Anthony Ventresque

anthony.ventresque@ucd.ie

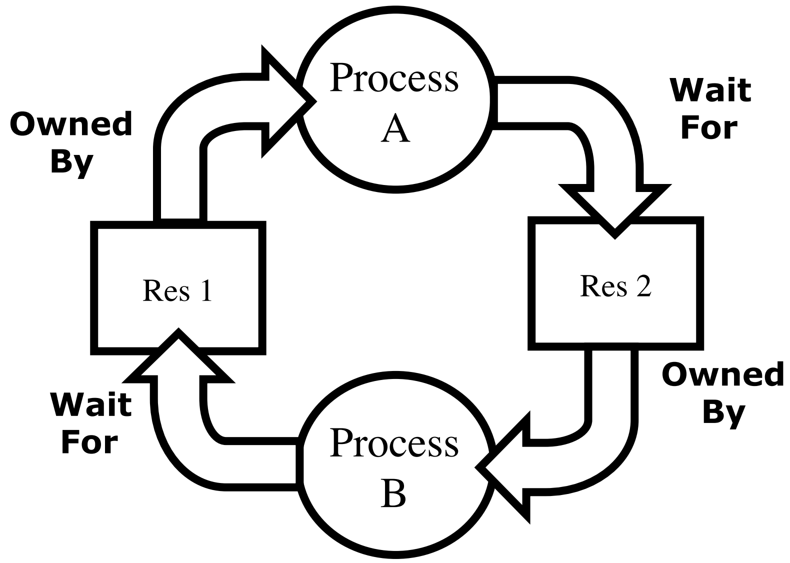
Deadlock and Starvation (2)

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Last Time: Starvation vs. Deadlock

- Starvation: process/thread waits indefinitely
  - Example, low-priority process/thread waiting for resources constantly in use by high-priority process/threads
- Deadlock: circular waiting for resources
  - Process A owns Res 1 and is waiting for Res 2
  - Process B owns Res 2 and is waiting for Res 1



- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Last Time: Four Necessary Conditions for Deadlock

1. **Mutual exclusion** (limited access)
   - At least one resource may be acquired exclusively by only one process at a time

2. **Hold-and-wait** (wait-for)
   - Processes may ask for resources while holding other resources

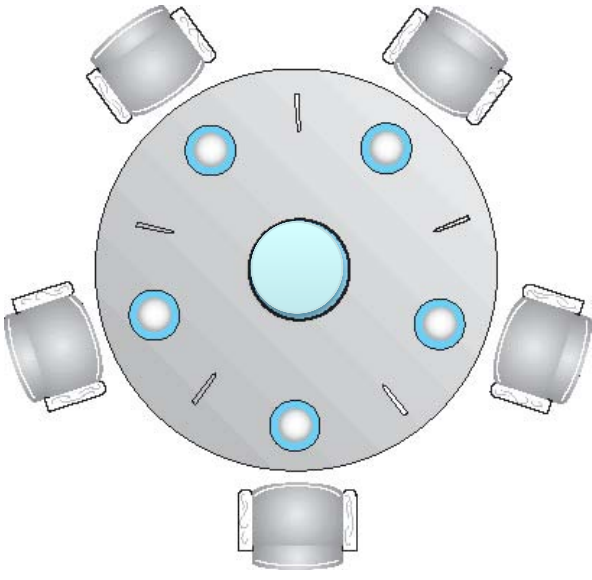
3. **No preemption**
   - Once allocated, resources are released only voluntarily by the process holding the resource, after process is finished with it

4. **Circular chain of request** (circular-wait)
   - Two or more processes locked in a circular chain in which each process is waiting for one or more resources that the next process in the chain is holding
   - Equivalent to a cycle in the RAG

Note: ***These are not sufficient conditions***

Last Time: Dining Philosophers

- Classic example proposed by Dijkstra (1971) to illustrate deadlock & starvation
- 5 philosophers living together
- Their life consists of two states: thinking or eating
- They have a position assigned at a round table on which there are 5 plates of noodles and 5 chopsticks
- A philosopher wishing to eat takes ***both chopsticks*** on the sides of their plate, and eats noodles
- No two philosophers can share a chopstick simultaneously

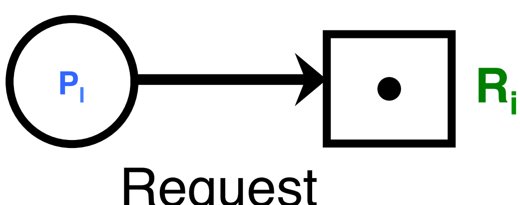


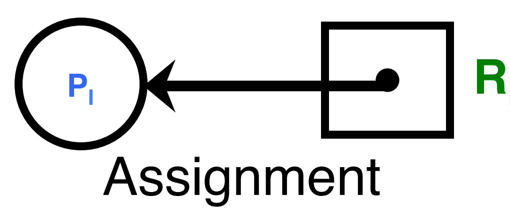
Philosopher = Process

Chopstick = Shared resource

Last Time: Resource-allocation Graph

- A RAG is a ***special directed graph***, since both vertices and directed edges are partitioned into ***two sets***
- Vertices:
  1. $P = \{P_1, \ldots, P_n\}$, all active processes in the system:
  2. $R = \{R_1, \ldots, R_m\}$, all resource types in the system:
  - also: number of • inside □ is the number of instances of the resource (two displays, three printers. . .)
- Directed edges:
  1. $P_i \rightarrow R_j$: request edge
  2. $R_j \rightarrow P_i$: assignment edge





Outline

- Techniques for Detecting Deadlocks
- Techniques for Preventing Deadlocks
- Techniques for Avoiding Deadocks

Methods for Deadlock Handling

Three possibilities:

1. Let deadlock occur, and do something about it afterwards
    - Deadlock ***detection*** & recovery
2. Never let deadlock occur
    - Deadlock ***prevention***
    - Deadlock ***avoidance***

3. Ignore the problem and pretend that deadlock never occurs
- This "strategy" is used by many desktop OSs
- It can be efficient, if the probability of deadlock is low
- It cannot be tolerated in mission-critical or real-time systems

