = [1,1]$
UNFINISHED = {T1,T3}

```
do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINSHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```

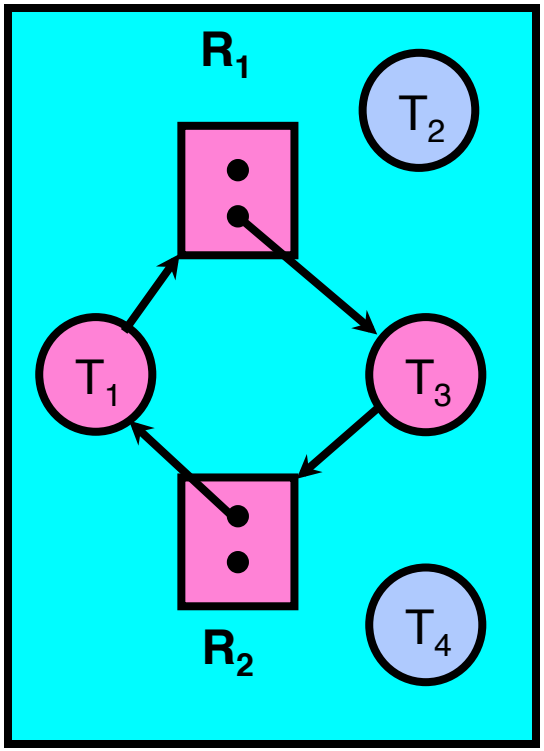


# Deadlock Detection Algorithm Example

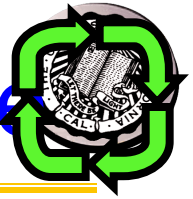


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT1] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT1]
         done = false
      }
   }
} until(done)
```

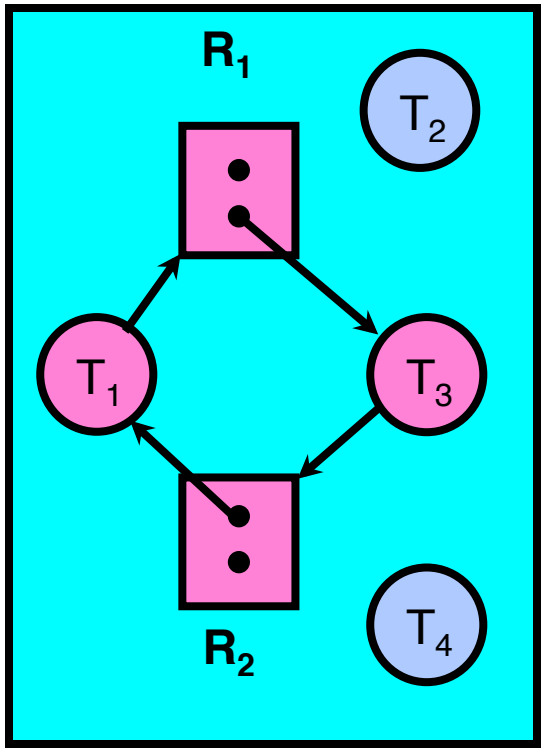


# Deadlock Detection Algorithm Example

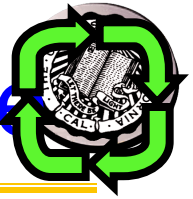


$[Request_{T1}] = [1,0]$; $Alloc_{T1} = [0,1]$
$[Request_{T2}] = [0,0]$; $Alloc_{T2} = [1,0]$
$[Request_{T3}] = [0,1]$; $Alloc_{T3} = [1,0]$
$[Request_{T4}] = [0,0]$; $Alloc_{T4} = [0,1]$
[Avail] = [1,1]
UNFINISHED = {T3}

```
do {
   done = true
   Foreach node in UNFINISHED {
     if ([Request_T1] <= [Avail]) {
       remove node from UNFINSHED
       [Avail] = [Avail] + [Alloc_T1]
       done = false
     }
   }
} until(done)
```

