Deadlock & Dining with my Phils

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Maya Huizar Sid Sannapareddy

Poll Everywhere

pollev.com/tqm

- Any planned courses for Fall 2025? Any Questions about PennOS?

Administrivia

- PennOS
  - Groups have been assigned
  - TA's have been assigned to groups
  - You have the first milestone, which needs to be done before end of day Tuesday the 8th. **TOMORROW**
  - Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification

- Videos containing some demos of a functioning PennOS posted on the schedule.

Administrivia

- PennOS Advice:
  - Will announce this on Ed as well
  - In your FAT code you may do something like this:

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

    - Sometimes though, the write and lseek will return a success, but it won't actually write to your file system
    - Most commonly happens with blocks near the end of the FAT
(as in blocks not in the allocation table but show up shortly after the end of the allocation table)
    - Most likely related to an issue between mmap and write
    - Shows up inconsistently!
    - What's the fix?
Just do it twice, that usually fixes it.

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

Poll Everywhere

pollev.com/tqm

- Any planned courses for Fall 2025? Any Questions about PennOS?

Deadlock Prevention Summary

- Prevent deadlocks by removing any one of the four deadlock preconditions
- But eliminating even one of the preconditions is often hard/impossible
  - Mutual Exclusion is necessary in a lot of situations
  - Forcing a lower priority process to release resources early requires rollback of execution
  - Not always possible to know all resources that an operating system or process will use upfront

Lecture Outline

- **Dining Philosophers**
- Deadlock Handling

Dining Philosophers

- Assume the following situation
  - There are N philosophers (computer scientists) that are trying to eat rice.
  - They only have one chopstick each!
    - Need two chopsticks to eat ☹
  - Alternate between two states:
    - Thinking
    - Eating
  - They are arranged in a circle with a chopstick between each of them

Dining Philosophers

- Philosophers have good table manners
  - Must acquire two chopsticks to eat
  - Only one philosopher can have a chopstick at a time
- Useful abstraction / “standard problem” try to achieve:
  - Deadlock Free
    - No state where no one gets to eat
  - Starvation Free
    - Solution guarantees that all philosophers occasionally eat
    - Ideally maximize parallel eating

First Solution Attempt

- If we number each philosopher 0 – N and then each chopstick is also 0 – N, we can model the problem with mutexes, each chopstick is a mutex and each philosopher is a thread
  - To eat, thread I must acquire lock **`I`** and **`I + 1`**
  - This ensures that each chopstick is only in use by one philosopher at a time

```
while (true) {
  pthread_mutex_lock(&chopstick[i]);
  pthread_mutex_lock(&chopstick[(i + 1) % N]);
  eat();
  pthread_mutex_unlock(&chopstick[(i + 1) % N]);
  pthread_mutex_unlock(&chopstick[i]);
  think();
}
```

Poll Everywhere

pollev.com/tqm

- What's wrong with this? Any Ideas on how to fix it?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
while (true) {
  pthread_mutex_lock(&chopstick[i]);
  pthread_mutex_lock(&chopstick[(i + 1) % N]);
  eat();
  pthread_mutex_unlock(&chopstick[(i + 1) % N]);
  pthread_mutex_unlock(&chopstick[i]);
  think();
}
```

Poll Everywhere

pollev.com/tqm

- What's wrong with this? Any Ideas on how to fix it?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
while (true) {
  pthread_mutex_lock(&chopstick[i]);
  pthread_mutex_lock(&chopstick[(i + 1) % N]);
  eat();
  pthread_mutex_unlock(&chopstick[(i + 1) % N]);
  pthread_mutex_unlock(&chopstick[i]);
  think();
}
```

Deadlock is possible: what happens if all threads pickup their left at the same time?

Second Attempt: Round Robin

- Our first attempt deadlocks.
- What if we instead we tried doing this “round robin”, we pass around a token that says “it is your turn to eat”
- Can this deadlock?
- What issues arise with this solution?

Second Attempt: Round Robin

- Our first attempt deadlocks.
- What if we instead we tried doing this "round robin", we pass around a token that says "it is your turn to eat"
- Can this deadlock?

No

- What issues arise with this solution?

Not parallel, just sequential eating ☹
Everyone guaranteed gets to eat though ☺

Third Attempt: Global Mutex

- What if instead, we add another “global” mutex that controls permission to pick up chopsticks. Once a philosopher has chopsticks, they can release the lock before they eat

- In our metaphor, this means that each philosopher “waits in line” to pick up chopsticks

- Can this deadlock?

- What issues arise with this solution?

