Parallel Programming

Deadlocks

Morgan Ericsson

Today

» Deadlocks

Remember channels

```go
func main() {
    ch := make(chan int)
    if res, ok := <-ch; ok {
        fmt.Println("Got", res)
    }
}
```

Will not work

- “Fatal error: all goroutines are asleep - deadlock!”
- We are blocked, waiting for a value on the channel
- Nobody is writing to the channel, so a value will never appear
- Deadlock

System model

- A system consists of a fine number of resources
  - Such as cpu cycles, files, I/O devices, ...
  - Can be single or multiple instances of a resource
- These resources are distributed among competing threads

Normal operation

- A thread can only use a resource in the following sequence:
  1. Request the resource. Will block until the resource becomes available.
  2. Use the resource
  3. Release the resource
- Just like a critical region

Deadlocks

- Suppose:
  - A thread holds *Resource A* and requests *Resource B*
  - and another holds *Resource B* and requests *Resource A*
- Both will remain blocked until the other finishes
  - So, deadlocked (blocked forever)

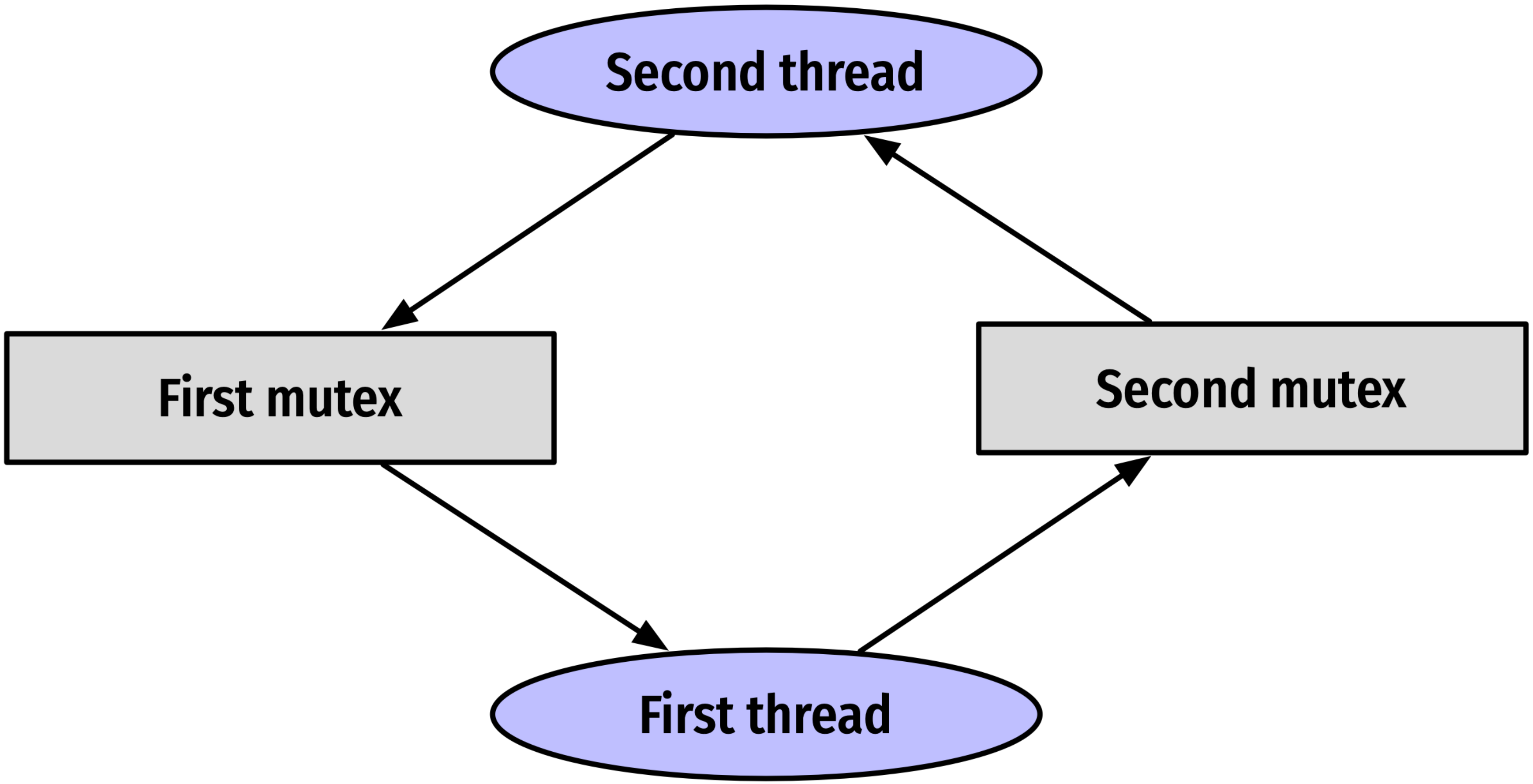
Deadlocks

A set of processes is *deadlocked* if each process in the set is *waiting for an event* that only *another process* in the set *can cause*

Example

```go
var lock1, lock2 sync.Mutex

go func() {
