CS 475
Operating Systems

Department of Mathematics
and Computer Science

Lecture 7
Deadlocks

Review of Semaphores (Last year's exam)

- This function pops an item off each stack and pushes the items onto the opposite stack.
  - If either stack was empty, both stacks should be left unchanged.
- Each stack object associates with a semaphore initialized to 1.
- Thread-Safe Version
  - **There are 3 bugs in the code. Name them.**

```
// 2+ threads run this function
void stackSwap(*stack1, *stack2) {

  wait(stack1->sem);
  thing1 = pop(stack1);
  if (thing1 != NULL) {
    wait(stack2->sem);
    thing2 = pop(stack2);
    if (thing2 != NULL) {
      push(stack2, thing1);
      push(stack1, thing2);
      signal(stack2->sem);
      signal(stack1->sem);
    }
  }
}
```

One Big Bug ... Mutual Starvation (Deadlock)

- What *could* happen if ... we have two stacks: **Sa** and **Sb**.
  - Thread 1 calls **stackSwap(Sa,Sb)** and Thread 2 calls **stackSwap(Sb,Sa)** concurrently?

```
void stackSwap(*stack1, *stack2) {

  wait(stack1->sem);
  thing1 = pop(stack1);
  if (thing1 != NULL) {

    wait(stack2->sem);
    thing2 = pop(stack2);
    if (thing2 != NULL) {
      push(stack2, thing1);
      push(stack1, thing2);
      signal(stack2->sem);
      signal(stack1->sem);
    }
  }
}
```

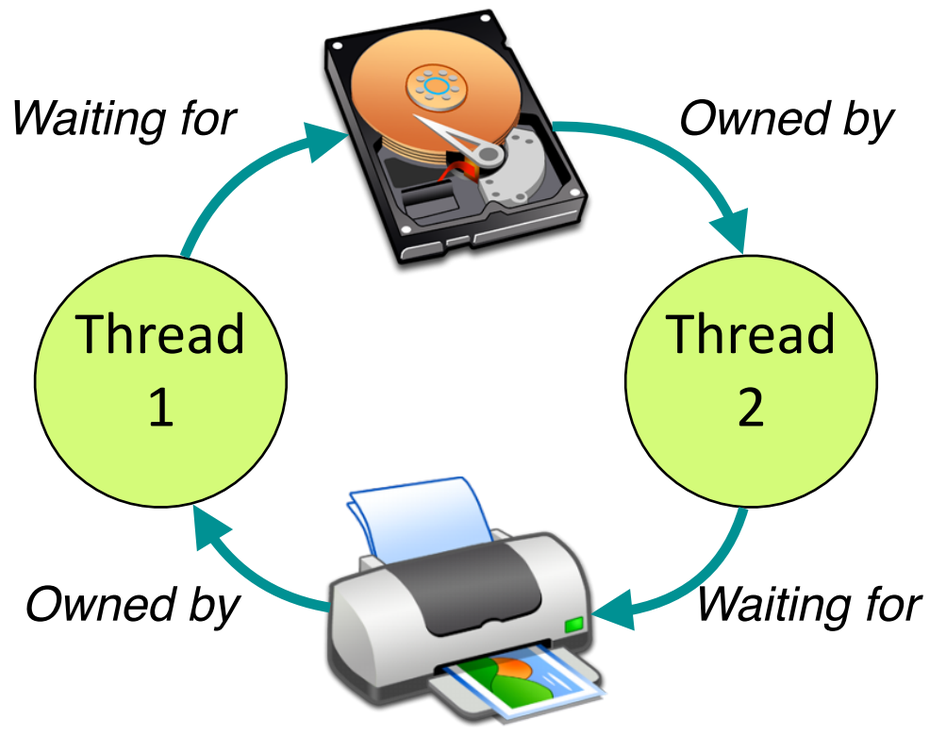
T1	T2
`stackSwap(Sa,Sb)`	
`wait(Sa->sem);`	
`thing1 = pop(Sa);`	
	`stackSwap(Sb,Sa)`
	`wait(Sb->sem);`
	`thing1 = pop(Sb)`
	`if (thing1 != ...`
`if (thing1 != ...`	
`wait(Sb->sem);`	
	`wait(Sa->sem);`
(wait forever)	(wait forever)

Deadlock and Starvation

- When synchronizing access across multiple threads...
  - Starvation: When a thread waits indefinitely for CPU execution.
  - Deadlock: When a set of threads are stuck waiting on an event that can only be caused by the another thread in the same set.

$$deadlock \implies starvation \quad \textit{(why?)}$$

$$starvation \not\implies deadlock \quad \textit{(why?)}$$



Another Example Deadlock

- Deadlocks are incredibly easy to produce.
  - Here, **lock1** and **lock2** are acquired in the opposite ordering.
  - [David's Code Example: deadlock.c]

```c
void* doStuff(void* args) {
        pthread_spin_lock(lock1);
        sleep(1);
        pthread_spin_lock(lock2);
        printf("Hi!\n");
        pthread_spin_unlock(lock2);
        pthread_spin_unlock(lock1);
}
```

Thread A

```c
void* doStuff2(void* args) {
        pthread_spin_lock(lock2);
        sleep(1);
        pthread_spin_lock(lock1);
        printf("Hi 2!\n");
        pthread_spin_unlock(lock1);
        pthread_spin_unlock(lock2);
}
```

Thread B

Topics for Today...

- Deadlock Conditions
- Ways to deal with deadlocks:
  - Deadlock Detection
    - Single Resource Allocation Graph
    - Multi-Resource Allocation Graph
  - Deadlock Avoidance
    - Banker's Algorithm

Four Deadlock Conditions

- **Necessary** and **Sufficient** conditions for deadlocks to occur:
  - "**Necessary**:" All 4 must exist for there to be a deadlock
    - If we take any of these conditions away, the system will be deadlock-free
  - "**Sufficient**:" If the 4 are shown to exist currently, then there *must* be a deadlock

- The four deadlock conditions include:
  - Mutual Exclusion
  - No Preemption
  - Hold and Wait
  - Circular Wait

Deadlock Conditions 1 & 2

Condition 1: Mutual Exclusion

- Only one thread can access the resource at any time

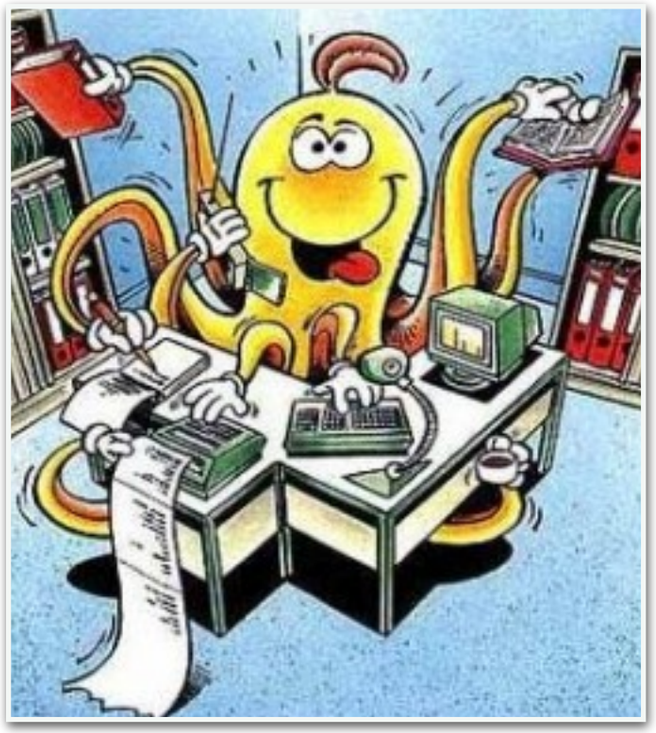
Condition 2: No Preemption

- A thread can't be *coerced* to release its hold on a resource.
- Why would preemption be considered unsafe?
  - *e.g., try forcibly taking a lock away from a thread*

Deadlock Condition 3: Hold and Wait

Condition 3: Hold and Wait

- When threads are allowed to:
  - *Hold* on to one or more resources, and be...
  - *Waiting* to acquire additional resources



To do my taxes, I need to acquire the following resources.

- A calculator
- Tax form 1040EZ
- My W2
- My spouse's W2
- My 1098

For each one I obtain, I don't put it back down for someone else to take.

Deadlock Condition 4: Circular Wait

Condition 4: Circular Wait

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-1} is waiting for a resource that is held by T_n
T_n is waiting for a resource that is held by T_1

My spouse needs calculator and her own W2 for something else.

I obtained the calculator.

She has her W2, which I need.

We're waiting for each other to give one up.

Example: Existence of 4 DL Conditions

- Mutual Exclusion
- No-Preemption
- Hold-and-Wait
- Circular Wait

Thread A

```
void* doStuff(void* args) {
        pthread_spin_lock(lock1);
        sleep(1);
        pthread_spin_lock(lock2);
        printf("Hi!\n");
        pthread_spin_unlock(lock2);
        pthread_spin_unlock(lock1);
}
```

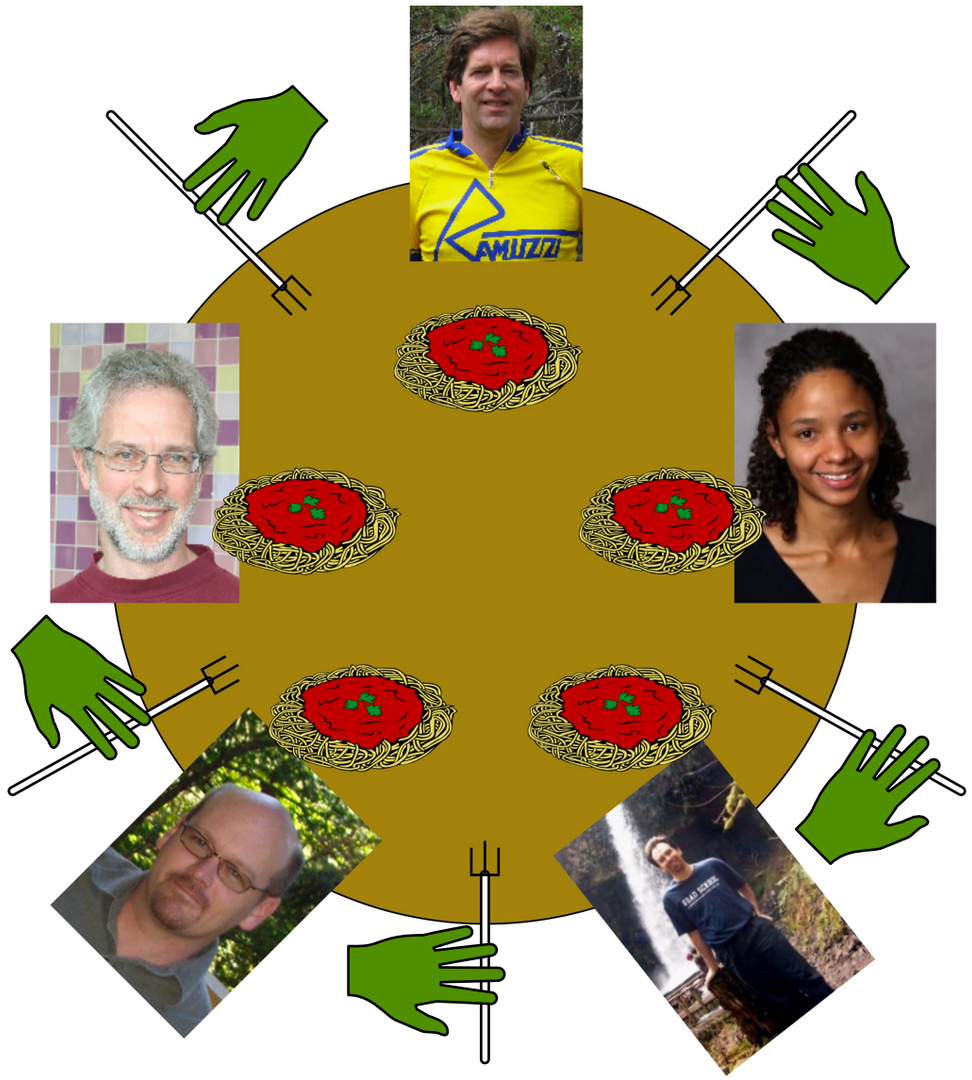
Thread B

```
void* doStuff2(void* args) {
        pthread_spin_lock(lock2);
        sleep(1);
        pthread_spin_lock(lock1);
        printf("Hi 2!\n");
        pthread_spin_unlock(lock1);
        pthread_spin_unlock(lock2);
}
```

Another Example: Dining Philosophers

- 5 forks, 5 philosophers. Need both ***left*** and ***right*** forks to eat

- Each philosopher has only two states:
  - Eating
    - Need to acquire both forks when eating
  - Thinking
    - Don't need forks to think
    - Put them both down



Deadlock:

*Everyone requests **right fork**, then...*

*... tries to grab their **left fork** (deadlock)*

What Can the OS Do?

- The OS should intervene when there's a deadlock.

- Several approaches toward handling deadlocks
  - **Prevent** deadlocks from ever happening by removing DL condition(s)
  - OS only allocates resources in such a way that it avoids unsafe situations (**Avoidance**)
- **Detection:** Allow the OS to enter a deadlocked state(!)
  - Detect it
  - Recover from it

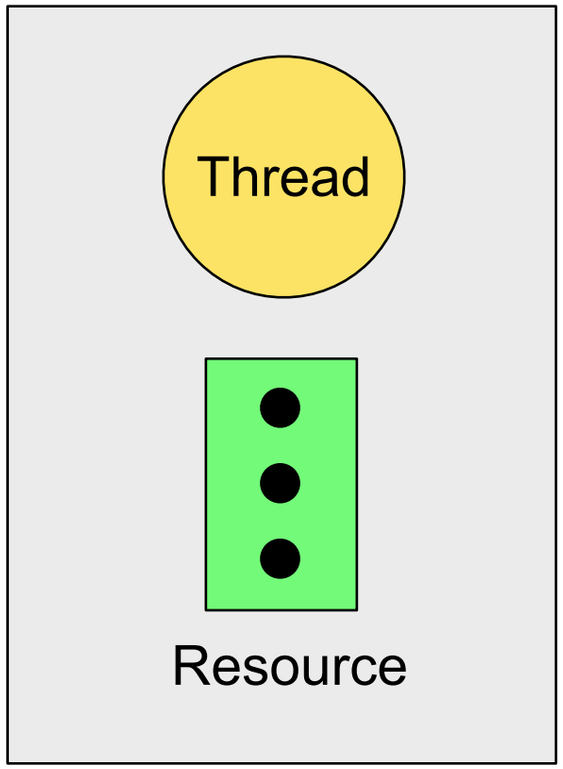
Topics for Today...

- Deadlock Conditions
- Ways to deal with deadlocks:
  - Deadlock Detection
    - Single Resource Allocation Graph
    - Multi-Resource Allocation Graph
  - Deadlock Avoidance
    - Banker's Algorithm

Modeling Resource Allocation (Graphs)

- Modeling deadlocks as graphs. Nodes can be:
  - Threads, or
  - Resource types
    - CPU, memory, disk, printer, semaphores, ...

Types of Nodes/Vertices



- Each resource type may have multiple instances
  - (Each instance is represented by ●)
  - There's one instance of a lock
  - A quad-core processor means we have 4 CPU instances
  - We may have 2 or more printers

Resource Allocation Graph (RAG)

- We can model resources' state using a *Directed Graph* $G = (V, E)$

- $V = T \cup R$ is partitioned into two types of vertices (nodes):
  - $T = \{t_1, \ldots, t_n\}$ is the current set of threads running in the system
  - $R = \{r_1, \ldots, r_m\}$ is the set of resource types in the system

- E (the set of edges) is partitioned into two types of directed edges:
  - Request edge: (t_i, r_j) exists when t_i requests an instance of resource r_j
  - Allocation edge: (r_j, t_i) exists when r_j is assigned to t_i

Resource Allocation Graph (Cont.)