Third Attempt: Global Mutex

- What if instead, we add another "global" mutex that controls permission to pick up chopsticks. Once a philosopher has chopsticks, they can release the lock before they eat

- In our metaphor, this means that each philosopher "waits in line" to pick up chopsticks

- Can this deadlock?

  No

- What issues arise with this solution?

  Not the most parallel, could result in sequential
  Not everyone guarantee gets to eat

Fourth Attempt: More Human Approach

- What if instead, if a philosopher fails to get a chopstick, it puts down any chopsticks it has, waits for a little bit and then tries again?

- Can we do this in code?
  - `pthread_mutex_trylock`: if the lock can't be acquired, return immediately
  - `pthread_mutex_timedlock`: timeout after trying to get a mutex for some specified amount of time

- Can this deadlock?
- What issues arise with this solution?

Fourth Attempt: More Human Approach

- What if instead, if a philosopher fails to get a chopstick, it puts down any chopsticks it has, waits for a little bit and then tries again?

- Can we do this in code?
  - `pthread_mutex_trylock`: if the lock can't be acquired, return immediately
  - `pthread_mutex_timedlock`: timeout after trying to get a mutex for some specified amount of time

No

- Can this deadlock?
- What issues arise with this solution?

Possible spinning and starvation

Fifth Attempt: Break the Symmetry

- What if the even numbered philosophers and odd numbered philosophers do things differently?
  - Even Numbered: Grab chopstick on their left and then right
  - Odd Numbered: Grab chopstick on their right and then left

- Can this deadlock?

- What issues arise with this solution?

Fifth Attempt: Break the Symmetry

- What if the even numbered philosophers and odd numbered philosophers do things differently?
  - Even Numbered: Grab chopstick on their left and then right
  - Odd Numbered: Grab chopstick on their right and then left

- Can this deadlock?

No

- What issues arise with this solution?