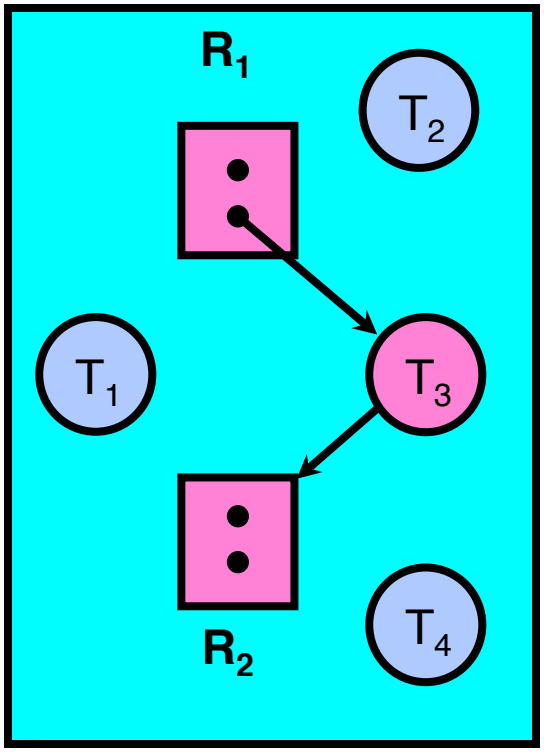


# Deadlock Detection Algorithm Example

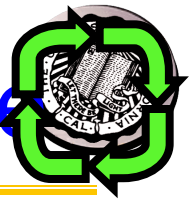


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,2]
UNFINISHED = {T3}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT1] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT1]
         done = false
      }
   }
} until(done)
```

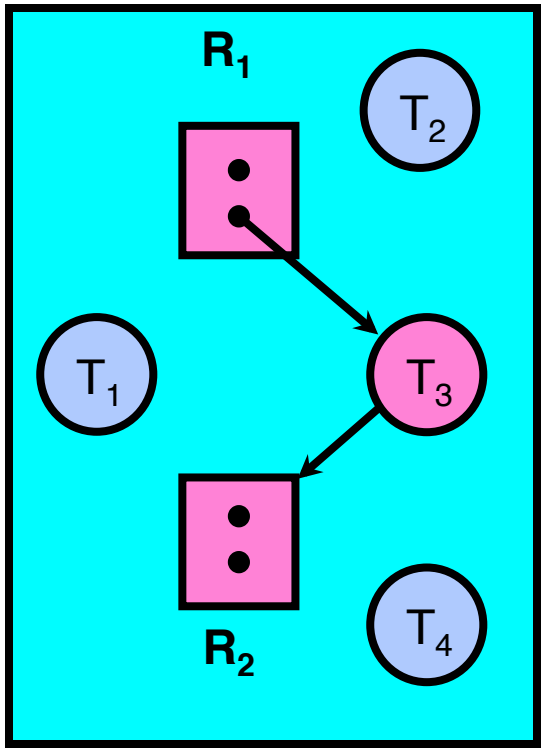


# Deadlock Detection Algorithm Example

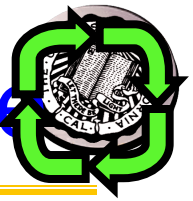


$[Request_{T1}] = [1,0]$; $Alloc_{T1} = [0,1]$
$[Request_{T2}] = [0,0]$; $Alloc_{T2} = [1,0]$
$[Request_{T3}] = [0,1]$; $Alloc_{T3} = [1,0]$
$[Request_{T4}] = [0,0]$; $Alloc_{T4} = [0,1]$
$[Avail] = [1,2]$
UNFINISHED = {T3}

```
do {
   done = true
   Foreach node in UNFINISHED {
     if ([Request_T1] <= [Avail]) {
       remove node from UNFINSHED
       [Avail] = [Avail] + [Alloc_T1]
       done = false
     }
   }
} until(done)
```

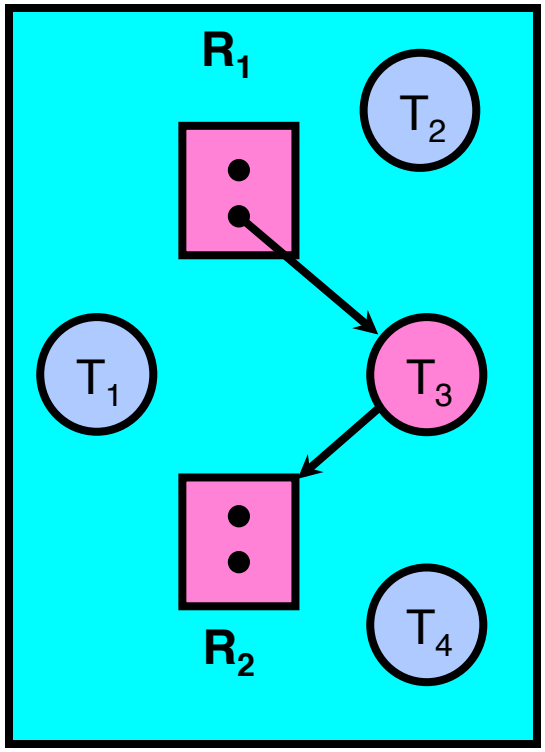


# Deadlock Detection Algorithm Example

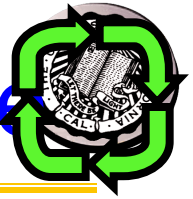


$[Request_{T1}] = [1,0]$; $Alloc_{T1} = [0,1]$
$[Request_{T2}] = [0,0]$; $Alloc_{T2} = [1,0]$
$[Request_{T3}] = [0,1]$; $Alloc_{T3} = [1,0]$
$[Request_{T4}] = [0,0]$; $Alloc_{T4} = [0,1]$
$[Avail] = [1,2]$
UNFINISHED = {T3}