    lock1.Lock()
    lock2.Lock()
    // Work
    lock2.Unlock()
    lock1.Unlock()
}()

go func() {
    lock2.Lock()
    lock1.Lock()
    // Work
    lock1.Unlock()
    lock2.Unlock()
}()
```

Graph



Required conditions for deadlock

1. *Mutual exclusion*
   - » Each resource assigned to one process or is available
2. *Hold and wait*
   - » Process holding resources can request additional
3. *No preemption*
   - » Previously granted resources cannot be forcibly taken away
4. *Circular wait*
   - » Must be a circular chain of two or more processes
   - » Each is waiting for resources held by the next member of the chain

All four must hold for a deadlock to occur

No preemption

- Two types of resources:
  - *Preemptable*, can be taken away with no ill effects
  - *Nonpreemptable*, will cause the process to fail if taken away
- Many are nonpreemptable, especially if some work has already taken place
  - Remember database transactions, rollback vs commit

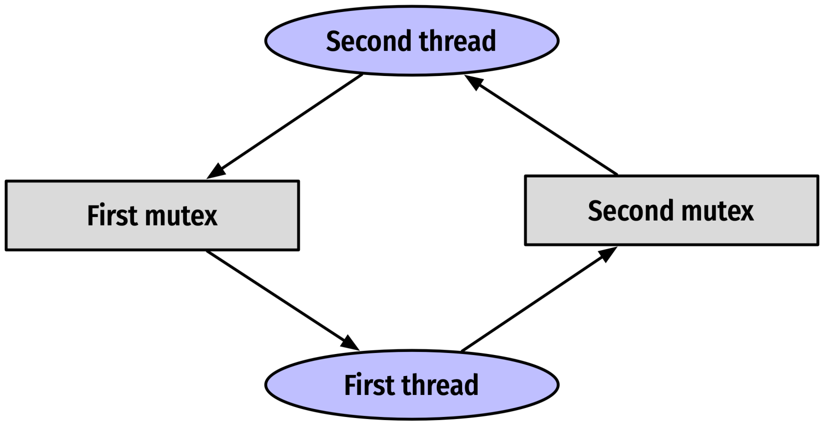
Hold and wait

- It is often possible to check or set a timeout when waiting for a lock
  - E.g. `lock.acquire(timeout=10)` (python)
- Can help prevent deadlocks, but may result in another liveness problem
  - Threads try for the second lock, fail, and release the first lock
  - They then retry, and again fail at the second lock and repeat