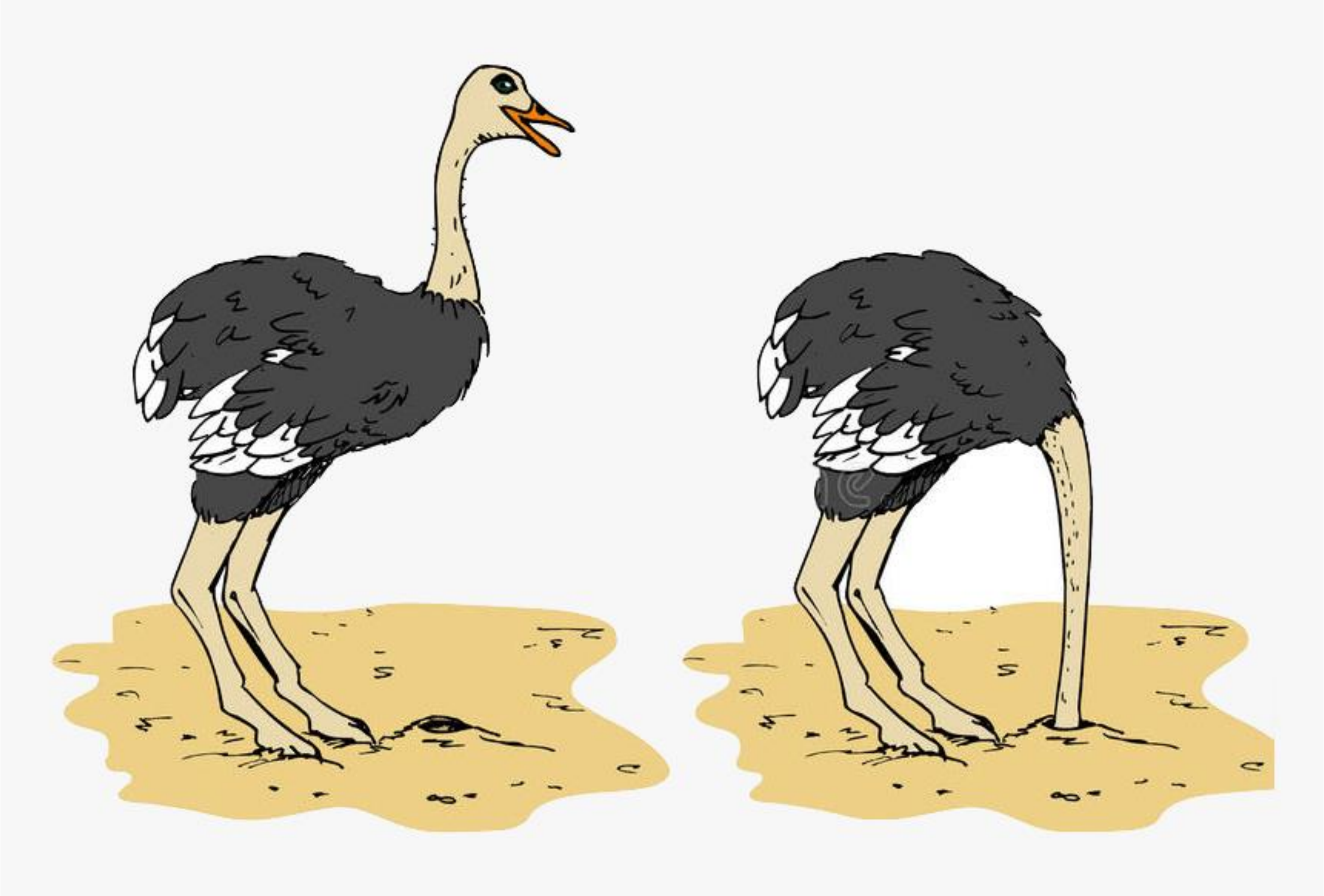
threads may still possibly starve

Lecture Outline

- Dining Philosophers
- **Deadlock Handling**

Deadlock Handling: Ostrich Algorithm

Deadlock Handling: Ostrich Algorithm



Ostriches don't actually do this, but it is an old myth

Deadlock Handling: Ostrich Algorithm

- Ignoring potential problems
  - Usually under the assumption that it is either rare, too expensive to handle, and/or not a fatal error

- Used in real world contexts, there is a real cost to tracking down every possible deadlock case and trying to fix it
  - Cost on the developer side: more time to develop
  - Cost on the software side: more computation for these things to do, slows things down

Deadlock Handling: Prevention

- Ad Hoc Approach
  - Key insights into application logic allow you to write code that avoids cycles/deadlock
  - Example: Dining Philosophers breaking symmetry with even/odd philosophers

- Exhaustive Search Approach
  - Static analysis on source code to detect deadlocks
  - Formal verification: model checking
  - Unable to scale beyond small programs in practice
    Impossible to prove for any arbitrary program (without restrictions)

Detection

- If we can't guarantee deadlocks won't happen, we can instead try to detect a deadlock just before it will happen **and then intervene**.

- Two big parts
  - Detection algorithm. This is usually done with tracking metadata and graph theory
  - The intervention/recovery. We typically want some sort of way to "recover" to a safe state when we detect a deadlock is going to happen

Detection Algorithms

- The common idea is to think of the threads and resources as a graph.
  - If there is a cycle: deadlock
  - If there is no cycle: no deadlock

- Finding cycles in a graph is a common algorithm problem with many solutions.

Poll Everywhere

pollev.com/tqm

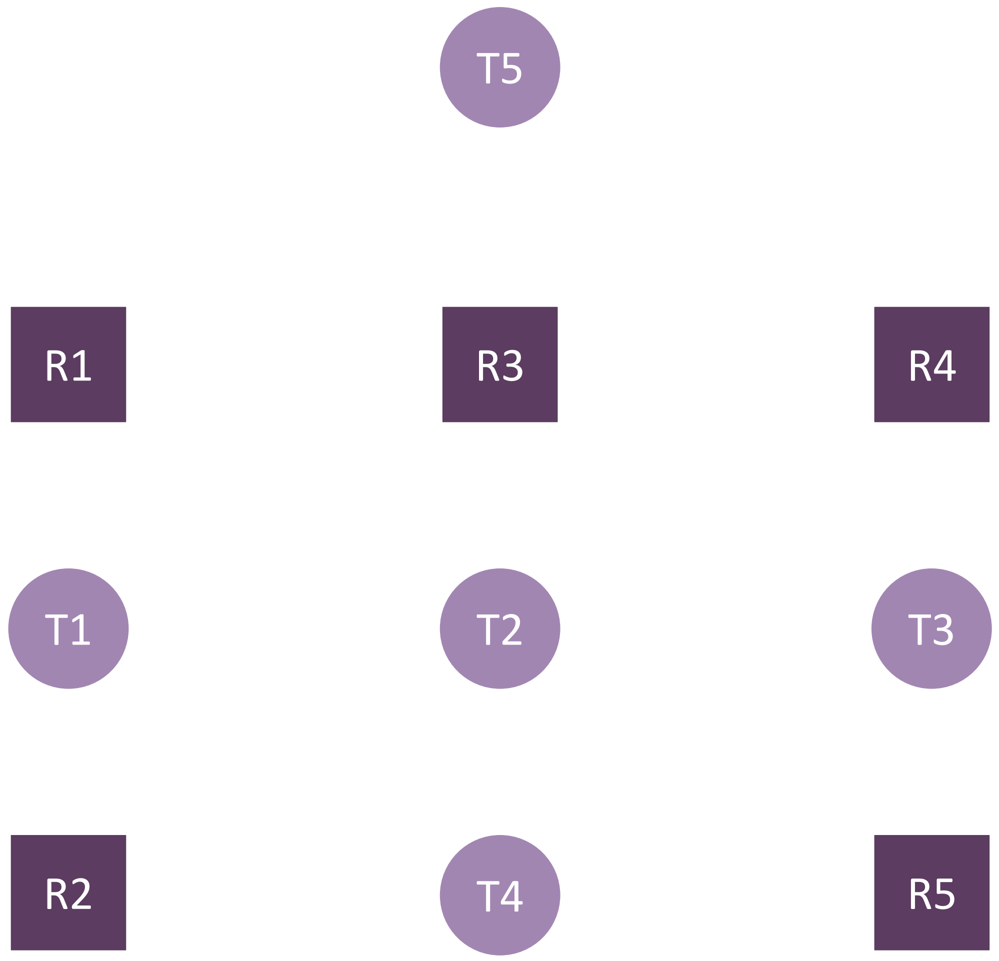
- Consider the following example with 5 threads and 5 resources that require mutual exclusion is this a deadlock?
  - Thread 1 has R2 but wants R1
  - Thread 2 has R1 but wants R3, R4 and R5
  - Thread 3 has R4 but wants R5
  - Thread 4 has R5 but wants R2
  - Thread 5 has R3