```
do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINSHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```

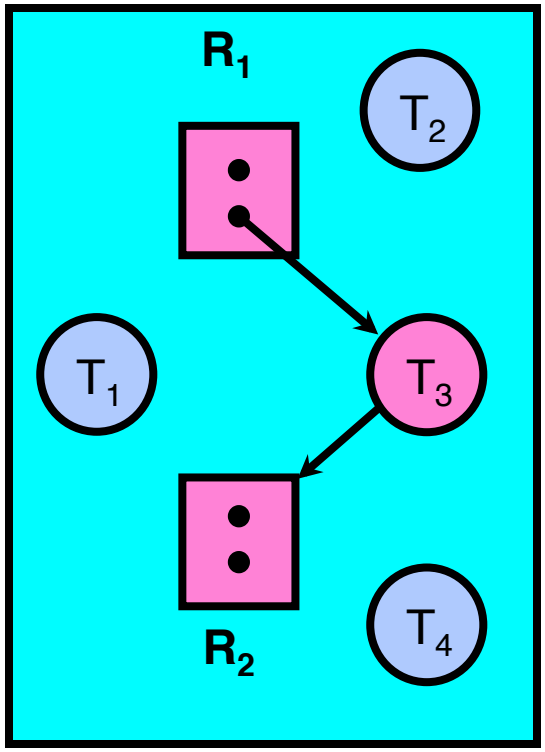


# Deadlock Detection Algorithm Example

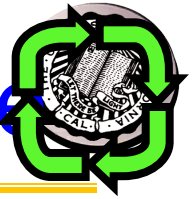


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [1,2]
UNFINISHED = {T3}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT3] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT3]
         done = false
      }
   }
} until(done)
```

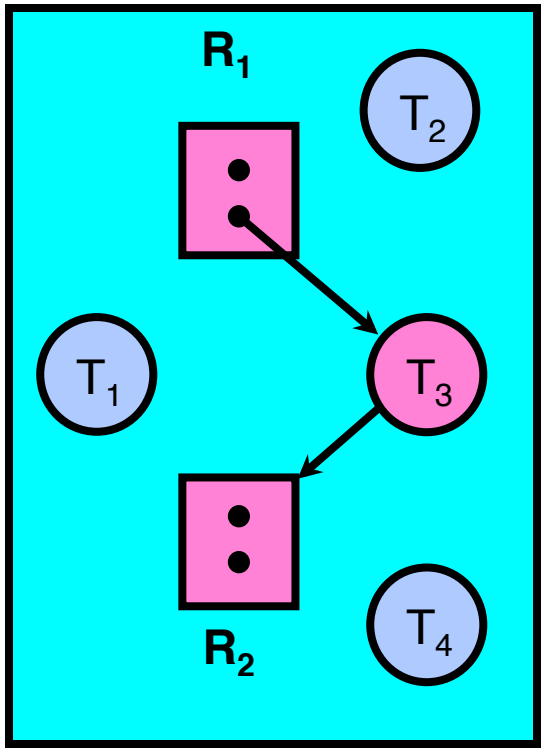


# Deadlock Detection Algorithm Example

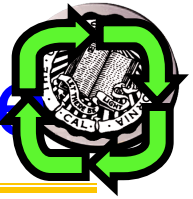


$[Request_{T1}] = [1,0]$; $Alloc_{T1} = [0,1]$
$[Request_{T2}] = [0,0]$; $Alloc_{T2} = [1,0]$
$[Request_{T3}] = [0,1]$; $Alloc_{T3} = [1,0]$
$[Request_{T4}] = [0,0]$; $Alloc_{T4} = [0,1]$
[Avail] = [1,2]
UNFINISHED = {}

