Fun with Concurrency ☺

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

Administrivia

- PennOS
  - You have the first milestone, which should have been done last week
  - Everyone should have already contacted their group, and should get started working on it.
  - Milestone 1 is due next week
    - Between Tuesday the 9th and Friday the 12th
    - Need to meet with TA again to show significant progress
    - Have a plan (a REAL plan) for how to complete the rest
  - Full Thing due ~April 22nd

Administrivia

- Check-in was due before today's lecture
  - Another one will be released this week, due sometime next week

- Exam grades posted
  - Remember the Clobber Policy. Many people benefit from this policy in my courses
  - Regrade are open and will stay open till April 5th at midnight.

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Dining Philosophers**
- Deadlock Prevention
- Deadlock Handling
- Parallel Analysis

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":
  - 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

- Can this deadlock? No

- 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 Prevention**
- Deadlock Handling
- Parallel Analysis

Previously: 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 ☹

Deadlock Definition

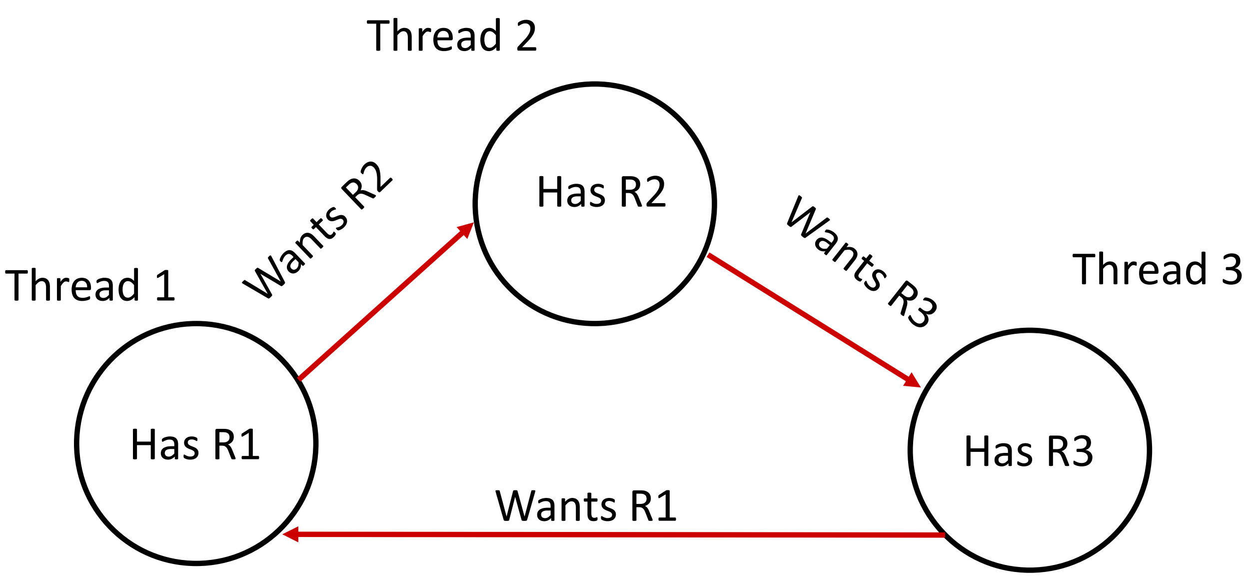
- A computer has multiple threads, finite resources, and the threads want to acquire those resources
  - Some of these resources require exclusive access
- A thread can acquire resources:
  - All at once
  - Accumulate them over time
  - If it fails to acquire a resource, it will (by default) wait until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



Poll Everywhere

pollev.com/tqm

- 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"
  - This is like one of our dining philosophers implementations
  - 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.
  - Consider dining philosophers, what if each thread was required to get the lowest numbered chopstick it wants first?

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

Deadlock Prevention Summary

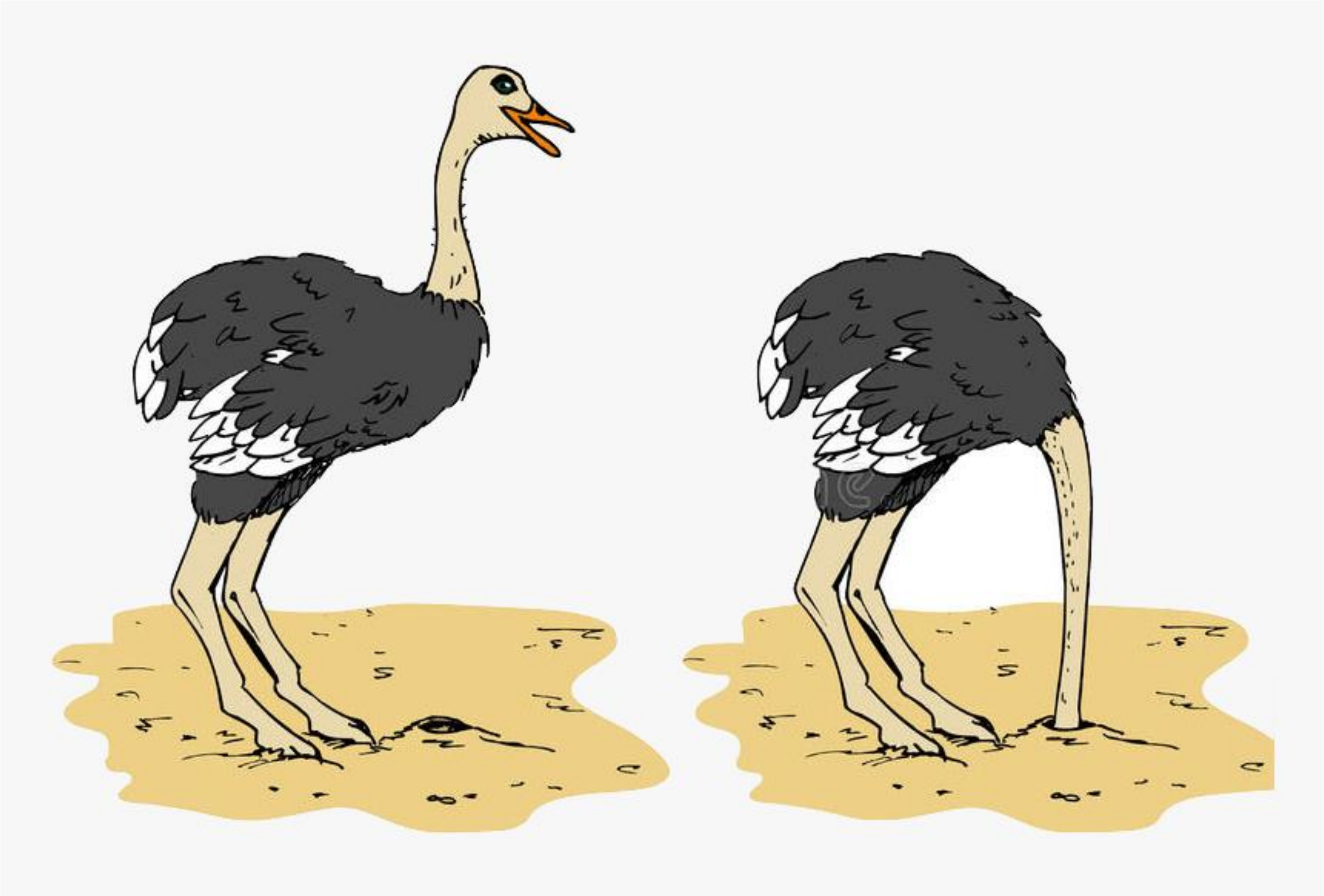
- Prevent deadlocks by removing any one of the four deadlock preconditions
- But eliminating even one of the preconditions is often hard/impossible
  - Mutual Exclusion is necessary in a lot of situations
  - Forcing a lower priority process to release resources early requires rollback of execution
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

