Threads & Deadlock

Computer Operating Systems, Spring 2025

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah	Bo Sun	Joy Liu	Susan Zhang	Zihao Zhou
Akash Kaukuntla	Connor Cummings	Khush Gupta	Vedansh Goenka	
Alexander Cho	Eric Zou	Kyrie Dowling	Vivi Li	
Alicia Sun	Haoyun Qin	Rafael Sakamoto	Yousef AlRabiah	
August Fu	Jonathan Hong	Sarah Zhang	Yu Cao	

Poll Everywhere

pollev.com/tqm

- Any planned courses for Fall 2025?

Administrivia

- PennOS
  - Groups have been assigned
  - TA's have been assigned to groups
  - You have the first milestone, which needs to be done sometime next week
  - 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
  - We will send emails to every group that had to be filled by course staff soon (let us know if you don't get this by the end of the week)

- Will post a small assignment with some readings sometime before next lecture
  - Details coming soon

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);
```

Lecture Outline

- **Mutex refresher**
- Mutex alternatives
  - tsl
  - Disable interrupts
  - Petersons
- Deadlocks
- Dining Philosophers
- Deadlock Handling

pthreads and Locks

- Another term for a lock is a mutex ("mutual exclusion")
  - `pthread.h` defines datatype `pthread_mutex_t`

- ```
  int pthread_mutex_init(pthread_mutex_t* mutex,
                         const pthread_mutexattr_t* attr);
  ```
  - Initializes a mutex with specified attributes
- ```
  int pthread_mutex_lock(pthread_mutex_t* mutex);
  ```
  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired
- ```
  int pthread_mutex_unlock(pthread_mutex_t* mutex);
  ```
  - Releases the lock
- ```
  int pthread_mutex_destroy(pthread_mutex_t* mutex);
  ```
  - "Uninitializes" a mutex – clean up when done

pthread Mutex Examples

- See `total.c`
  - Data race between threads
- See `total_locking.c`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.c`

Poll Everywhere

pollev.com/tqm

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

- Thread-1 executes line 8 while Thread-2 executes line 21.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 15 while Thread-2 executes line 15.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

```
1  // global variables
2  pthread_mutex_t lock;
3  int g =    0;
4  int k = 0;
5
6  void fun1() {
7    pthread_mutex_lock(&lock);
8    g += 3;
9    pthread_mutex_unlock(&lock);
10   k++;
11 }
12
13 void fun2(int a, int b) {
14   g += a;
15   a += b;
16   k = a;
17 }
18
19 void fun3() {
20   pthread_mutex_lock(&lock);
21   g = k + 2;
22   pthread_mutex_unlock(&lock);
23 }
```

Poll Everywhere

pollev.com/tqm

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

- Thread-1 executes line 8 while Thread-2 executes line 14
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

```
1   // global variables
2   pthread_mutex_t lock;
3   int g =    0;
4   int k = 0;
5
6   void fun1() {
7     pthread_mutex_lock(&lock);
8     g += 3;
9     pthread_mutex_unlock(&lock);
10    k++;
11  }
12
13  void fun2(int a, int b) {
14    g += a;
15    a += b;
16    k = a;
17  }
18
19  void fun3() {
20    pthread_mutex_lock(&lock);
21    g = k + 2;
22    pthread_mutex_unlock(&lock);
23  }
```

Lecture Outline

- Mutex refresher
- **Mutex alternatives**
  - **tsl**
  - **Disable interrupts**
  - **Petersons**
- Deadlocks
- Dining Philosophers
- Deadlock Handling

TSL

- TSL stands for **T**est and **S**et **L**ock, sometimes just called **test-and-set**.

- TSL is an atomic instruction that is guaranteed to be atomic at the hardware level

- `TSL R, M`
  - Pass in a register and a memory location
  - R gets the value of M
  - M is set to 1 AFTER setting R

TSL to implement Mutex

- A mutex is pretty much this:

```
pthread_mutex_lock(lock) {
    prev_value = TSL(lock);

    // if prev_value = 1, then it was already locked
    while (prev_value == 1) {
        block();
        prev_value = TSL(lock);
    }
}

pthread_mutex_unlock(lock) {
   lock = 0;
   wakeup_blocked_threads(lock);
}
```

Disabling Interrupts

- If data races occur when one thread is interrupted while it is accessing some shared code….

  What is we don’t switch to other threads while executing that code?

- This can be done by disabling interrupts: no interrupts means that the clock interrupt won’t go off and interrupt the currently running thread

Disabling Interrupts

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0

```
disable_interrupts();
++sum_total;
enable_interrupts();
```

Thread 1

```
disable_interrupts();
++sum_total;
enable_interrupts();
```

Disabling Interrupts

- Advantages:
  - This is one way to fix this issue

- Disadvantages
  - This is usually overkill
  - This can stop threads that aren't trying to access the shared resources in the critical section. May stop threads that are executing other processes entirely
  - If interrupts disabled for a long time, then other threads will starve
  - In a multi-core environment, this gets complicated

Discuss

- Lets try a more complicated software approach..
- We create two threads running thread_code, one with **`arg = 0`**, other thread has **`arg = 1`**
- Each thread tries to increment sum_total. Does this work?