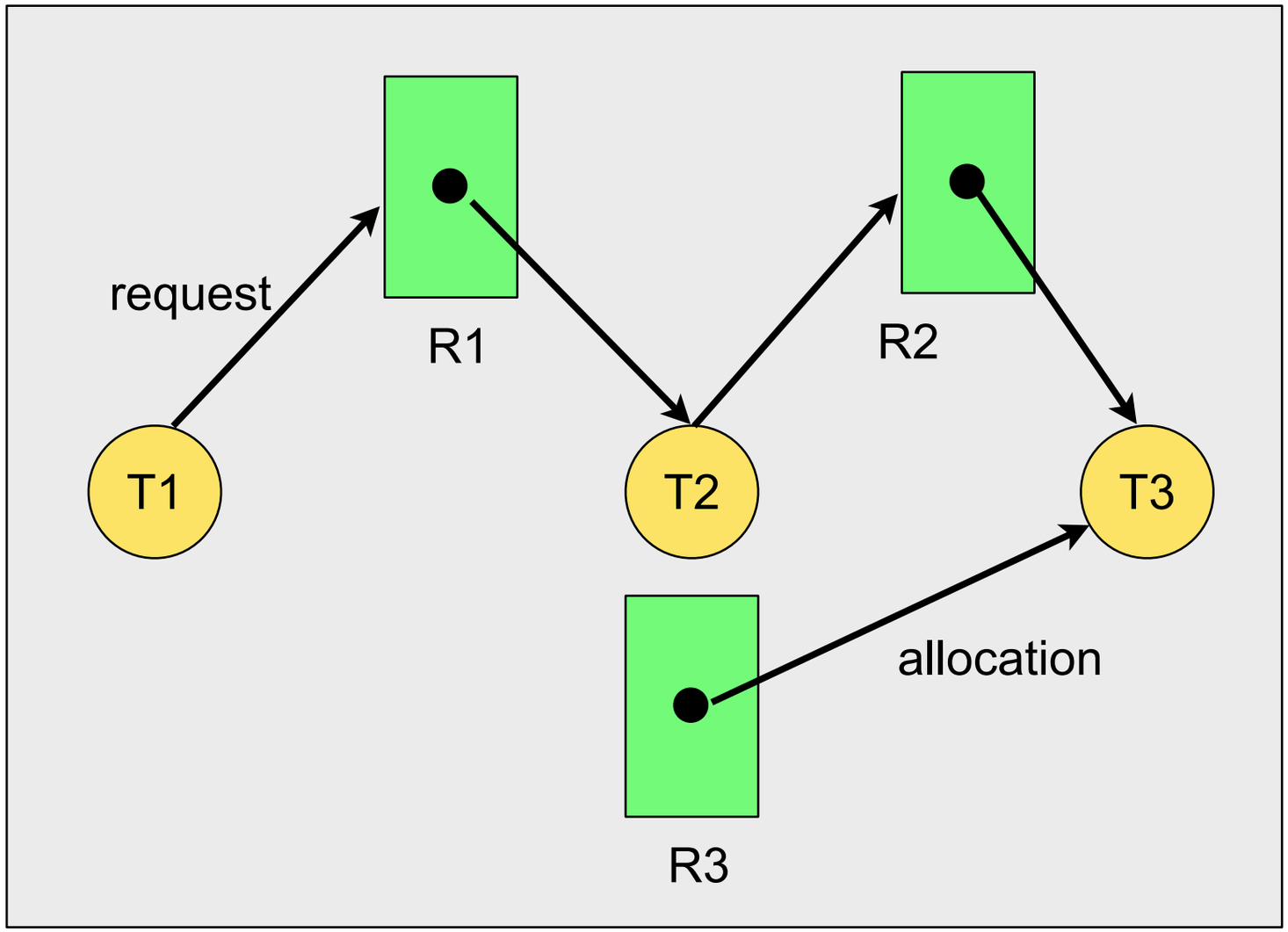
- How to build a RAG?

- Event-driven
  - When a request for a resource is made, **add** request edge
  - When a request is fulfilled, **change** request edge to allocation edge
  - When a thread releases a resource, **remove** allocation edge

- Therefore, the OS needs to update allocation and deallocation requests
  - `acquire()` and `release()` must be system calls

Example with Single-Instance Resources

- For simplicity, suppose our system does not have multiple resource instances: One CPU, One Printer, ...



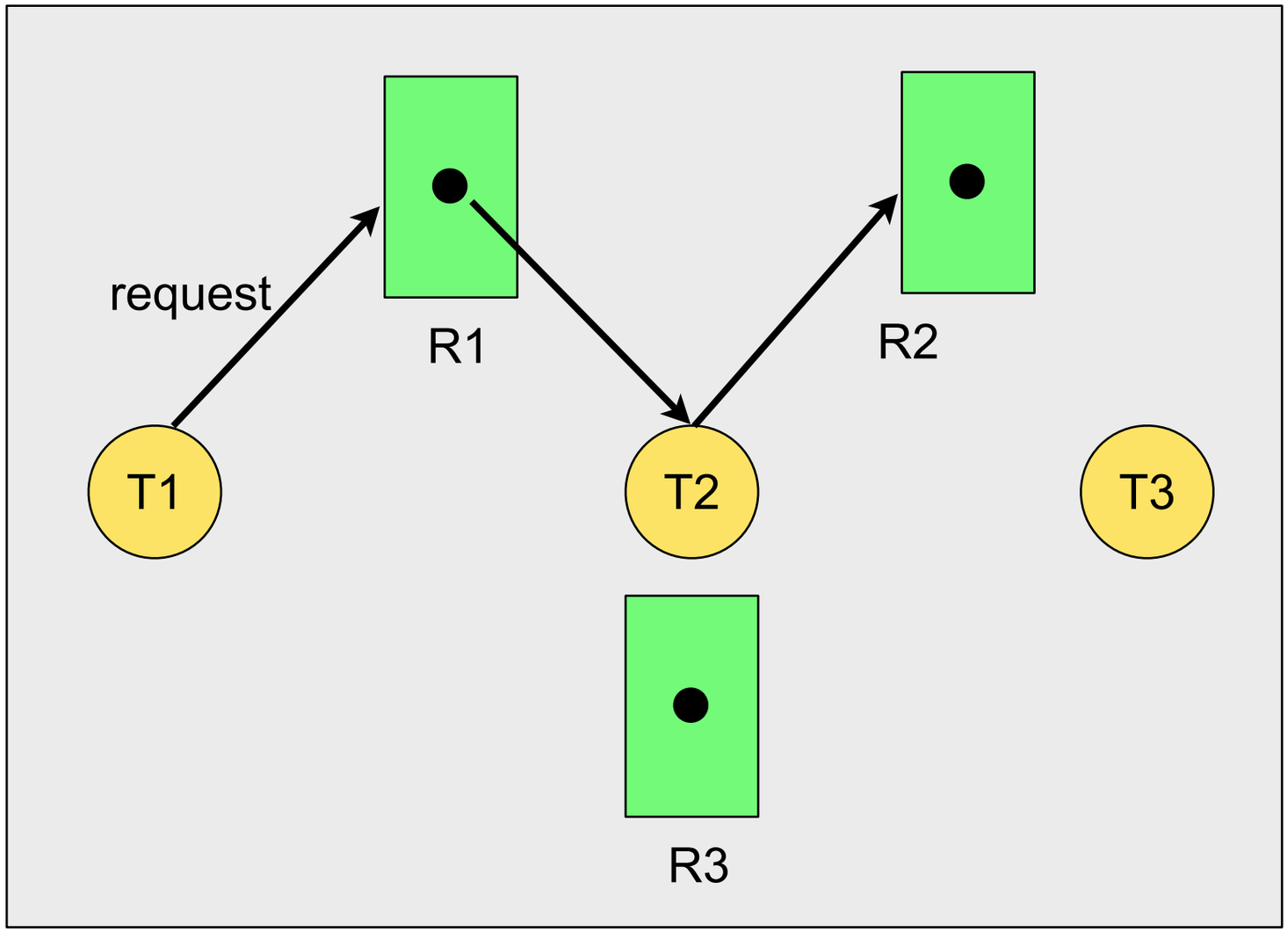
Is there a Deadlock in this RAG?

How to check?

- Can all threads finish execution?
- Simulate execution and try to "reduce the edges"

Example with Single-Instance Resources

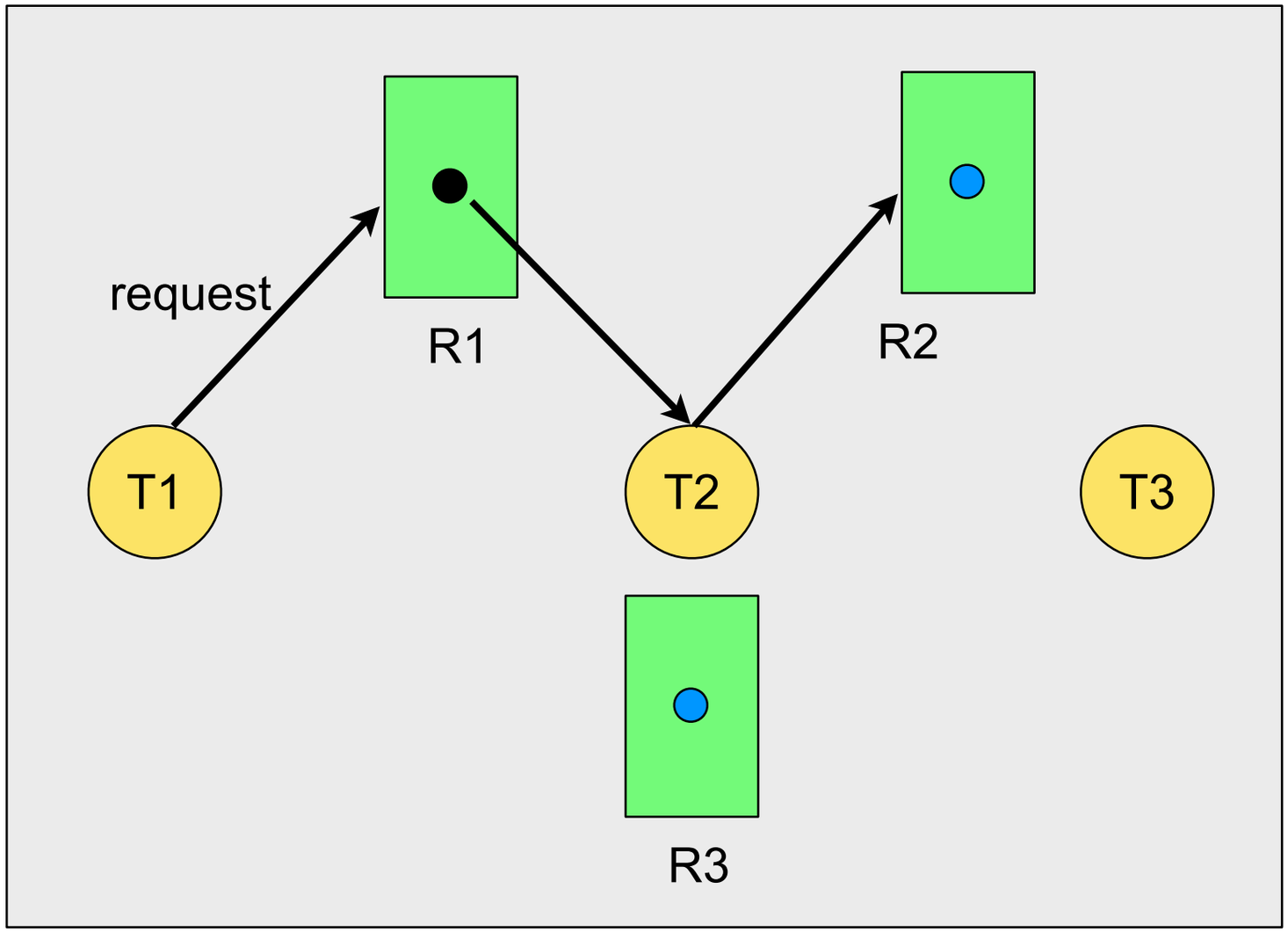
- For simplicity, suppose our system does not have multiple resource instances: One CPU, One Printer, ...



T3 has all the resources it needs and can finish execution!

Example with Single-Instance Resources

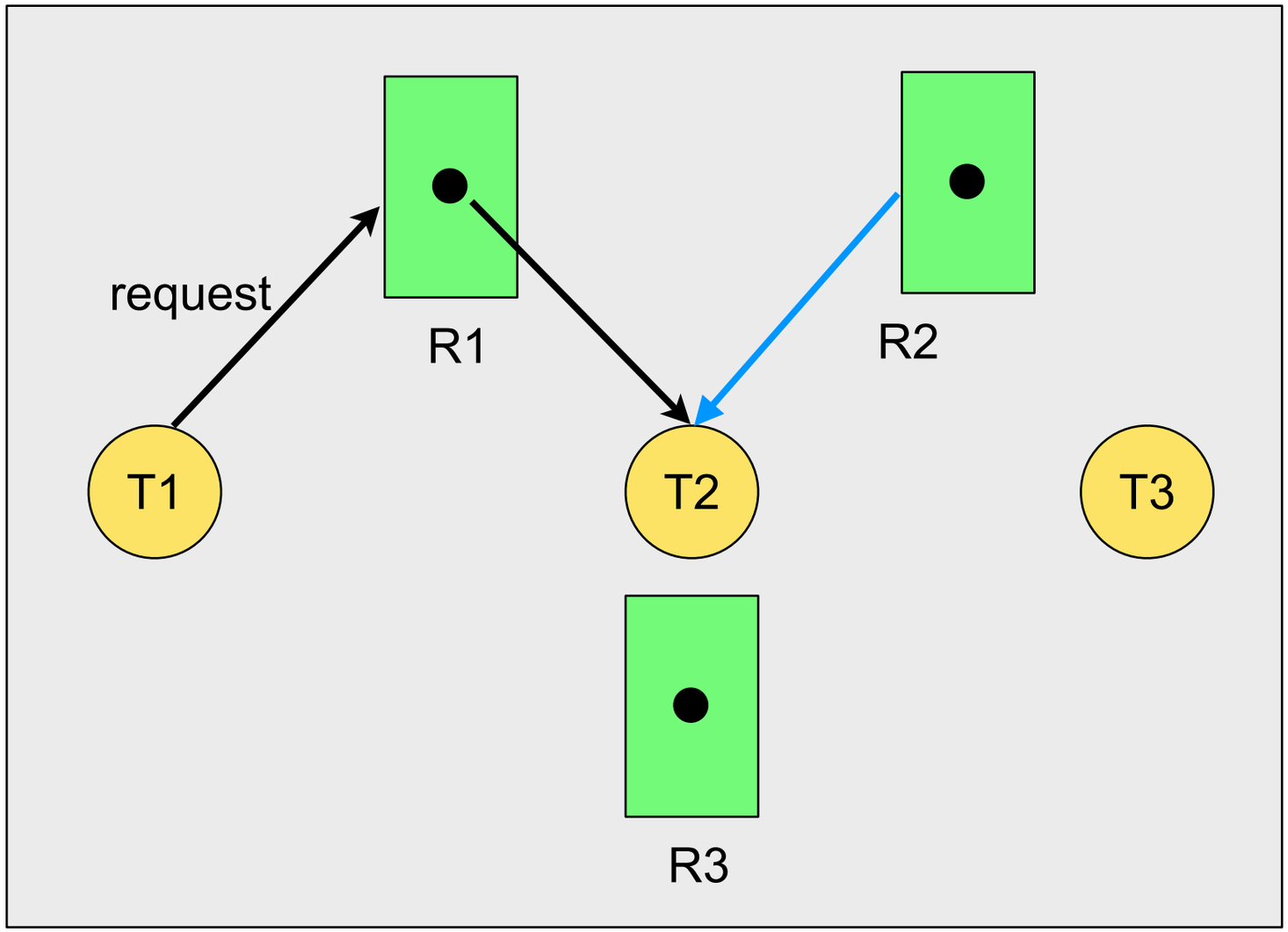
- For simplicity, suppose our system does not have multiple resource instances: One CPU, One Printer, ...