Deadlock Detection and Recovery

Detection: Scan the RAG to find cycles

- Periodically
- or during low system utilisation periods

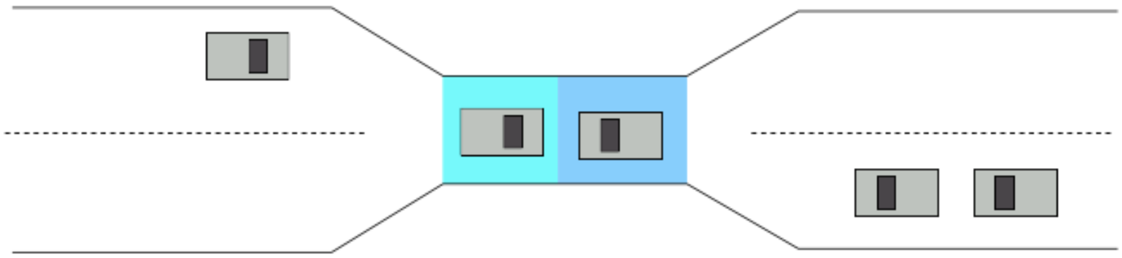
Recovery strategies:

1. ***Process termination***
   - Abort one deadlocked process at a time until deadlock cycle eliminated: costly, and maybe slow (deadlock detection after each termination)
   - Abort all deadlocked processes: very costly, faster
   - System reboot: even more costly, but fastest

2. ***Resource pre-emption***: successively pre-empt processes from resources until deadlock cycle broken; issues:
   - ***Victim selection order:*** pre-empt according to cost function
   - ***Rollback*****:** victim must be rolled back to a safe prior state; information must be kept consistent (extreme: total rollback)
   - ***Starvation*****:** will resources always be pre-empted from the same process? (we must take this into account in cost function)

Example: Deadlock Recovery at Bridge Crossing

- Consider the figure below



  - Bridge section allows one-way traffic only
  - Bridge section = resource (critical section); car = process
- If deadlock occurs: it can be resolved if one car reverses
  - Pre-empt resource and roll back
- Several cars may have to reverse if a deadlock occurs
  - Different costs for different "victims"
- Starvation is possible

Deadlock Prevention

Prevention policies are based on eliminating the possibility of at least one of the necessary conditions for deadlock.

Possibilities:

- Always avoid mutual exclusion: some resources can be shared by an unlimited number of processes (e.g., read-only file)

  Issue: Some resources are nonshareable (e.g., printer)

- Always avoid hold-and-wait. Two ways to do it:
  1. Don’t allow waiting for a resource while holding resources; or
  2. Have each process request and be allocated all its resources before execution

  Issue: low resource utilization, starvation possible

Techniques for Preventing Deadlock

- Make all threads request everything they'll need at the beginning
  - *Problem:* Predicting future is hard, tend to over-estimate resources
  - *Example:* Don't leave home until we know no one is using any intersection between here and where you want to go!

- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,…)
    - Make tasks request disk, then memory, then…

Deadlock Avoidance

- Avoidance policies are based on the system having a priori information available
  - *A-priori information:* processes declare the ***maximum number of resources*** of each type that they may need at the start

- Deadlock-avoidance algorithms dynamically monitor the system state to ensure no circular waits based on this information

- <u>Issues</u>
  - Hard to implement, as we need to accurately predict the future
  - It assumes processes eventually release their resources, but this could be a long time

Deadlock Avoidance: Safe State

- **Definition:** safe state of a system
  - State in which resources can be allocated to each process (up to maximum requested) while avoiding deadlock

- Formally: a state is safe if there exists a safe sequence of processes $< P_1, P_2, \ldots, P_n >$ such that
  - The resource requests that P_i can make are satisfiable by:
    1. Currently available resources, plus
    2. Resources held by $P_1, P_2, \cdots P_{i-1}$

- A safe sequence is an ***ordered*** arrangement of all processes
  - We create it <u>sequentially</u>, starting with one process, then two, etc

Deadlock Avoidance: Safe State

- If no safe sequence exists, the system state is unsafe

state	*deadlock*
safe	impossible
unsafe	possible (but not sure)

- State safety is a worst-case analysis:
  Therefore an unsafe state is a necessary, but not sufficient, condition for deadlock

Deadlock Avoidance: Banker's Algorithm (Dijkstra)

- It allows to check whether satisfying a request for resources will put the system in a <u>safe state</u> or not
  - The request is only satisfied if the new state is safe
  - It is a conservative algorithm

- Reason for the name:
  - It may be used by a banker to ensure that cash is never allocated in such a way that the bank can no longer satisfy the needs of all its customers

Banker’s Algorithm: Basic Structure

- The banker’s algorithm involves two sub-algorithms:

1. **Safety Algorithm:** to verify that a system state is safe

2. **Resource-request algorithm:** to verify whether allocating the requested resources will take the system to a new safe state

Banker's Algorithm

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

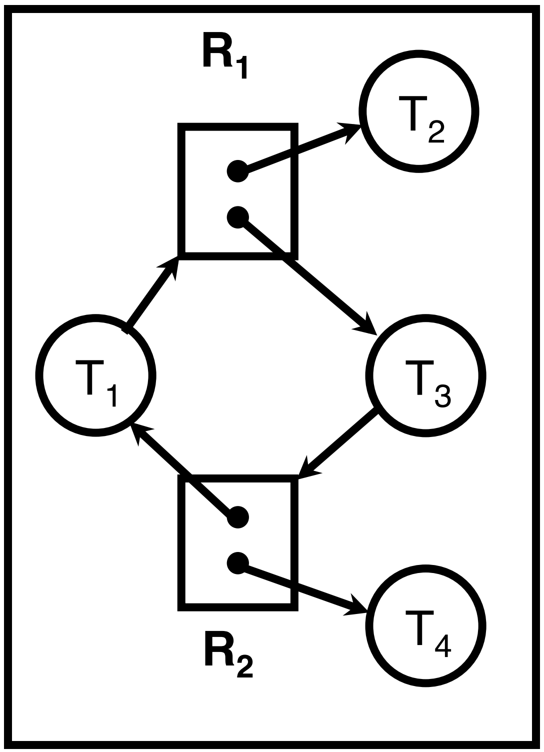
`[FreeResources]:`	Current free resources each type
`[Alloc`$_X$`]:`	Current resources held by thread X
`[Max`$_X$`]:`	**Max resources requested by thread X**

```
[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)
```

