Deadlocks

umw cs405

much of the material from Prof. Dahlin, U Texas &
from textbook

Outline

- Deadlock
  - definition
  - conditions for its occurrence
  - solutions: breaking deadlocks, avoiding deadlocks
  - efficiency vs. complexity
- Other hard liveliness problems:
  - priority inversion, starvation, denial of service

Definitions

Resources

- threads - active
- resources - passive; things needed by the thread to do its job (CPU, disk space, memory).
- 2 kinds of resources:
  - preemptable: can take it away (CPU)
  - Non-preemptable: must leave w/ thread (disk space)

Resources cont'd

- lock/mutual exclusion - a kind of resource
  - a set of data that a thread needs exclusive access to to do a job.
- Is a lock preemptable or non-preemptable?

Starvation v. Deadlock

* Starvation: thread waits indefinitely (e.g., some other thread is using the resource)
* Deadlock: circular waiting for resources.
* Deadlock implies starvation but not vice versa.

Deadlock Example

THREAD A	THREAD B
`x.acquire()`	`y.acquire()`
`y.acquire()`	`x.acquire()`

Deadlock Example

THREAD A	THREAD B
`x.acquire()`	`y.acquire()`
`y.acquire()`	`x.acquire()`

DEADLOCK: A SET OF BLOCKED PROCESSES EACH HOLDING A RESOURCE AND WAITING TO ACQUIRE A RESOURCE HELD BY ANOTHER IN THE SET.

Deadlock in Kansas:

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

Law passed by Kansas Legislature.

Conditions for Deadlock

Motivation

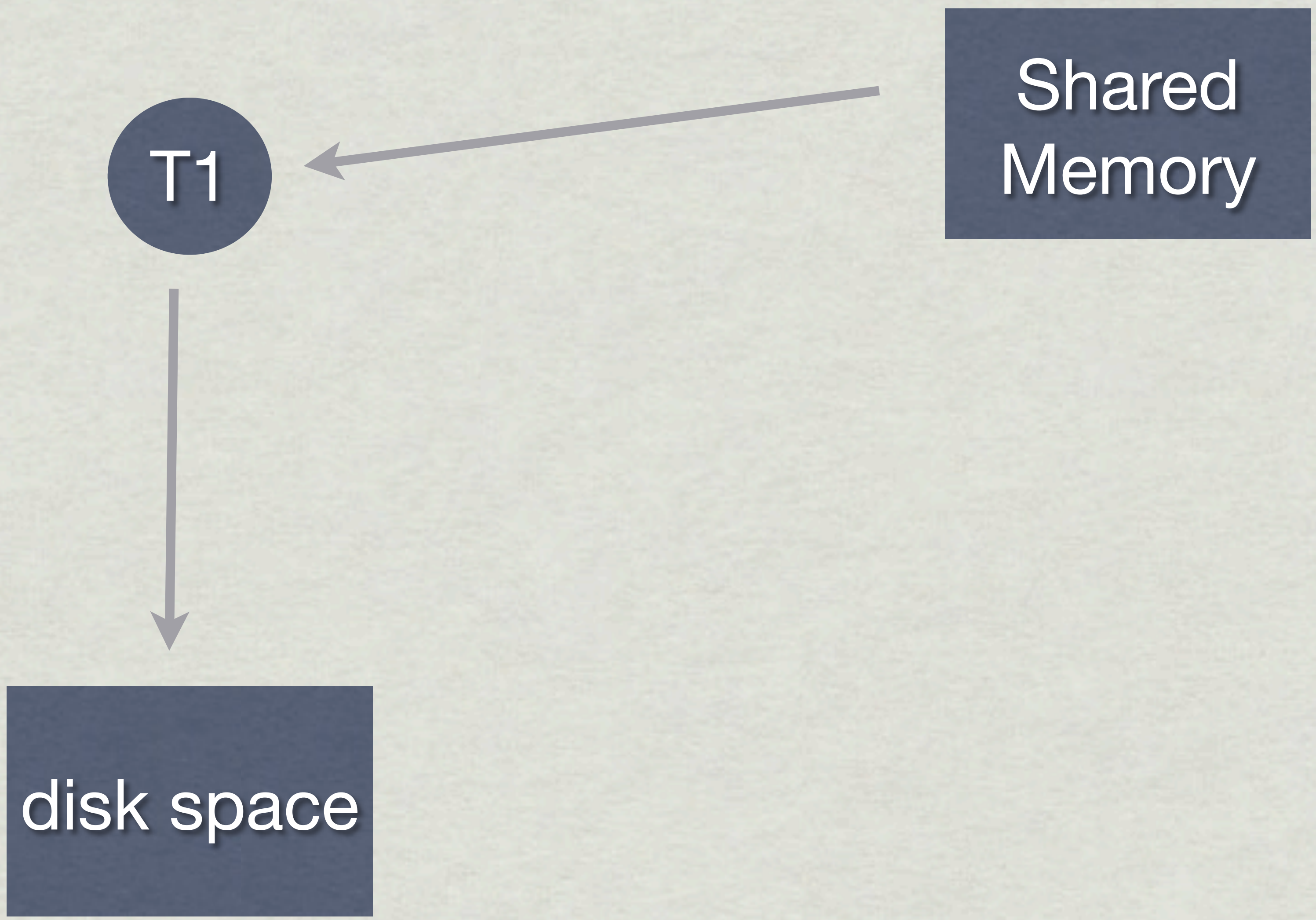
- Deadlock can happen with any type of resource
- Can occur with multiple resources (you can't decompose the problem to solve deadlock for each resource)