```
int sum_total = 0;
bool flag[2] = {false, false};
int turn = 0

void thread_code(int arg) {
  int me = arg;

  flag[me] = true;
  turn = 1 - me;
  while((flag[1-me] == true) && (turn != me)) { }
  ++sum_total;
  flag[me] = false;
}
```

Check the index of the other thread

Peterson's Algorithm

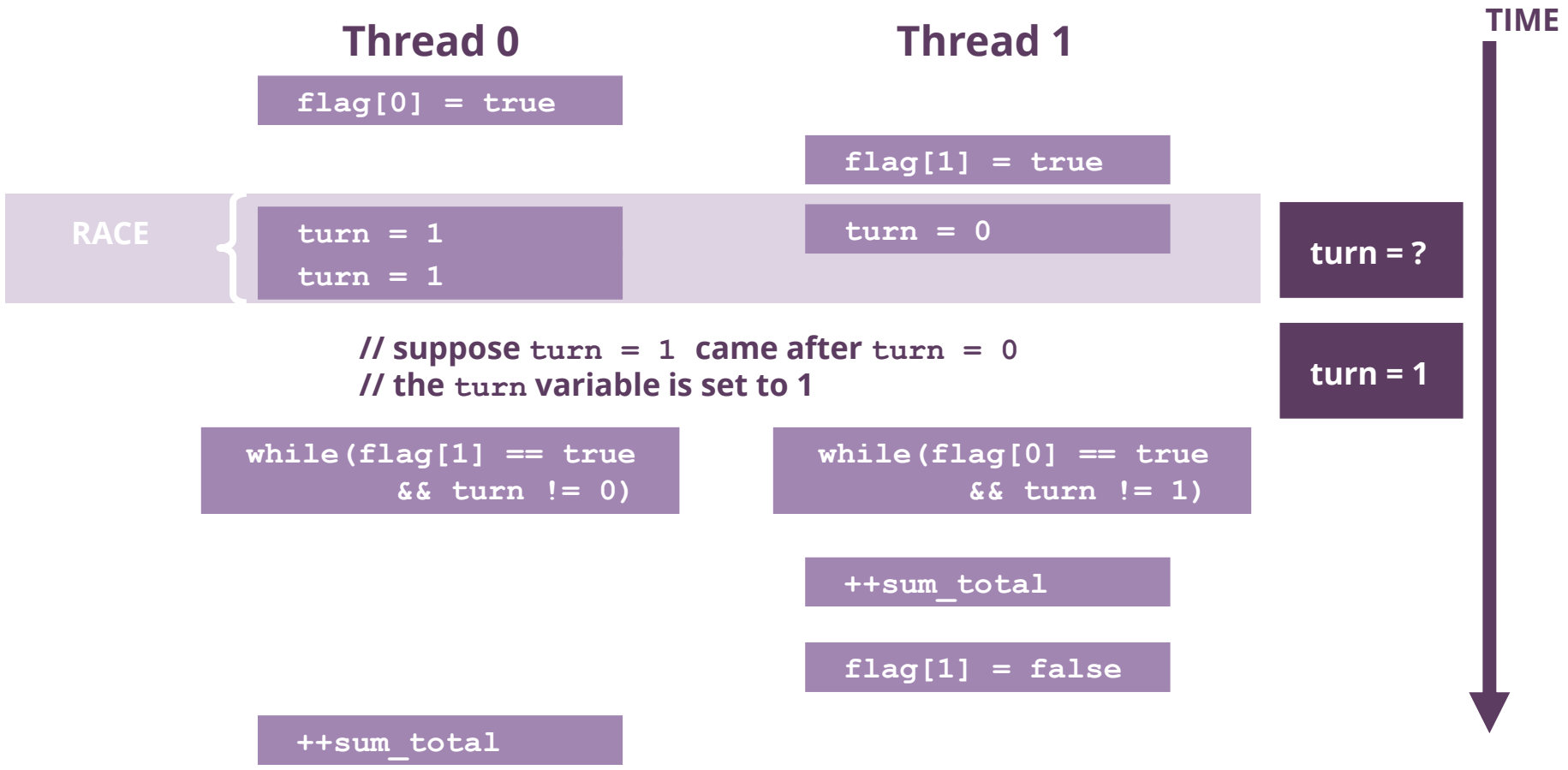
- What we just did was Peterson's algorithm
- Why does it work? (using an analogy)
  - Each thread first declares that they want to enter the critical section by setting their flag
  - Each thread then states (once) that the other should "go first".
    - This is done by setting the turn variable to 1 – me
    - One of these assignments to the turn variable will happen last, that is the one that decides who goes first
  - One of the thread goes first (decided by the value of turn) and accesses the critical section, before saying it is done (by changing their flag to false)

Peterson's Algorithm

- What we just did was Peterson's algorithm
- Why does it work?
  - Case1:
    If P0 enters critical section, flag[0] = true, turn = 0. It enters the critical section successfully.
  - Case2:
    If P0 and P1 enter critical section, flag[0] and flag[1] = true

    Race condition on turn. Suppose P0 sets turn = 0 first. Final value is turn = 1. P0 will get to run first.

Explanation



Peterson's Assumptions

- Some operations are atomic:
  - Reading from the flag and turn variables cannot be interrupted
  - Writing to the flag and turn variables cannot be interrupted
  - E.g setting turn = 1 or 0 will set turn to 0 or 1, you can be interrupted before or after, but not "during" when turn may have some intermediate value that is not 0 or 1

- That the instructions are executed in the specific order laid out in the code

Atomicity

- Atomicity: An operation or set of operations on some data are *atomic* if the operation(s) are indivisible, that no other operation(s) on that same data can interrupt/interfere.

- Aside on terminology:
  - Often interchangeable with the term “Linearizability”
  - Atomic has a different (but similar-ish) meaning in the context of data bases and ACID.

Aside: Instruction & Memory Ordering

- Do we know that **t** is set before **g** is set?