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,T2,T3,T4}

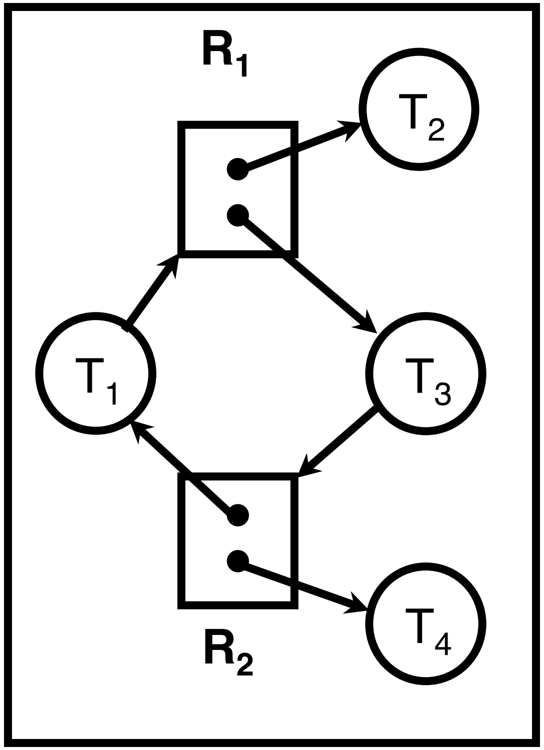
```
do {
  done = true
  Foreach node in UNFINISHED {
    if ([Request_node] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



Deadlock Detection Algorithm: Example

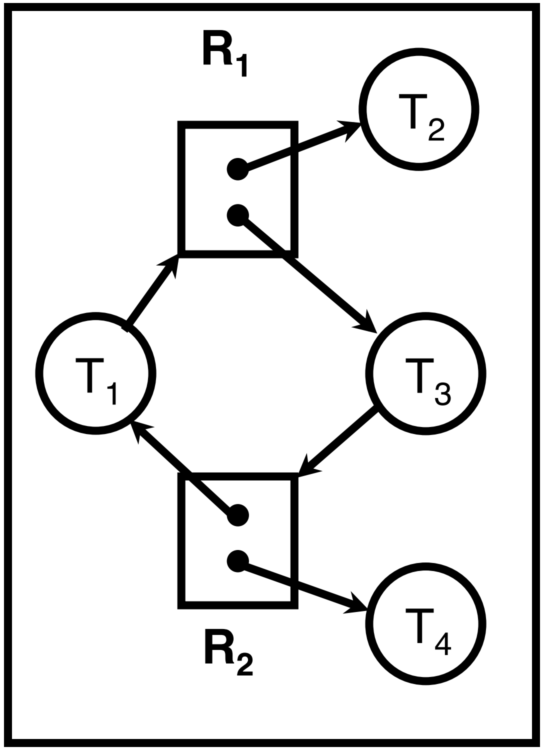
```
[Request_T1] = [1,0]; [Alloc_T1] = [0,1]
[Request_T2] = [0,0]; [Alloc_T2] = [1,0]
[Request_T3] = [0,1]; [Alloc_T3] = [1,0]
[Request_T4] = [0,0]; [Alloc_T4] = [0,1]
[Avail] = [0,0]
UNFINISHED = {T1,T2,T3,T4}

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

False



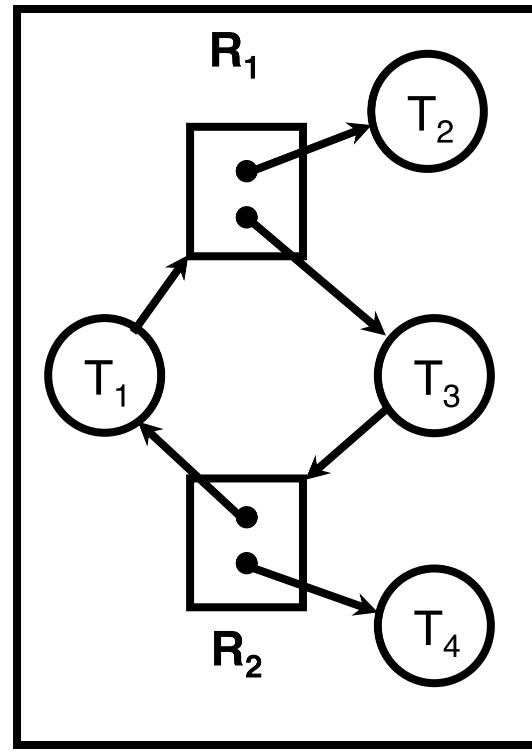
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [Allocnode]
      done = false
    }
  }
} until(done)
```



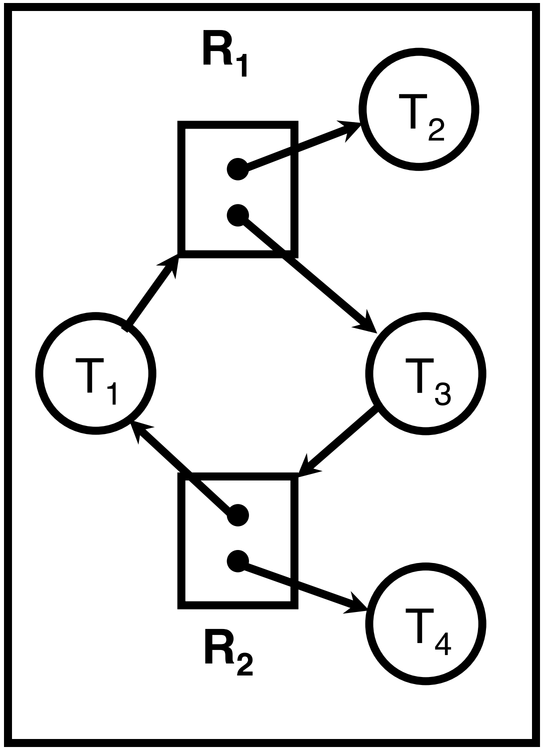
Deadlock Detection Algorithm: Example