Example

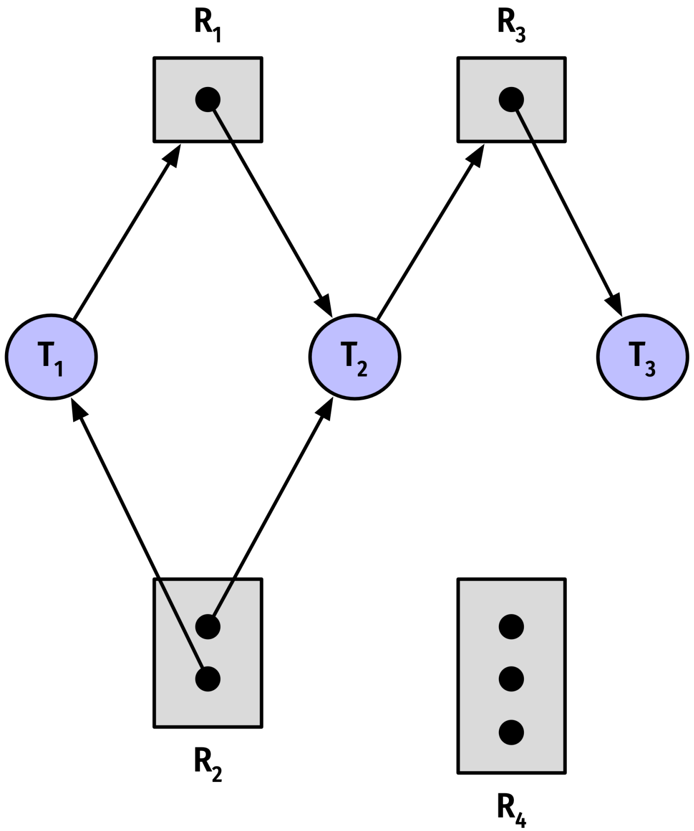
```
var lock1, lock2 sync.Mutex

for {
    for {
        lock1.Lock()
        if ok := lock2.tryLock(); !ok {
            lock1.Unlock()
        }
        // runtime.Gosched()
        // time.Sleep(xxx)
    }
    // Do Work
    lock2.Unlock()
    lock1.Unlock()
}
```

Resource-Allocation Graphs



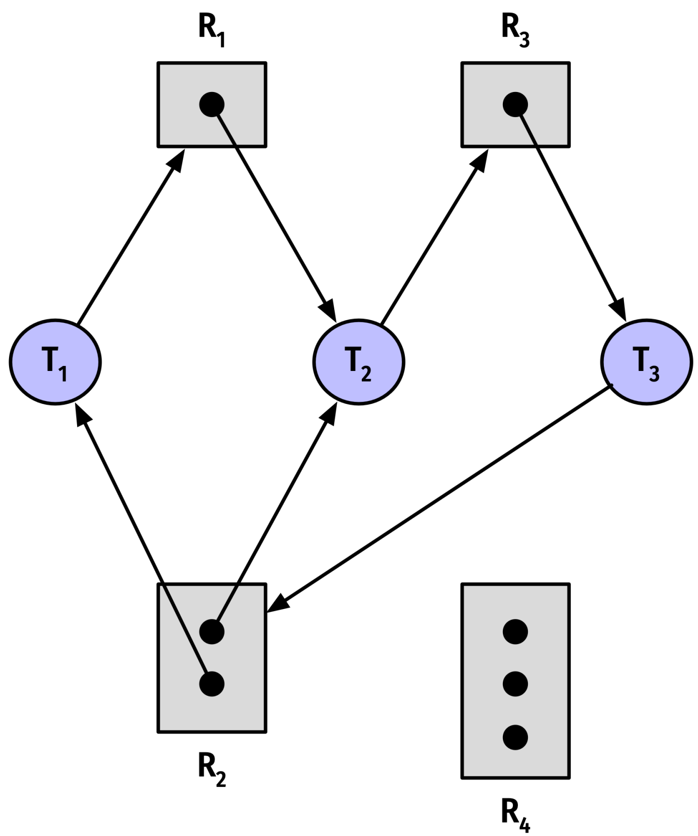
» Vertices are of two types:
  » T (threads)
  » (resource types)
» A directed edge is called a request edge
» A directed edge is called an assignment edge

Example 1



- holds one instance and one instance of .
- Requests one instance of .
- Is the system deadlocked?
  - Why/why not?