```
bool g = false;
int t = 0

void some_func(int arg) {
  t = arg;
  g = true;
}
```

Aside: Instruction & Memory Ordering

- The compiler may generate instructions with different ordering if it does not appear that it will affect the semantics of the function
  - Since `g = true;` is not affected by `t = arg;` then either one could execute first.

- The Processor may also execute these in a different order than what the compiler says

- Why? Optimizations on program performance
  - If you want to know more, look into “Out-of-Order Execution” and “Memory Order”

Aside: Memory Barriers

- How do we fix this?

- We can emit special instructions to the CPU and/or compiler to create a "memory barrier"
  - "all memory accesses before the barrier are guaranteed to happen before the memory accesses that come after the barrier"

  - A way to enforce an order in which memory accesses are ordered by the compiler and the CPU

  - This is done for us when we mark a variable as atomic or use a lock.

Lecture Outline

- Mutex refresher
- Mutex alternatives
  - tsl
  - Disable interrupts
  - Petersons
- **Deadlocks**
- Dining Philosophers
- Deadlock Handling

Liveness

- Liveness: A set of properties that ensure that threads execute in a timely manner, despite any contention on shared resources.

- When `pthread_mutex_lock();` is called, the calling thread blocks (stops executing) until it can acquire the lock.
  - What happens if the thread can never acquire the lock?

Liveness Failure: Releasing locks

- If locks are not released by a thread, then other threads cannot acquire that lock

- See `release_locks.c`
  - Example where locks are not released once critical section is completed.

Liveness Failure: Deadlocks

- Consider the case where there are two threads and two locks
  - Thread 1 acquires lock1
  - Thread 2 acquires lock2
  - Thread 1 attempts to acquire lock2 and blocks
  - Thread 2 attempts to acquire lock1 and blocks

Neither thread can make progress ☹

- See `milk_deadlock.c`

- Note: there are many algorithms for detecting/preventing deadlocks

Liveness Failure: Mutex Recursion

- What happens if a thread tries to re-acquire a lock that it has already acquired?

- See `recursive_deadlock.c`

- By default, a mutex is not re-entrant.
  - The thread won't recognize it already has the lock, and block until the lock is released

Aside: Recursive Locks

- Mutex's can be configured so that you it can be re-locked if the thread already has locked it. These locks are called *recursive locks* (sometimes called *re-entrant locks*).

- Acquiring a lock that is already held will succeed
- To release a lock, it must be released the same number of times it was acquired

- Has its uses, but generally discouraged.

Deadlock Definition

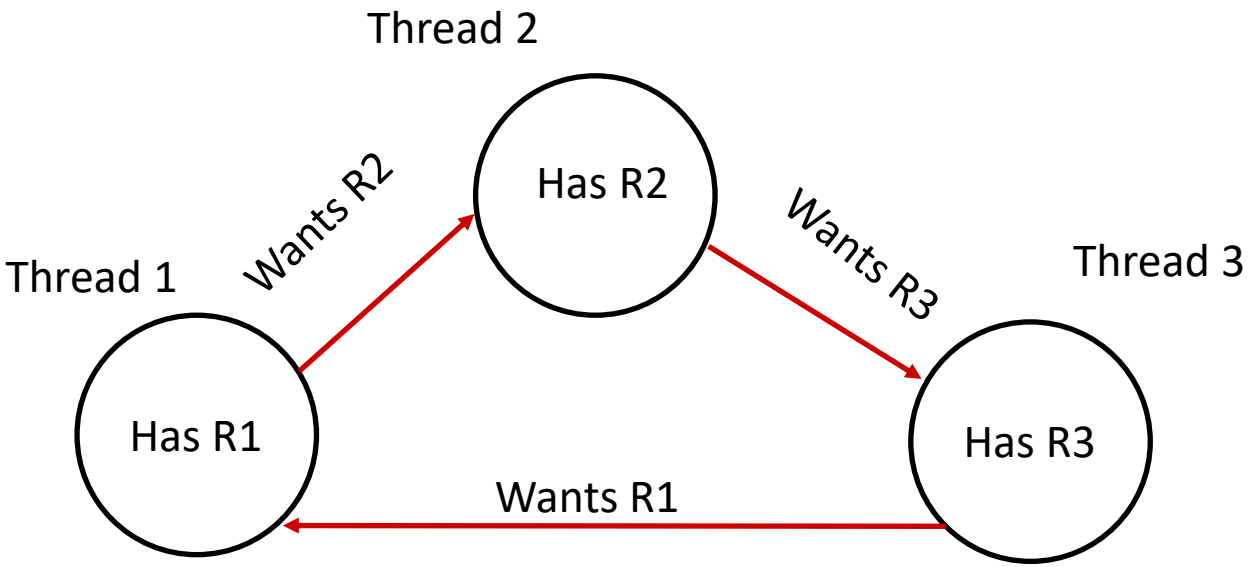
- A computer has multiple threads, finite resources, and the threads want to acquire those resources
  - Some of these resources require exclusive access
- A thread can acquire resources:
  - All at once
  - Accumulate them over time
  - If it fails to acquire a resource, it will (by default) wait until it is available before doing anything
- Deadlock: Cyclical dependency on resource acquisition so that none of them can proceed
  - Even if all unblocked threads release, deadlock will continue

Preconditions for Deadlock

- Deadlock can only happen if these occur simultaneously:
  - **Mutual Exclusion**: at least one resource must be held exclusively by one thread
  - **Hold and Wait**: a thread must be holding a resource, requesting a resource that is held by a thread, and then waiting for it.
  - **No preemption**: A resource is held by a thread until it explicitly releases it. It cannot be preempted by the OS or something else to force it to release the resource
  - **Circular Wait**:
    Can be a chain of more than 2 threads
    Each thread must be waiting for a resource that is held by another thread. That other thread must waiting on a resource that forms a chain of dependency

Circular Wait Example

- A cycle can exist of more than just two threads:



Discuss

- Can a thread deadlock if there is only one thread?

Deadlock Prevention

- If we can remove the conditions for deadlock, we could avoid prevent deadlock from every happening

Deadlock Prevention: Mutual Exclusion

- **Mutual Exclusion**: at least one resource must be held exclusively by one thread

- You usually need mutual exclusion or you don't, so it is hard to avoid.
- Some resources require exclusive access

- A lot of work done related to this
  - called: Lock-free programming, Lock-less programming, or Non-blocking algorithms
  - General idea is to take advantage of operations that are atomic at the hardware level when sharing is needed

Deadlock Prevention: Hold and Wait

- **Hold and Wait**: a thread must be holding a resource, requesting a resource that is held by a thread, and then waiting for it.

- What if we had each thread acquire all resources it needs in the beginning "at once"
  - Not always practical, a thread may not know ahead of time all the resources it will need

Deadlock Prevention: No Preemption

- **No preemption**: A resource is held by a thread until it explicitly releases it. It cannot be preempted by the OS or something else to force it to release the resource

- If we force a thread to release a resource, how do we ensure it is in a valid state?
  - Undoing actions and recovering valid state is complex (more on this next lecture)

Deadlock Prevention: Circular Wait

- **Circular Wait**: Each thread must be waiting for a resource that is held by another thread. That other thread must waiting on a resource that forms a chain of dependency

- Break cycles in resource acquisition.
- We could enforce an ordering to resource acquisition.

- Challenge: Still we may not know all resources we need ahead of time

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

- Mutex refresher
- Mutex alternatives
  - tsl
  - Disable interrupts
  - Petersons
- Deadlocks
- **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();
}
```