An instance of R2 and R3 have been freed!

Example with Single-Instance Resources

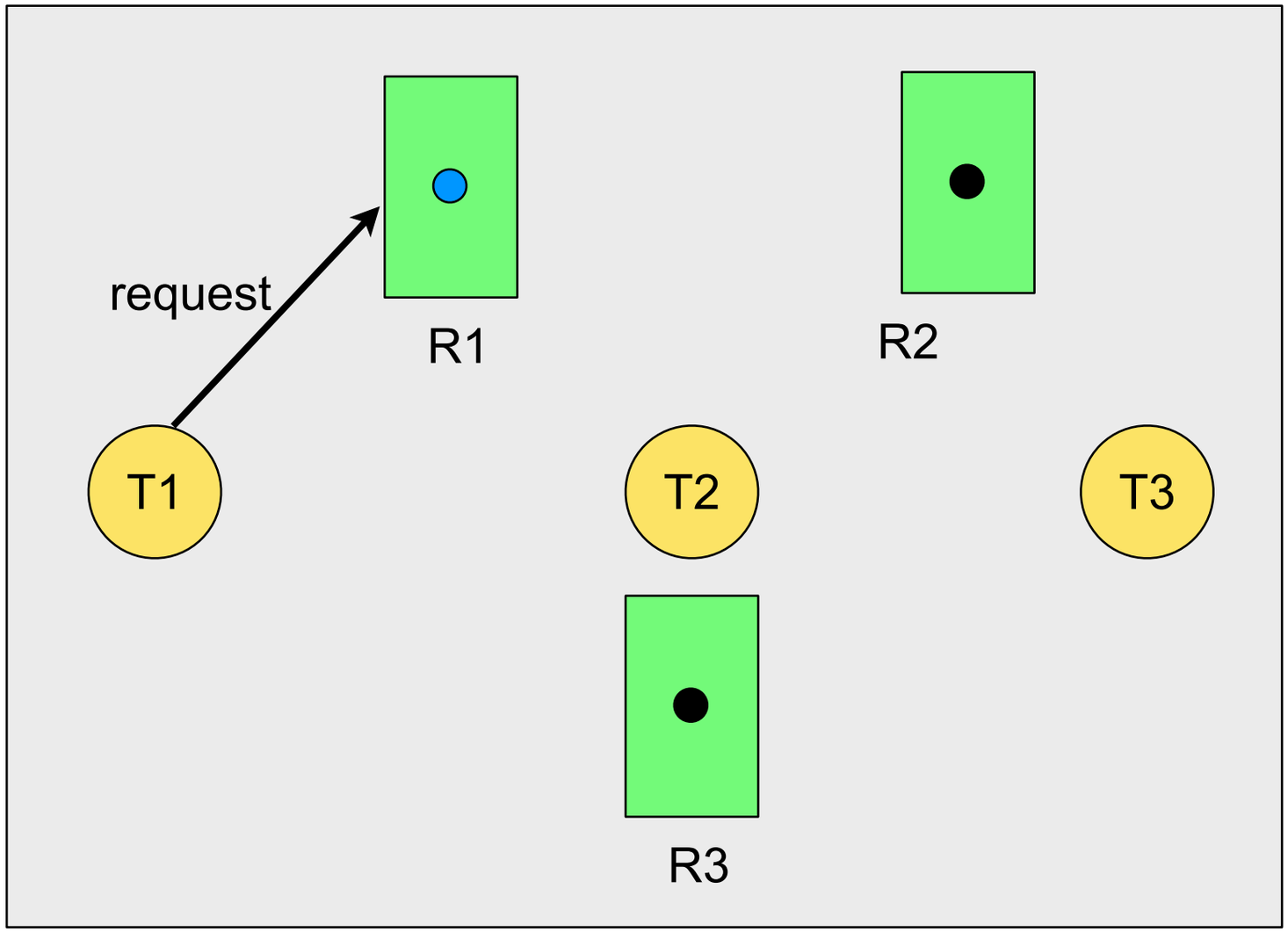
- For simplicity, suppose our system does not have multiple resource instances: One CPU, One Printer, ...



An instance of R2 can now be assigned to T2.

Example with Single-Instance Resources

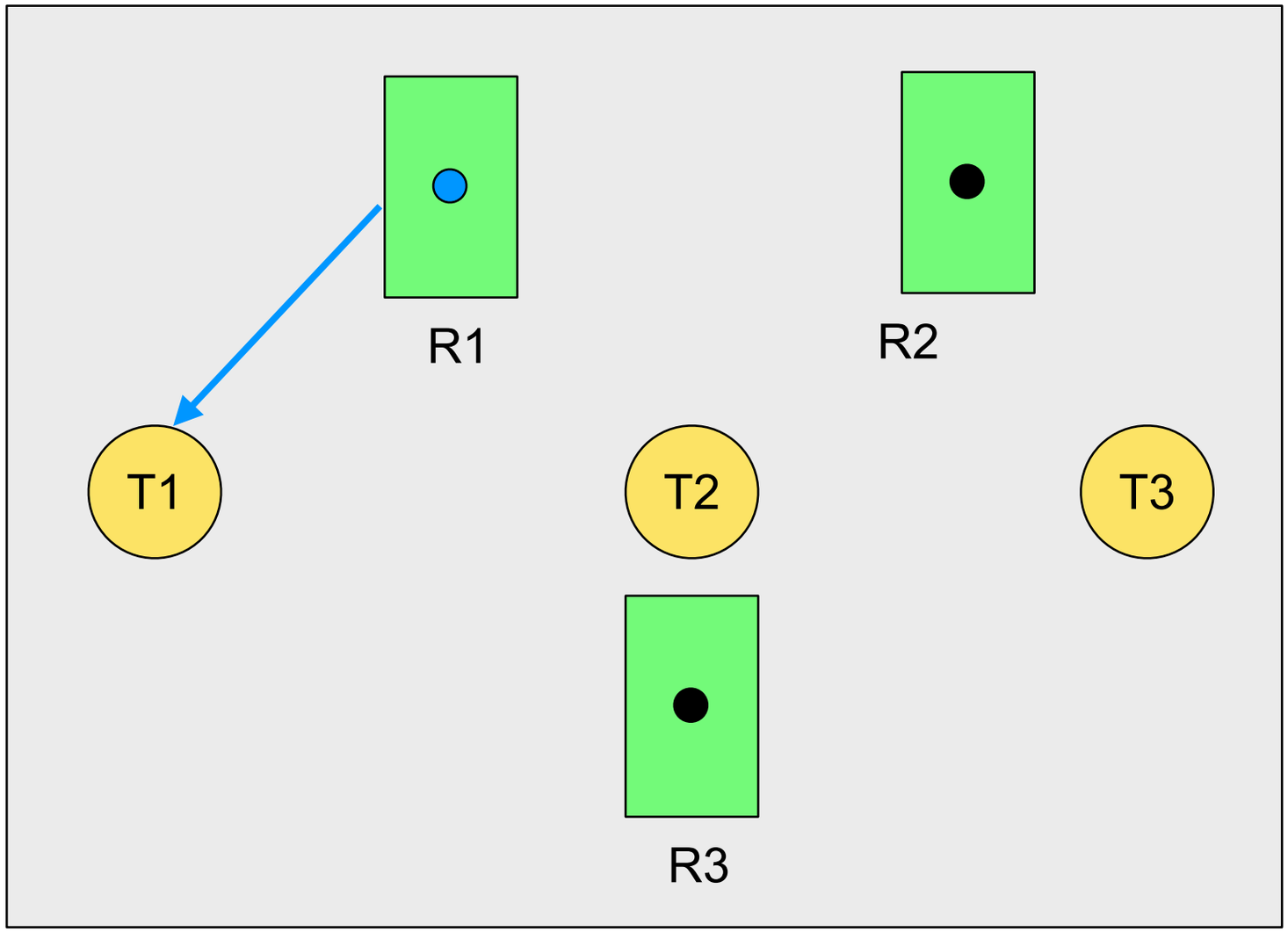
- For simplicity, suppose our system does not have multiple resource instances: One CPU, One Printer, ...



T2 has all the resources it needs and can finish execution!

Example with Single-Instance Resources

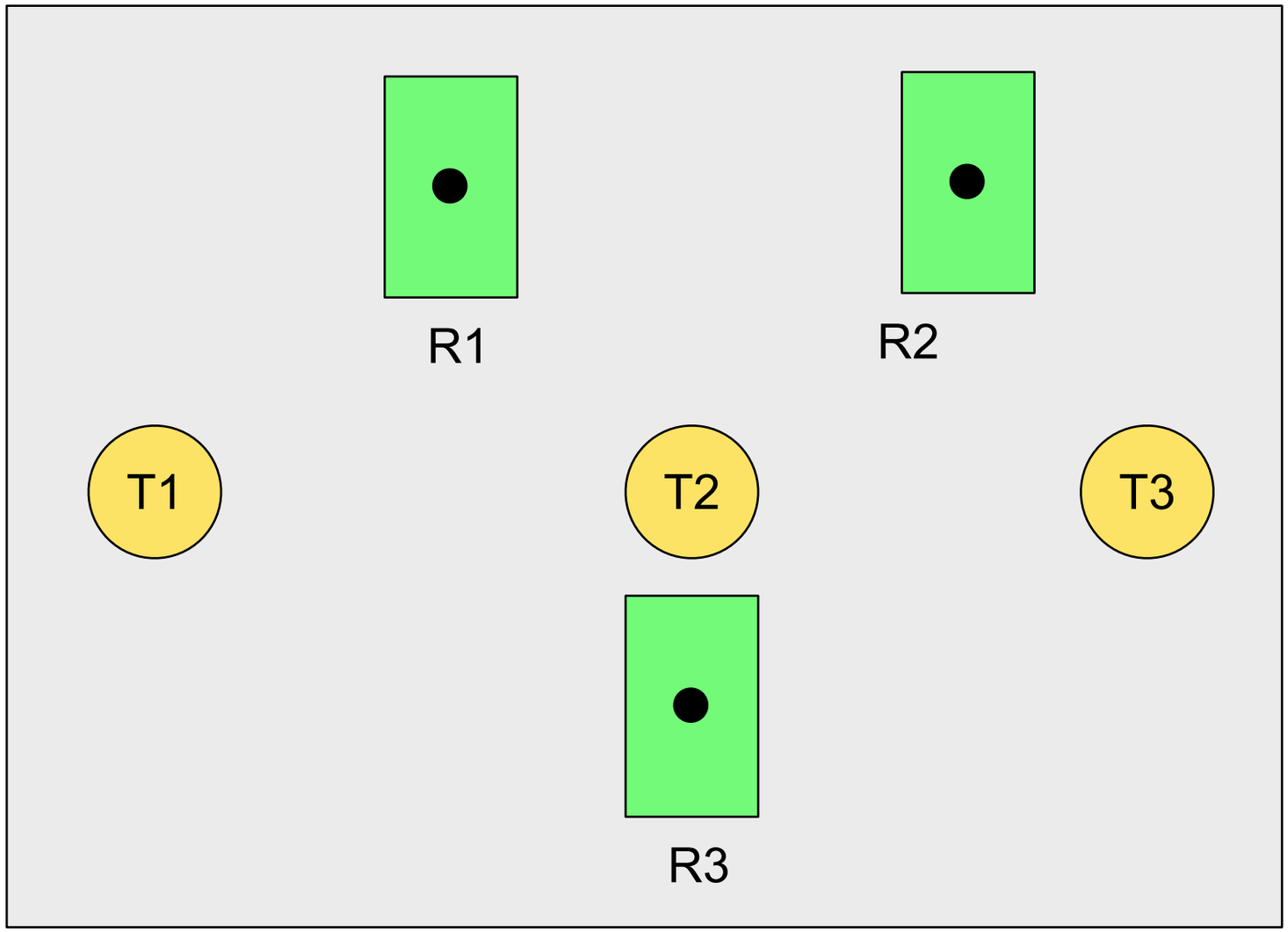
- For simplicity, suppose our system does not have multiple resource instances: One CPU, One Printer, ...



An instance of R1 can be allocated to T1.

Example with Single-Instance Resources

- For simplicity, suppose our system does not have multiple resource instances: One CPU, One Printer, ...

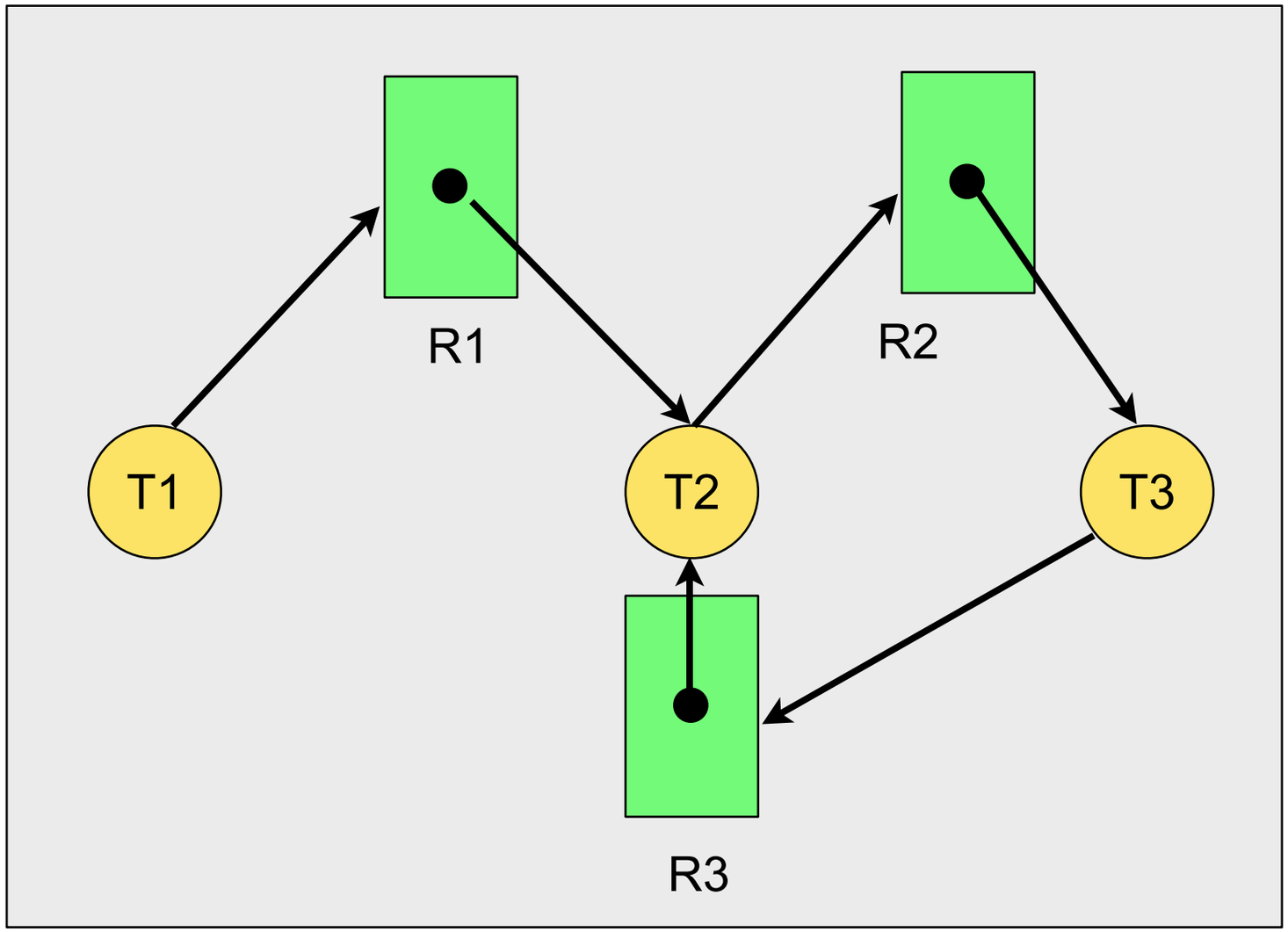


T1 can finish.