```
[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,T2,T3,T4}

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



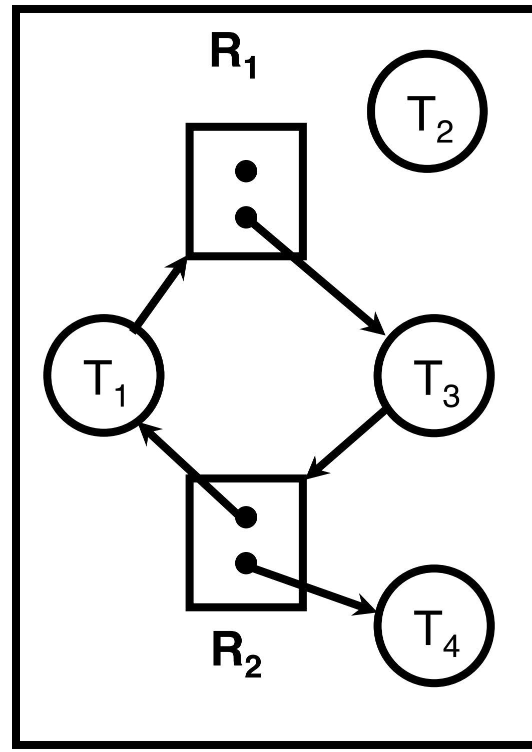
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [Alloc_T2]
      done = false
    }
  }
} until(done)
```



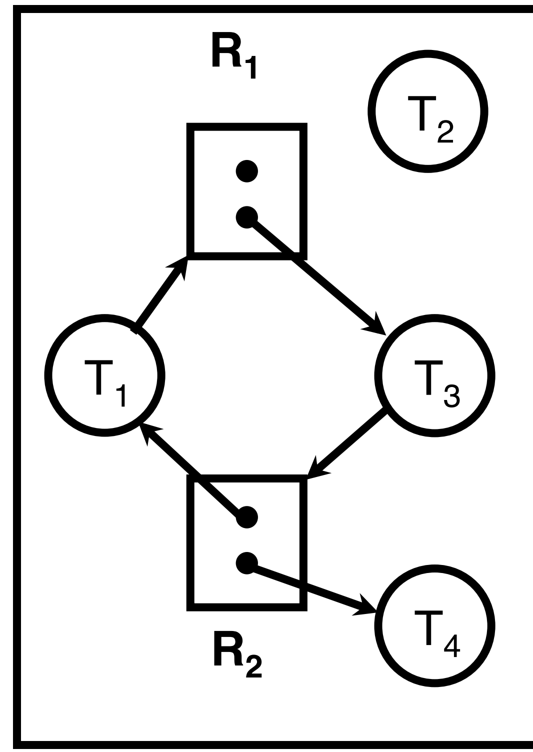
Deadlock Detection Algorithm: Example

```
[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] = [1,0]
UNFINISHED = {T1,T3,T4}

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



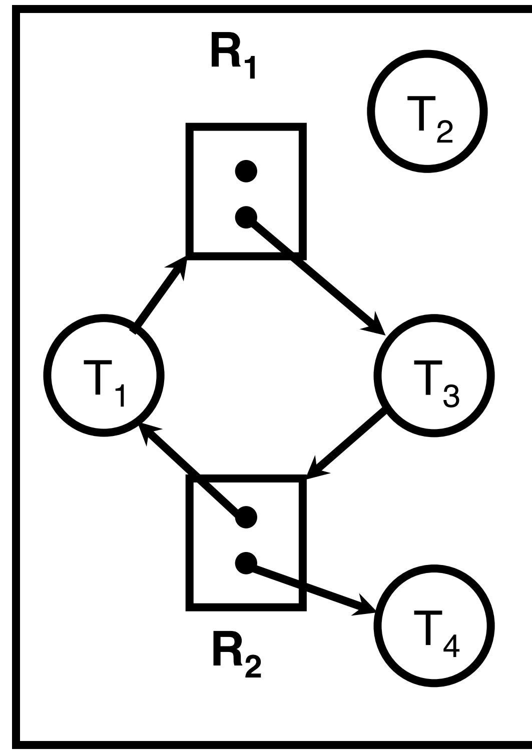
Deadlock Detection Algorithm: Example

```
[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] = [1,0]
UNFINISHED = {T1,T3,T4}


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



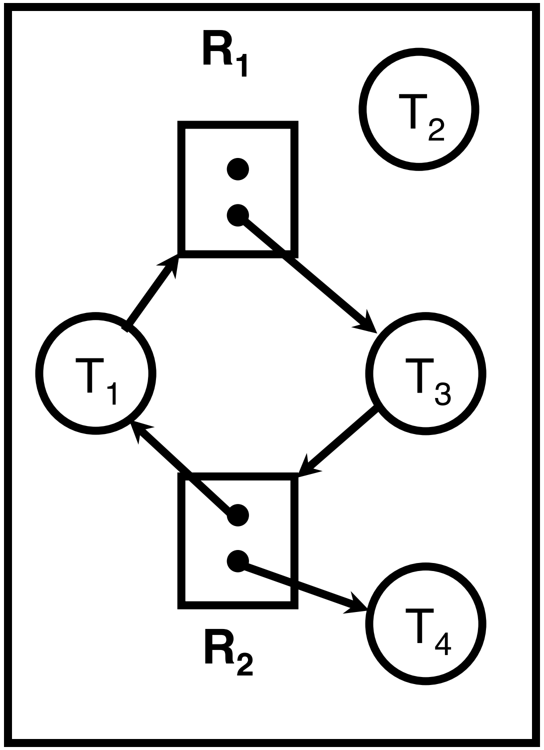
Deadlock Detection Algorithm: Example

```
[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] = [1,0]
UNFINISHED = {T1,T3,T4}