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?

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?

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?

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?

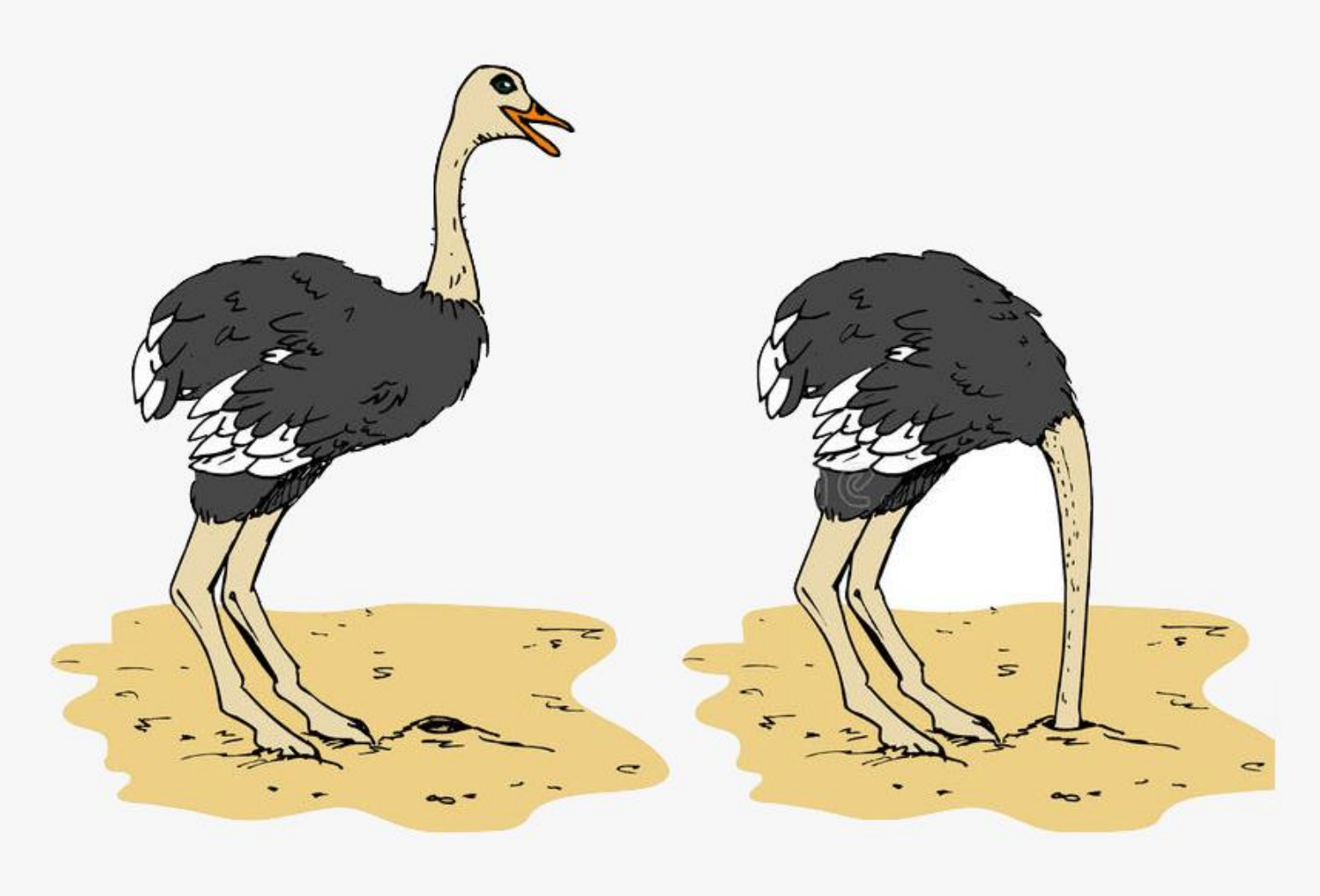
- What issues arise with this solution?