T1, T2, T3 all finished: deadlock free!

Another Example

- How about now? Try reducing edges like earlier.



Is there a deadlock now?

What is different about this and the previous RAG?

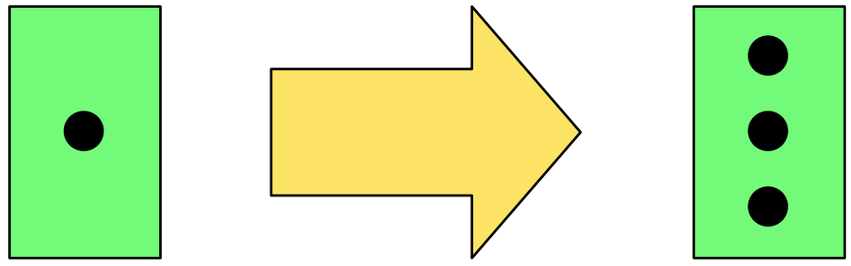
Can you identify the four deadlock conditions in this RAG?

RAG for Single-Instance Resources

- Deadlock detection is trivial for single-instance resources:
  - No Cycle => No Deadlock 
    - Because *Circular Wait* is a necessary condition for a deadlock, and you show it doesn't exist!
  - Cycle => Deadlock 
    - All four conditions would hold in the presence of a cycle
    - ... which is **sufficient** for the presence of a deadlock

- Cycle detection is just a variant of the Depth First Search (DFS) algorithm.
  - Complexity: $O(|V| + |E|)$

RAGs with Multi-Instance Resources (mRAG)

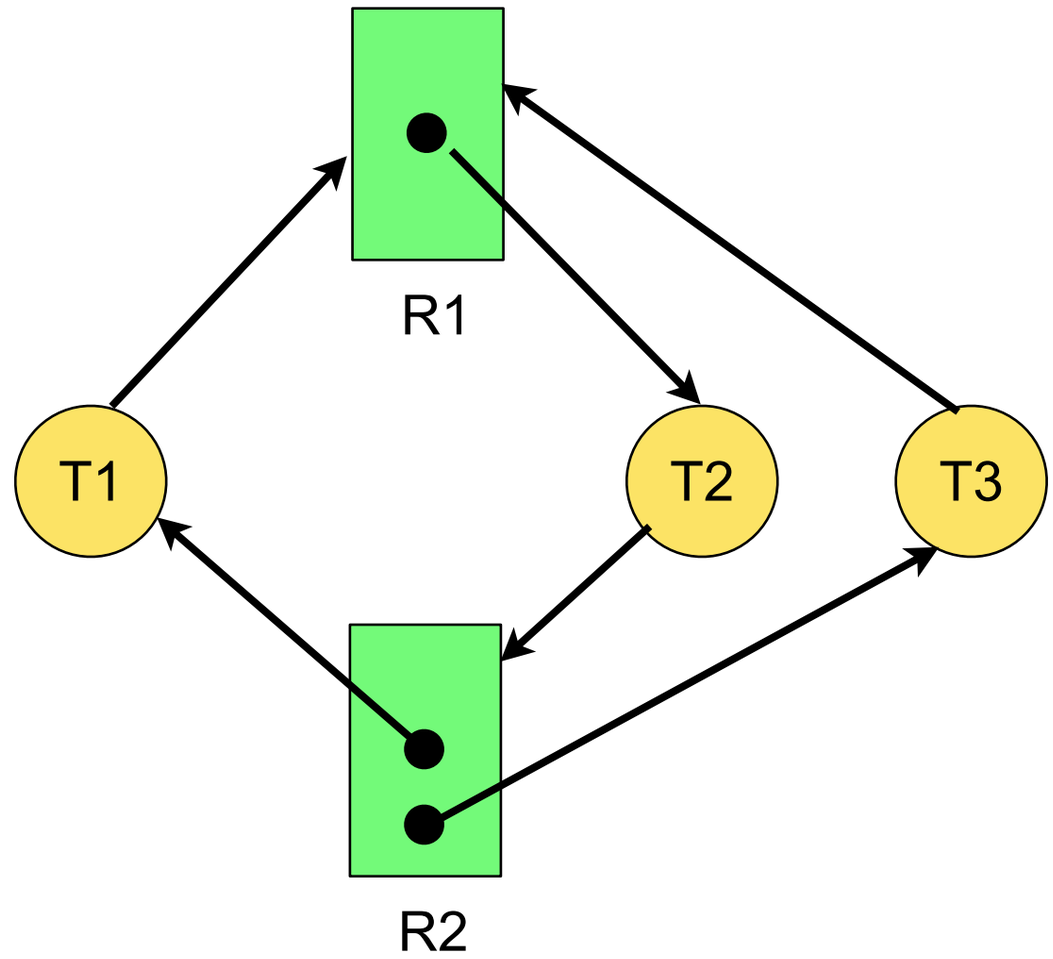


- In general, we should consider multiple resource instances
  - In practice, we would have multiple CPU cores
  - We might have multiple disks, printers, graphics cards

- *New question: Would the same cycle-detection algorithm scale to multi-instance resources?*

mRAG Example

- Check whether cycle exists.
- Is there a deadlock?

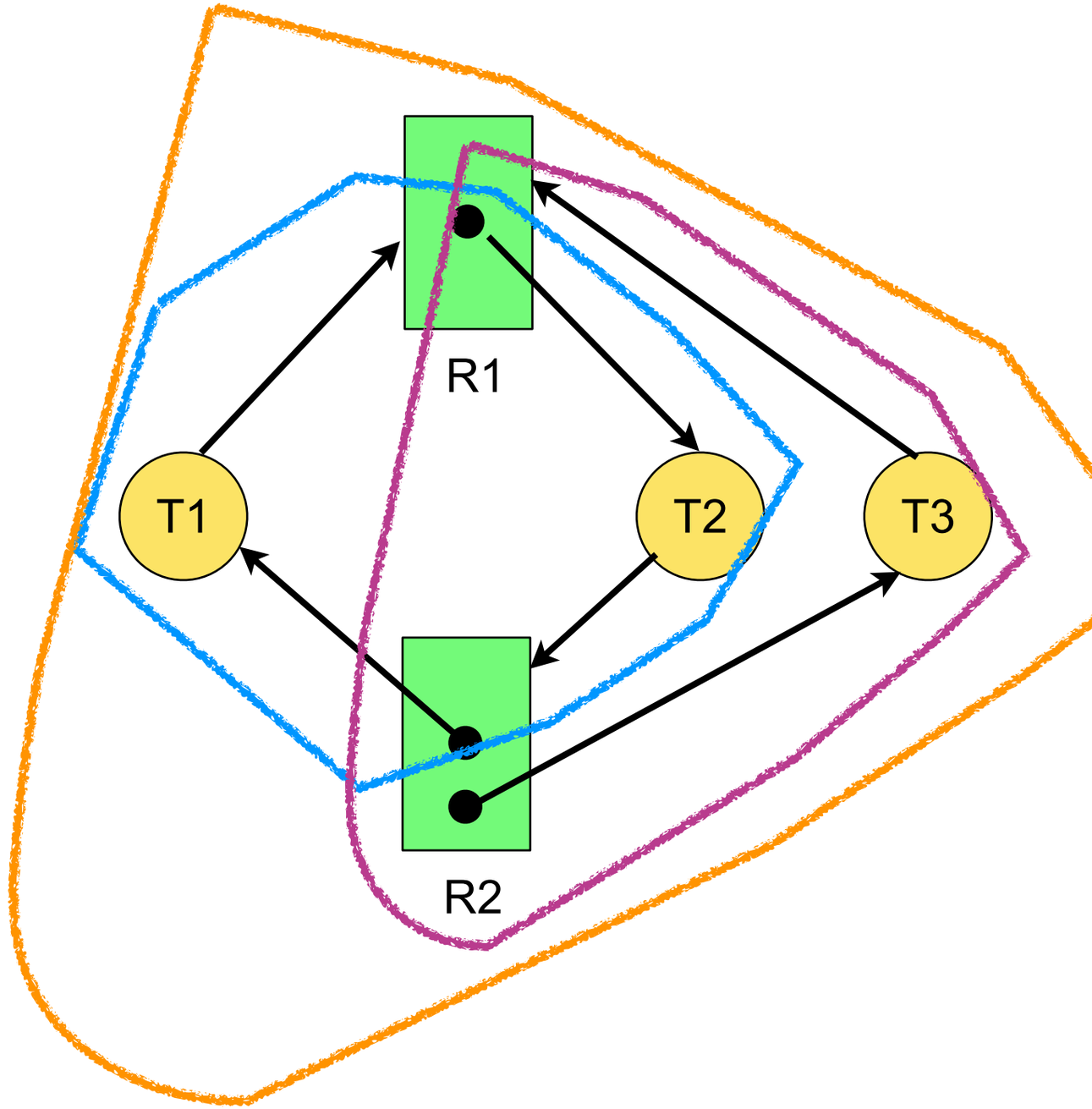


mRAG Example (2)

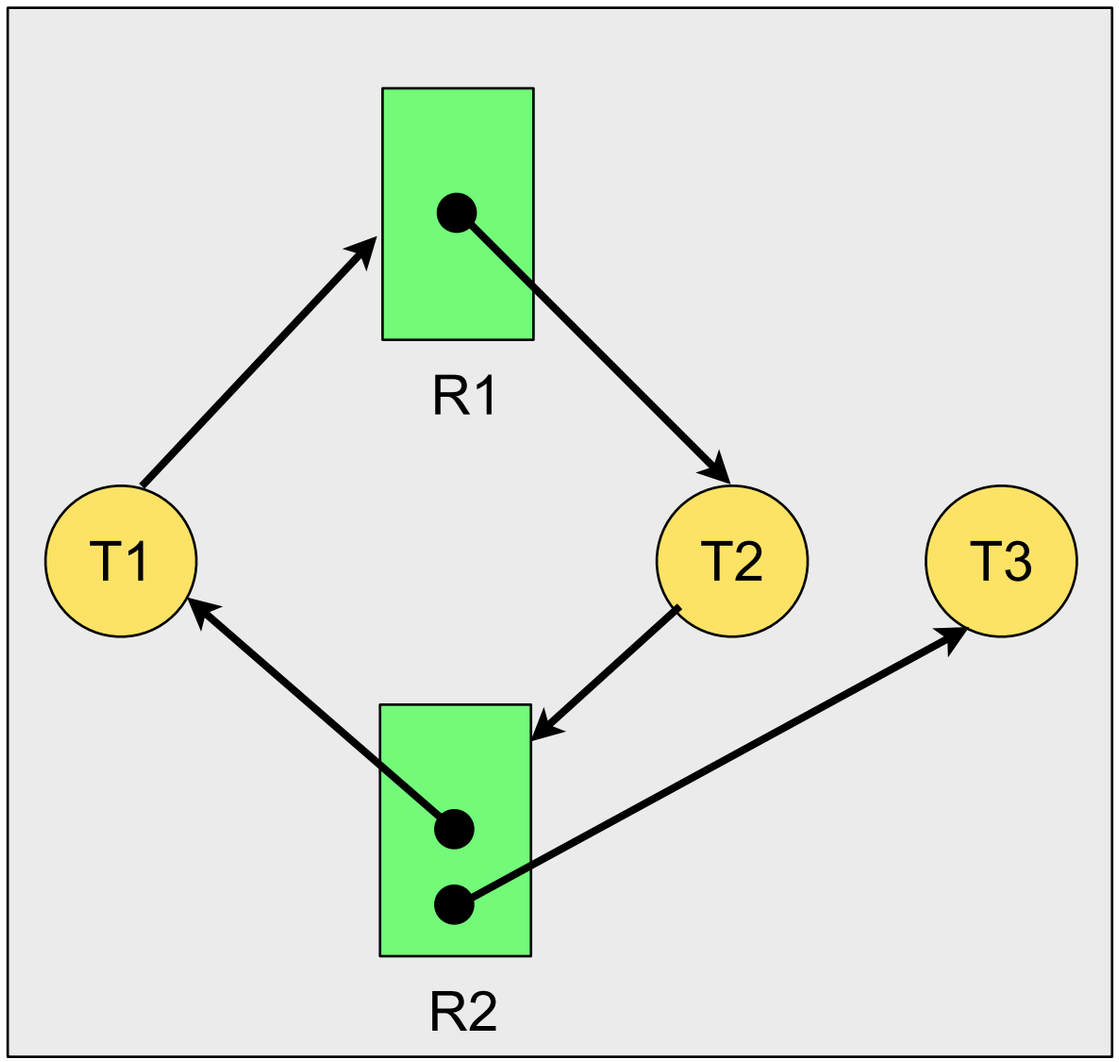
- Cycle exists... (actually several)

  ...and there is a deadlock!

- The cycle-detection algorithm seems to work for mRAGs also!

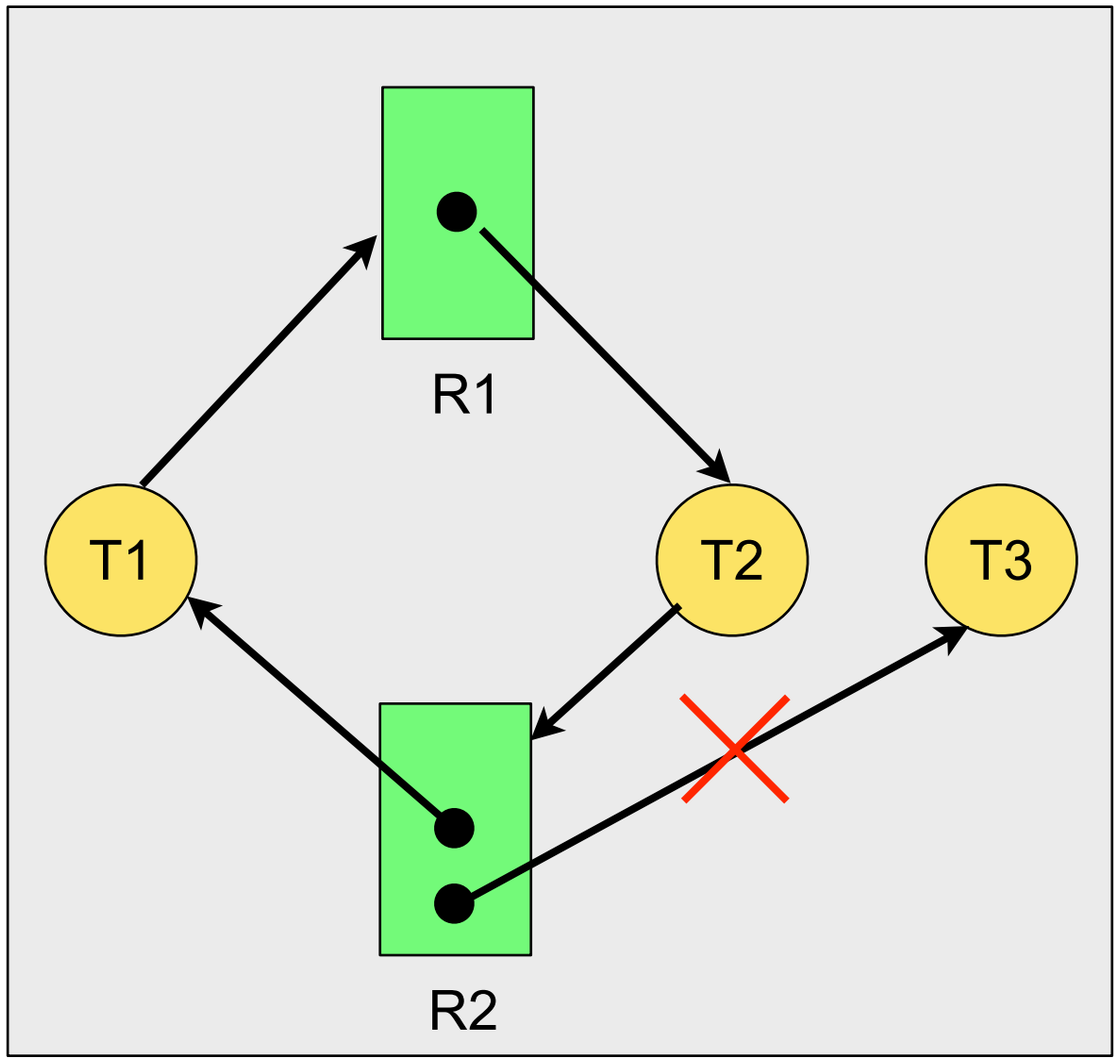


Another mRAG Example (1)