Example

Thread 1 holds a lock for shared memory

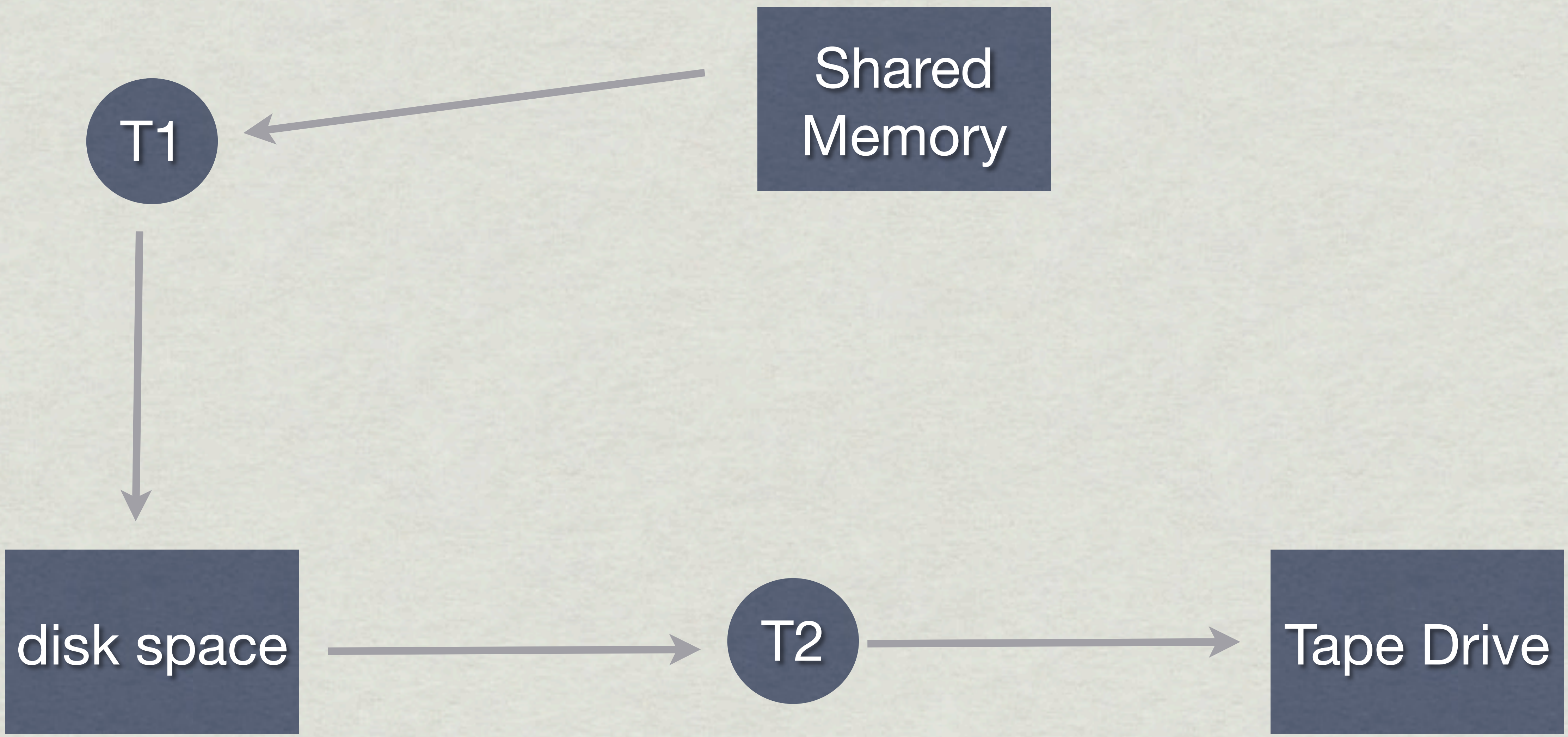
Example

Thread 1 wants disk space so it waits



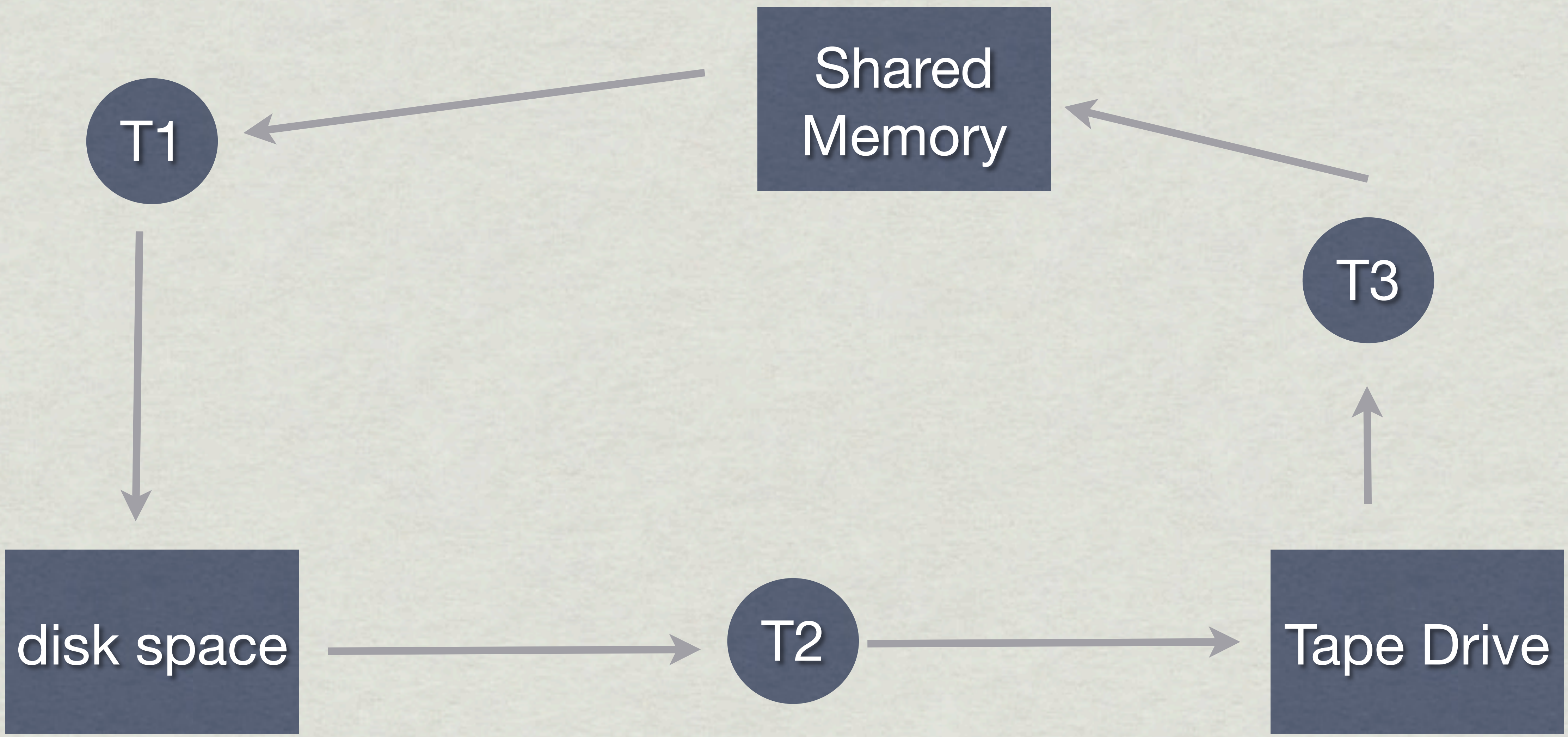
Example

Thread 2 has a lock on the disk space and is waiting on the tape drive



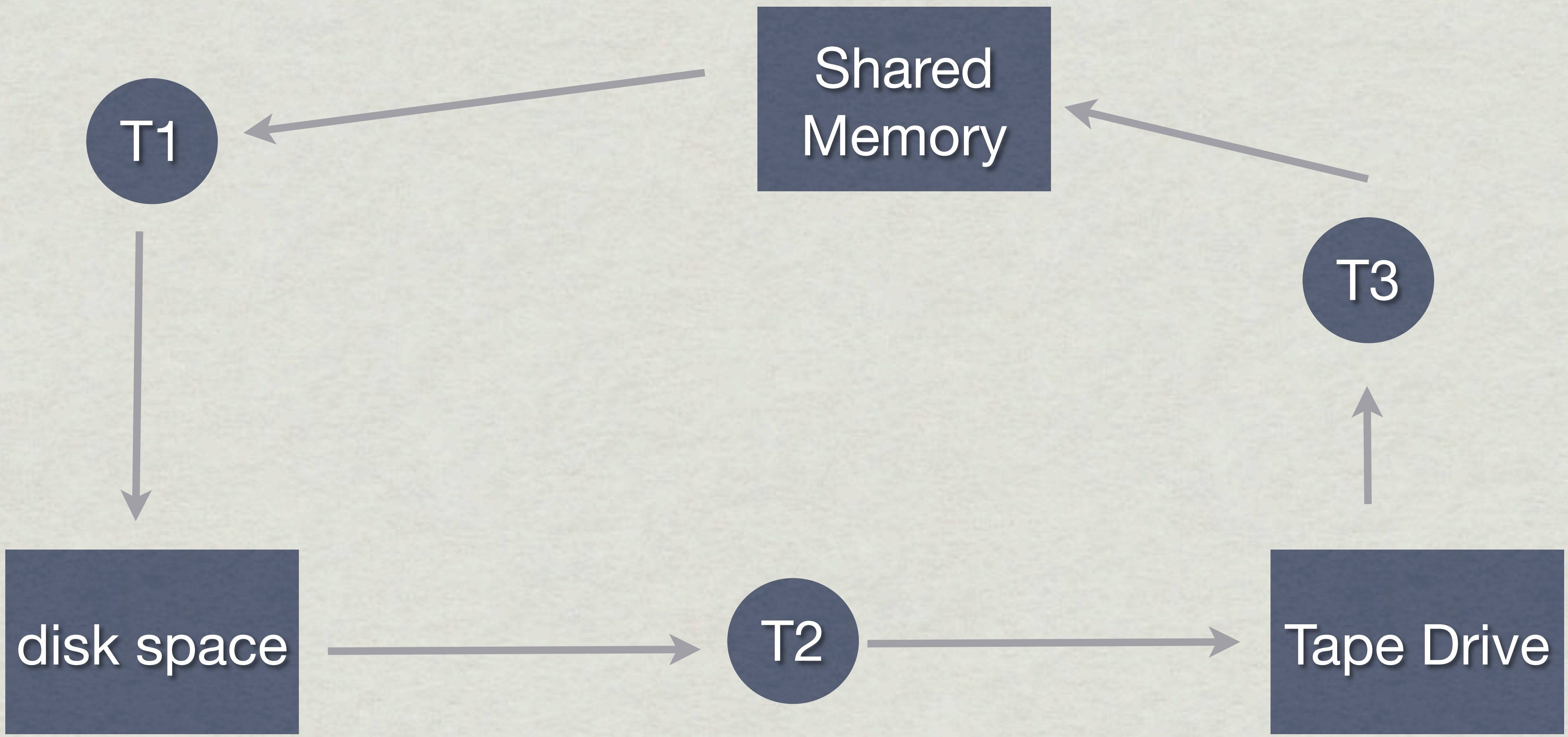
Example