Resource Allocation Graph

- We can represent this deadlock with a graph:
  - Each resource and thread is a node
  - If a thread has a resource, draw an arrow pointing at the thread form that resource
  - If a thread wants to acquire a resource but can’t, draw an arrow pointing at the resource from the thread trying to acquire it

Resource Allocation Graph Example

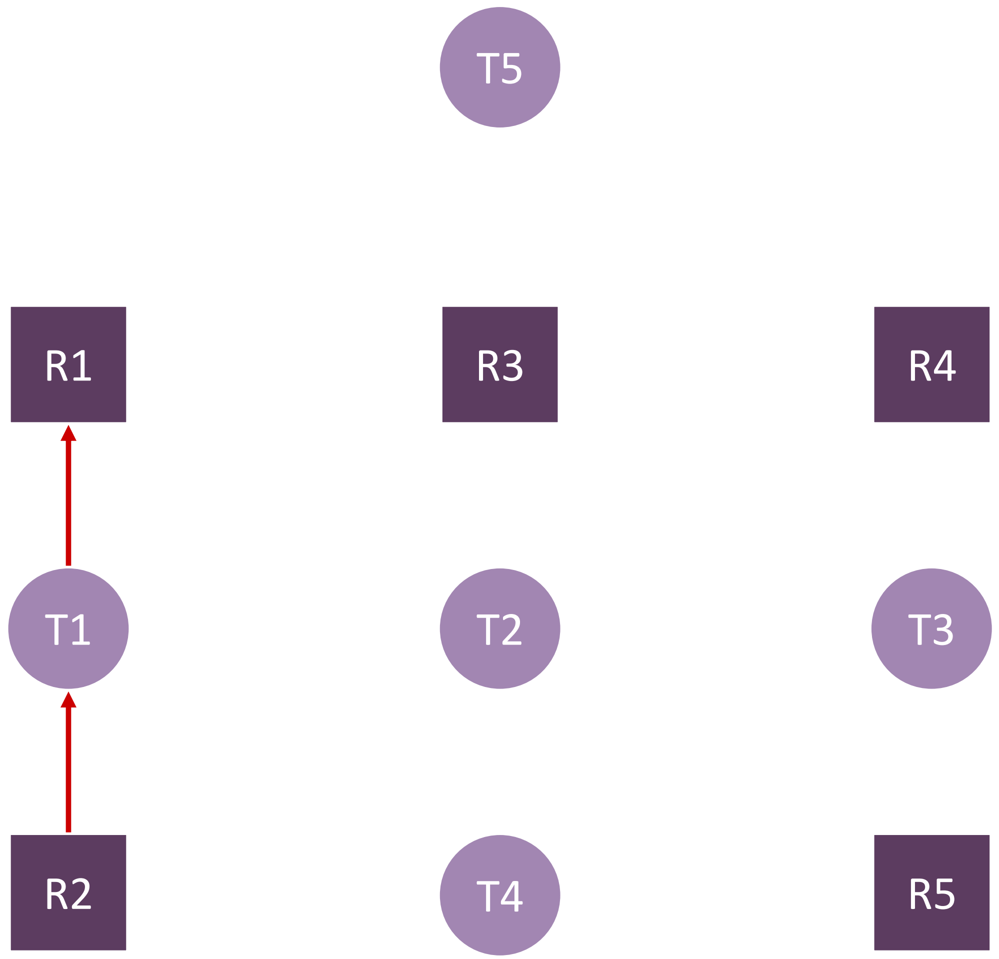
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