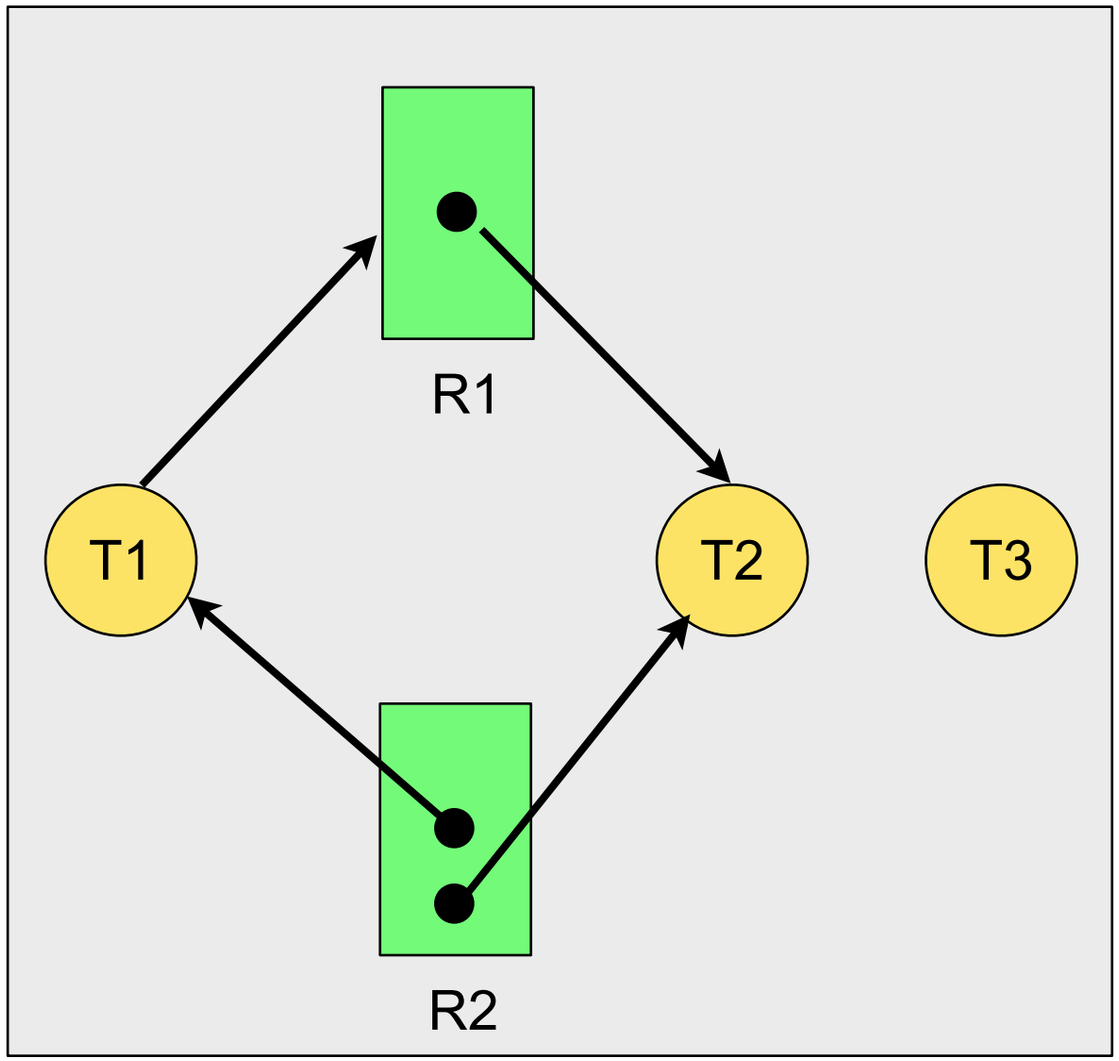
- Let's make a small change
  - Remove request edge (t_3, r_1)
- A cycle *still* exists.
- Deadlock?

Another mRAG Example (2)



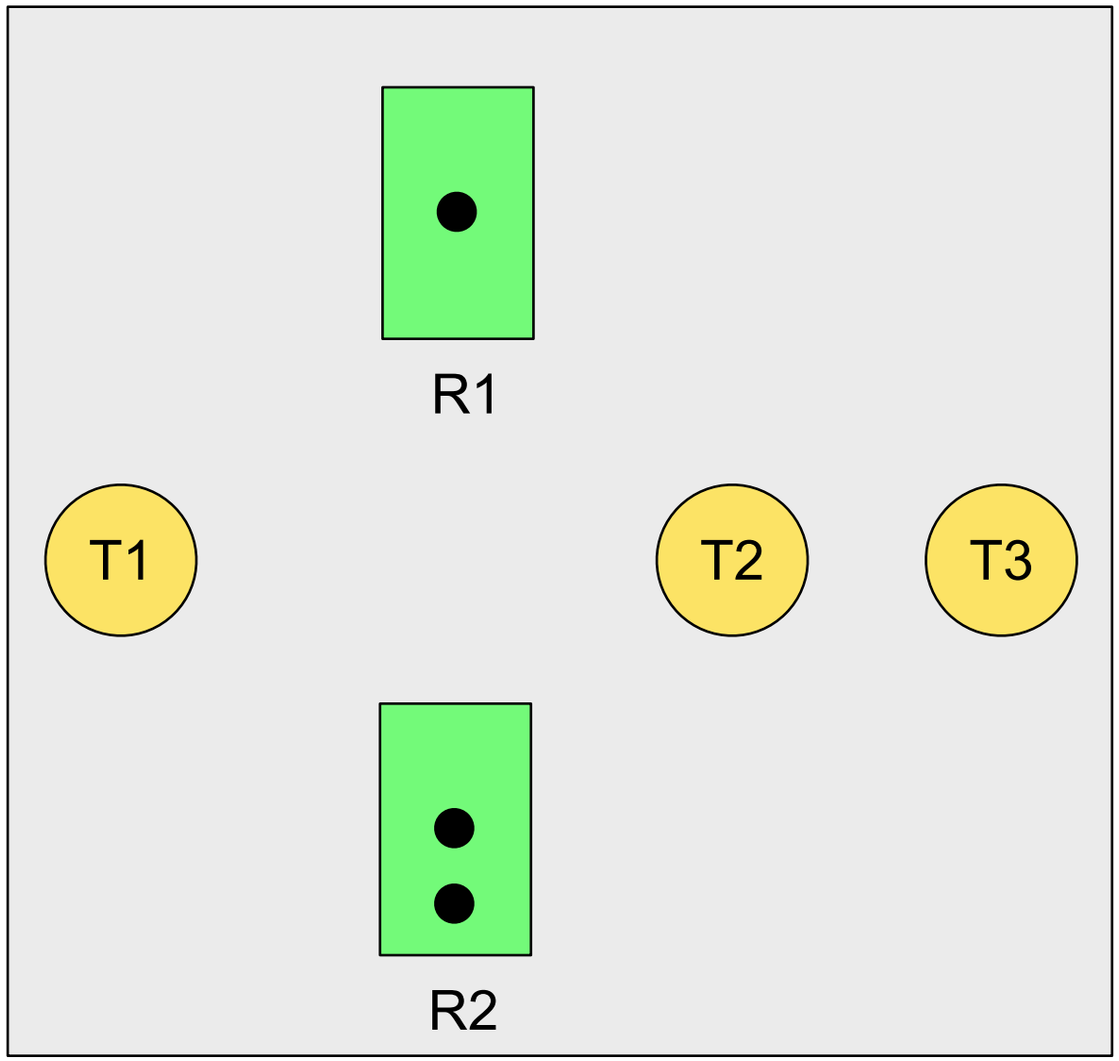
- T3 is not waiting, so it finishes
  - Allocation edge removed,
  - Instance of R2 reclaimed by OS

Another mRAG Example (3)



- The freed instance of R2 gets assigned to T2
- Request edge transforms into allocation edge

Another mRAG Example (4)

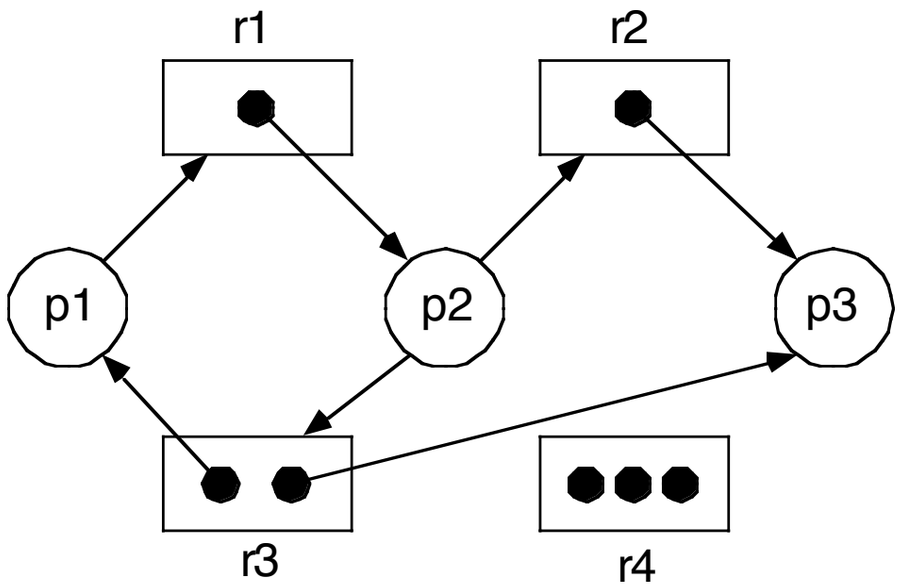


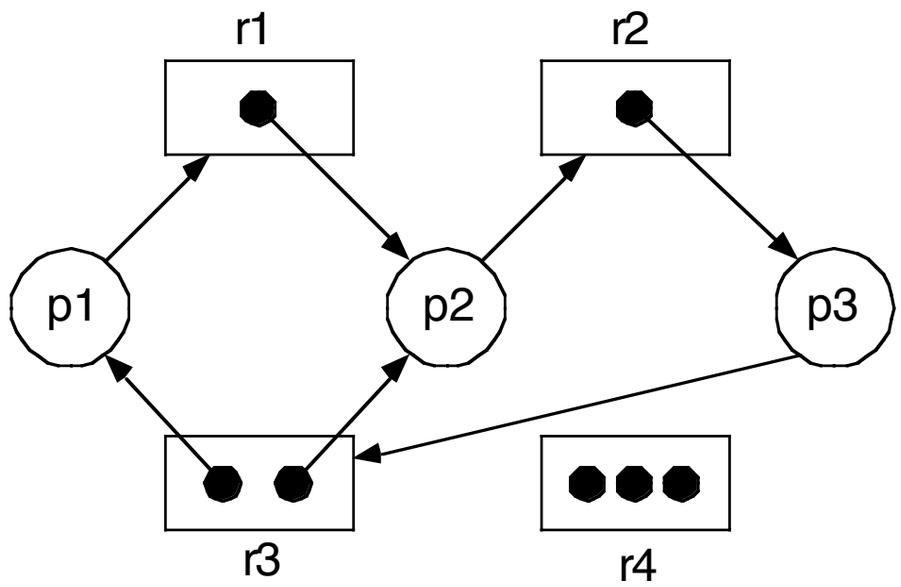
- Cycle is now removed!
  - T2 can finish, then T1 can finish!
  - ***No more deadlock!***
- What we know: For multi-resource instances,
  - No cycle in RAG => No deadlock
  - Cycle in RAG => No clue!

Towards Deadlock Detection of mRAGs

- If a cycle is not strong enough for deadlock detection of mRAGs, then what is?

- **Observe:** One of the following mRAGs has a deadlock.
  - They both have cycles...
  - But what's *different* about the nature of their cycles?



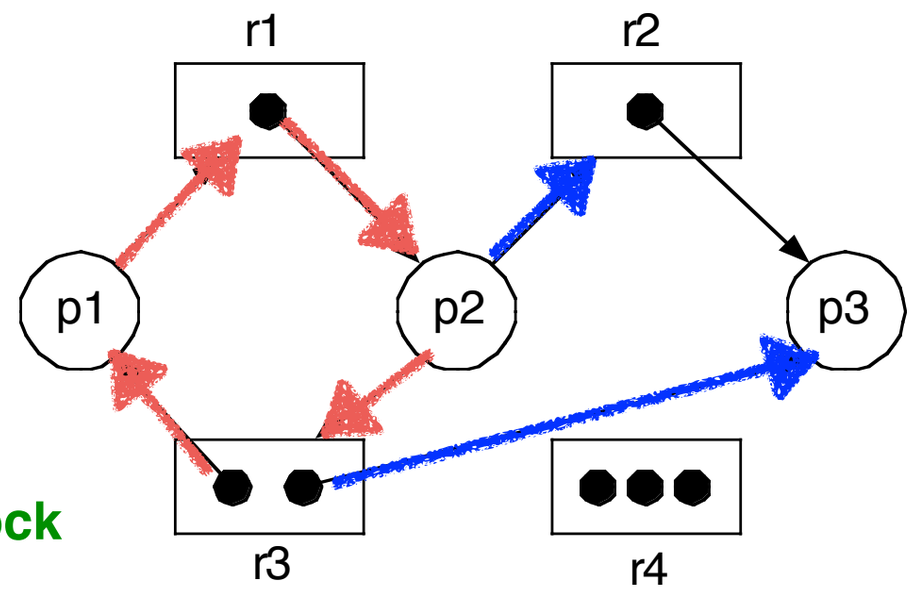


Def'n: Knots!

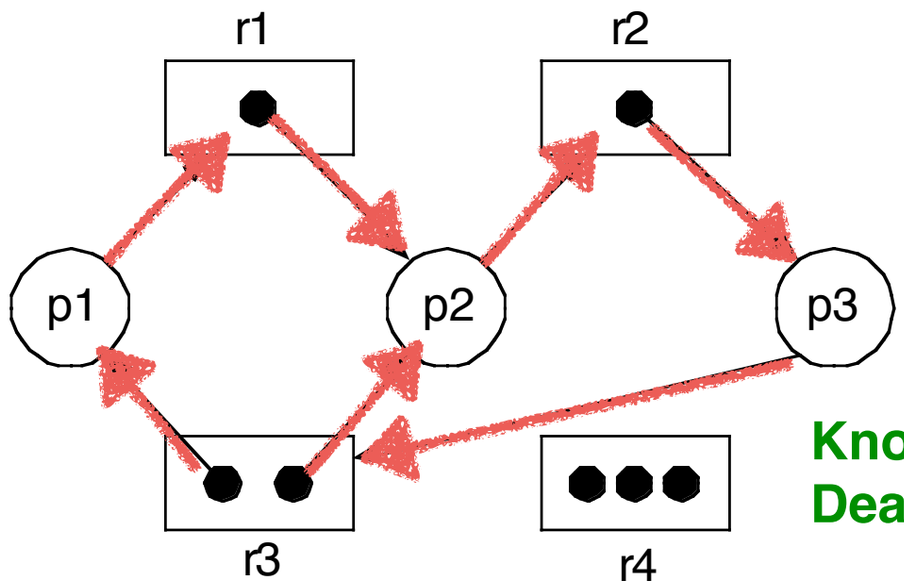
- A *Knot K* in a directed graph $G = (V, E)$ is a subgraph such that:
  - K is the *maximal* strongly-connected component (SCC)
    - SCC? Any two nodes $u, v \in K$ must be *reachable* from each other
    - Maximal? No other SCC is a proper subset of K
  - No nodes outside of K is reachable from K
    - *(Essentially, a knot is an SCC you can't escape.)*

Maximal Reachability (SCC)

Escape outside of SCC



No knot,
No deadlock



Knot present,
Deadlock!