Thread 3 holds a lock on the tape drive and is waiting on shared memory



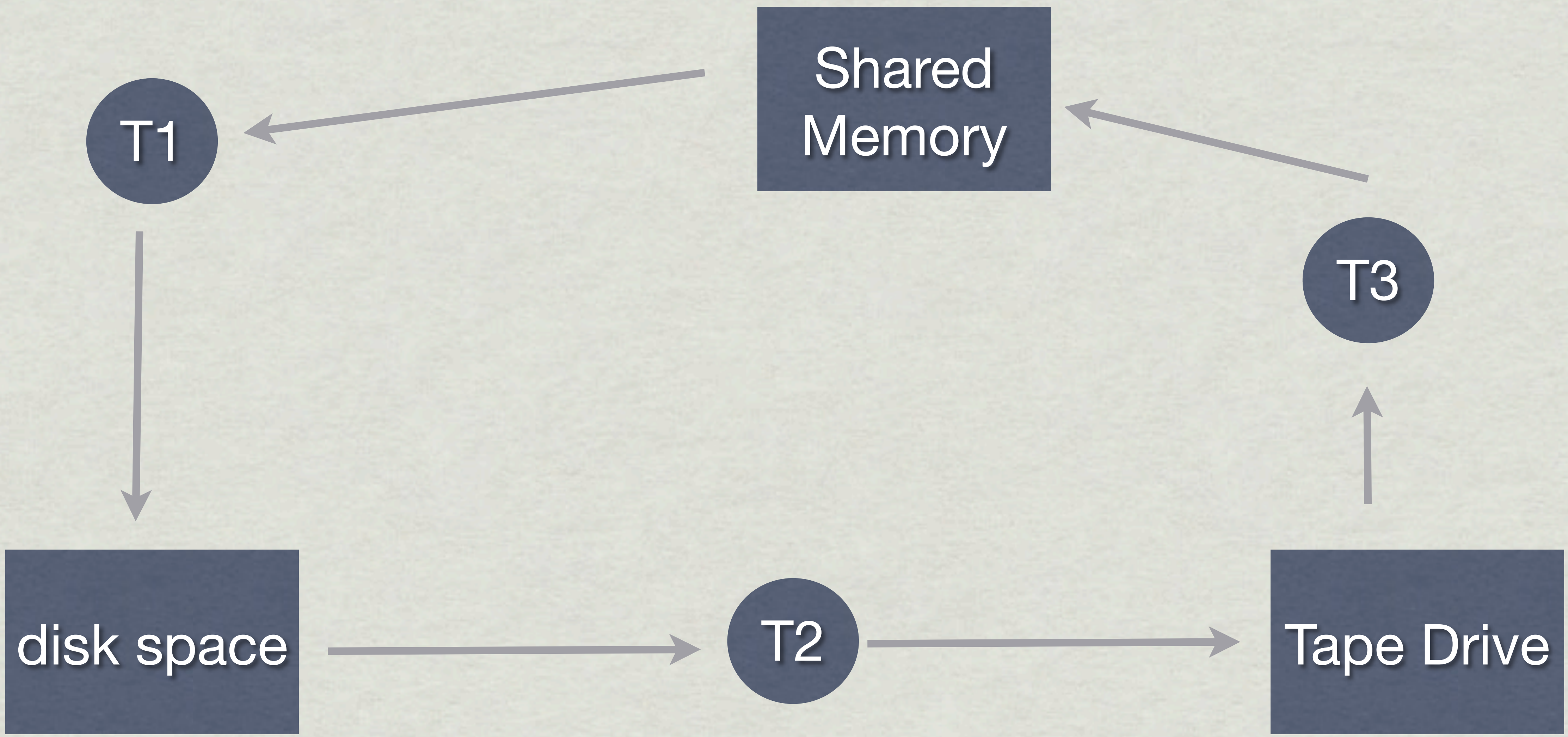
Example

Each is waiting for the other to release.



Deadlock

Each is waiting for the other to release.



deadlock

can occur whenever there is waiting

PROJECT

DINING PHILOSOPHERS

DEADLOCK

DATE 27-09-2012

CLIENT PHILOSOPHY DEPT.