Lecture Outline

- Mutex refresher
- Mutex alternatives
  - tsl
  - Disable interrupts
  - Petersons
- Deadlocks
- 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

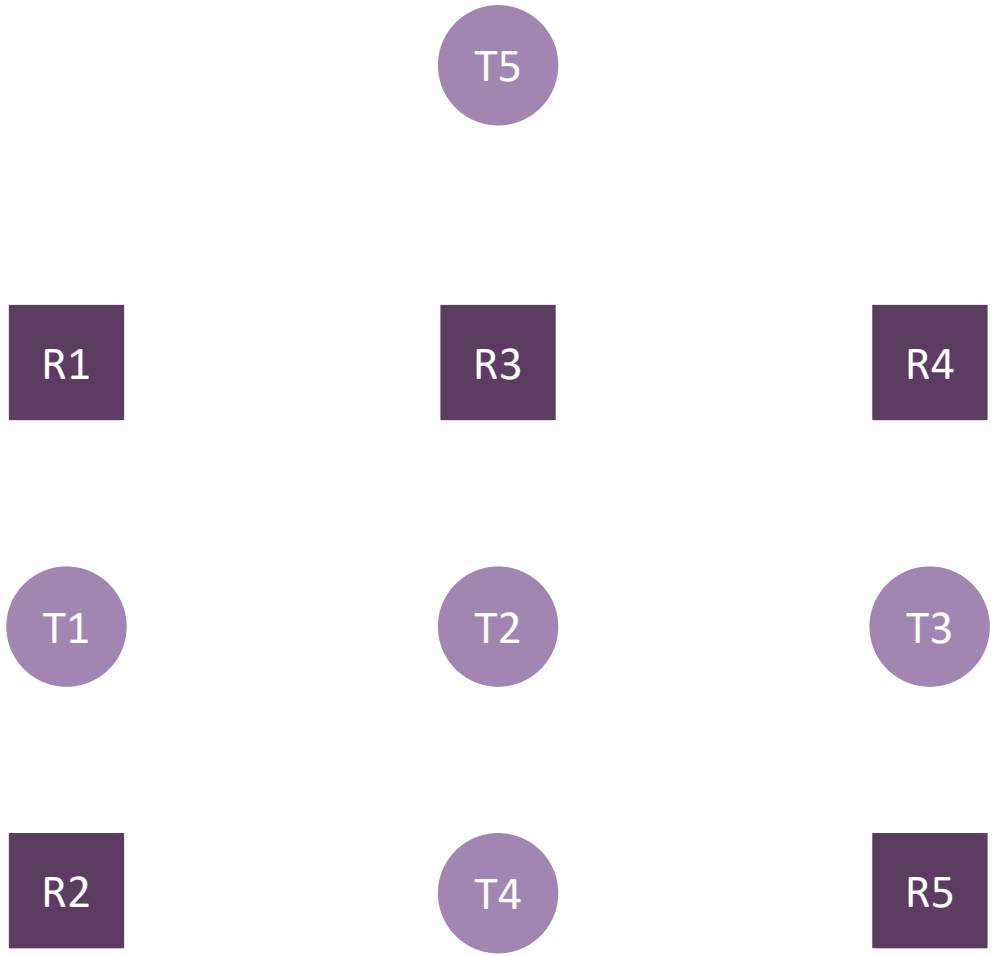
- Consider the following example with 5 threads and 5 resources that require mutual exclusion is this a deadlock?
  - Thread 1 has R2 but wants R1
  - Thread 2 has R1 but wants R3, R4 and R5
  - Thread 3 has R4 but wants R5
  - Thread 4 has R5 but wants R2
  - Thread 5 has R3

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

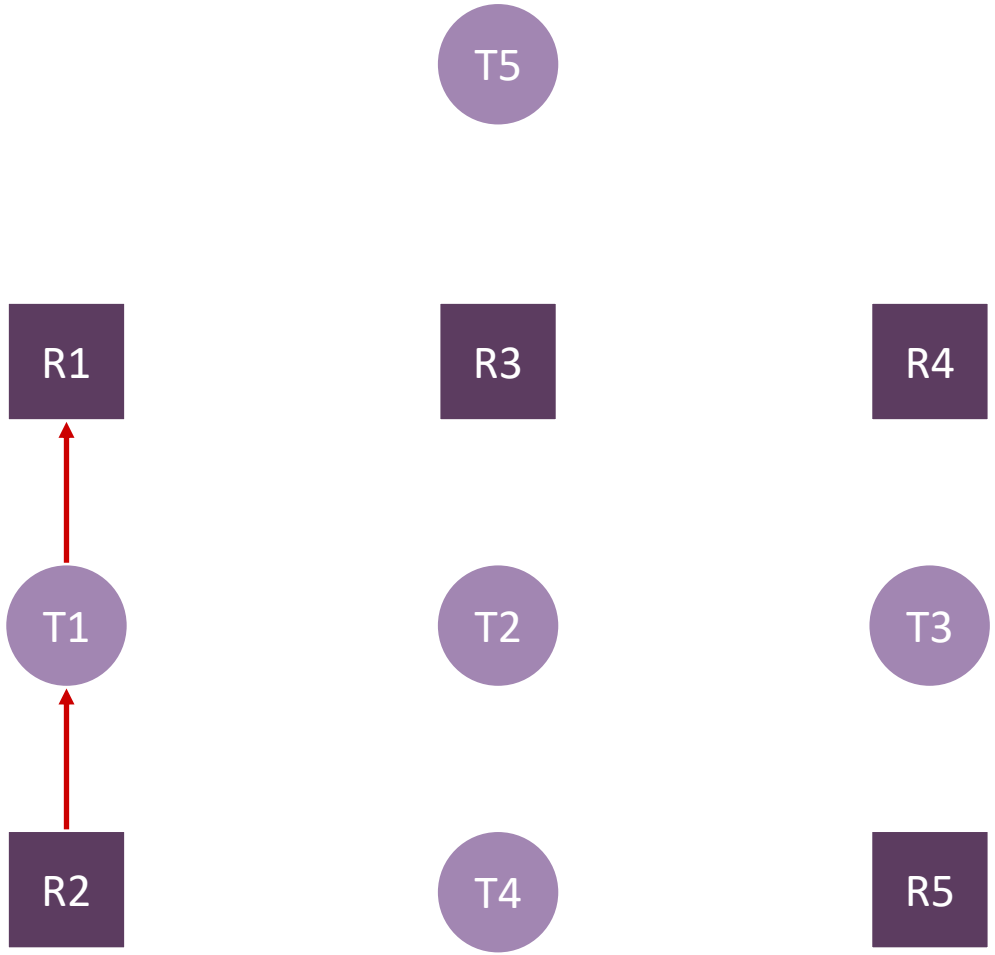
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