Lecture Outline

- Dining Philosophers
- Deadlock Prevention
- **Deadlock Handling**
  - **Ostrich**
  - **Prevention**
  - **Detection**
  - **Avoidance**
- Parallel Analysis

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

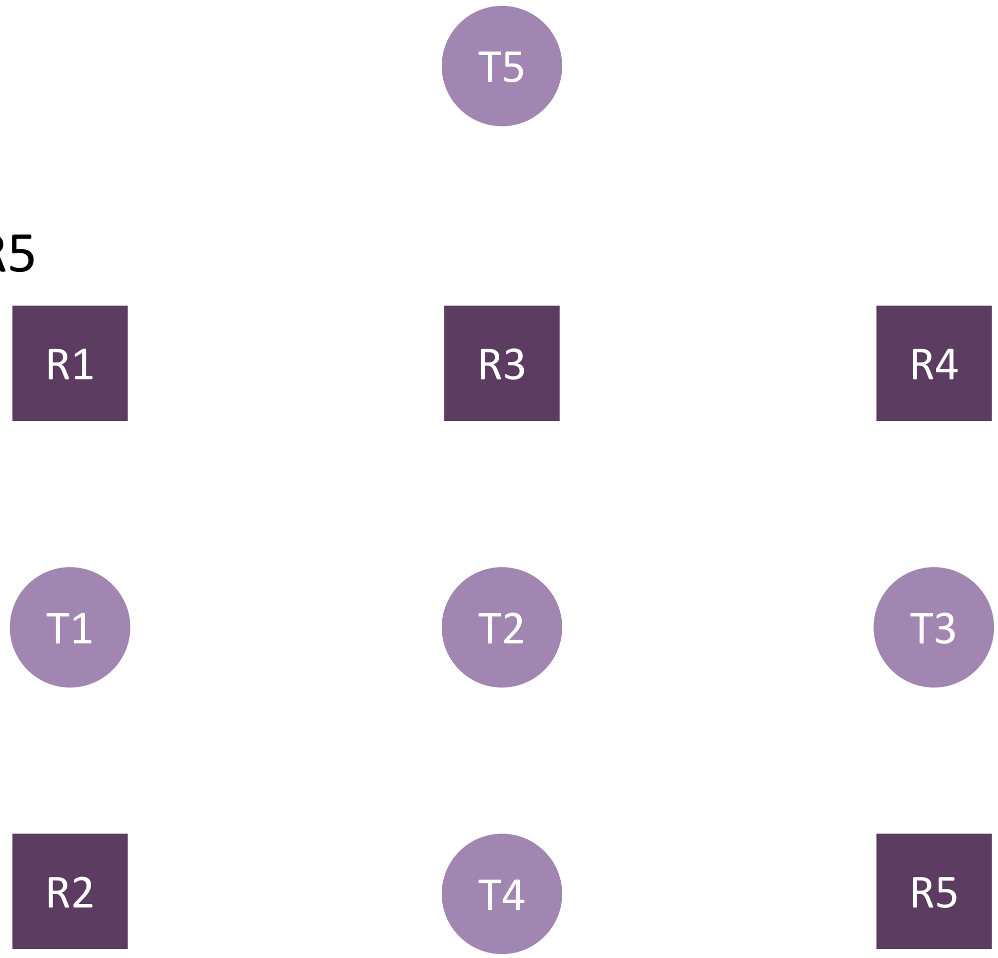
- Consider the following example with 5 threads and 5 resources that require mutual exclusion is this a deadlock?
  - Thread 1 has R2 but wants R1
  - Thread 2 has R1 but wants R3, R4 and R5
  - Thread 3 has R4 but wants R5
  - Thread 4 has R5 but wants R2
  - Thread 5 has R3

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

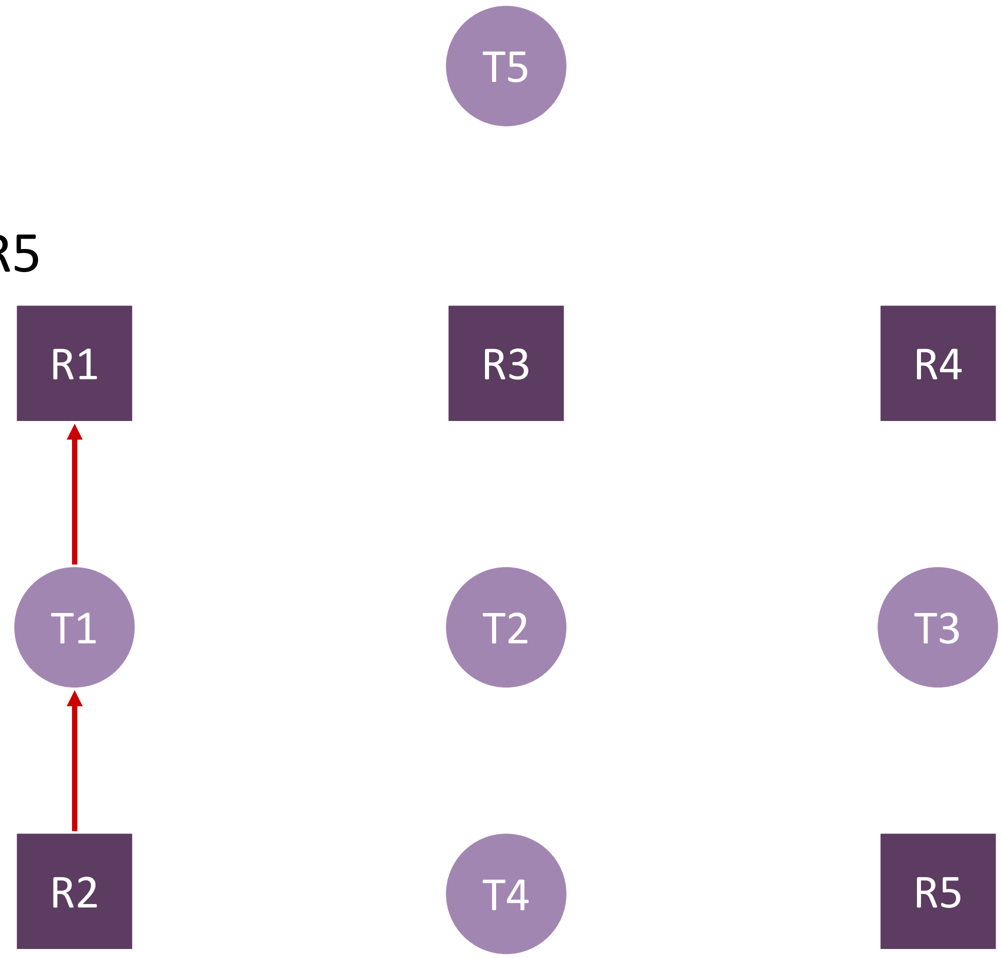
- Thread 1 has R2 but wants R1
- Thread 2 has R1 but wants R3, R4 and R5
- Thread 3 has R4 but wants R5
- Thread 4 has R5 but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