Example 2

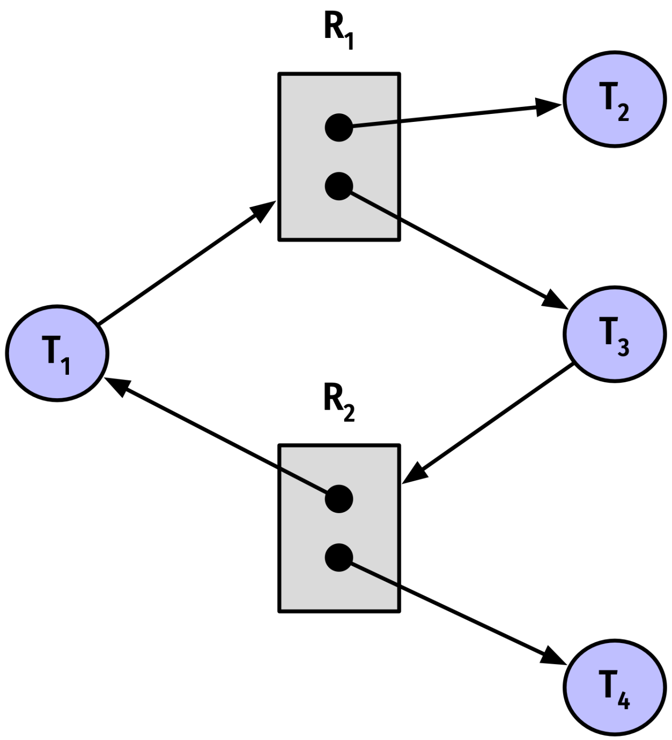


- Is the system deadlocked now?
  - Why/why not?

Several important ideas

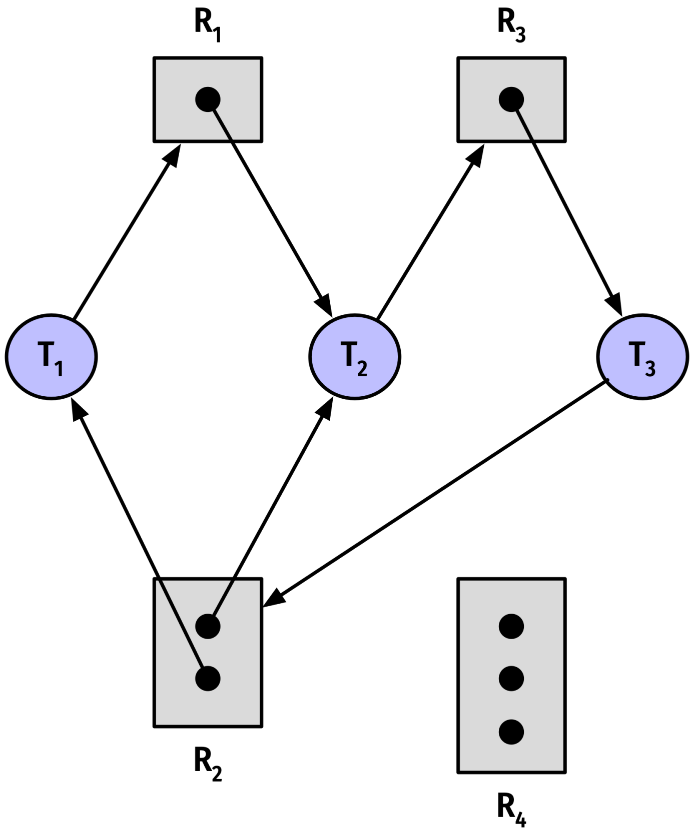
- Cycles in the resource allocation graph indicate that there may be a deadlock
  - There is one if there is only one instance per resource type
  - There may be one if there are multiple instances per resource type
- Each thread involved in the cycle is deadlocked

Example 3



- Is the system deadlocked?
  - Why/why not?

Back to example 2



» Which threads are deadlocked?

Several important ideas

- We can detect whether the system might be deadlocked
- We can identify which threads are deadlocked
- So, we should be able to avoid, prevent, or resolve deadlocks?

What can we do about deadlocks?

- Allow the system to enter a deadlocked state, *detect* it, and *recover*
  - Common approach in database systems
  - Abort and rollback a transaction if deadlocked
- Use a protocol to *prevent* or *avoid* deadlocks, ensuring that the system will never enter a deadlocked state
- *Ignore* the problem and pretend that deadlocks never occur
  - Push the problem to developers
  - Common approach in most operating systems

Deadlock prevention

» Remember, there are four necessary conditions for deadlocks to occur

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

» If at least one of these cannot hold, we can prevent deadlocks from occurring

Mutual exclusion

- Holds if at least one resource type is nonsharable
- There are several types of sharable resources
  - E.g., read-only files and data structures
- But non-sharable resources were the reason we introduced synchronization
  - So, we must assume that they must exist
  - Hence, removing mutual exclusion is generally not possible

Hold and wait

- We have already seen that it is possible to remove hold and wait
  - Use timeouts when requesting resources
- The drawback is that a thread must either
  - Allocate all resources “up front”
  - Release allocated resources before requesting new
- Neither is great
  - Low utilization, risk of starvation

No preemption

- If a thread holding resources requests another and has to wait, preempt all resources it is holding
- Or, when a thread is requesting a resource that is not available, check if it can be preempted
  - E.g., if the thread holding it is waiting for something