Def'n: Knots!

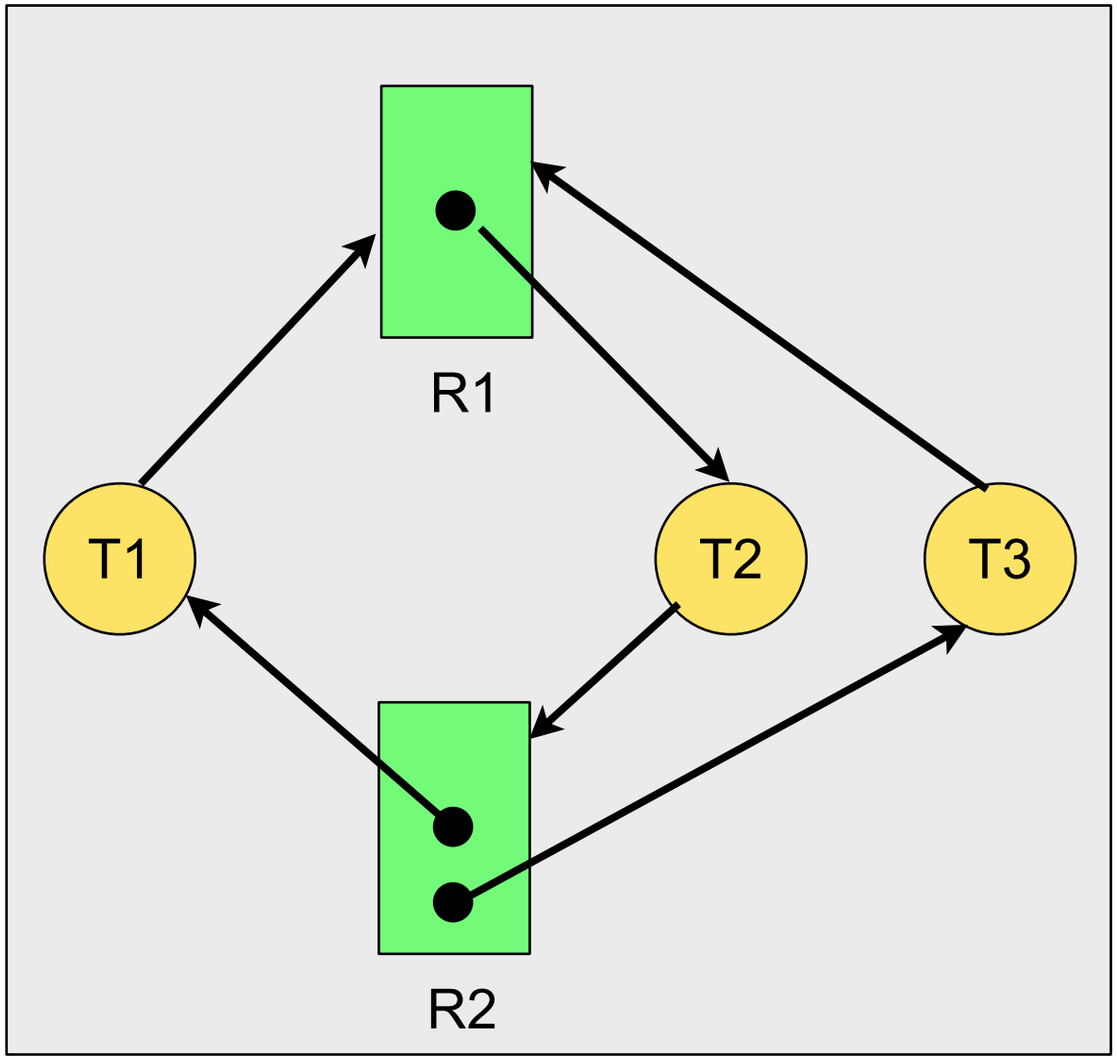
- A *Knot K* in a directed graph $G = (V, E)$ is a subgraph such that:
  - K is the *maximal* strongly-connected component (SCC)
    - SCC? Any two nodes $u, v \in K$ must be *reachable* from each other
    - Maximal? No other SCC is a proper subset of K
  - No nodes outside of K is reachable from K
    - Essentially, a knot is a cycle you can't escape.

- The claim for mRAGs:
  - Knot $\Longrightarrow$ Deadlock
    - All 4 necessary & sufficient conditions holds in the presence of a knot
  - No Knot *?=>* No Deadlock *(Not sure yet, let's check later)*

Back to mRAG Example 1

- Is there a knot?
  - What are the strongly-connected components (SCCs)?

- Step 1: List the SCCs
  - *SCC1 = {T1, R1, T2, R2}*
  - *SCC2 = {T2, R1, T3, R2}*
  - *SCC3 = {T1, T2, T3, R1, R2}*
  - ***SCC3 is the maximal SCC since SCC1 ⊂ SCC3, and SCC2 ⊂ SCC3.***

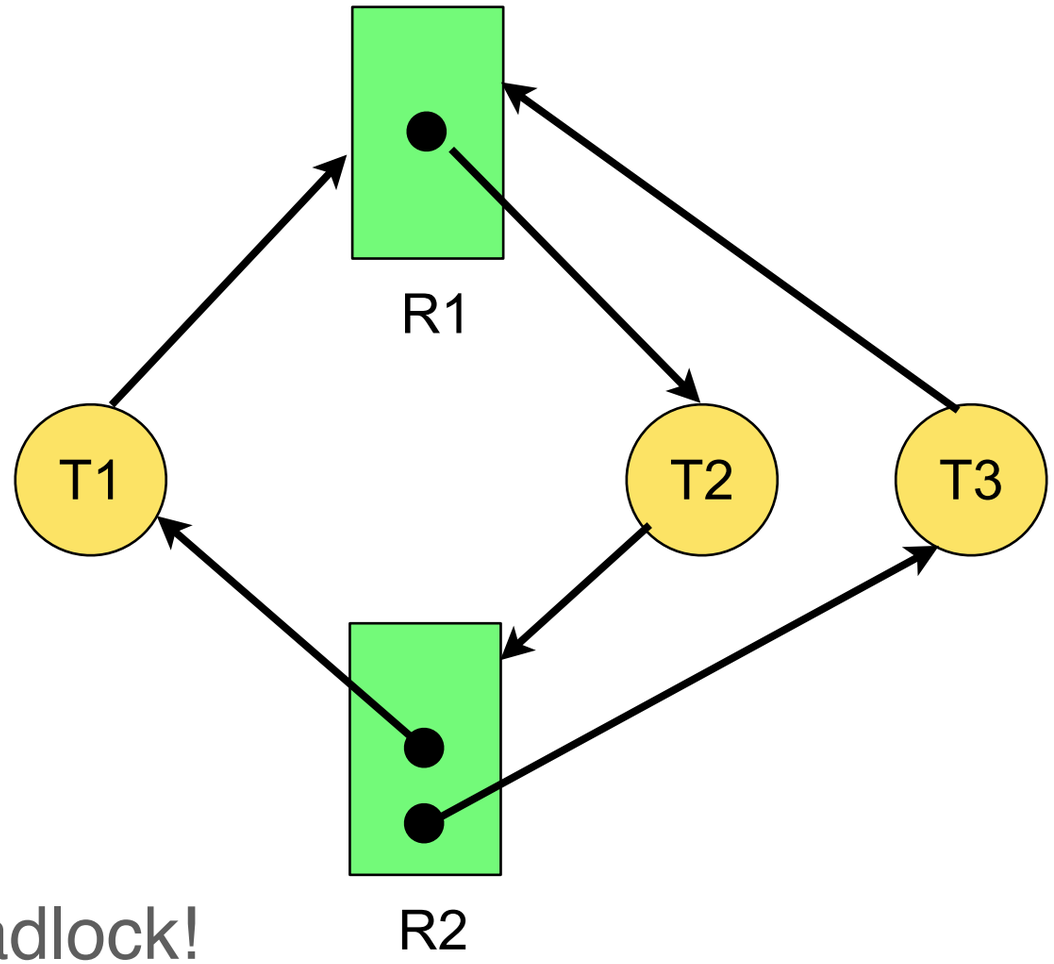


Back to mRAG Example 1

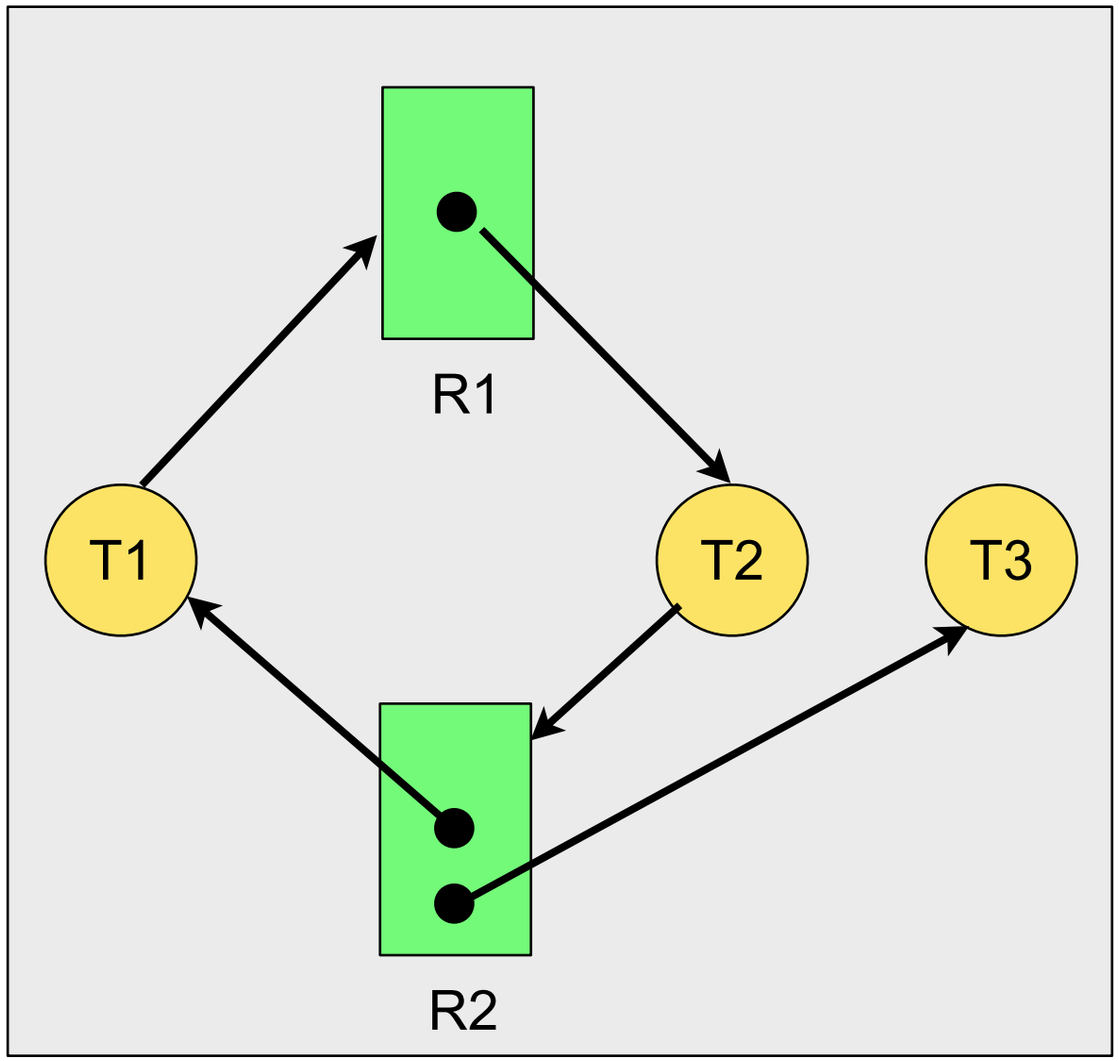
- Is there a knot?
  - What are the strongly-connected components (SCCs)?

- Step 1: List the SCCs:
  - *SCC1 = {T1, R1, T2, R2}*
  - *SCC2 = {T2, R1, T3, R2}*
  - *SCC3 = {T1, T2, T3, R1, R2}*

- Step 2: Is *SCC3* contained?
  - Yes!
  - So SCC3 is a **knot**, and there ***is*** a deadlock!