Conditions for deadlock

without ALL these, can't have deadlock

1. limited access (mutex, bounded buffer, etc)
2. no preemption (if someone has a resource, we can't take it away.
3. multiple independent requests (wait while holding)
4. circular waiting

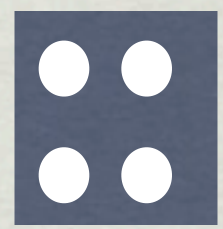
resource allocation graph

a way to describe deadlocks

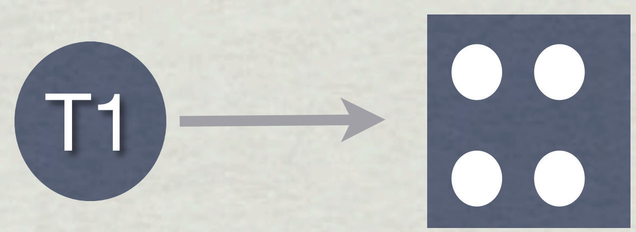
Symbol Resources

- thread

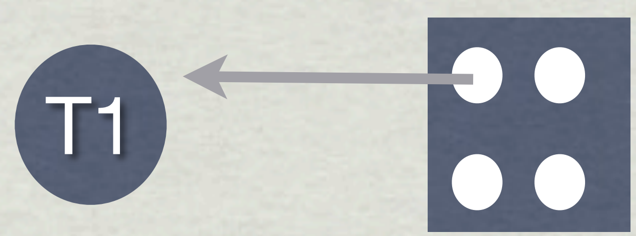
- resource w/ four instances



- thread requests instance

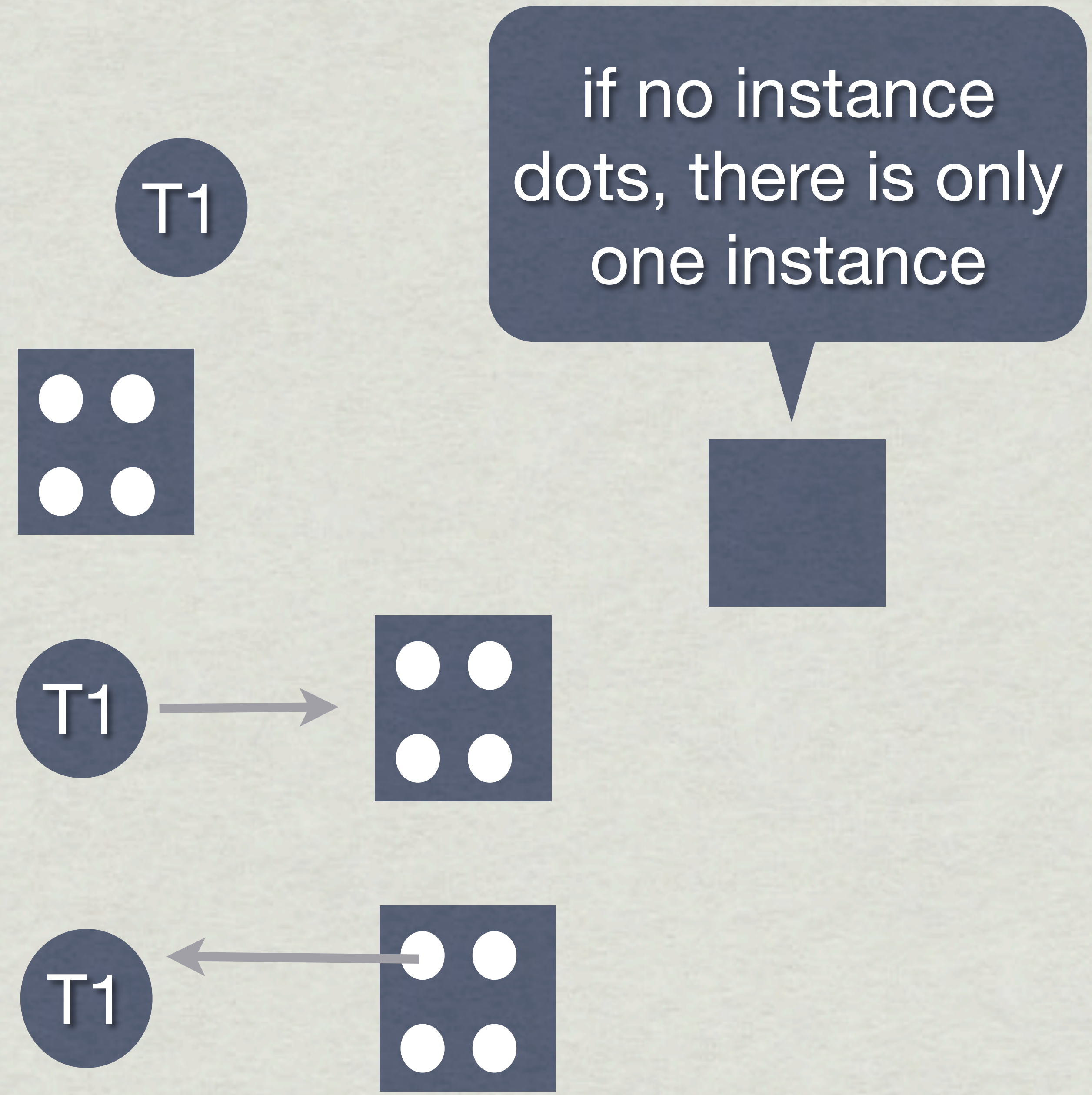


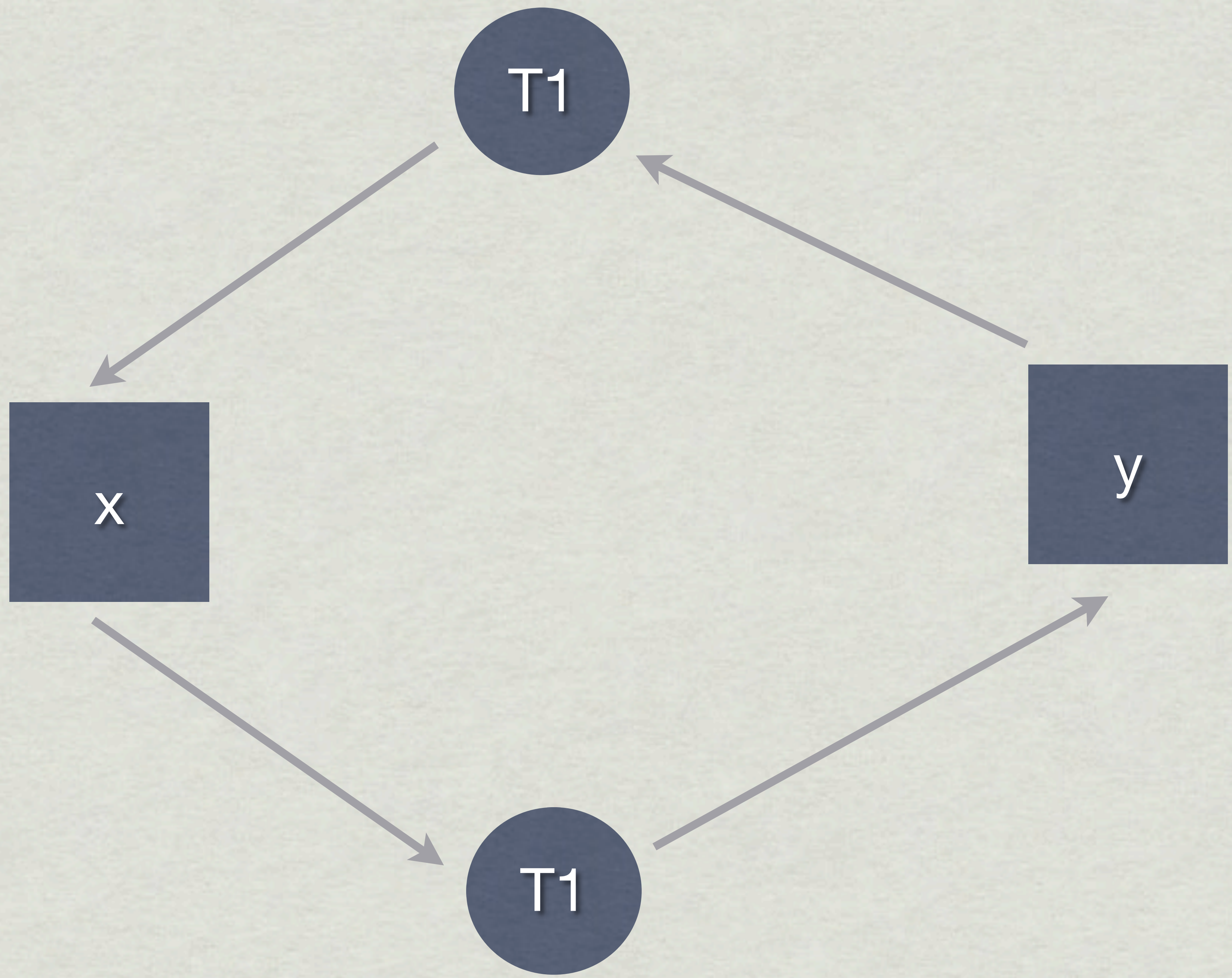
- thread is holding a resource instance



Symbol Resources

- thread
- resource w/ four instances
- thread requests instance
- thread is holding a resource instance