No preemption

- Works well if resources can be preempted easily
  - E.g., CPU registers, database transactions
- Not so well for others, which is generally where deadlocks appear
  - E.g., mutex and semaphores

Circular wait

- We can break circular wait by enforcing an order in which resources must be allocated
  - Generally, by assigning a total order to resources, e.g., resources 1, 2, 3, etc.,
  - Resources can only be allocated in strictly increasing order, so not Resource 3, then Resource 1
  - If several of the same types should be allocated, they need to be allocated in a single request

Circular wait

The main problem is establishing the total order - Simple if few resources, problematic if hundreds - Can use hashcodes or ids, e.g., `id(mylock)` - Up to developers to maintain

Deadlock avoidance

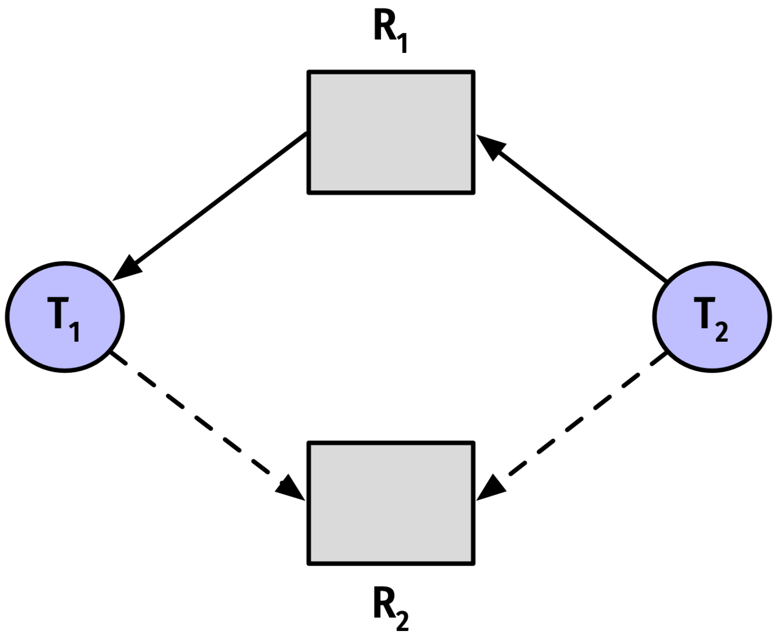
- Evaluate each request and decide whether is will keep the system in a safe (from deadlocks) state or not
  - Only grant requests that keep the system in a safe state

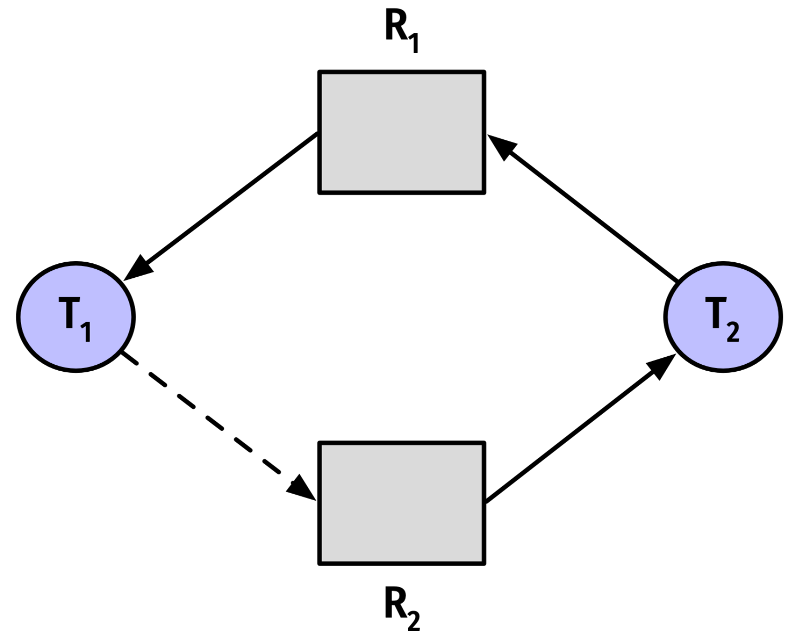
Deadlock avoidance

- Requires additional information about the threads
  - Which resource it will allocate, and ...
  - ... the maximum number of each
- System maintains the allocation state
- Ensures that circular wait cannot occur