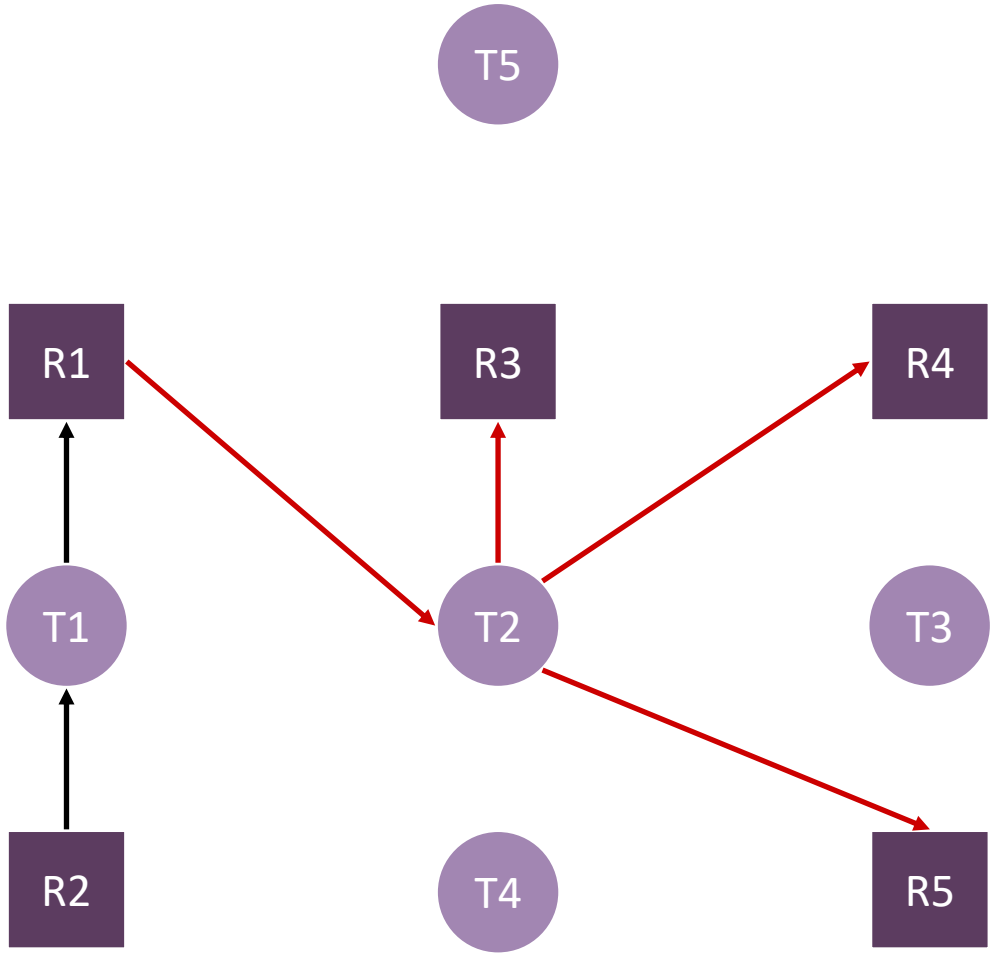
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