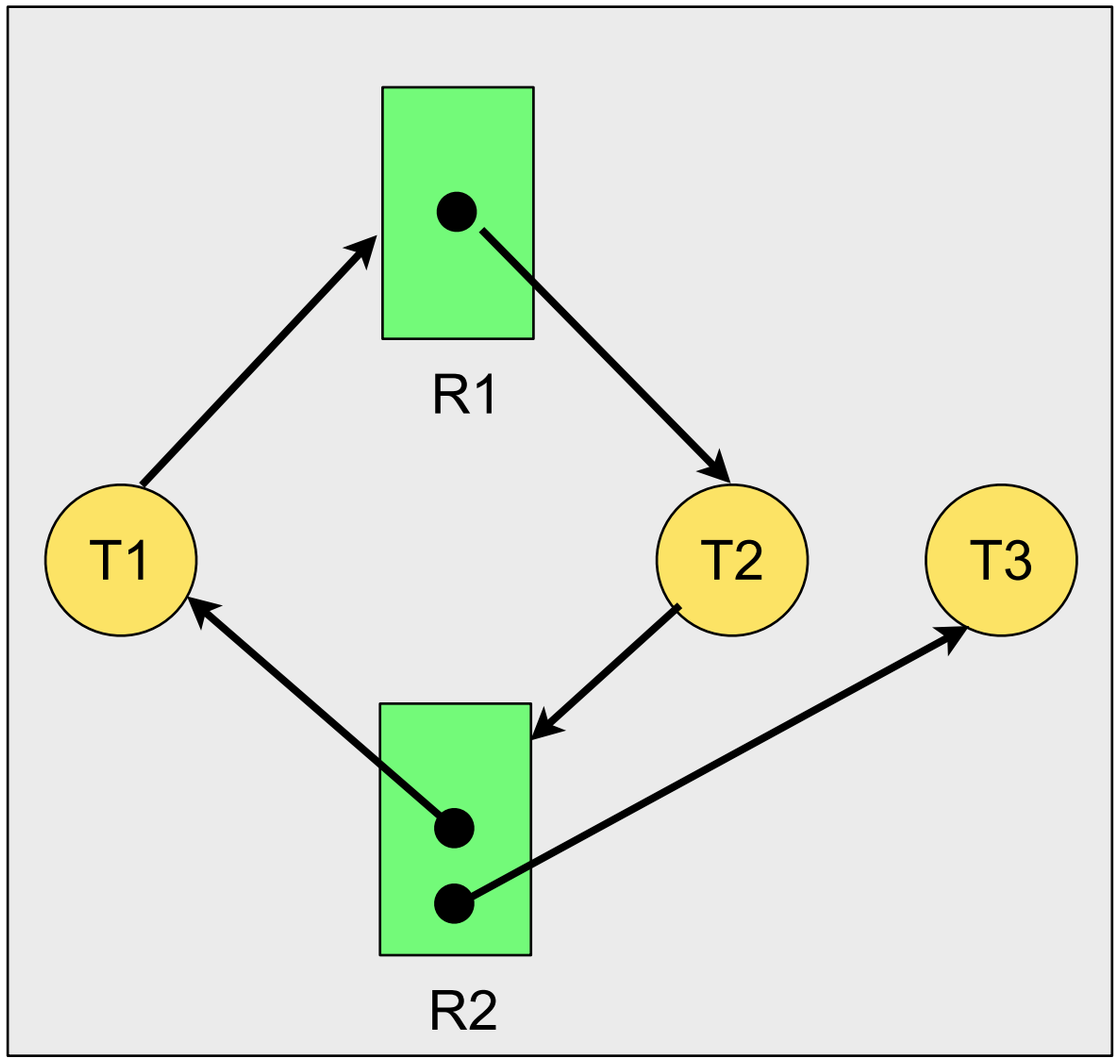


Back to mRAG Example 2



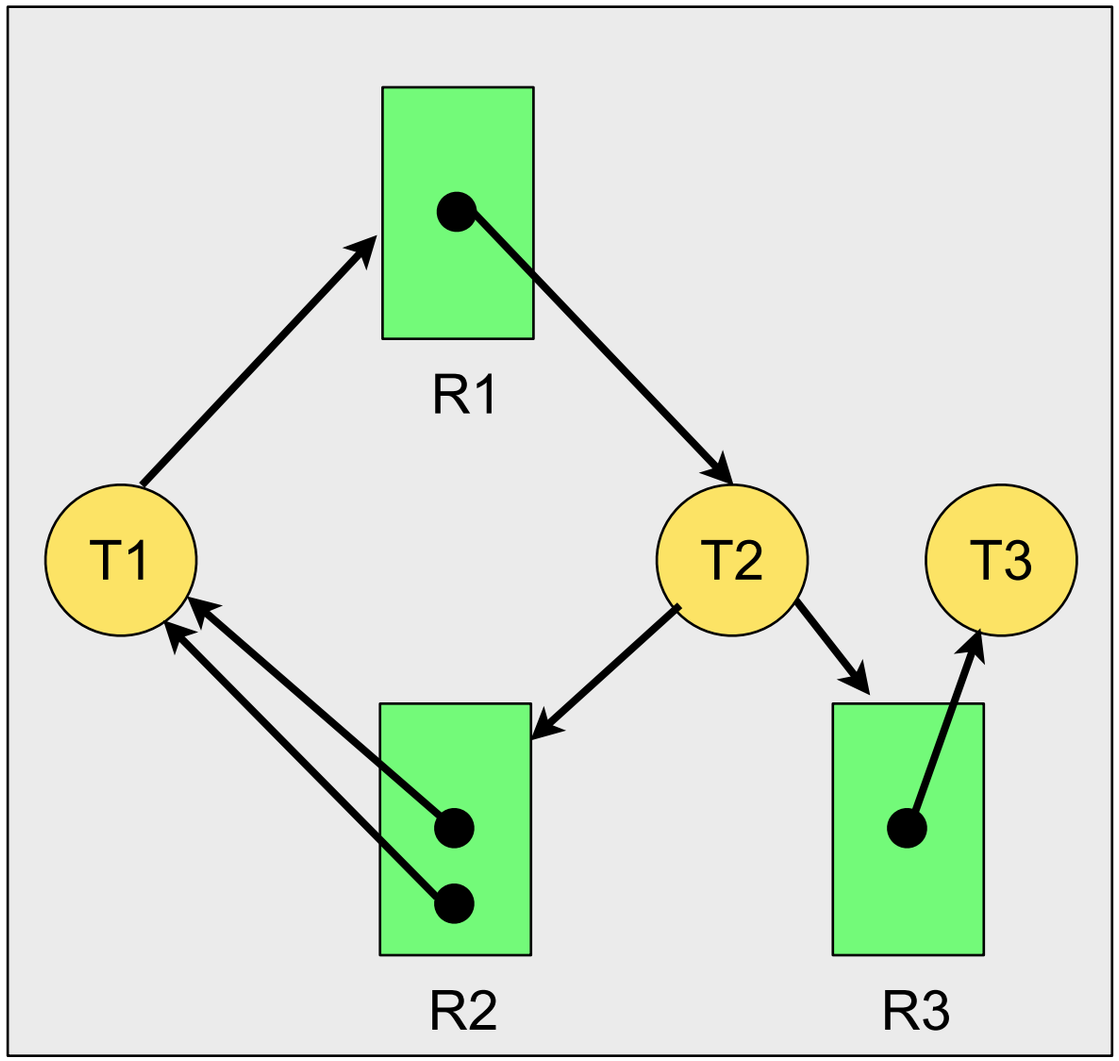
- How about this example?
  - There's no deadlock here.
- List the SCCs (just one)
  - SCC1 = {T1, R1, T2, R2}
- Is SCC1 a knot?

Back to mRAG Example 2



- How about this example?
  - There's no deadlock here.
- List the SCCs (just one)
  - SCC1 = {T1, R1, T2, R2}
- Is SCC1 a knot?
  - No! T3 is reachable and it's outside of SCC1. And no deadlock!

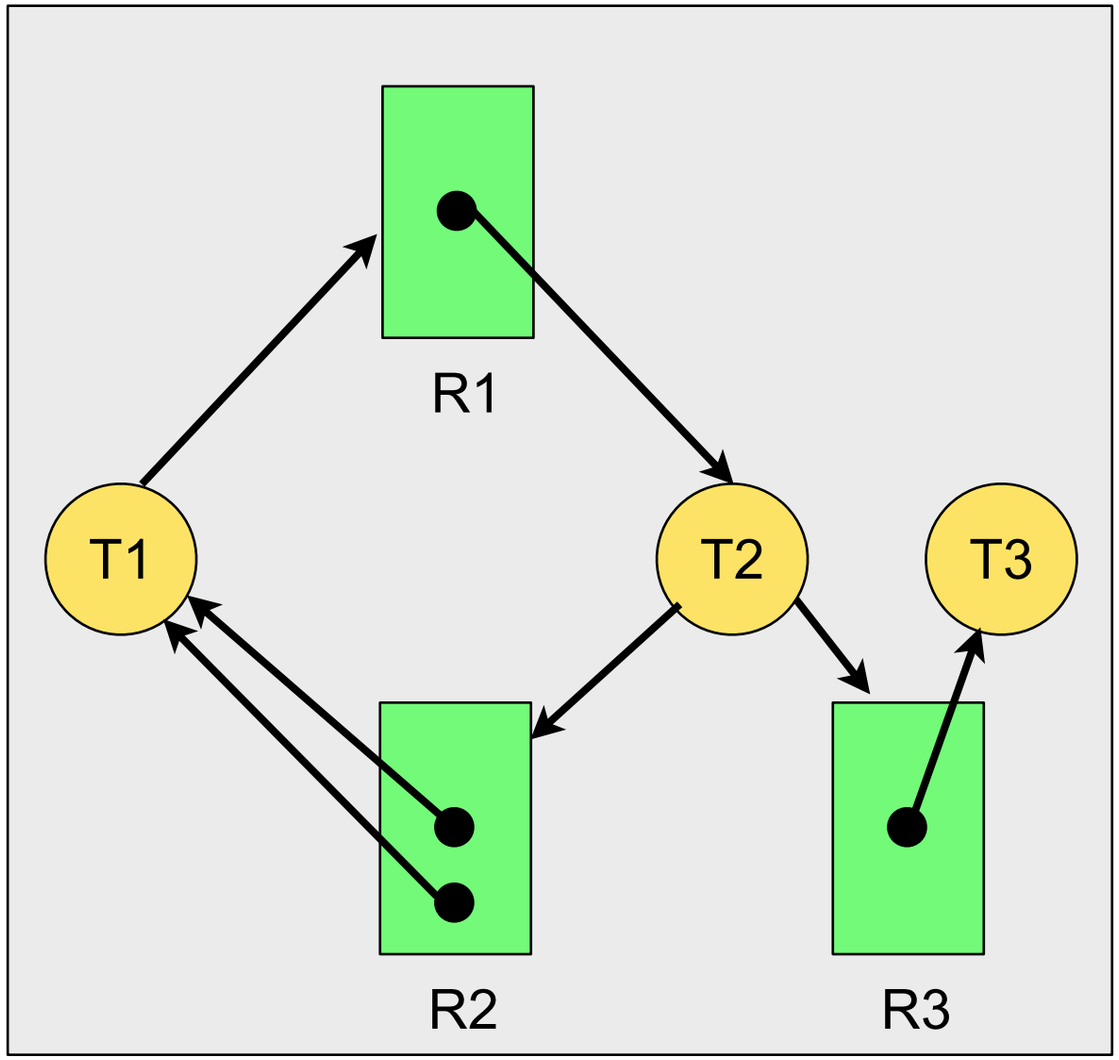
One More mRAG Example...



- T3 is a sink that every other node in the SCC can reach!
- So no knot here... but let's confirm by reducing the edges!
- (Drat... there's no knot, but there's still a deadlock)

There's no knot! There must not be deadlock?!

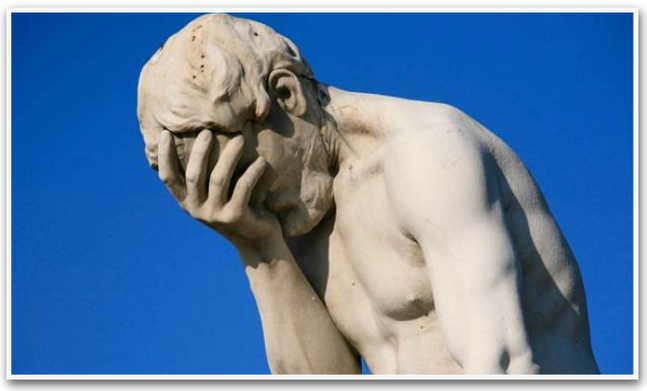
One More mRAG Example... (2)



There's no knot! We don't know anything.

- No knot, but a deadlock exists!

- **Conclusion for mRAGS:**
  - Cycle => No clue
  - No Cycle => No Deadlock ✓
  - Knot => Deadlock ✓
  - No Knot => No clue



Summary of RAGs

- **Single-Resource Instances** (RAGs):
  - Existence of cycle is necessary and sufficient for a deadlock
  - This can be used in practice
    - *Most resources cannot tolerate multiple simultaneous accesses*
  - Example: Linux (lockdep)

- **Multi-Resource Instance** (mRAGs):
  - Existence of a cycle is necessary but insufficient for a deadlock
  - Existence of a knot is sufficient for deadlock but unnecessary
  - We can't use knot detection reliably: False negatives are possible!

Deadlock Recovery

- Recovery
  - So we detect a deadlock in a single-instance RAG. Now what?

- Options:
  - *Do nothing* (This is what Linux does; detect and do nothing)
  - *Terminate* an deadlocked thread to force it to give up resources
  - *Roll back* an offending thread
    - Pick an thread that is involved in the deadlock
    - Rollback execution to a previous state *(how?)*, then replay
      - Hope for a deadlock-free schedule to play out

Recovery after Detection (Cont.)

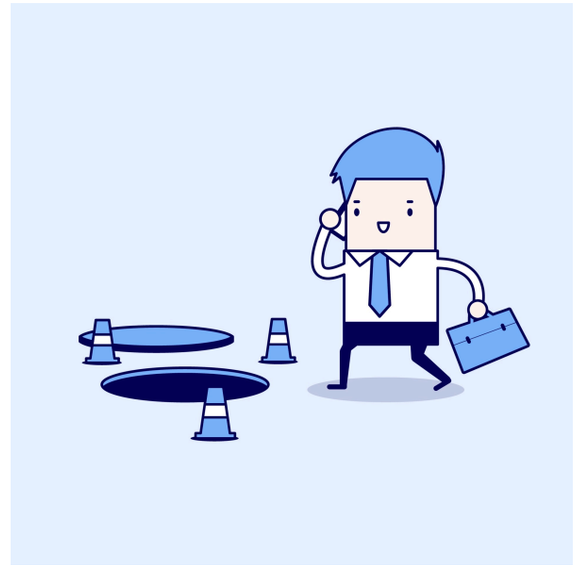
- How to implement rollbacks?
- Need to take a *checkpoint* (snapshot of complete system state) periodically.
  - Roll back to last check point, replay all the instructions again.
  - Sounds expensive *(It is!)*
  - Can't things replay exactly the same way? (Yep)
  - What happens to OS performance?

Topics for Today...

- Deadlock Conditions
- Ways to deal with deadlocks:
  - Deadlock Detection
    - Single Resource Allocation Graph
    - Multi-Resource Allocation Graph
  - Deadlock Avoidance
    - Dijkstra's Bankers Algorithm

Deadlock Avoidance

- Multi-instance RAGs are a problem.
  - We can show when a deadlock doesn't exist, but...
  - A deadlock can't be reliably detected
- Another option: Deadlock Avoidance
  - What if we knew all the threads' resource needs in advance?
    - "For me to run, I'll need 3 P's and 2 M's"
    - (Isn't always possible to know in advance)
  - OS keeps track of current system state
  - OS allocates resources in a way that takes OS from one "safe" state to another



Bankers Algorithm (Dijkstra)

- Risk-averse resource request algorithm
  - Simulate the execution after each resource request
    - Grant the request if the simulation leads to a deadlock-free (safe) state
  - *Works for multi-instance resources!*

What's a "*Safe" System State*?

There exists an execution sequence $T_1, T_2, \ldots, T_n$ such that

T_1 can obtain *all* its resources and finish,

T_2 can obain *all* its resources and finish, ...

System Model in Banker's Algorithm

n: The number of processes/threads

m: The number of resource types

$Available = (V_1, \ldots, V_m)$ V_j is the number of still **available** resources of type j

$$Alloc = \begin{pmatrix} A_{1,1} & \ldots & A_{1,m} \\ \vdots & \ddots & \vdots \\ A_{n,1} & \ldots & A_{n,m} \end{pmatrix}$$

$A_{i,j}$ is the current **allocation** of resource j to thread i

$$Max = \begin{pmatrix} M_{1,1} & \ldots & M_{1,m} \\ \vdots & \ddots & \vdots \\ M_{n,1} & \ldots & M_{n,m} \end{pmatrix}$$

$M_{i,j}$ is the **maximum demand** of thread i for resource j over the thread's lifetime

Example System State

- Threads: `T1,..,T5` and resource types: `Printer, Disk, Semaphore`

$$Available = \begin{array}{c} \begin{pmatrix} \texttt{P} & \texttt{D} & \texttt{S} \\ 3 & 3 & 2 \end{pmatrix} \end{array}$$

$$Max = \begin{array}{c} \begin{matrix} \texttt{P} & \texttt{D} & \texttt{S} \end{matrix} \\ \begin{pmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{pmatrix} \begin{matrix} \texttt{T1} \\ \texttt{T2} \\ \texttt{T3} \\ \texttt{T4} \\ \texttt{T5} \end{matrix} \end{array}$$

$$Alloc = \begin{array}{c} \begin{matrix} \texttt{P} & \texttt{D} & \texttt{S} \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{matrix} \texttt{T1} \\ \texttt{T2} \\ \texttt{T3} \\ \texttt{T4} \\ \texttt{T5} \end{matrix} \end{array}$$

- What's the total number of printers we have? **10**
- How many semaphores does T4 still need? **1**
- Say T3 finishes executing.
  - What must change about the state? ***Available* adds the row at *Alloc[3]***
- *Is the OS currently in a safe state? (Yes! But how do we know?)*

Is the Current System State in a Safe State?

```
m = number of resource types (3)
n = number of threads (5)

boolean isSafe(Available, Alloc, Need) {
    Work[] = Available.clone();                    Step 1: Initialization
    Finish[] = (0, 0, ..., 0)

    while (exists unfinished thread i && Need[i] ≤ Work) {
        // pretend that thread i finishes execution
        // then OS can reclaim thread i's allocated resources
        Work += Alloc[i]
        Finish[i] = 1
    }
    // there's an execution order in which all threads
    // can finish. System is safe!
    if (Finish == (1, 1, ..., 1))
        return true
    // there could be a deadlock!
    return false
}
```

$$Available = \begin{matrix} P & D & S \\ \end{matrix} \begin{pmatrix} 3 & 3 & 2 \end{pmatrix}$$

$$Alloc = \begin{matrix} P & D & S \\ \end{matrix} \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

$$Need = \begin{matrix} P & D & S \\ \end{matrix} \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

Is the Current System State in a Safe State?

```
m = number of resource types (3)
n = number of threads (5)

boolean isSafe(Available, Alloc, Need) {

    Work[] = Available.clone();
    Finish[] = (0, 0, ..., 0)

    while (exists unfinished thread i && Need[i] ≤ Work) {
        // pretend that thread i finishes execution
        // then OS can reclaim thread i's allocated resources
        Work += Alloc[i]
        Finish[i] = 1
    }                                          Step 2: Simulation
    // there's an execution order in which all threads
    // can finish. System is safe!
    if (Finish == (1, 1, ..., 1))
        return true
    // there could be a deadlock!
    return false
}
```

Simplified Notation

Let X and Y be vectors of length n.