Safe state

Example





Banker's algorithm

» Algorithm to check whether a request is safe or not

» Requires several data structures. Assume threads and resource types

  » , the number of available resources for each type

  » , the maximum a thread needs of each resource type

  » , the current allocation to each thread

  » ,

Example

	Allocation			Max			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			

Banker’s algorithm

» Check whether the system is in a safe state or not:

1. Set . Set for each thread .
2. Find an index such that and . If no such index exist, go to 4.
3. Set and . Go to 2.
4. If is true for all , the system is in a safe state.

Banker's algorithm

» When a resource is requested:

1. If continue to 2, else raise an error.
2. If continue to 3, else wait.
3. Pretend to allocate the requested resources and check whether the resulting state is safe. If unsafe, restore and have the thread wait.

    »

    »

    »

Example

	Allocation			Max			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			

	Need		
	A	*B*	*C*
T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

Safe state? Yes, T_1, T_3, T_4, T_2, T_0 is a safe sequence.

Example

	Allocation			Max			Available			Need		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	7	5	3	3	3	2	7	4	3
T_1	2	0	0	3	2	2				1	2	2
T_2	3	0	2	9	0	2				6	0	0
T_3	2	1	1	2	2	2				0	1	1
T_4	0	0	2	4	3	3				4	3	1

What if T_1 requests 1 *A* and 2 *C*? $request_1 = (1, 0, 2)$

Ok, since T_1, T_3, T_4, T_0, T_2 is a safe sequence.