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



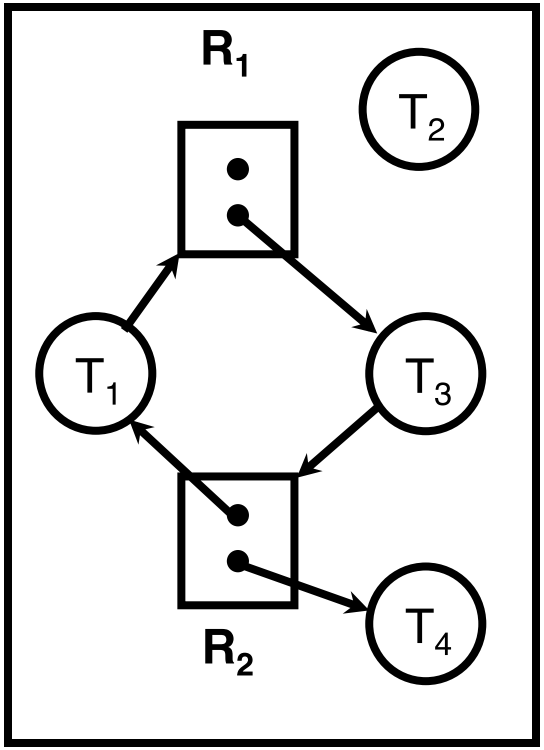
Deadlock Detection Algorithm: Example

```
[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] = [1,0]
UNFINISHED = {T1,T3,T4}

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



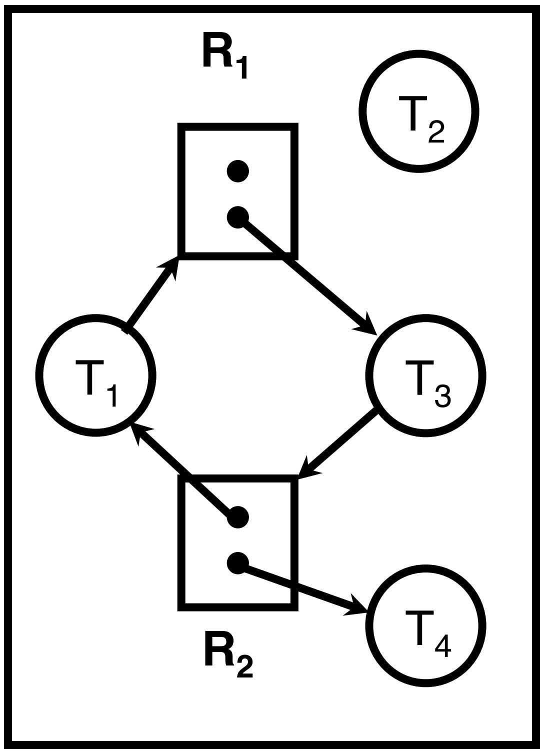
Deadlock Detection Algorithm: Example

```
[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] = [1,0]
UNFINISHED = {T1,T3,T4}

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



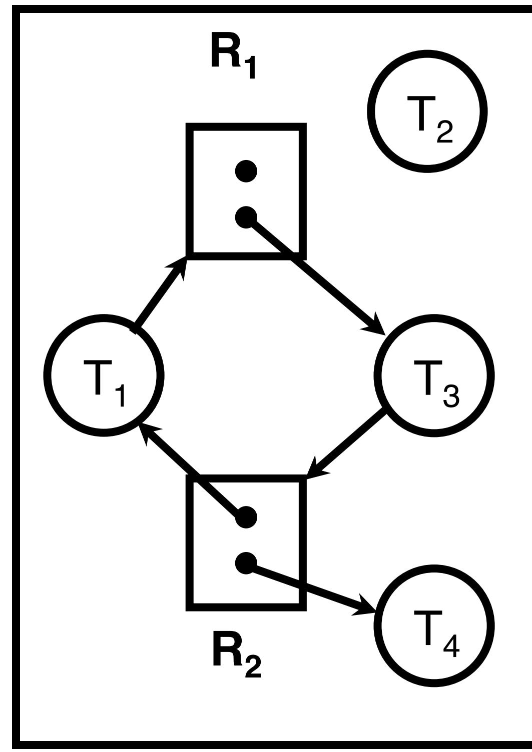
Deadlock Detection Algorithm: Example

```
[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] = [1,0]
UNFINISHED = {T1,T3,T4}

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



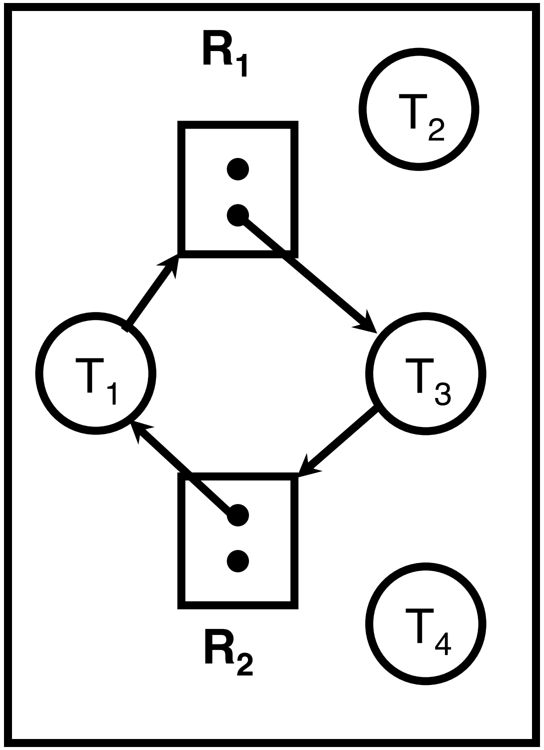
Deadlock Detection Algorithm: Example

```
[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] = [1,0]
UNFINISHED = {T1,T3}

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



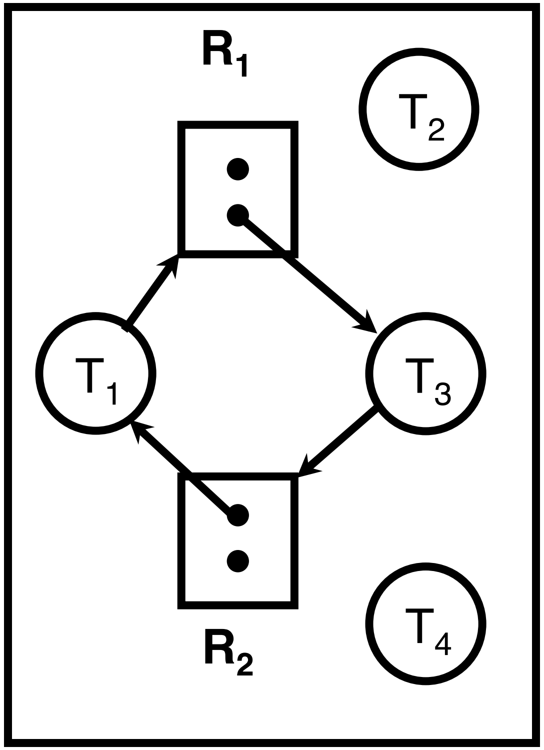
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [Alloc_T4]
      done = false
    }
  }
} until(done)
```



Deadlock Detection Algorithm: Example

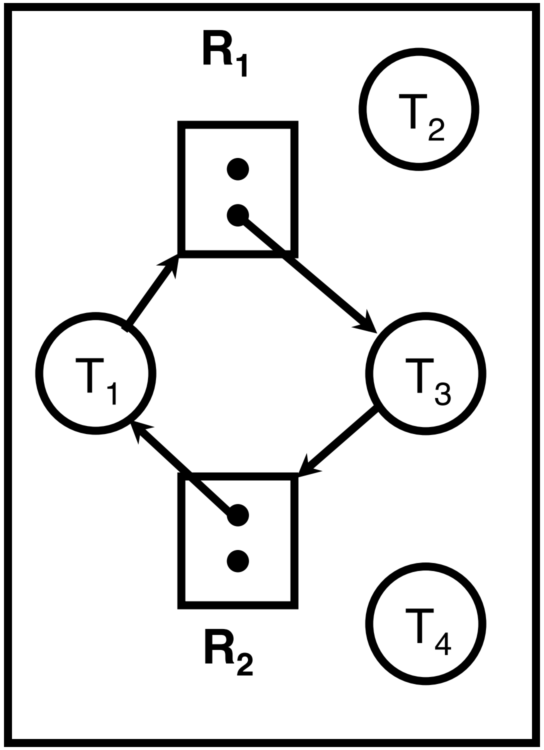
```
[RequestT1] = [1,0]; [AllocT1] = [0,1]
[RequestT2] = [0,0]; [AllocT2] = [1,0]
[RequestT3] = [0,1]; [AllocT3] = [1,0]
[RequestT4] = [0,0]; [AllocT4] = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