We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all i

$$Available = \begin{array}{c} \begin{matrix} P & D & S \end{matrix} \\ \begin{pmatrix} 3 & 3 & 2 \end{pmatrix} \end{array}$$

$$Alloc = \begin{array}{c} \begin{matrix} P & D & S \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \end{array} \begin{matrix} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

$$Need = \begin{array}{c} \begin{matrix} P & D & S \end{matrix} \\ \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \end{array} \begin{matrix} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

Yes, we're in a safe state!!!

```
m = number of resource types (3)
n = number of threads (5)

boolean isSafe(Available, Alloc, Need) {

   Work[] = Available.clone();
   Finish[] = (0, 0, ..., 0)

   while (exists unfinished thread i && Need[i] ≤ Work) {
      // pretend that thread i finishes execution
      // then OS can reclaim thread i's allocated resources
      Work += Alloc[i]
      Finish[i] = 1
   }
   // there's an execution order in which all threads
   // can finish. System is safe!
   if (Finish == (1, 1, ..., 1))
      return true
   // there could be a deadlock!
   return false                                   Step 3: Determine safety
}
```

$$Available = \begin{matrix} P & D & S \\ \end{matrix} \begin{pmatrix} 3 & 3 & 2 \end{pmatrix}$$

$$Alloc = \begin{matrix} P & D & S \\ \end{matrix} \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

$$Need = \begin{matrix} P & D & S \\ \end{matrix} \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

An Unsafe Example: *Available = (0 1 1)*

```
m = number of resource types (3)
n = number of threads (5)

boolean isSafe(Available, Alloc, Need) {

   Work[] = Available.clone();
   Finish[] = (0, 0, ..., 0)

   while (exists unfinished thread i && Need[i] ≤ Work) {
      // pretend that thread i finishes execution
      // then OS can reclaim thread i's allocated resources
      Work += Alloc[i]
      Finish[i] = 1
   }
   // there's an execution order in which all threads
   // can finish. System is safe!
   if (Finish == (1, 1, ..., 1))
      return true
   // there could be a deadlock!
   return false
}
```

$$Available = \begin{matrix} \text{P} & \text{D} & \text{S} \\ (& 0 & 1 & 1 &) \end{matrix}$$

$$Alloc = \begin{matrix} \text{P} & \text{D} & \text{S} & \\ \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} & \begin{matrix} \text{T1} \\ \text{T2} \\ \text{T3} \\ \text{T4} \\ \text{T5} \end{matrix} \end{matrix}$$

$$Need = \begin{matrix} \text{P} & \text{D} & \text{S} & \\ \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} & \begin{matrix} \text{T1} \\ \text{T2} \\ \text{T3} \\ \text{T4} \\ \text{T5} \end{matrix} \end{matrix}$$

Banker's: What Good Is a Safety Checker?

- We want the OS to use the safety algo to determine whether a resource request should be granted (safe) or denied (unsafe)
  - *This* is Banker's Algorithm
- Back to previous example:
  - Let's say **T2** requests 1 printer and 2 semaphores
  - *Should the OS grant this request?*

$$Req[2] = \begin{matrix} P & D & S \\ (& 1 & 0 & 2 &) \end{matrix}$$

$$Available = \begin{matrix} P & D & S \\ (& 3 & 3 & 2 &) \end{matrix}$$

$$Max = \begin{matrix} P & D & S \\ \begin{pmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{pmatrix} \end{matrix} \begin{matrix} \text{T1} \\ \text{T2} \\ \text{T3} \\ \text{T4} \\ \text{T5} \end{matrix}$$

$$Alloc = \begin{matrix} P & D & S \\ \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \end{matrix} \begin{matrix} \text{T1} \\ \text{T2} \\ \text{T3} \\ \text{T4} \\ \text{T5} \end{matrix}$$

Banker's Algorithm

```
requestResources(Req[i], Available, Max, Alloc) {
    // Create and initialize the Need matrix
    Need = Max - Alloc

    if (Req[i] > Need[i])
        perror("Thread %d requested more than it needs\n", i)
    else if (Req[i] > Available)
        // can't fulfill request right now, try again later
        wait(thread[i])
    else {
        // update system state as if the request was granted
        Alloc[i]  += Req[i]
        Need[i]   -= Req[i]
        Available -= Req[i]

        if (isSafe(Available, Alloc, Need))
            grantRequest(thread[i])
        else {
            // restore system state
            Alloc[i]  -= Req[i]
            Need[i]   += Req[i]
            Available += Req[i]
            wait(thread[i])
        }
    }
}
```

$$Req[2] = \begin{matrix} P & D & S \\ (& 1 & 0 & 2 &) \end{matrix}$$

$$Available = \begin{matrix} P & D & S \\ (& 3 & 3 & 2 &) \end{matrix}$$

$$Alloc = \begin{matrix} P & D & S & \\ \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} & \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix} \end{matrix}$$

$$Max = \begin{matrix} P & D & S & \\ \begin{pmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{pmatrix} & \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix} \end{matrix}$$

Banker's Algorithm

```
requestResources(Req[i], Available, Max, Alloc) {
    // Create and initialize the Need matrix
    Need = Max - Alloc                                  Step 1: Initialization

    if (Req[i] > Need[i])
        perror("Thread %d requested more than it needs\n", i)
    else if (Req[i] > Available)
        // can't fulfill request right now, try again later
        wait(thread[i])
    else {
        // update system state as if the request was granted
        Alloc[i]  += Req[i]
        Need[i]   -= Req[i]
        Available -= Req[i]

        if (isSafe(Available, Alloc, Need))
            grantRequest(thread[i])
        else {
            // restore system state
            Alloc[i]  -= Req[i]
            Need[i]   += Req[i]
            Available += Req[i]
            wait(thread[i])
        }
    }
}
```

$$Req[2] = \begin{matrix} P & D & S \\ (& 1 & 0 & 2 &) \end{matrix}$$

$$Available = \begin{matrix} P & D & S \\ (& 3 & 3 & 2 &) \end{matrix}$$

$$Alloc = \begin{matrix} P & D & S \\ \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} & \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix} \end{matrix}$$

$$Need = \begin{matrix} P & D & S \\ \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} & \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix} \end{matrix}$$

Banker's Algorithm

```
requestResources(Req[i], Available, Max, Alloc) {
    // Create and initialize the Need matrix
    Need = Max - Alloc
```

Step 2: Sanity Check

```
    if (Req[i] > Need[i])
        perror("Thread %d requested more than it needs\n", i)
    else if (Req[i] > Available)
        // can't fulfill request right now, try again later
        wait(thread[i])
    else {
        // update system state as if the request was granted
        Alloc[i]  += Req[i]
        Need[i]   -= Req[i]
        Available -= Req[i]

        if (isSafe(Available, Alloc, Need))
            grantRequest(thread[i])
        else {
            // restore system state
            Alloc[i]  -= Req[i]
            Need[i]   += Req[i]
            Available += Req[i]
            wait(thread[i])
        }
    }
}
```

$$Req[2] = \begin{matrix} P & D & S \\ (& 1 & 0 & 2 &) \end{matrix}$$

$$Available = \begin{matrix} P & D & S \\ (& 3 & 3 & 2 &) \end{matrix}$$

$$Alloc = \begin{matrix} P & D & S \\ \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \end{matrix} \begin{matrix} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

$$Need = \begin{matrix} P & D & S \\ \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \end{matrix} \begin{matrix} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

Banker's Algorithm

```
requestResources(Req[i], Available, Max, Alloc) {
    // Create and initialize the Need matrix
    Need = Max - Alloc

    if (Req[i] > Need[i])
        perror("Thread %d requested more than it needs\n", i)
    else if (Req[i] > Available)
        // can't fulfill request right now, try again later
        wait(thread[i])
    else {
        // update system state as if the request was granted
        Alloc[i]  += Req[i]
        Need[i]   -= Req[i]
        Available -= Req[i]

        if (isSafe(Available, Alloc, Need))
            grantRequest(thread[i])
        else {
            // restore system state
            Alloc[i]  -= Req[i]
            Need[i]   += Req[i]
            Available += Req[i]
            wait(thread[i])
        }
    }
}
```

Step 3: Assume request is granted...

$$Req[2] = \begin{matrix} P & D & S \\ (& 1 & 0 & 2 &) \end{matrix}$$

$$Available = \begin{matrix} P & D & S \\ (& \mathbf{2} & \mathbf{3} & \mathbf{0} &) \end{matrix}$$

$$Alloc = \begin{matrix} P & D & S & \\ \begin{pmatrix} 0 & 1 & 0 \\ \mathbf{3} & \mathbf{0} & \mathbf{2} \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} & \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix} \end{matrix}$$

$$Need = \begin{matrix} P & D & S & \\ \begin{pmatrix} 7 & 4 & 3 \\ \mathbf{0} & \mathbf{2} & \mathbf{0} \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} & \begin{matrix} T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix} \end{matrix}$$

Banker's Algorithm

```
requestResources(Req[i], Available, Max, Alloc) {
    // Create and initialize the Need matrix
    Need = Max - Alloc

    if (Req[i] > Need[i])
        perror("Thread %d requested more than it needs\n", i)
    else if (Req[i] > Available)
        // can't fulfill request right now, try again later
        wait(thread[i])
    else {
        // update system state as if request was granted..
        Alloc[i]  += Req[i]
        Need[i]   -= Req[i]
        Available -= Req[i]

        if (isSafe(Available, Alloc, Need))
            grantRequest(thread[i])
        else {
            // restore system state
            Alloc[i]  -= Req[i]
            Need[i]   += Req[i]
            Available += Req[i]
            wait(thread[i])
        }
    }
}
```

Step 4: Run the safety algorithm

(In this case, yes, Thread 2's request is granted!)

$$Req[2] = \begin{matrix} P & D & S \\ (1 & 0 & 2) \end{matrix}$$

$$Available = \begin{matrix} P & D & S \\ (\mathbf{2} & \mathbf{3} & \mathbf{0}) \end{matrix}$$

$$Alloc = \begin{matrix} P & D & S \\ \begin{pmatrix} 0 & 1 & 0 \\ \mathbf{3} & \mathbf{0} & \mathbf{2} \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \end{matrix} \begin{matrix} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

$$Need = \begin{matrix} P & D & S \\ \begin{pmatrix} 7 & 4 & 3 \\ \mathbf{0} & \mathbf{2} & \mathbf{0} \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix} \end{matrix} \begin{matrix} \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{matrix}$$

Evaluation of Banker's Algorithm

- Summary: fulfill resource requests such that deadlocks can't happen
- Pros
  - Can detect potential deadlocks for multi-instance resources
- Cons
  - Need a crystal ball: *Max* resource requirements must be known
  - Overhead costs *(run for every resource request)*
    - Resource-Request Algorithm
      - Due in large part to Safety Algorithm
    - Time and space complexities? $O(n^2)$ where
      n = max(number of threads, number of resource types)

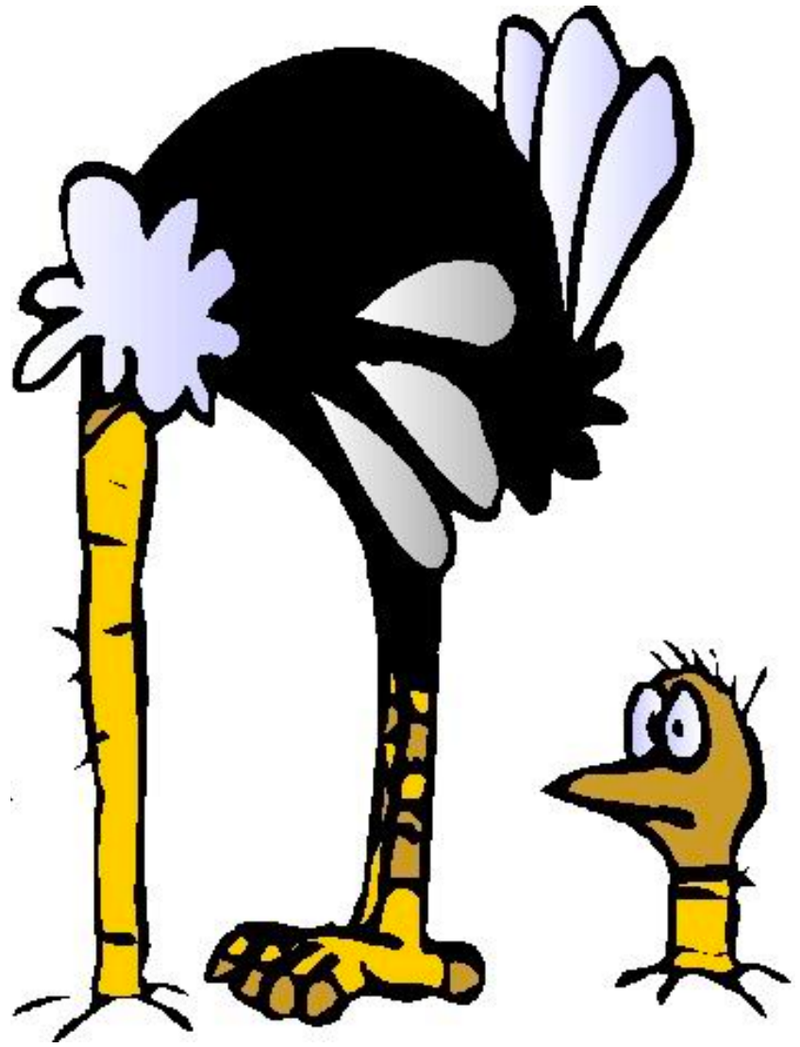
Topics for Today...

- Deadlock Conditions
- Ways to deal with deadlocks:
  - Deadlock Detection
    - Single Resource Allocation Graph
    - Multi-Resource Allocation Graph
  - Deadlock Avoidance
    - Banker's Algorithm
  - The Ostrich Approach

Ostrich Approach

- Key insight: Sometimes more cost-effective to let just deadlocks happen

- *Ostrich Approach*
  - OS sticks head in sand
  - Leave user to deal with deadlocks
    - Kill the offending thread(s) or process(es) or reboot



- In Linux, user can detect deadlocks.
- In Java, you can look at a threaddump: `ctrl+\`

Evaluating Deadlock Handling Approaches

- Deadlock Avoidance
  - Too much overhead per resource request
- Deadlock Detection
  - Run infrequently: could miss deadlock state
  - Run too often: high overhead
    - Could still miss deadlock state
  - Rollbacks are also costly

In Conclusion...

- Deadlock != Starvation
  - Starvation: waiting indefinitely
  - Deadlock: circular waiting for resources
- Know what *"necessary and sufficient"* means
- Know the four conditions for a deadlock
- Know the different ways to deal with deadlocks

In Conclusion...

- Many ways to deal with deadlocks, most are impractical
  - Understand implications of prevention schemes
  - Know Banker's Algorithm
  - Know RAGs
    - Single Resource vs Multi-Resource
    - Existence of Cycles and Knots, and what they tell us

- Next Topic: Memory Management