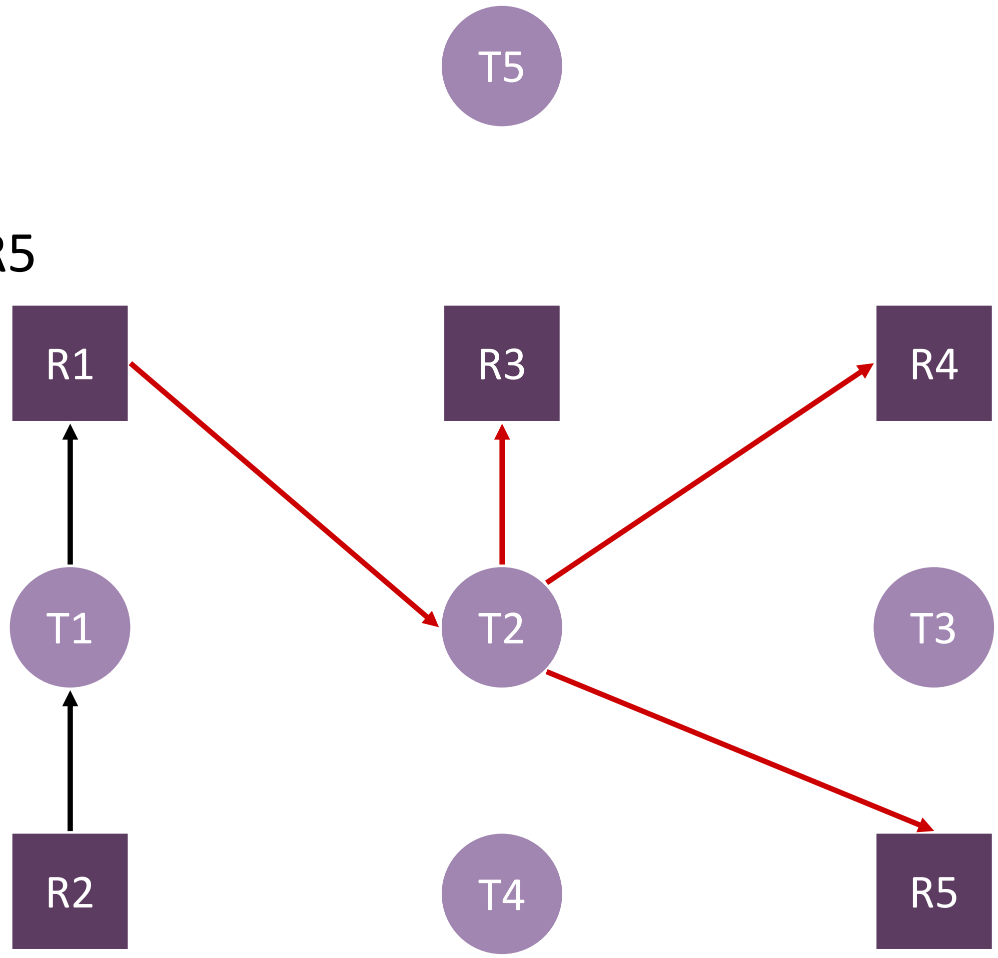
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

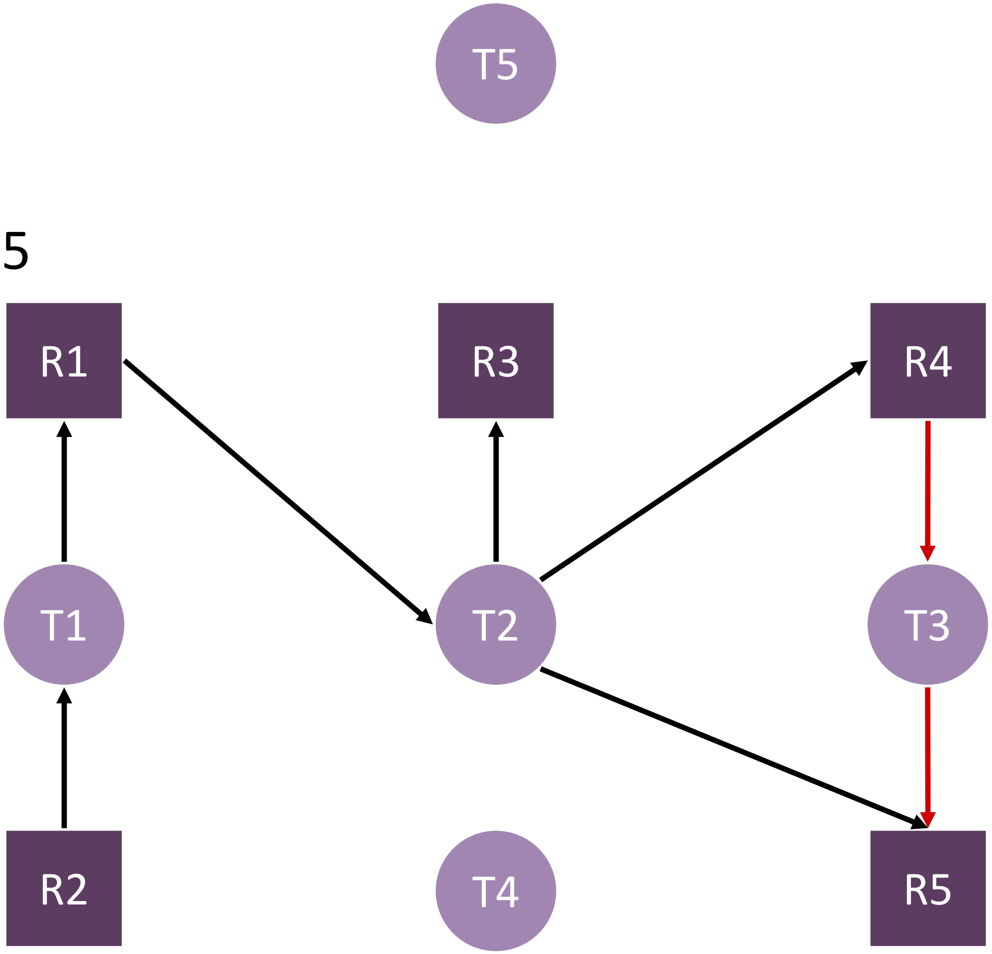
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

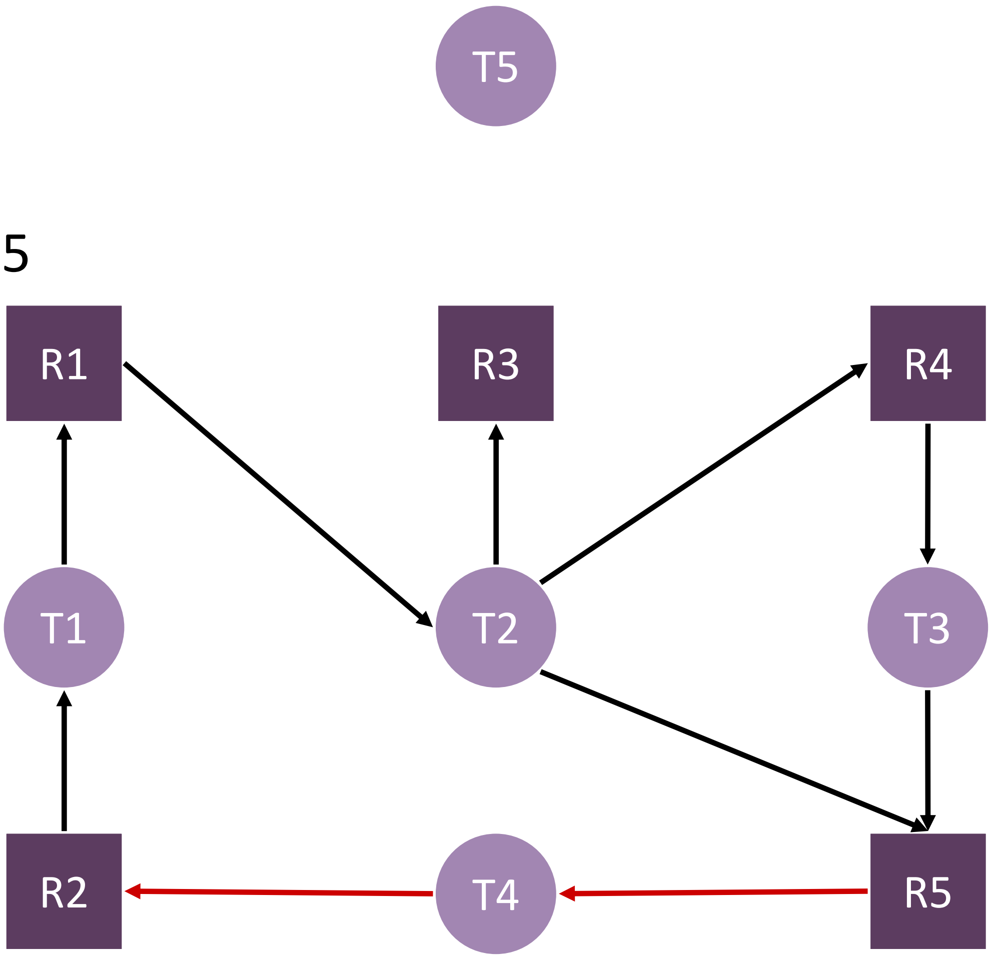
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