Deadlock detection

1. Check whether a deadlock has occurred
2. If so, recover from the deadlock

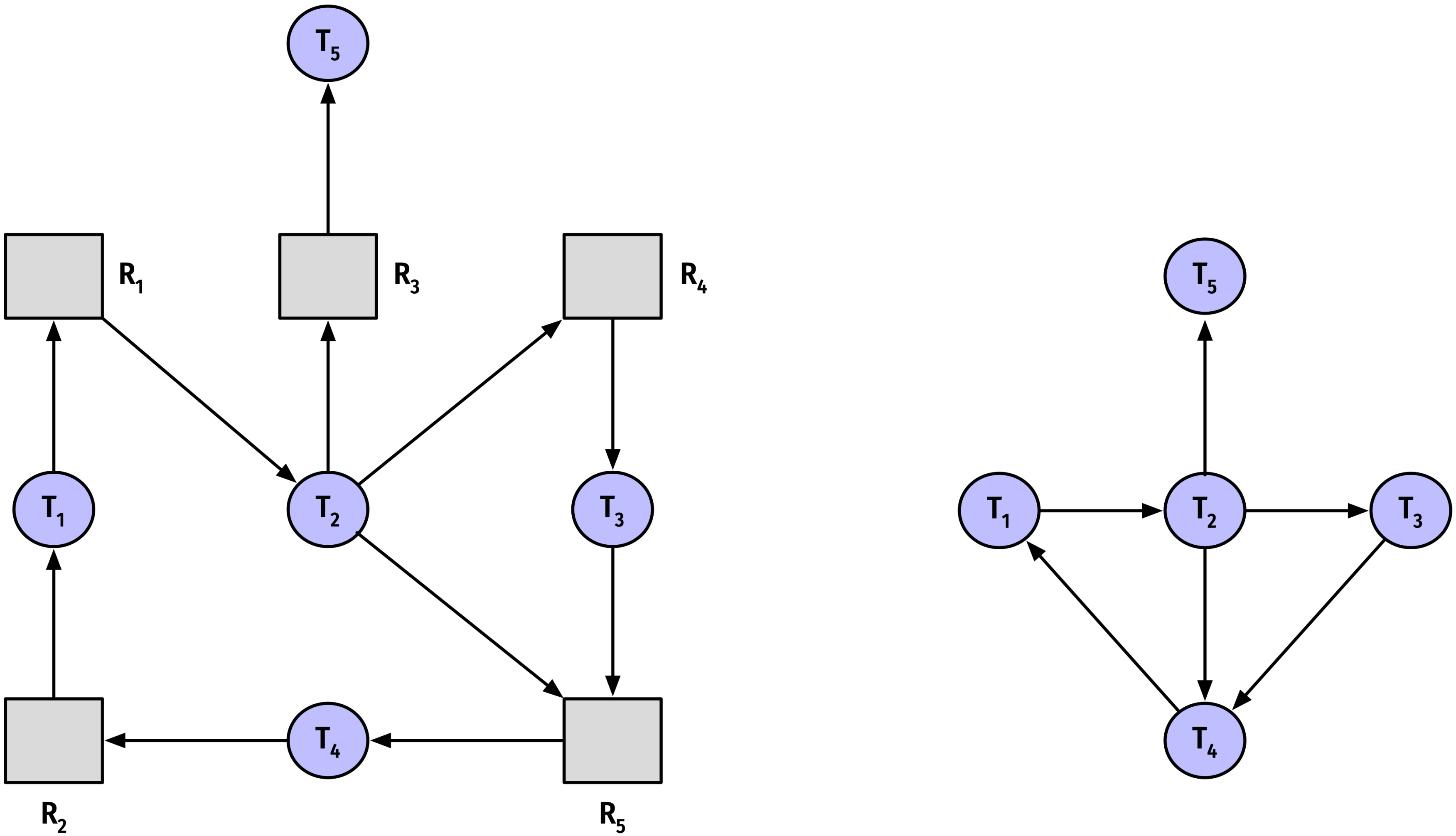
Wait-for graph

- If there is only a single instance of each resource type, a wait-for graph can be used
- A wait-for graph is a resource-allocation graph where the resource vertices have been removed
  - To only show threads waiting for each other

Wait-for graph

- If there is a cycle in the wait-for graph, the system is deadlocked
- The system maintains the graph and periodically checks it for cycles
  - Cycle checking has quadratic complexity.

Example



Several instances of a resource?

» We can use something similar to Banker's algorithm

1. Set . For each thread , set if the thread's allocation is 0, else false.
2. Find an index such that and . If no such exists, go to 4.
3. Set and . Go to step 2.
4. If is false for some (), the system is deadlocked and thread is deadlocked.

Example

	Allocation			Request			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

Is the system deadlocked?

No, sequence T_0, T_2, T_3, T_1, T_4 allows all to finish.

Example

	Allocation			Request			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	*1*			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

Is the system deadlocked now? Yes!

When should the algorithm run

- Both detection algorithms are quite expensive and just generate overhead
  - If no deadlocks are detected, the execution time is just wasted
- When to run depends on:
  - How often is a deadlock likely to occur?
  - How many threads will be affected by it when it occurs?

When should the algorithm run

- If deadlocks occur frequently, run the algorithm frequently
  - Fast detection limits the number of threads involved and the cost of the deadlock
- High overhead running often or even at every request
  - Perhaps run only at set intervals (e.g., every hour) or when CPU utilization drops below a specific threshold

When should the algorithm run

» Note that if the algorithm is invoked at arbitrary times, it is not possible to determine which threads "caused" the deadlock

Recovery from deadlock

- » We can recover by terminating threads/processes or preempting resources
- » Terminating processes
  - » Abort all deadlocked threads/processes
  - » Abort one thread/process at a time until the cycle is broken

Recovery from deadlock

- Both methods are potentially expensive
  - Long-running threads/processes might be aborted, and results are lost/need to be recomputed
  - One at a time causes overhead since the algorithm must be run after each termination
- And problematic
  - What if the thread/process is modifying a shared resource, e.g., a file? How can we guarantee integrity?

Recovery from deadlock

- Resource preemption
  - What should be done with the thread/process if we preempt its resources? Can it be rolled back to a safe state? What if not? Terminate?
  - How do we ensure that there is no starvation? How can we guarantee that resources are not always preempted from the same thread/process?
    - If we use heuristics, it is likely that the same thread/process will always be chosen and never be able to complete its task

Do nothing?

- Might be a reasonable choice, given the cost and the potentially severe issues of avoidance, detection, and prevention.
- Pushes the problem to the next “tier,” e.g., from operating system to developers or developers to end users

Do nothing?

- Depends on how common and how expensive to deal with
  - Reasonable to detect and recover in databases
  - Reasonable to push to developers in operating systems
- But, good idea to take measures as a developer, e.g., by attacking circular wait

Next time

- We now know the basics of
  - Threads and processes
  - Shared memory
  - Synchronisation
- And algorithms!
- So, time to move on to concurrent algorithms