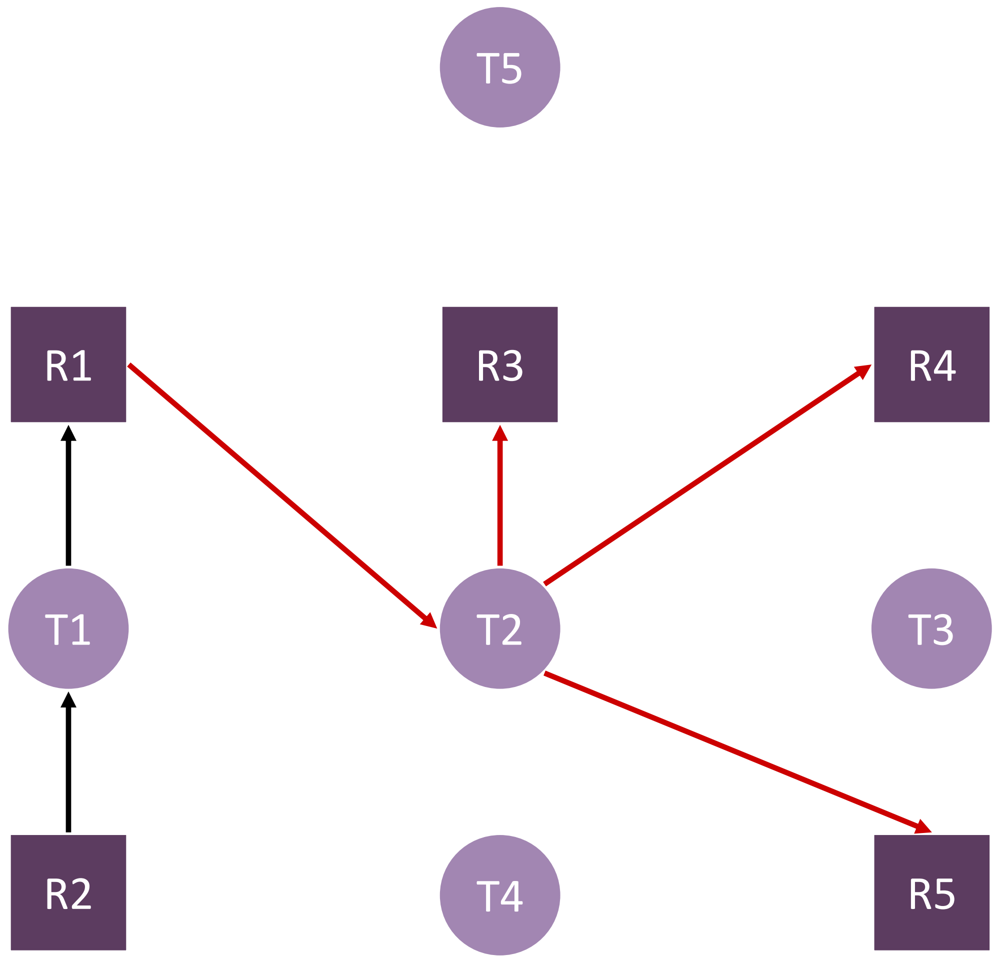
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