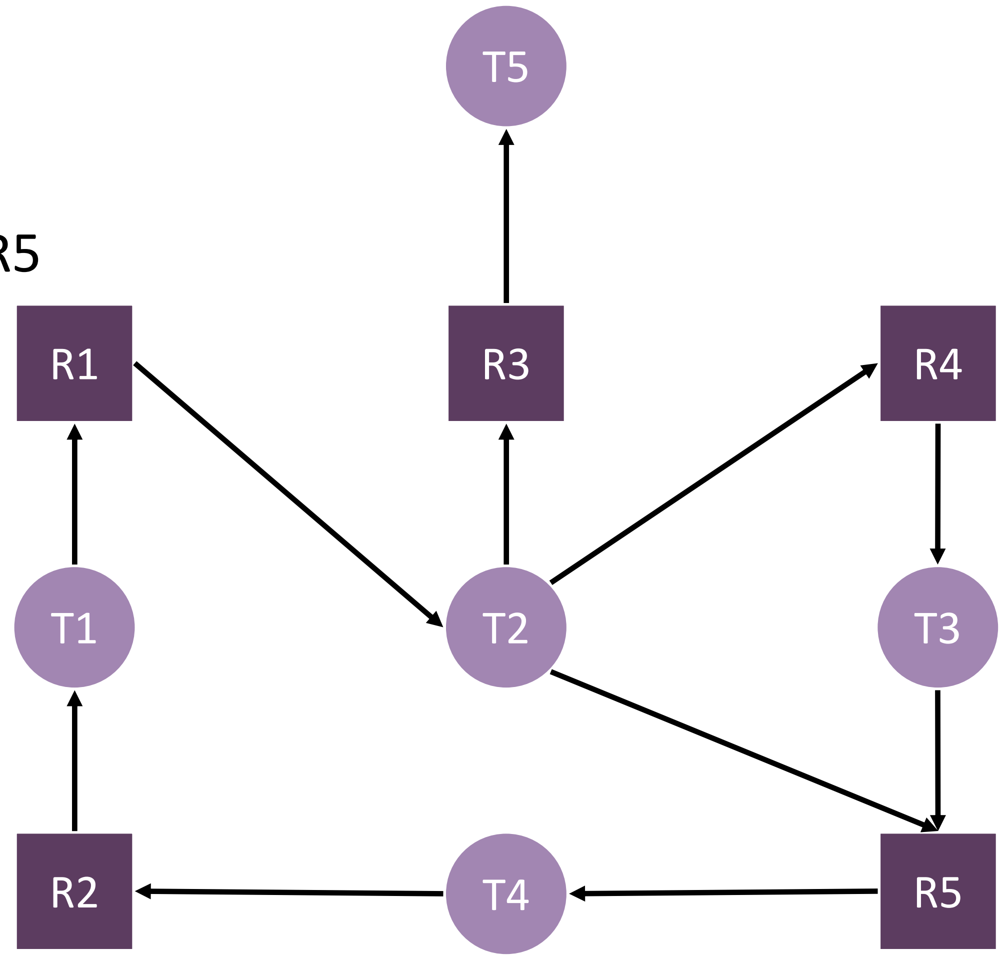
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



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

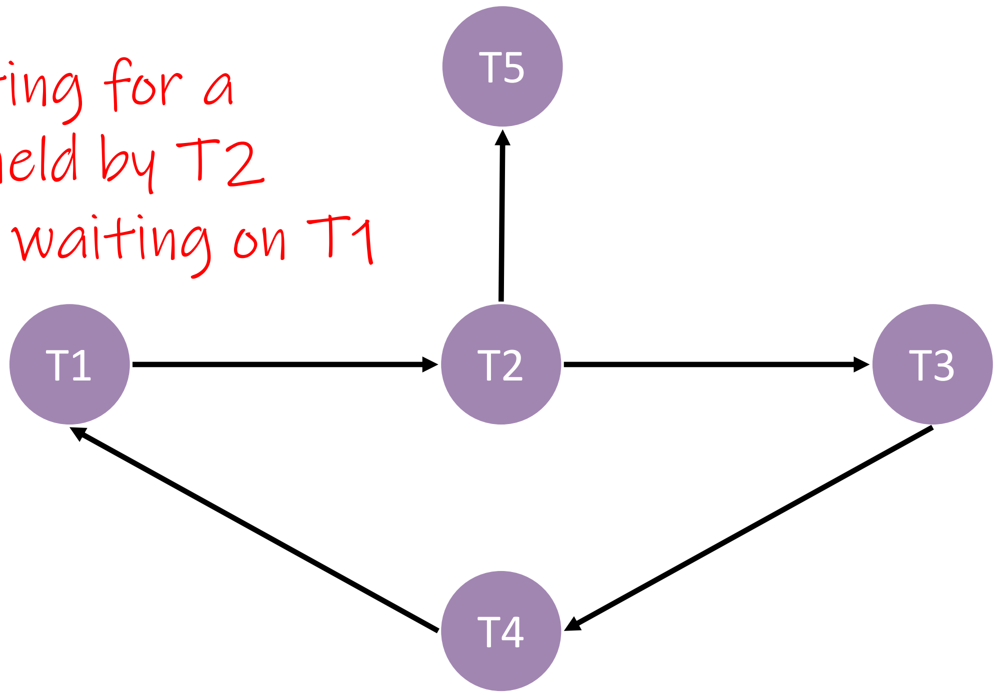


Resource Allocation Graph

Alternate graph

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

T1 is waiting for a resource held by T2 and T4 is waiting on T1



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

Lecture Outline

- Dining Philosophers
- Deadlock Prevention
- Deadlock Handling
- **Parallel Analysis**
  - **Recurrences**
  - **Amdahl's Law**

Parallel Algorithms

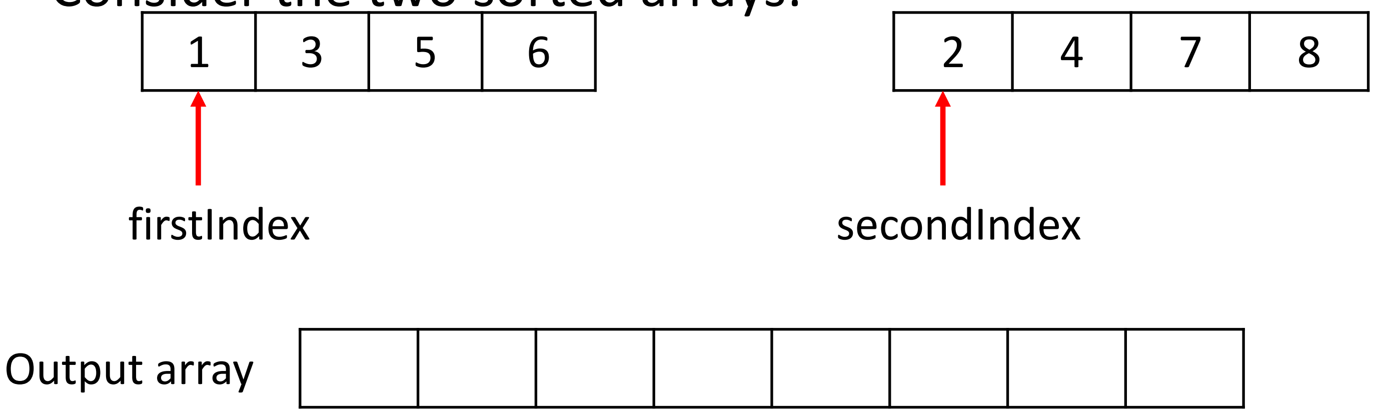
- One interesting applications of threads is for faster algorithms