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



Deadlock Detection Algorithm: Example

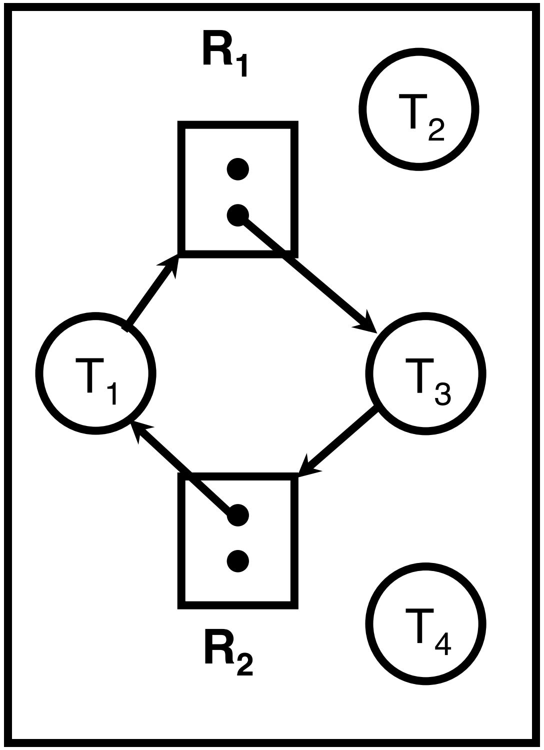
```
[Request_T1] = [1,0]; [Alloc_T1] = [0,1]
[Request_T2] = [0,0]; [Alloc_T2] = [1,0]
[Request_T3] = [0,1]; [Alloc_T3] = [1,0]
[Request_T4] = [0,0]; [Alloc_T4] = [0,1]
[Avail] = [1,1]
UNFINISHED = {T1,T3}

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

False



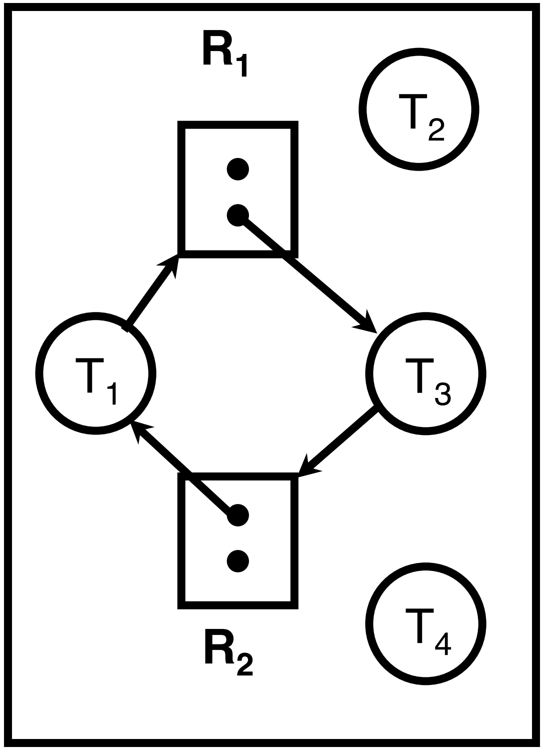
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



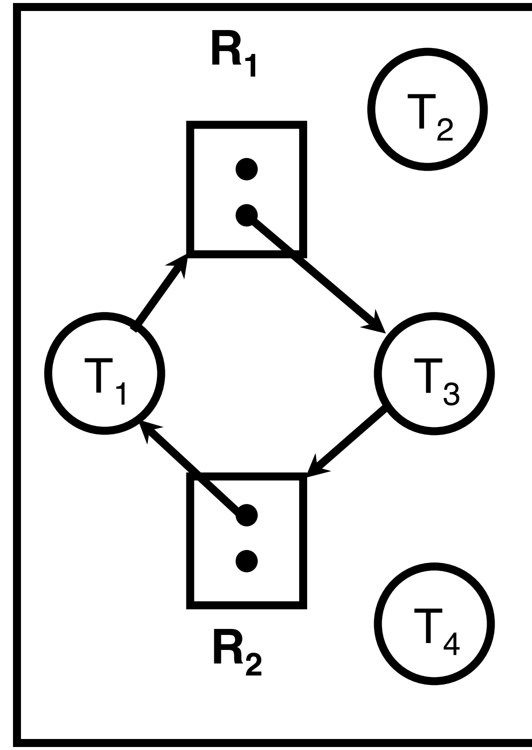
Deadlock Detection Algorithm: Example