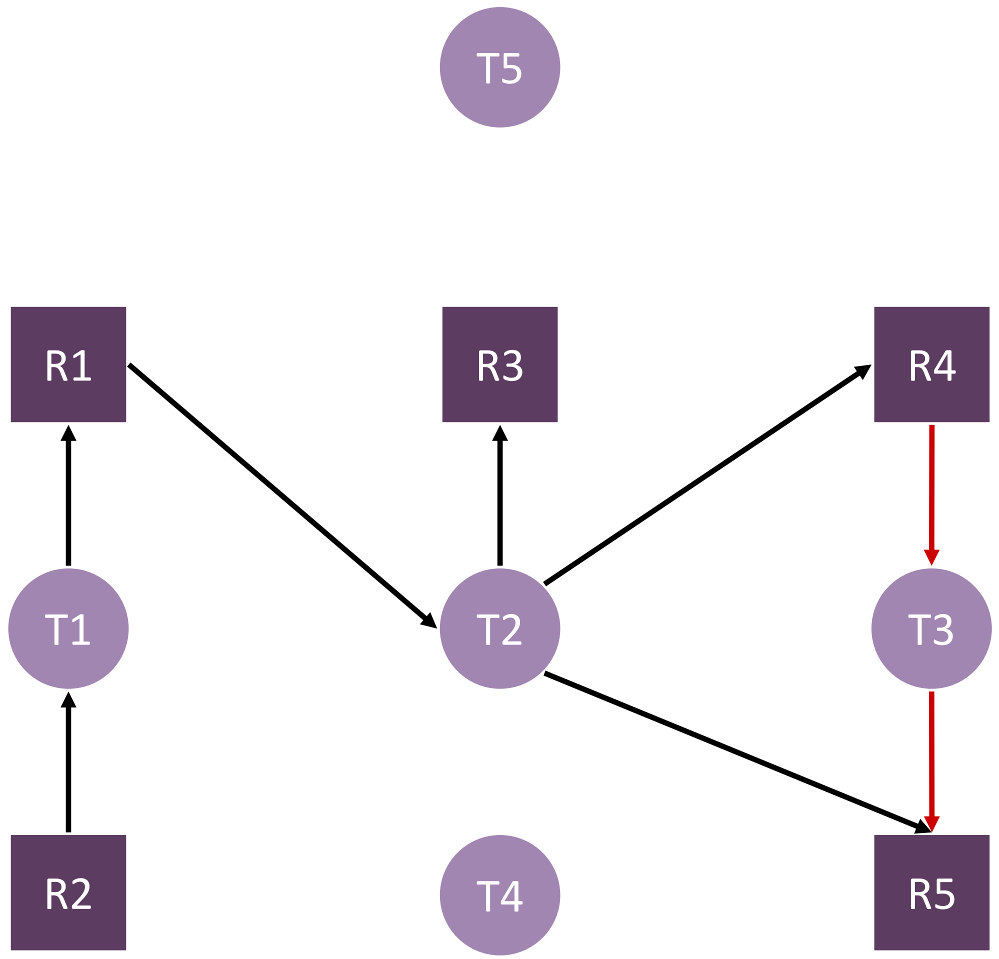
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

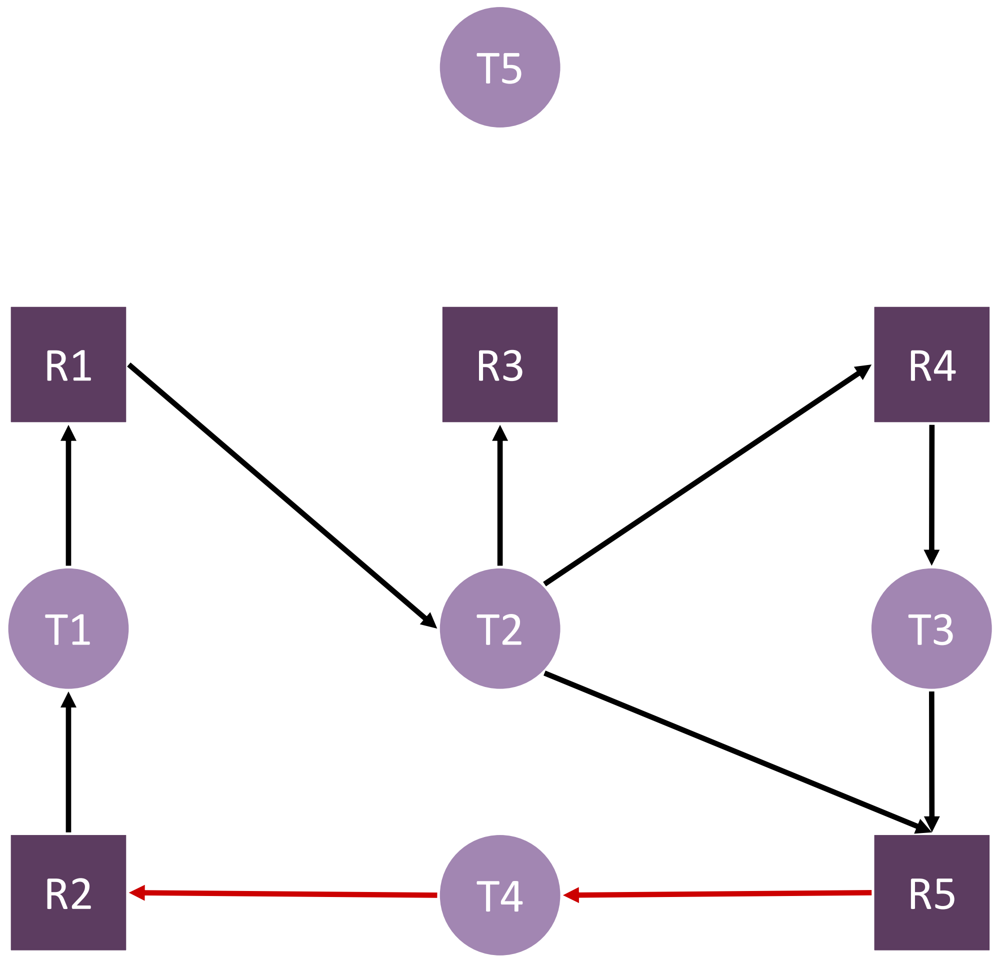
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