- Common Example: Merge sort

Merge Sort: Core Ideas

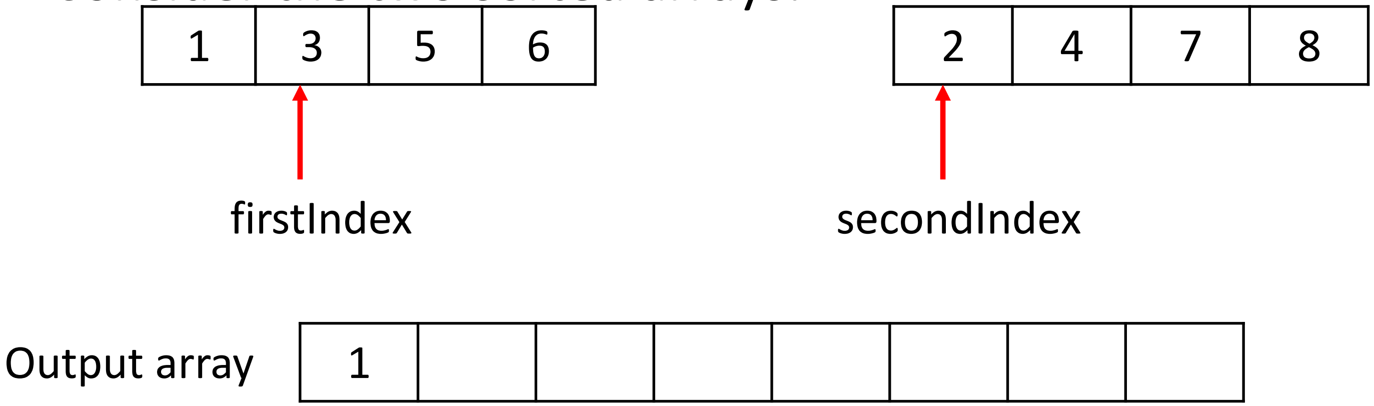
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



Merge Sort: Core Ideas

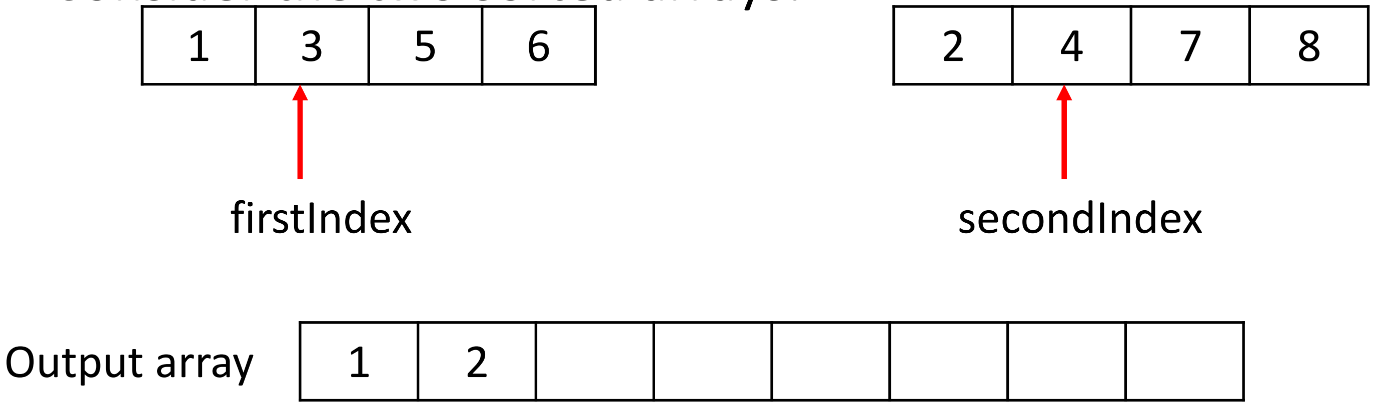
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

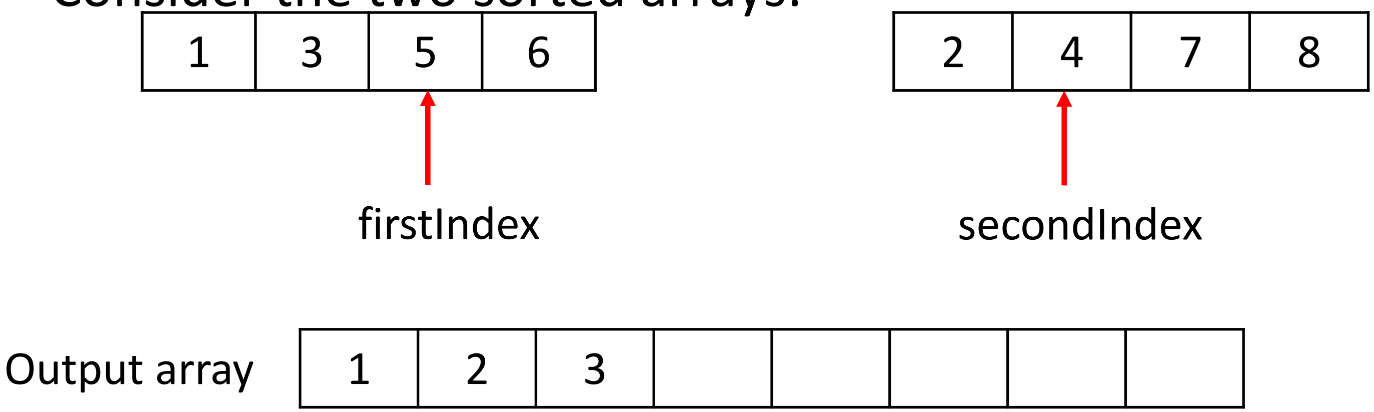
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

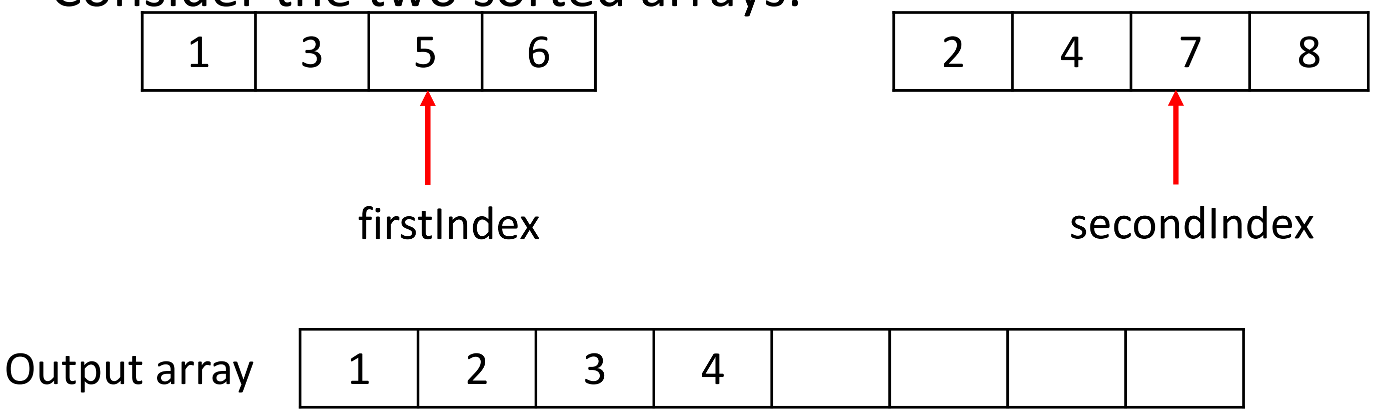
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