```
[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_T1] <= [Avail]) {
      remove node from UNFINISHED
      [Avail] = [Avail] + [Alloc_T1]
      done = false
    }
  }
} until(done)
```



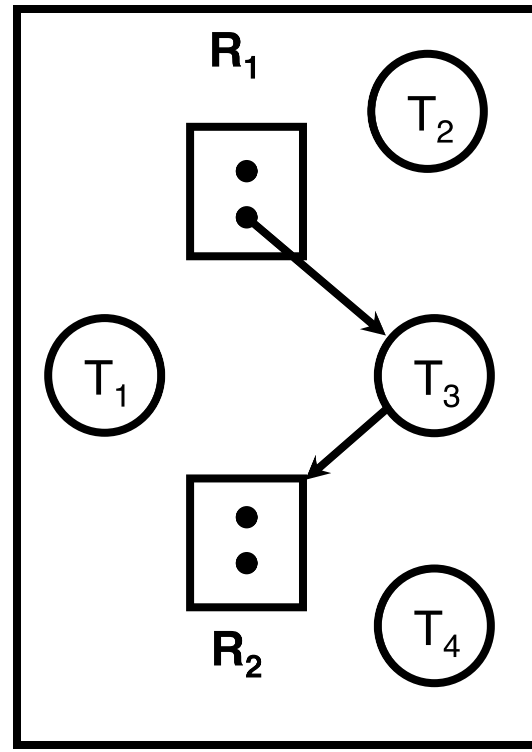
Deadlock Detection Algorithm: Example

```
[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] = [1,1]
UNFINISHED = {T3}

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



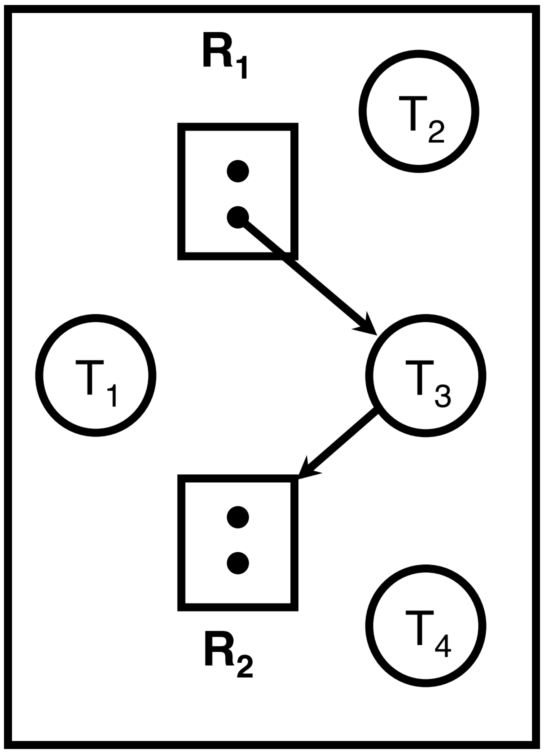
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [AllocT1]
      done = false
    }
  }
} until(done)
```



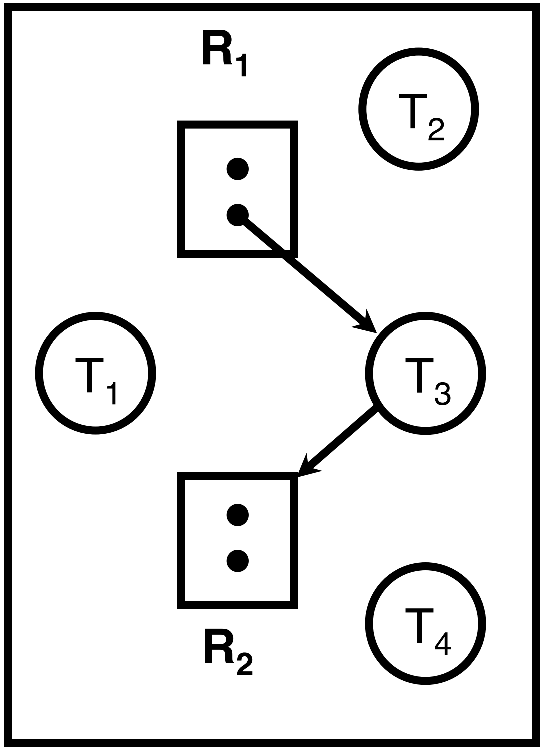
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [Alloc_T1]
      done = false
    }
  }
} until(done)
```



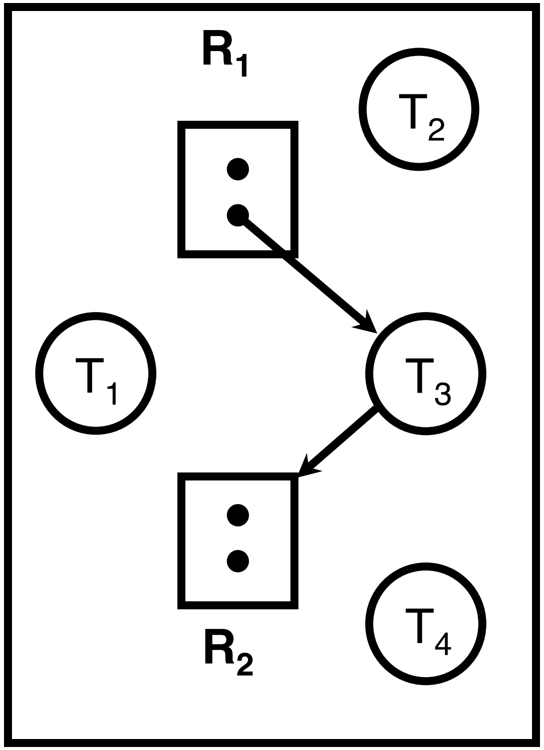
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [Alloc_node]
      done = false
    }
  }
} until(done)
```



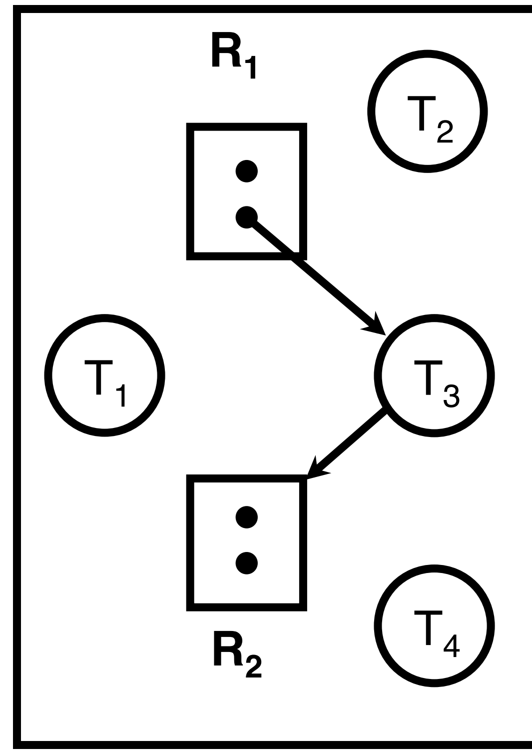
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
         [Avail] = [Avail] + [AllocT3]
         done = false
      }
   }
} until(done)
```