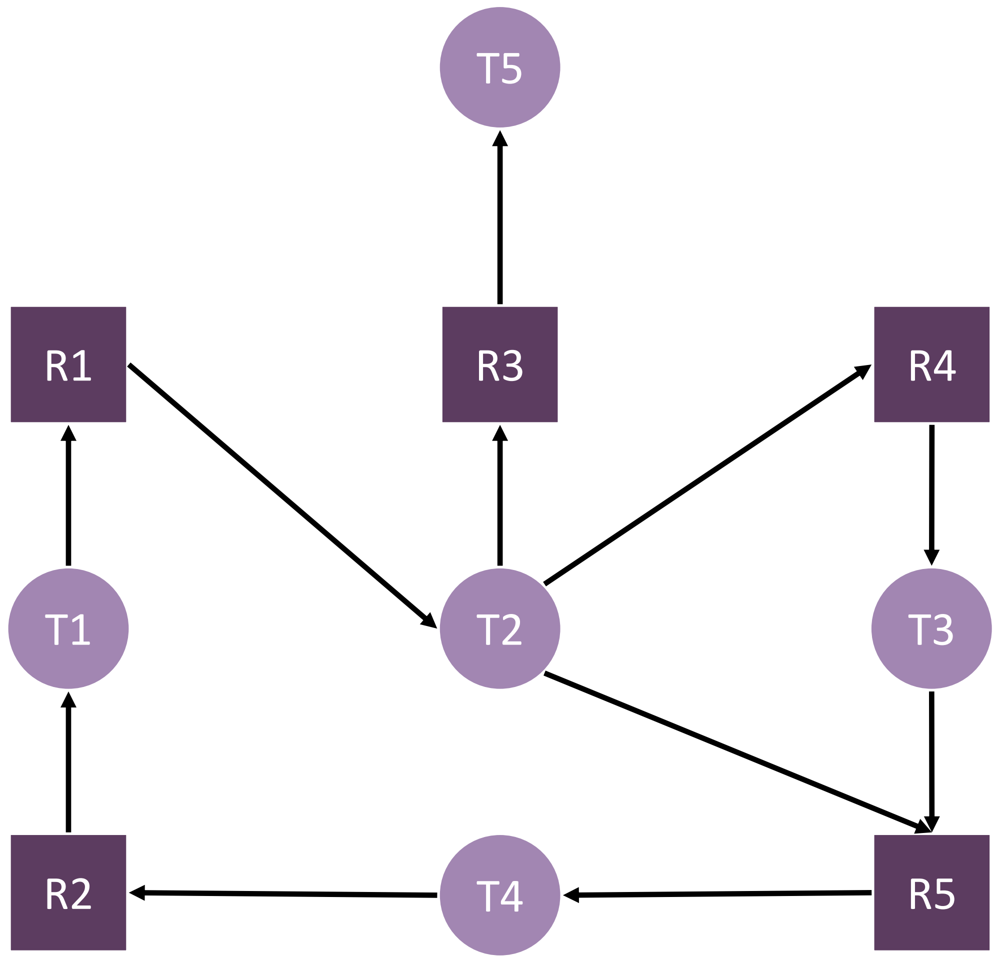
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