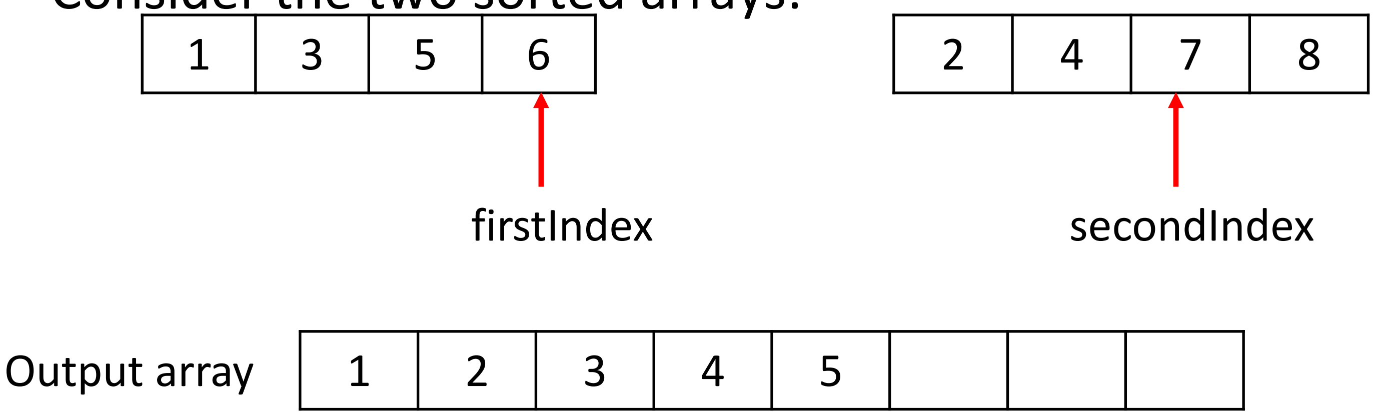
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

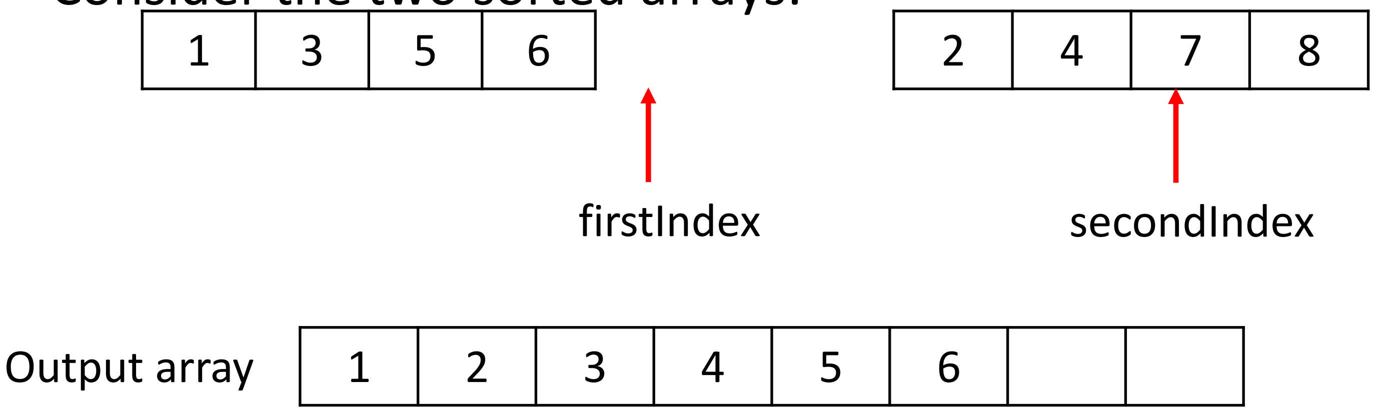
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



Merge Sort: Core Ideas

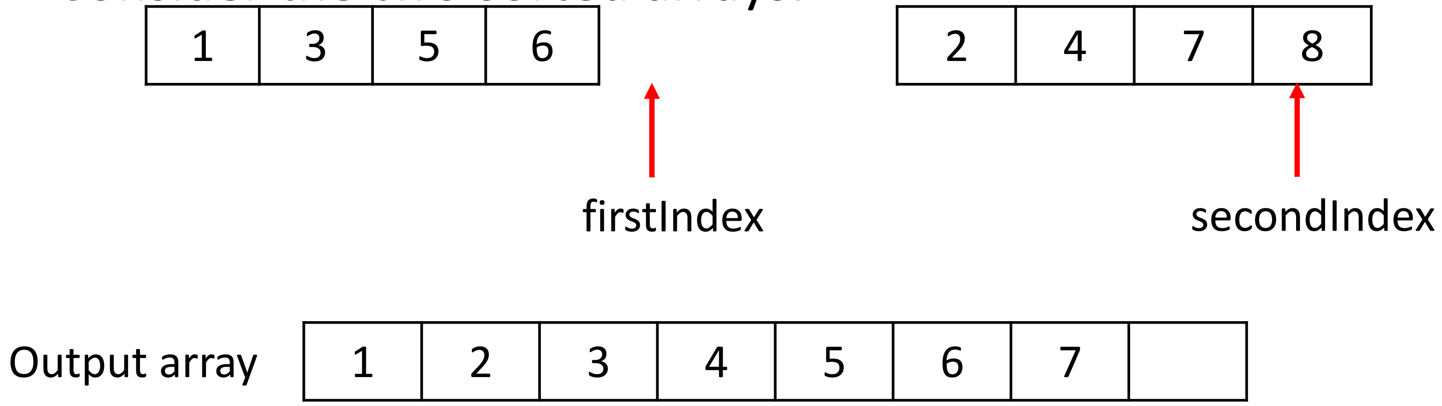
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



Merge Sort: Core Ideas

- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:

1	3	5	6

2	4	7	8

firstIndex

secondIndex

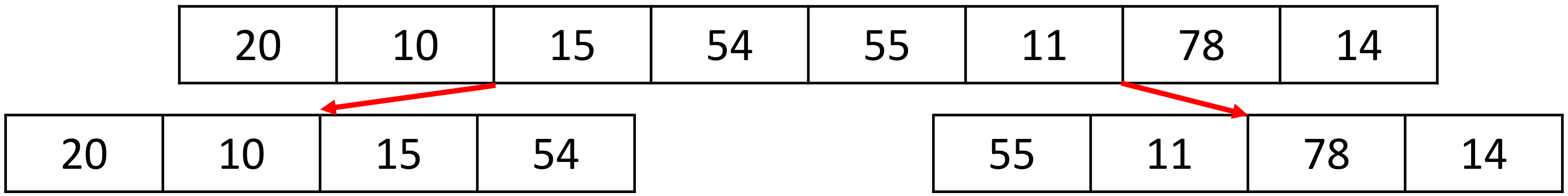
Output array

1	2	3	4	5	6	7	8

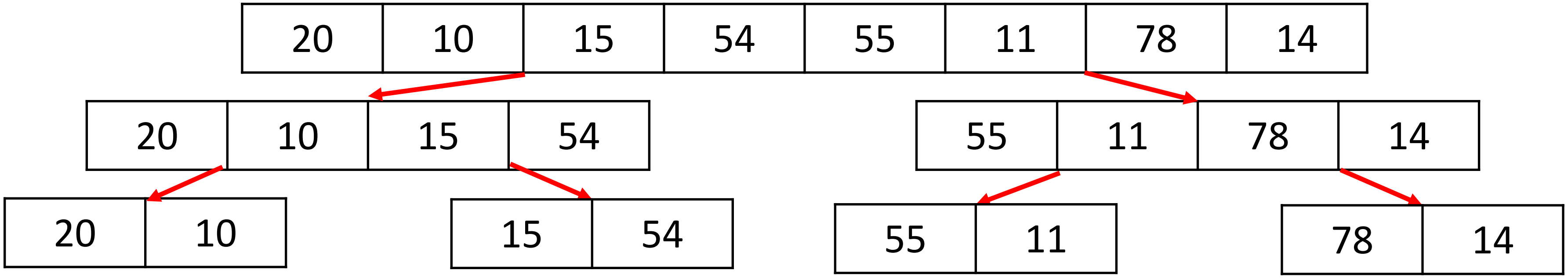
Merge Sort: High Level Example

20	10	15	54	55	11	78	14

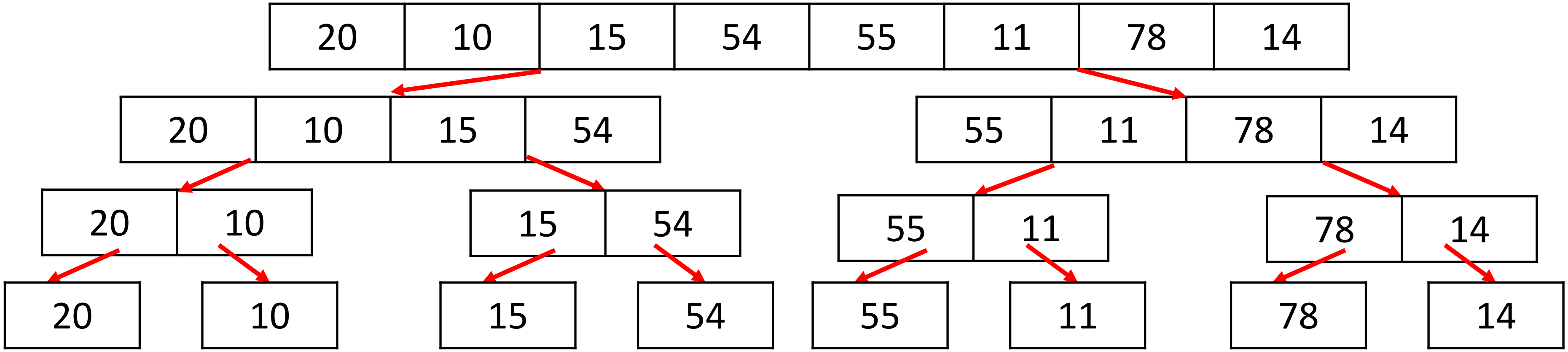
Merge Sort: High Level Example



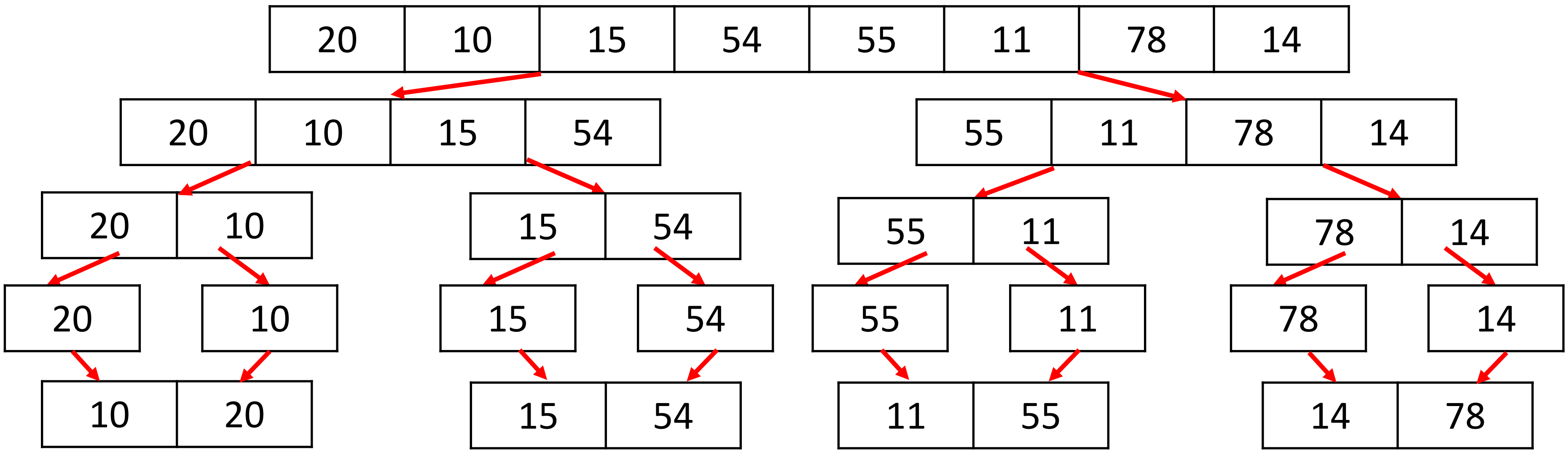
Merge Sort: High Level Example



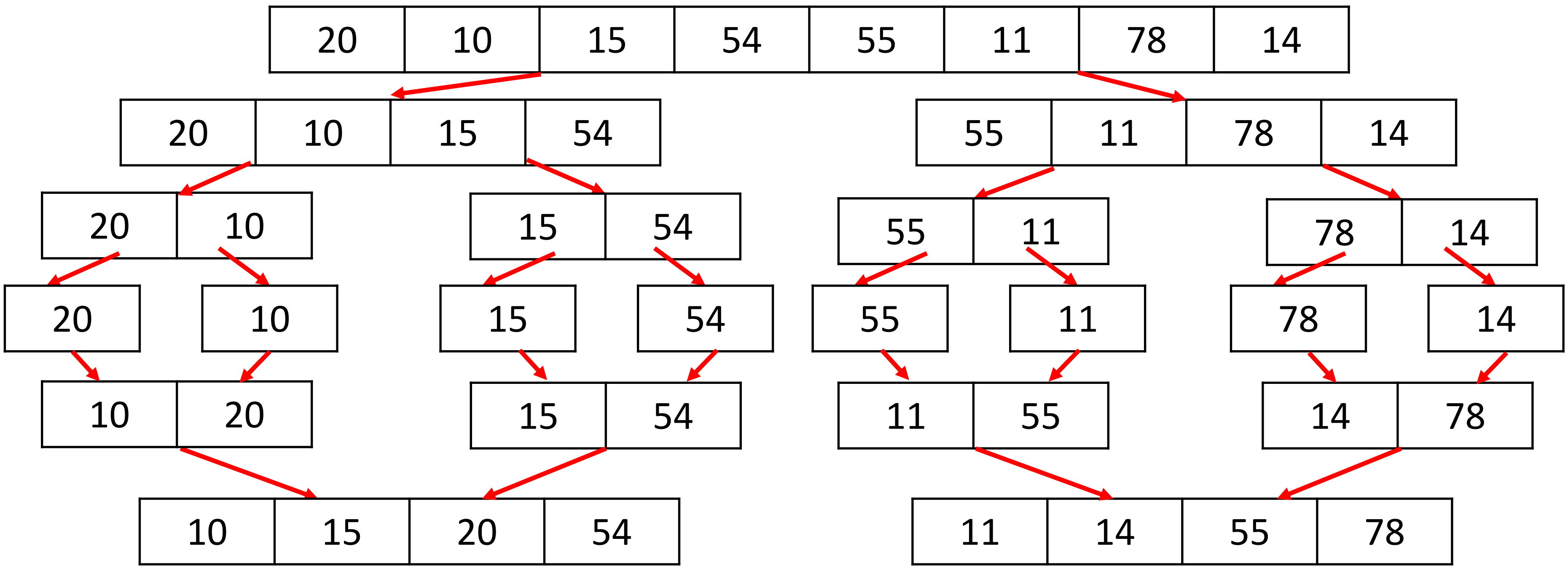
Merge Sort: High Level Example



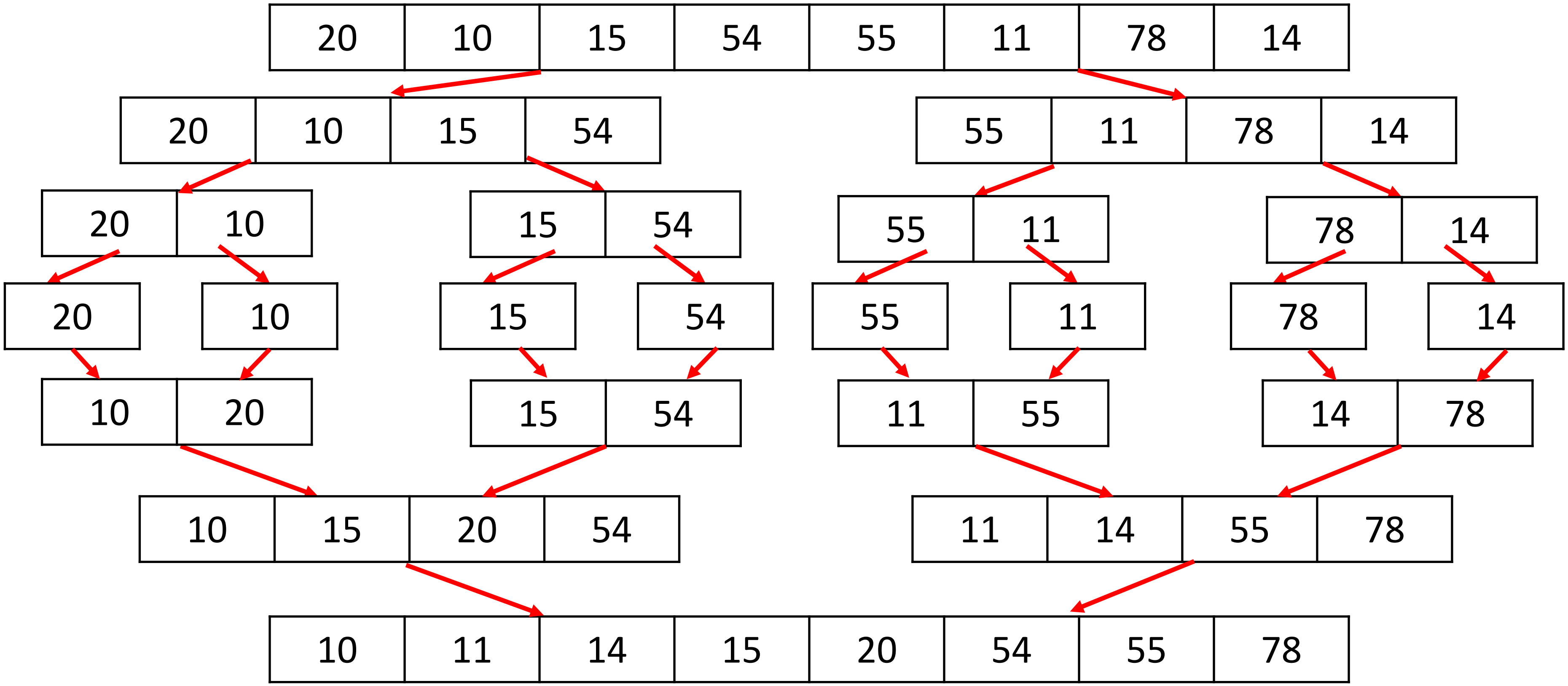
Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort Algorithmic Analysis

- Algorithmic analysis of merge sort gets us to O(n * log(n)) runtime.

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;
  merge_sort(arr, lo, mid);  // sort the bottom half
  merge_sort(arr, mid, hi);  // sort the upper half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

- We recurse $\log_2(N)$ times, each recursive “layer” does O(N) work

Merge Sort Algorithmic Analysis

- We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;

  // sort bottom half in parallel
  pthread_create(merge_sort(arr, lo, mid));
  merge_sort(arr, mid, hi);  // sort the upper half

  pthread_join(); // join the thread that did bottom half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

  - Now we are sorting both halves of the array in parallel!

Poll Everywhere

pollev.com/tqm

- We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {
  // lo high start at 0 and arr.length respectively
  int mid = (lo + hi) / 2;

  // sort bottom half in parallel
  pthread_create(merge_sort(arr, lo, mid));
  merge_sort(arr, mid, hi);  // sort the upper half

  pthread_join(); // join the thread that did bottom half

  // combine the upper and lower half into one sorted
  // array containing all eles
  merge(arr[lo : mid], arr[mid : hi]);
}
```

  - Now we are sorting both halves of the array in parallel!
  - **How long does this take to run?**
  - **How much work is being done?**

Parallel Algos:

Will not test you on this

- We can define **T(n)** to be the running time of our algorithm

- We can split up our work between two parts, the part done sequentially, and the part done in parallel
  - T(n) = sequential_part + parallel_part
  - T(n) = O(n) *merging* + T(n/2) *sort half the array*
    - This is a recursive definition

- If we start recurring…
  - T(n) = O(n) + O(n/2) + T(n/4)
  - T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)