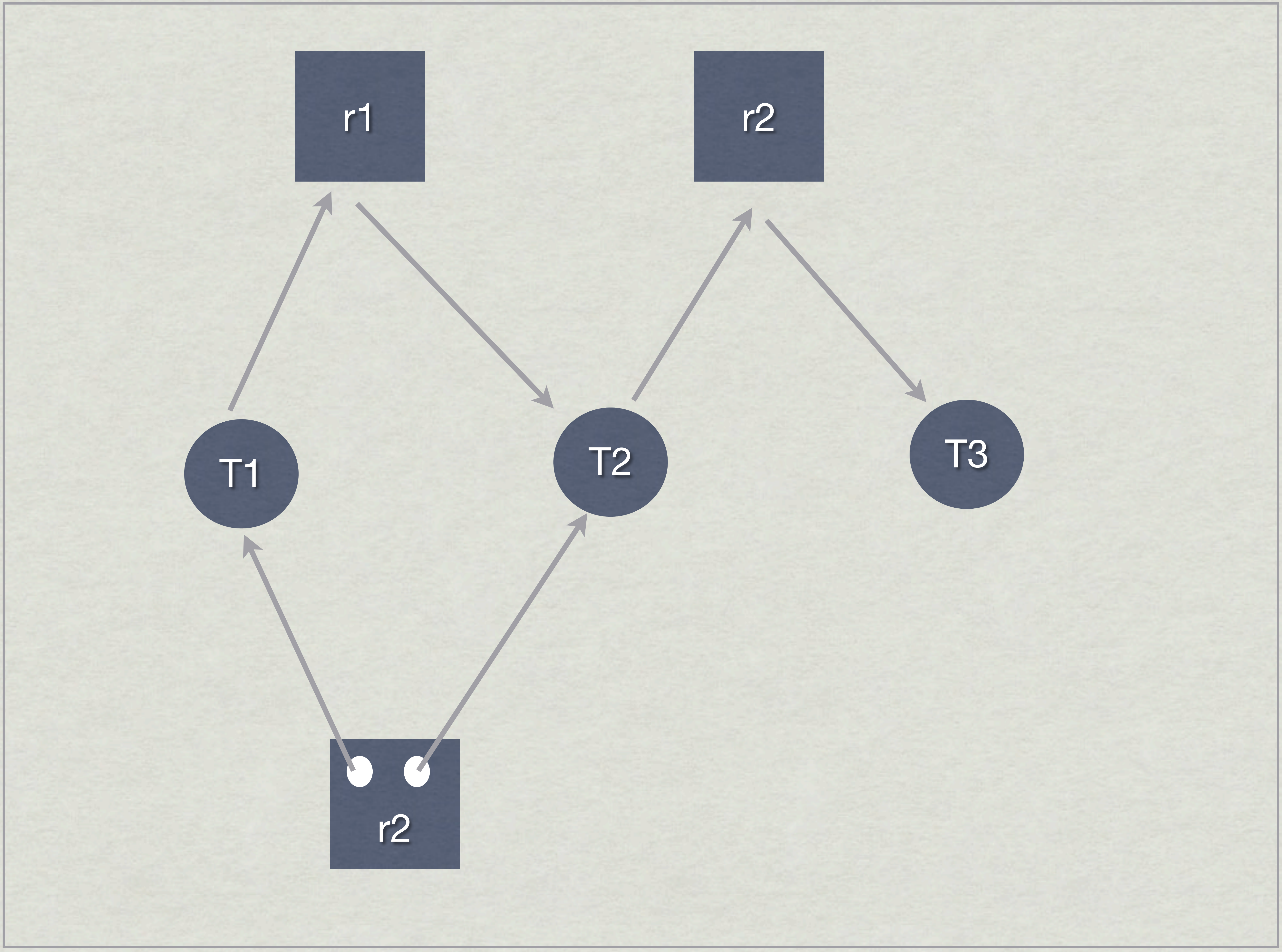

T1
x
y
T1

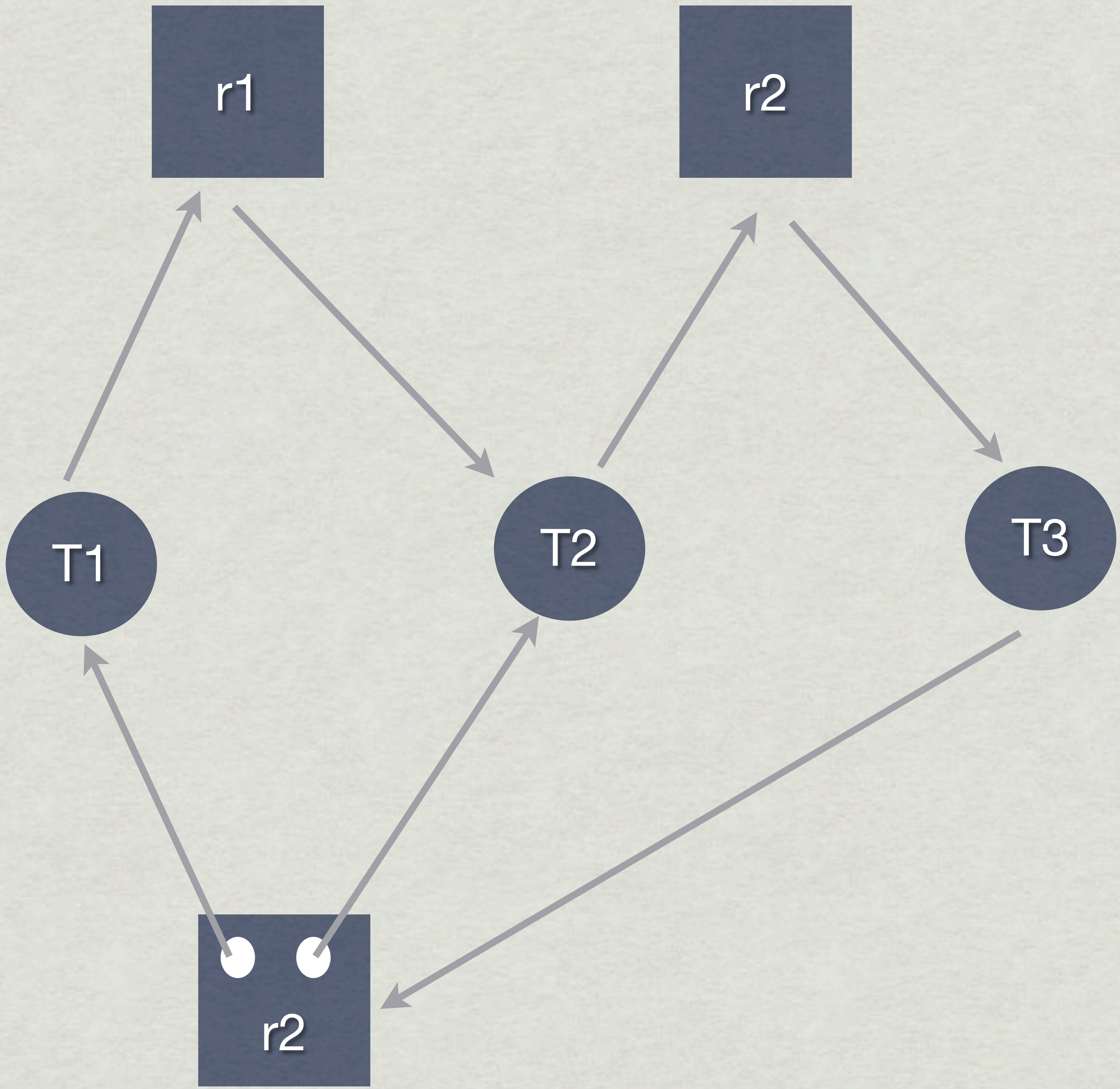
resource allocation graph

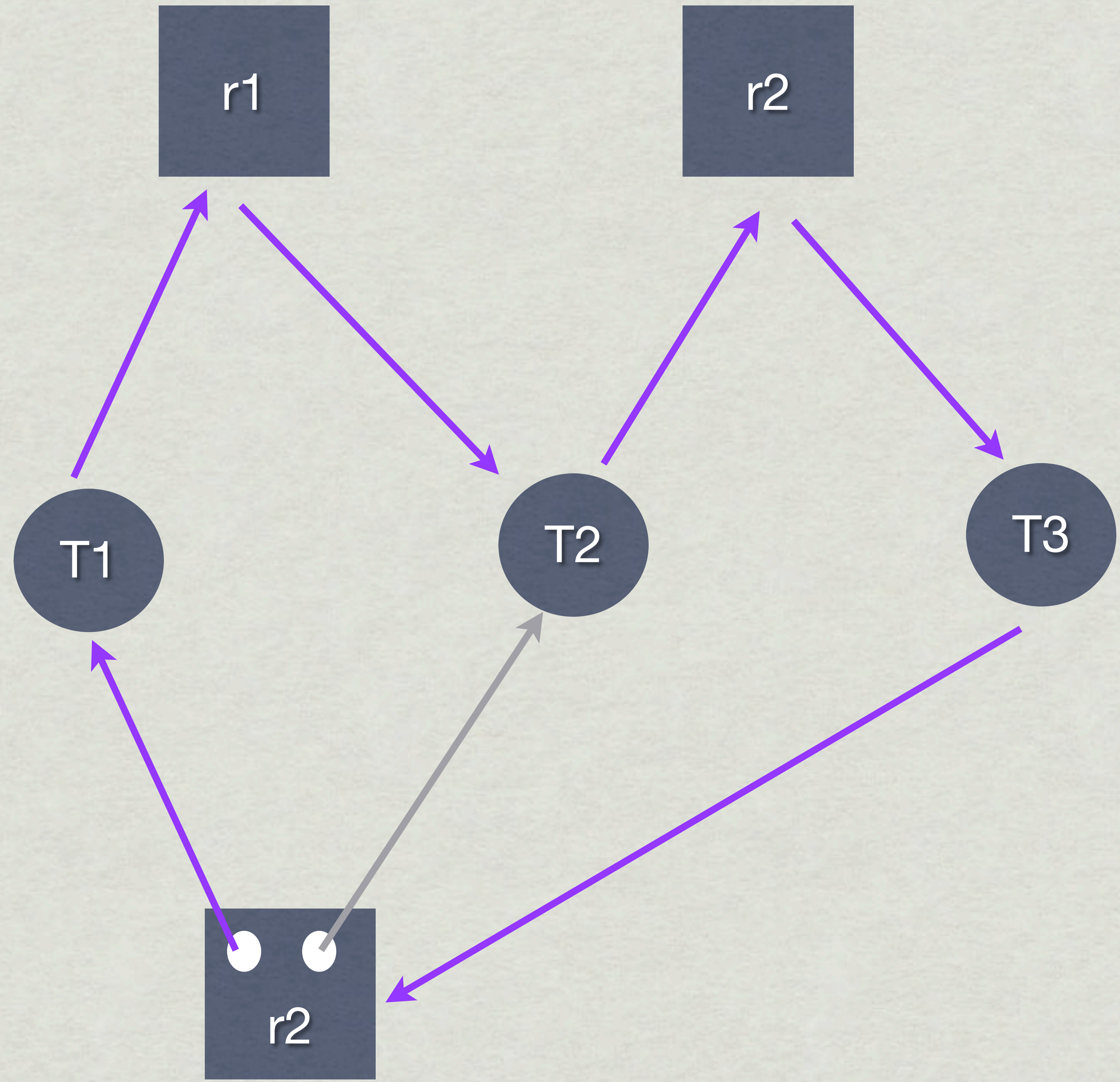
- no cycles ➜ no deadlock exists
- cycle ➜ deadlock may exist
    - if one instance of each resource both necessary and sufficient condition
    - if multiple instances, necessary but not sufficient.

Deadlock or not?