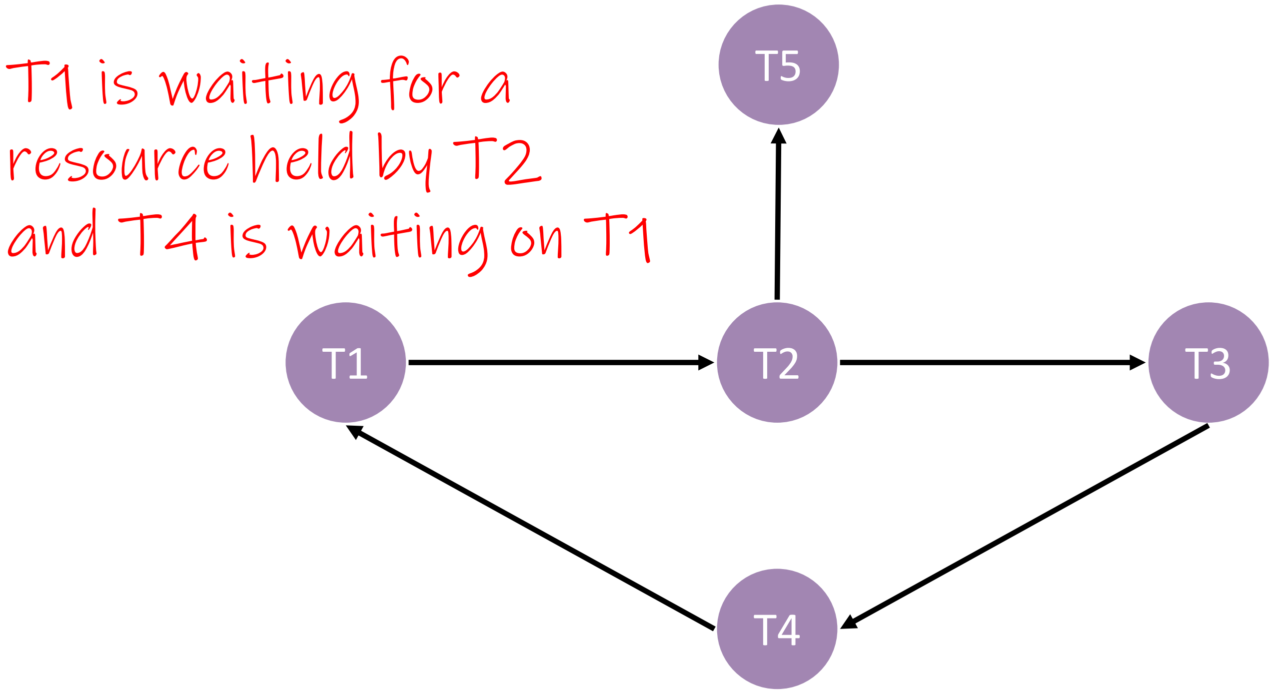
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Alternate graph

- Instead of also representing resources as nodes, we can have a "wait for" graph, showing how threads are waiting on each other



Wait For Graph

Recovery after Detection

- Preemption:
  - Force a thread to give up a resource
  - Often is not safe to do or impossible
- Rollback:
  - Occasionally checkpoint the state of the system, if a deadlock is detected then go back to the checkpointed “Saved state”
  - Used commonly in database systems
  - Maintaining enough information to rollback and doing the rollback can be expensive
- Manual Killing:
  - Kill a process/thread, check for deadlock, repeat till there is no deadlock
  - Not safe, but it is simple

Overall Costs

- Doing Deadlock Detection & Recovery solves deadlock issues, but there is a cost to memory and CPU to store the necessary information and check for deadlock

- This is why sometimes the ostrich algorithm is preferred

Avoidance

- Instead of detecting a deadlock when it happens and having expensive rollbacks, we may want to instead avoid deadlock cases earlier

- Idea:
  - Before it does work, it submits a request for all the resources it will need.
  - A deadlock detection algorithm is run
    - If acquiring those resources would lead to a deadlock, deny the request. The calling thread can try again later
    - If there is no deadlock, then the thread can acquire the resources and complete its task
  - The calling thread later releases resources as they are done with them

Avoidance

- Pros:
  - Avoids expensive rollbacks or recovery algorithms

- Cons:
  - Can't always know ahead of time all resources that are required
  - Resources may spend more time being locked if all resources need to be acquired before an action is taken by a thread, could hurt parallelizability
    - Consider a thread that does a very expensive computation with many shared resources.
    - Has one resources that is only updated at the end of the computation.
    - That resources is locked for a long time and other threads that may need it cannot access it

Aside: Bankers Algorithm

- This gets more complicated when there are multiple copies of resources, or a finite number of people can access a resources.

- The Banker's Algorithm handles these cases
  - But I won't go into detail about this
  - There is a video linked on the website under this lecture you can watch if you want to know more