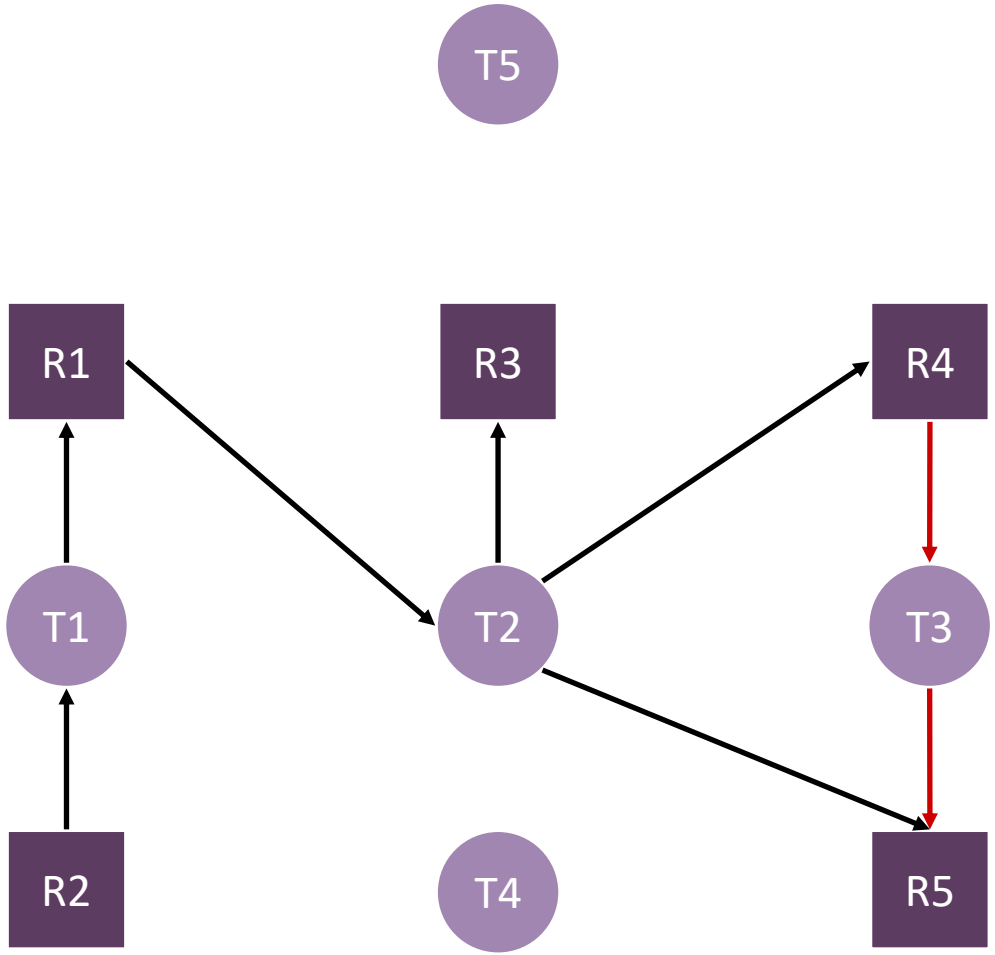
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

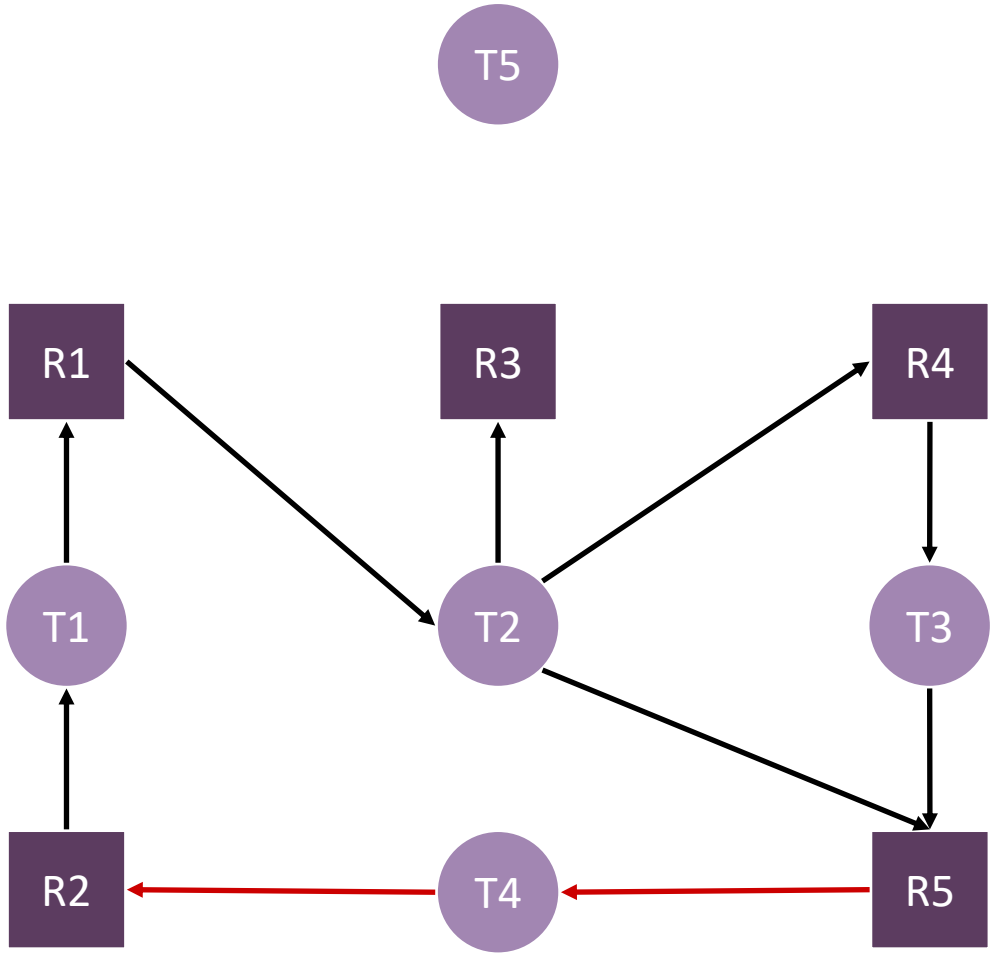
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