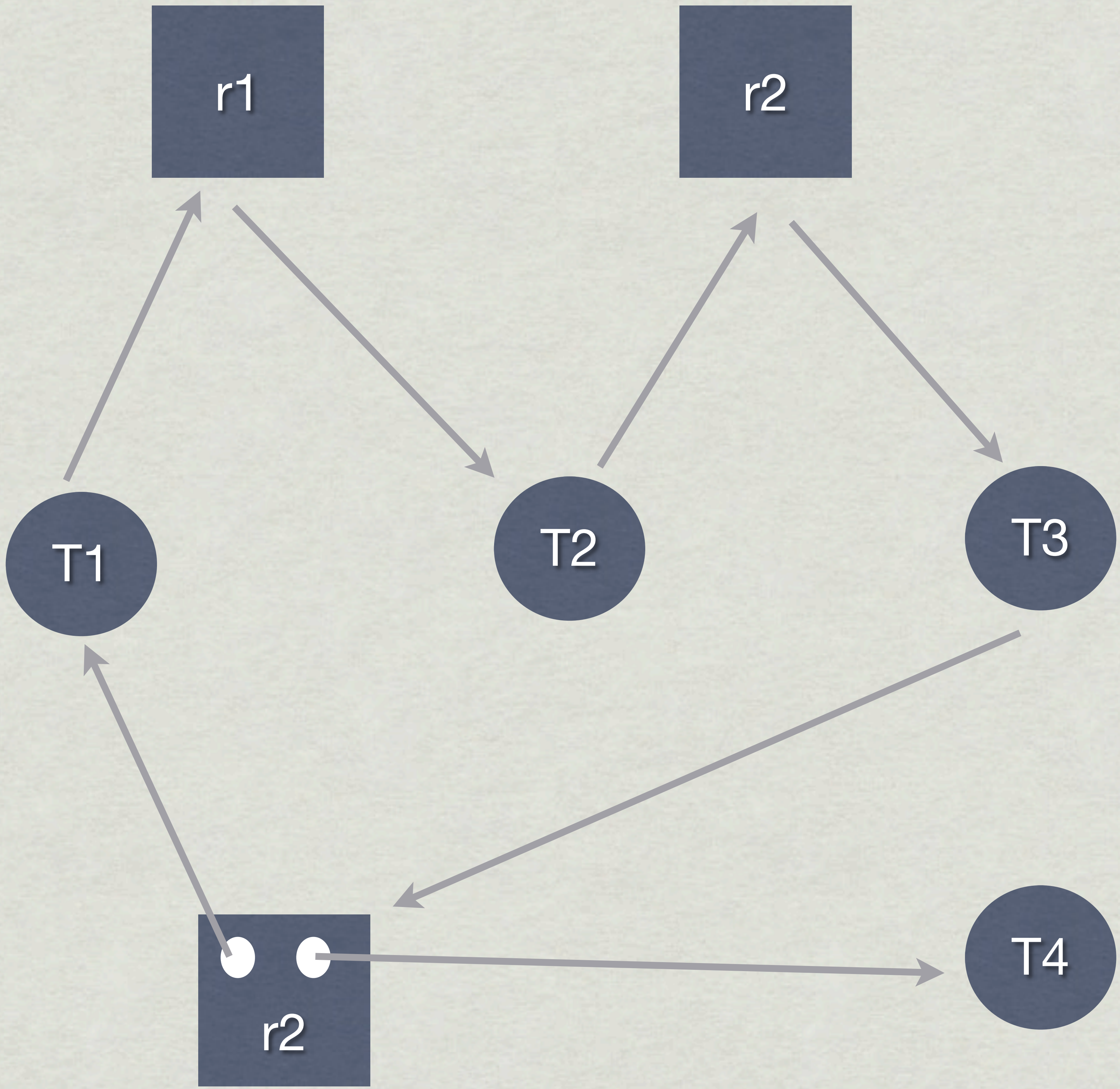
a quiz


r1
r2
T1
T2
T3
r2


r1
r2
T1
T2
T3
r2



r1
r2
T1
T2
T3
r2


r1
r2
T1
T2
T3
r2
T4

Ignore the problem and pretend deadlocks never occur

used by most operating systems including UNIX.

Detect and fix

```
scan graph
detect cycles
fix them    (the hard part)
```

How to fix?

- shoot thread. force it to give up resources. not always possible.
  - thread holding mutex - if we force it to give it up the world could end up inconsistent.
- roll back actions of deadlocked threads ("transactions"). common database technique.

Preventing deadlock

key idea: get rid of one of the four necessary conditions

What are those conditions?

Conditions for deadlock

without ALL these, can't have deadlock

1. limited access (mutex, bounded buffer, etc)
2. no preemption (if someone has a resource, we can't take it away.
3. multiple independent requests (wait while holding)
4. circular waiting

avoiding deadlock hard

Thread 1	Thread 2
Grab A	**Grab B**
Grab C	wait for C
Wait for B	...

Ideas?

1. limited access (mutex, bounded buffer, etc)
2. no preemption (if someone has a resource, we can't take it away.
3. multiple independent requests (wait while holding)
4. circular waiting

Develop an order

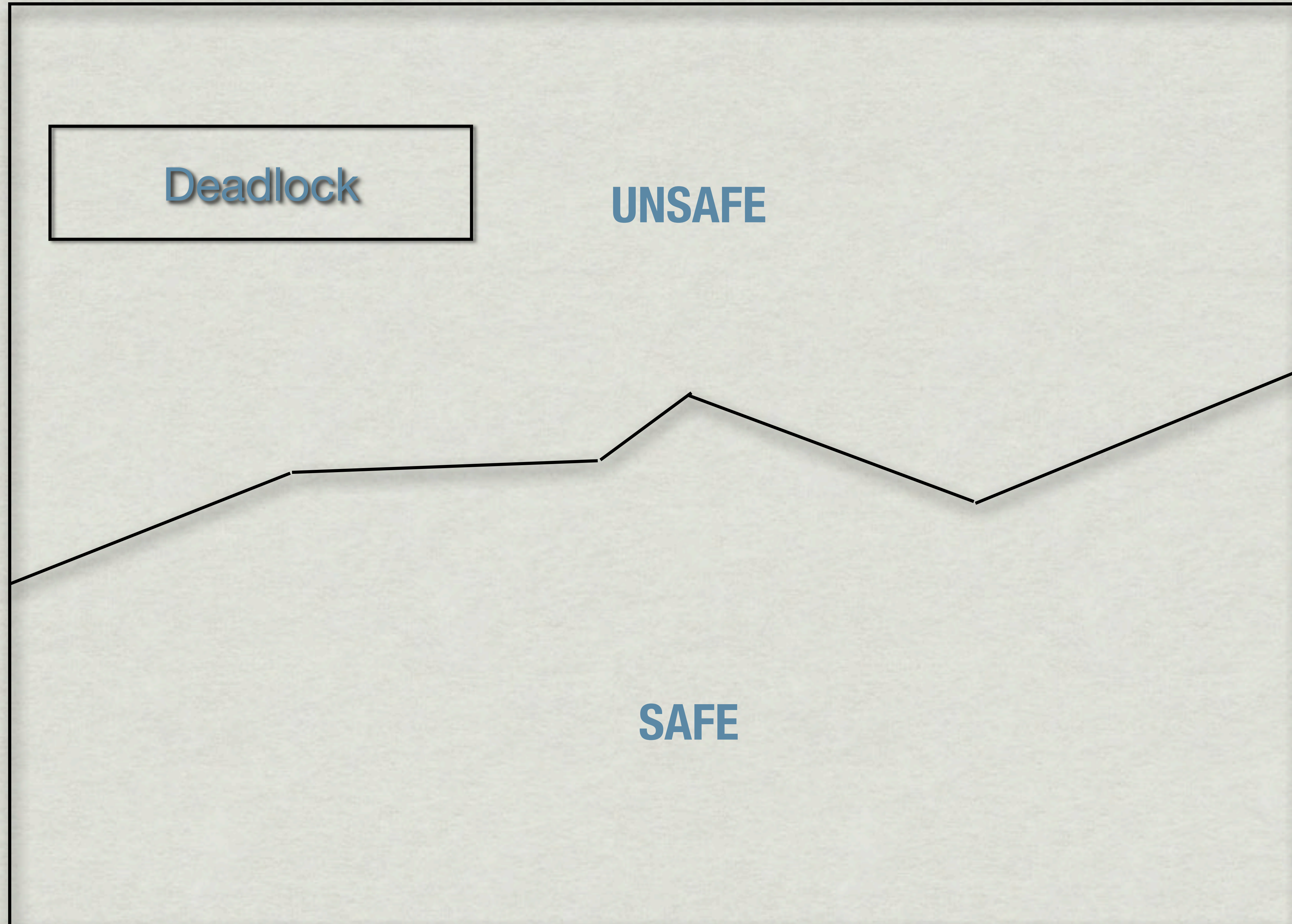
* each resource given a number
* threads need to request resources in the correct order
* problem?