```
do {
   done = true
   Foreach node in UNFINISHED {
     if ([Request_T3] <= [Avail]) {
       remove node from UNFINSHED
       [Avail] = [Avail] + [Alloc_T3]
       done = false
     }
   }
} until(done)
```

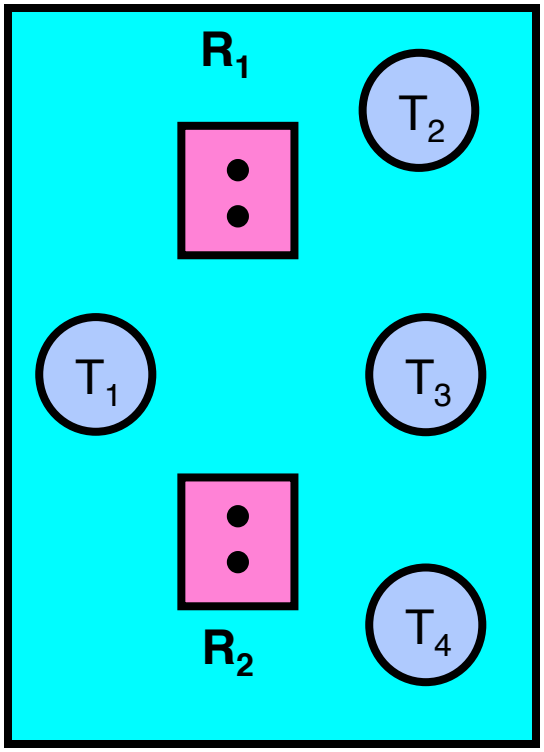


# Deadlock Detection Algorithm Example

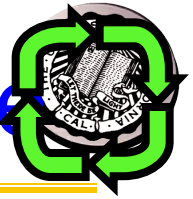


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [2,2]
UNFINISHED = {}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT3] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT3]
         done = false
      }
   }
} until(done)
```

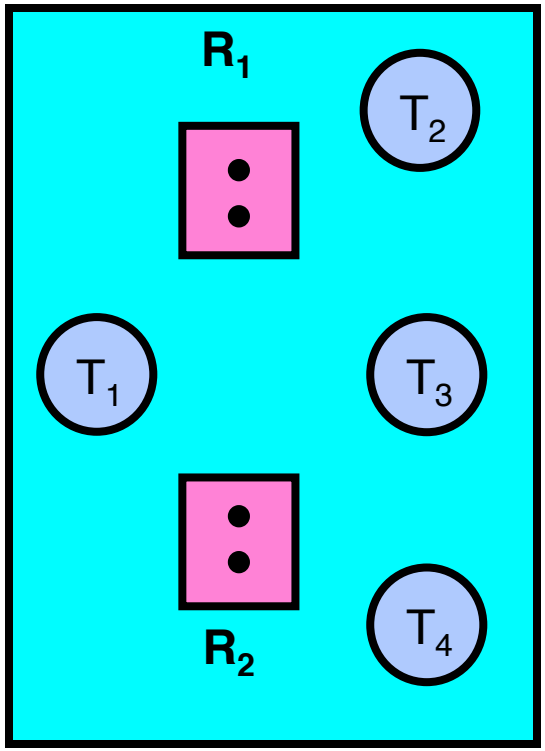


# Deadlock Detection Algorithm Example

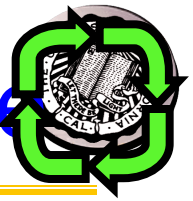


```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [2,2]
UNFINISHED = {}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT3] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT3]
         done = false
      }
   }
} until(done)
```