Parallel Algos:

Will not test you on this

- If we start recurring…
  - T(n) = O(n) + O(n/2) + T(n/4)
  - T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)
  - …
  - Eventually we stop, there is a limit to the length of the array. And we can say an array of size 1 is already sorted, so T(1) = O(1)
- This approximates to T(n) = ~2 * O(n) = O(n)
  - This parallel merge sort is O(n), but there are further optimizations that can be done to reach ~O(log(n))
- There is a lot more to parallel algo analysis than just this, I am just giving you a sneak peek

Amdahl's Law

- For most algorithms, there are parts that parallelize well and parts that don't. This causes adding threads to have diminishing returns
  - (even ignoring the overhead costs of creating & scheduling threads)

- Consider we have some parallel algorithm $T_1 = 1$
  - The 1 subscript indicates this is run on 1 thread
  - we define the work for the entire algorithm as 1

- We define S as being the part that can be parallelized
  - $T_1 = S + (1 - S)$ *// (1-S) is the sequential part*

Amdahl's Law

- For running on one thread:
  - $T_1 = (1 - S) + S$

- If we have P threads and perfect linear speedup on the parallelizable part, we get
  - $T_P = (1\text{-}S) + \frac{S}{P}$

- Speed up multiplier for P threads from sequential is:
  - $\frac{T_1}{T_P} = \frac{1}{1-S+\frac{S}{P}}$

Amdahl's Law

- Let’s say that we have 100000 threads (P = 100000) and our algorithm is only 2/3 parallel? (s = 0.6666..)
  - $\frac{T_1}{T_p} = \frac{1}{1-0.6666+\frac{0.6666}{100000}} = 2.9999\ times\ faster\ than\ sequential$

- What if it is 90% parallel? (S = 0.9):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.9+\frac{0.9}{100000}} = 9.99\ times\ faster\ than\ sequential$

- What if it is 99% parallel? (S = 0.99):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.99+\frac{0.99}{100000}} = 99.99\ times\ faster\ than\ sequential$

Limitation: Hardware Threads

- These algorithms are limited by hardware.
- Number of Hardware Threads: The number of threads can genuinely run in parallel on hardware

- We may be able to create a huge number of threads, but only run a few (e.g. 4) in parallel at a time.

- Can see this information in with **`lscpu`** in bash
  - A computer can have some number of CPU sockets
  - Each CPU can have one or more cores
  - Each Core can run 1 or more threads