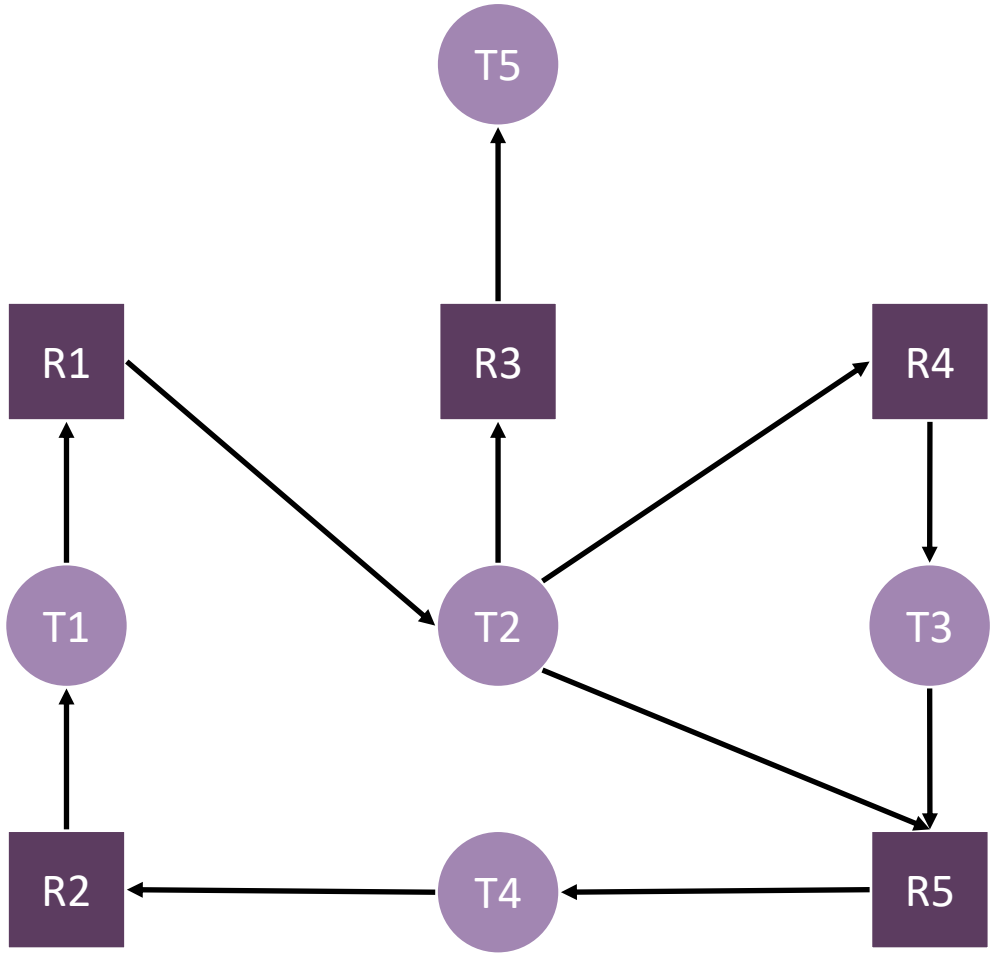
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

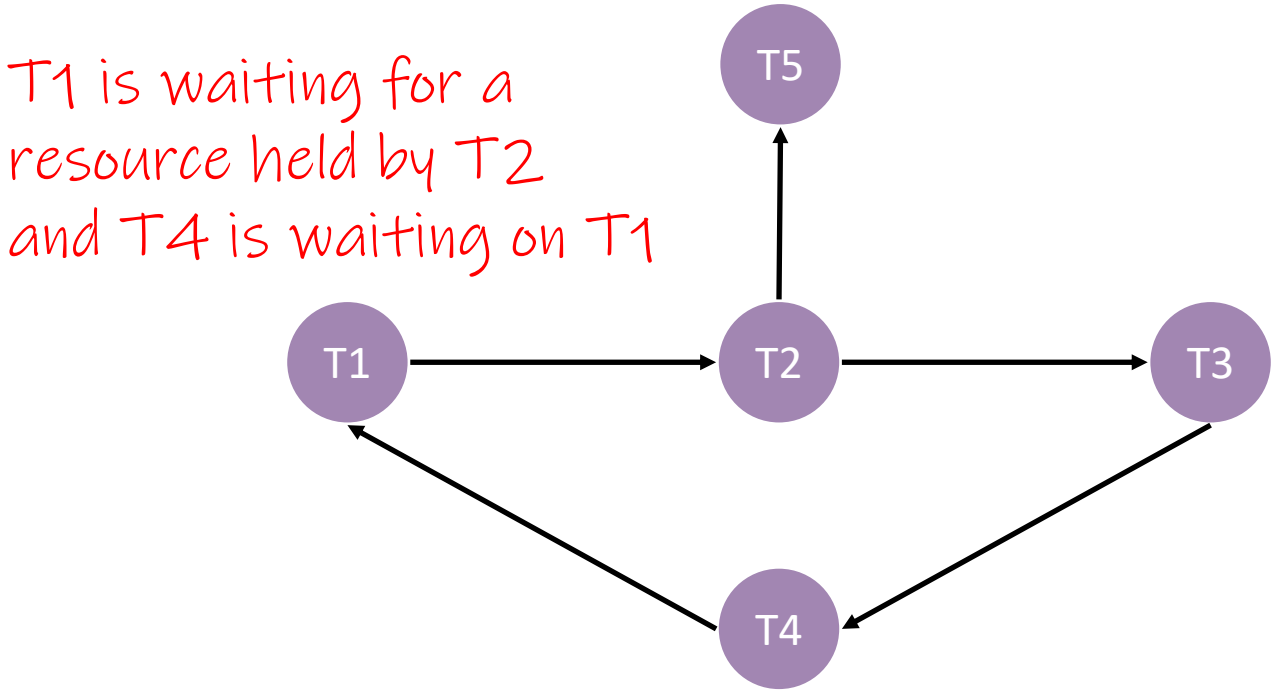
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



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