Concurrency Wrap-up
Deadlock Handling

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- Full PennOS is due Wed Nov 29
  - You will schedule a time to meet with your TA to demonstrate your working code
  - Can submit via gradescope now
  - Reach out to TA’s to schedule PennOS Demo
- Check-in due before Lecture next week
- Recitation after class is open OH

Poll Everywhere

pollev.com/tqm

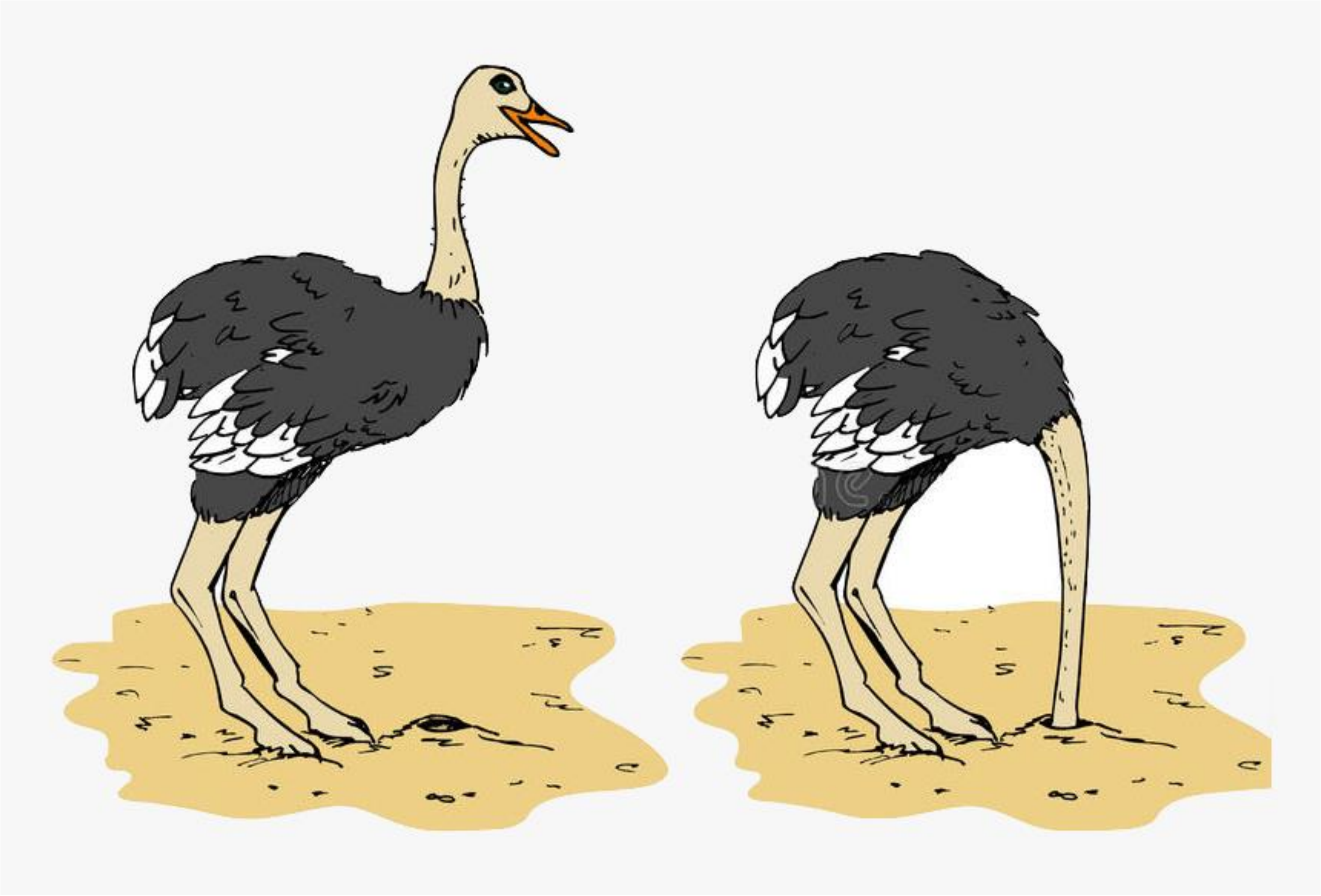
- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Deadlock Handling (start)**
  - **Ostrich**
  - **Prevention**
  - **Detection**
  - **Avoidance**
- Parallel Analysis
  - Recurrences
  - Amdahl's Law