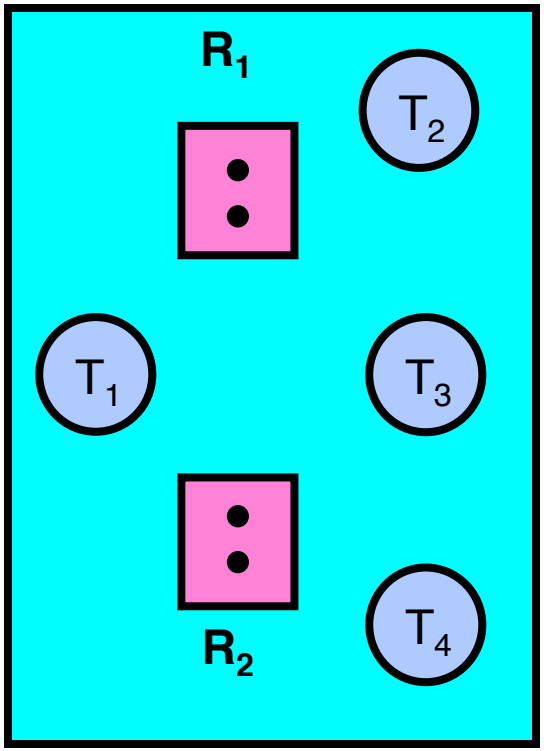


# Deadlock Detection Algorithm Example



```
[RequestT1] = [1,0]; AllocT1 = [0,1]
[RequestT2] = [0,0]; AllocT2 = [1,0]
[RequestT3] = [0,1]; AllocT3 = [1,0]
[RequestT4] = [0,0]; AllocT4 = [0,1]
[Avail] = [2,2]
UNFINISHED = {}

do {
   done = true
   Foreach node in UNFINISHED {
      if ([RequestT3] <= [Avail]) {
         remove node from UNFINSHED
         [Avail] = [Avail] + [AllocT3]
         done = false
      }
   }
} until(done)
```




DONE!

# Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max remaining that might be needed by any thread

- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with $T_1$ requesting all remaining resources, finishing, then $T_2$ requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

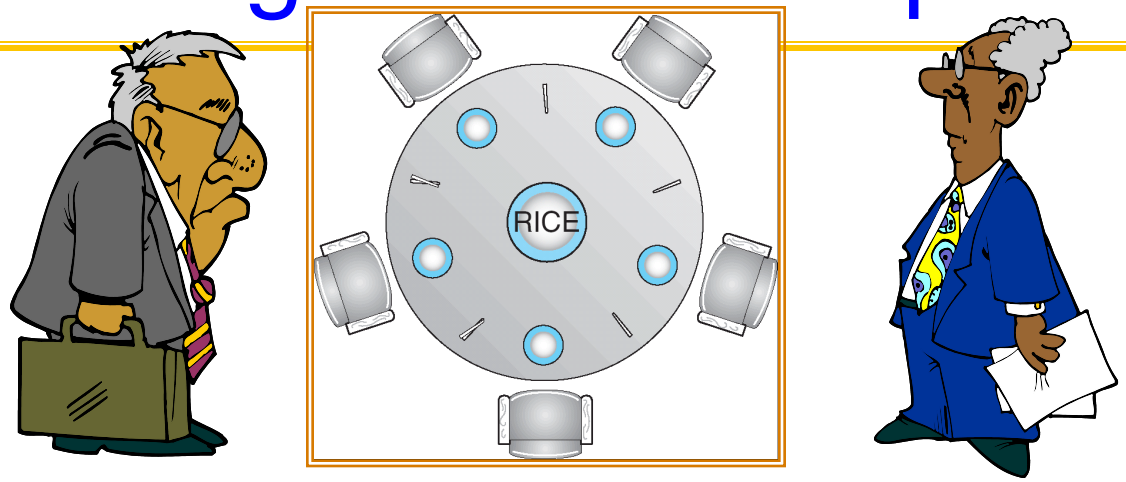
# Banker's Algorithm

- Technique: pretend each request is granted, then run deadlock detection algorithm, substitute ([Request$_{node}$] ≤ [Avail]) → ([Max$_{node}$]-[Alloc$_{node}$] ≤ [Avail])

```
[FreeResources]:          Current free resources each type
   [Alloc_X]:             Current resources held by thread X
     [Max_X]:             Max resources requested by thread X

     [Avail] = [FreeResources]
   Add all nodes to UNFINISHED
   do {
      done = true
      Foreach node in UNFINISHED {
     if ([Max_node]-[Alloc_node]<= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc_node]
            done = false
         }
      }
   } until(done)
```

# Banker's Algorithm Example




- Banker's algorithm with dining philosophers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - Not last chopstick
    - Is last chopstick but someone will have two afterwards
  - What if k-handed philosophers? Don't allow if:
    - It's the last one, no one would have k
    - It's 2nd to last, and no one would have k-1
    - It's 3rd to last, and no one would have k-2
    - …

# Summary: Deadlock

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - ∃ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Deadlock detection and preemption
- Deadlock prevention
  - Loop Detection, Banker's algorithm