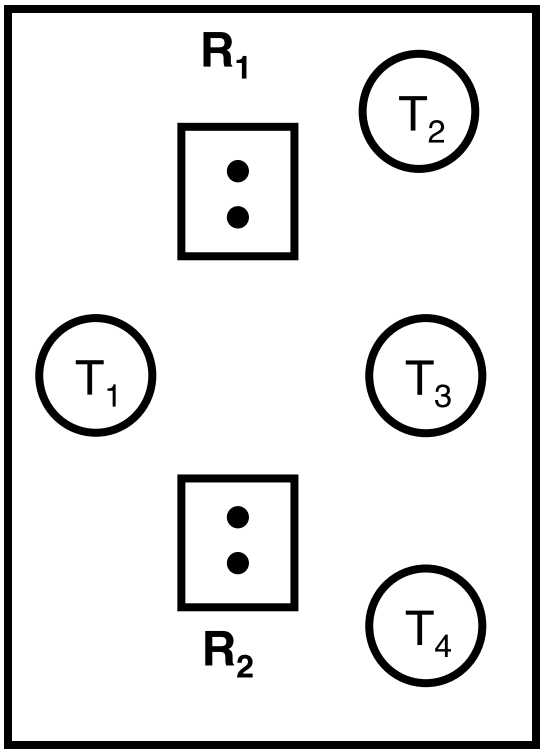
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
       [Avail] = [Avail] + [Alloc_T3]
       done = false
     }
   }
} until(done)
```



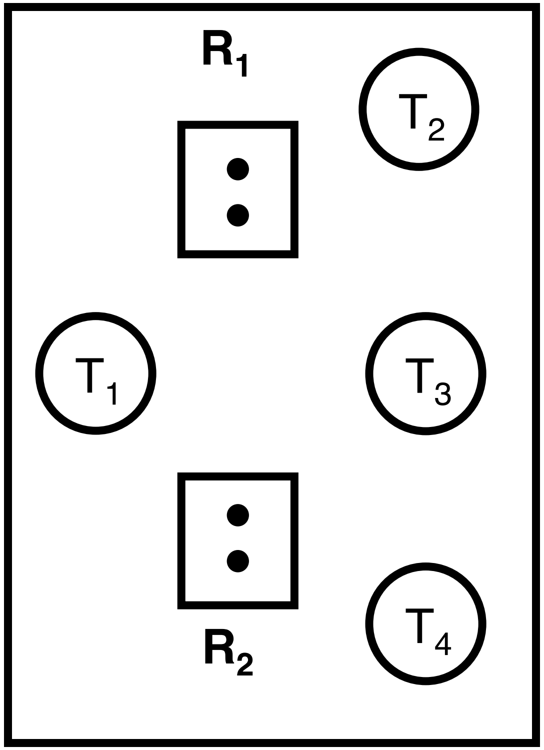
Deadlock Detection Algorithm: Example

```
[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] = [2,2]
UNFINISHED = {}

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



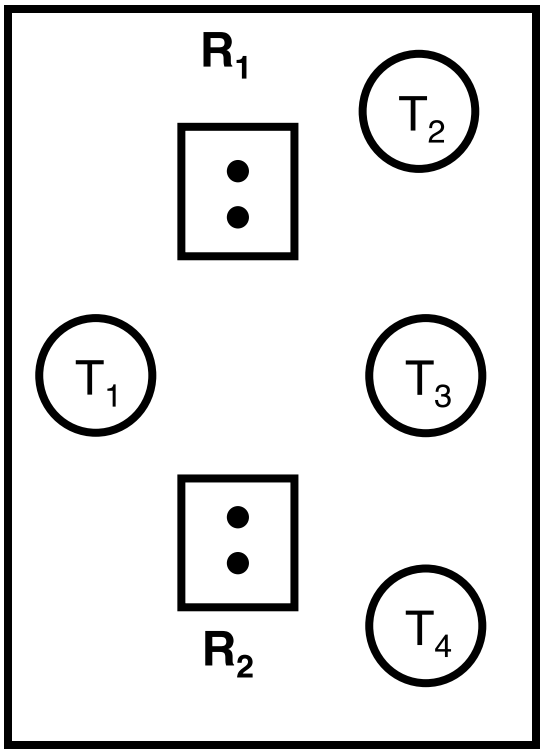
Deadlock Detection Algorithm: Example

```
[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 UNFINISHED
      [Avail] = [Avail] + [AllocT3]
      done = false
    }
  }
} until(done)
```



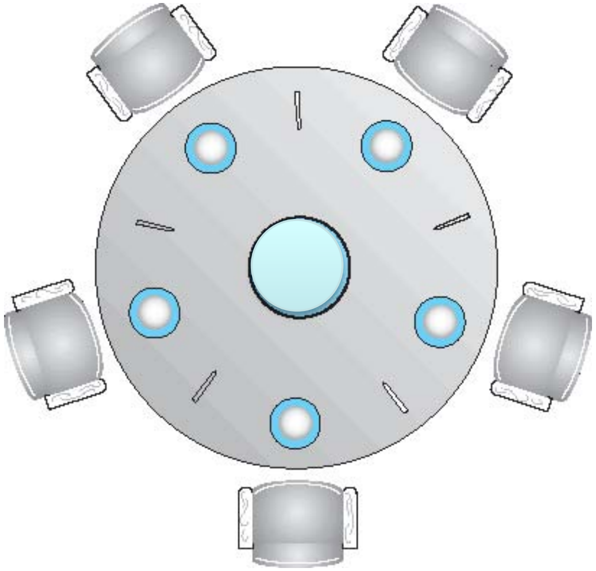
Deadlock Detection Algorithm: Example

```
[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] = [2,2]
UNFINISHED = {}

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



DONE!

Dining Philosophers 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

Conclusion

- Methods for Handling Deadlock
  1. Let deadlock occur, and do something about it afterwards
  2. Never let deadlock occur
  3. Ignore the problem and pretend that deadlock never occurs
- Safe state of a system: State in which resources can be allocated to each process while avoiding deadlock
- Banker's algorithm
  - **Safety Algorithm**
  - **Resource-request algorithm**