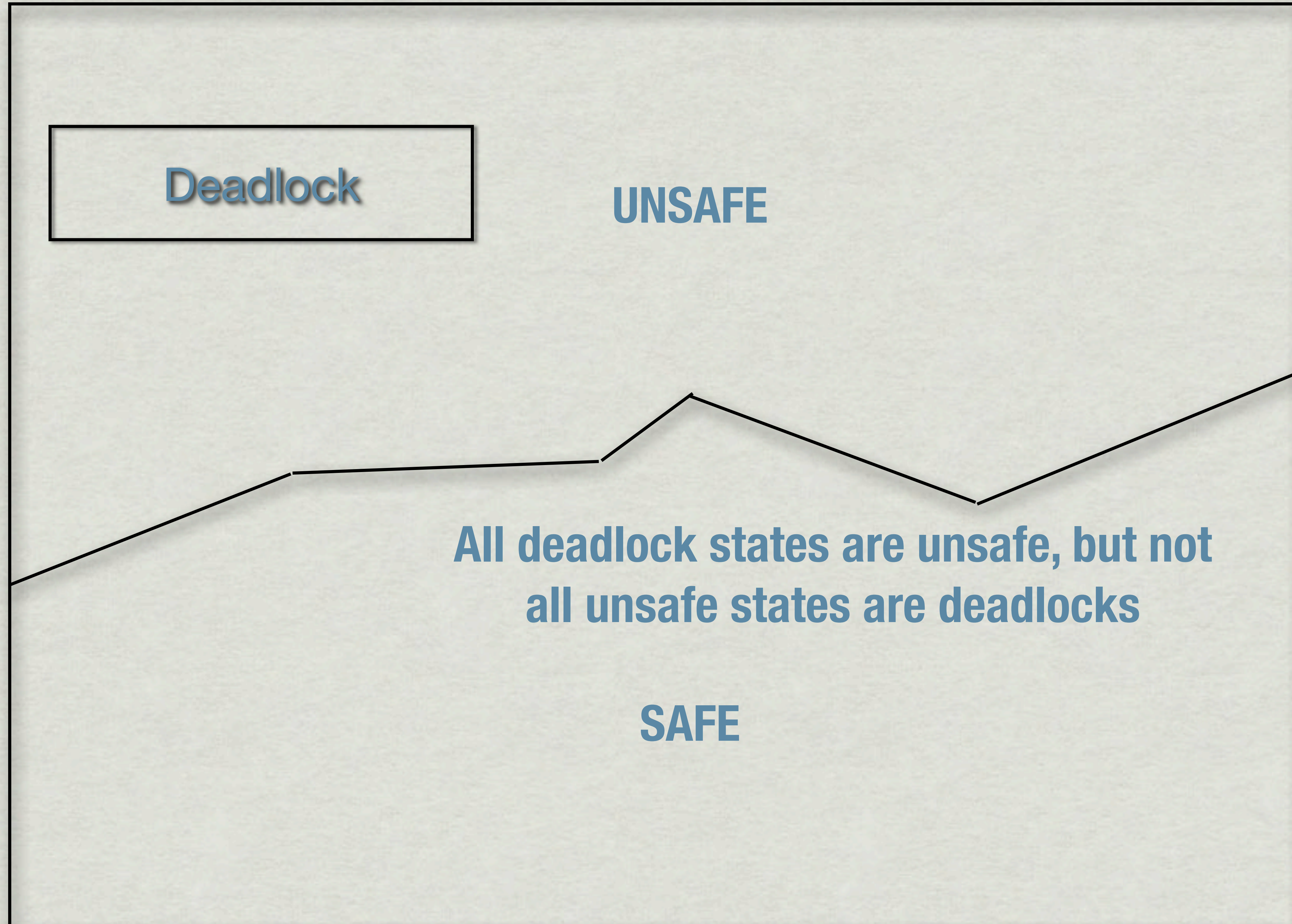
Deadlock Avoidance

An alternative to deadlock prevention

Key concept: safe state

In a safe state there exists some ordering of resource grants that guarantees all processes can complete without deadlock


Deadlock
UNSAFE
SAFE


Deadlock
UNSAFE
All deadlock states are unsafe, but not all unsafe states are deadlocks
SAFE

Our Goal:

Keep everything in a safe state

Banker's Algorithm

allow the sum of maximum resource needs of all current threads to be greater than the total resources, as long as there is some way for all threads to finish without getting into deadlock

Banker’s Algorithm

- need to state maximum resource needs in advance
- allocate resources dynamically

```
/
// Invariant: the system is in a safe state
//
ResourceMgr::Request(ResourceID resource,
                RequestorID thread){
        lock.acquire();
        assert(system is in a safe state);

        while(the state that would result from
           giving resource to thread is not safe){
                cv.wait(&mutex);
            }
        update state by giving resource to thread
        assert(system is in a safe state);
        lock.release();
}
```

```
/
// Invariant: the system is in a safe state
//
ResourceMgr::Request(ResourceID resource,
                RequestorID thread){
        lock.acquire();
        assert(system is in a safe state);

        while(the state that would result from
           giving resource to thread is not safe){
                cv.wait(&mutex);
            }
        update state by giving resource to thread
        assert(system is in a safe state);
        lock.release();
}
```

THE TRICK IS HOW TO DETERMINE IF THERE IS A SAFE SEQUENCE

```
Max[i,j] // max resource j needed by process i
Alloc[i,j] // current allocation of resource j to process i
Need[i,j] = Max[i,j] – Alloc[i,j]
Avail[j] // number of resource j available
```

```
TestSafe(Max[], Alloc[], Need[], Avail[]){
     Work[] = avail[]
     Finish[] = 0,0,0,... // Boolean; is process i finished?
            repeat{
         find i s.t. finish[i] = false and need[i] < work
         if no such i exists
            if finish[i] = true forall i return true
            else return false
         else
            work = work + alloc[i]
            finish[i] = true
```