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

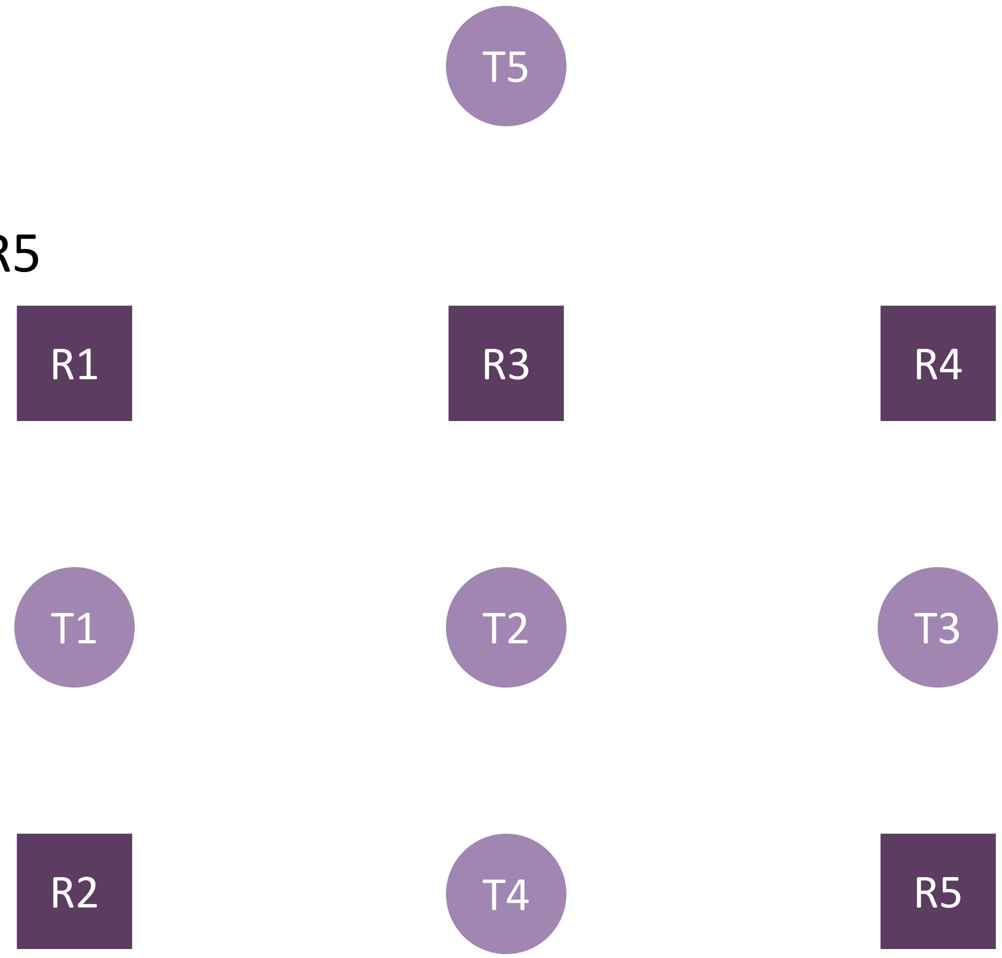
- Consider the following example with 5 threads and 5 resources that require mutual exclusion is this a deadlock?
  - Thread 1 has R2 but wants R1
  - Thread 2 has R1 but wants R3, R4 and R5
  - Thread 3 has R4 but wants R5
  - Thread 4 has R5 but wants R2
  - Thread 5 has R3

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

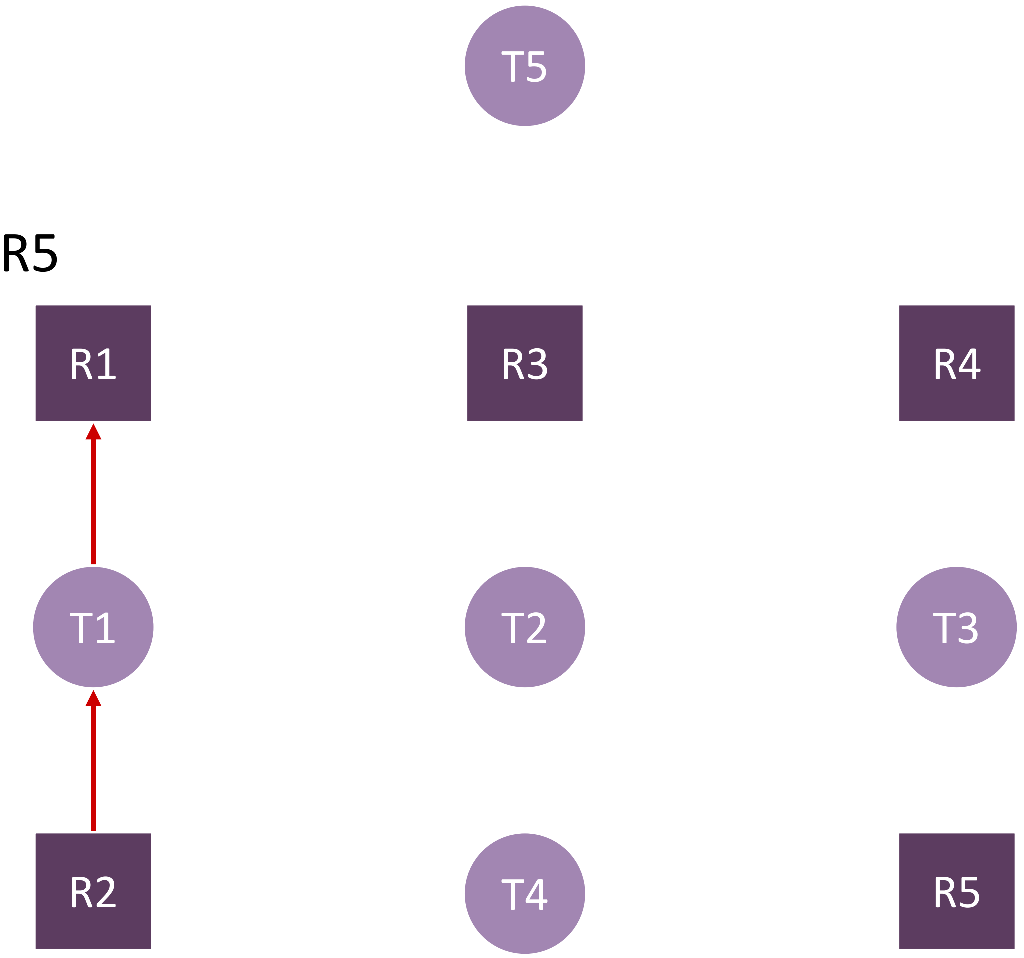
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

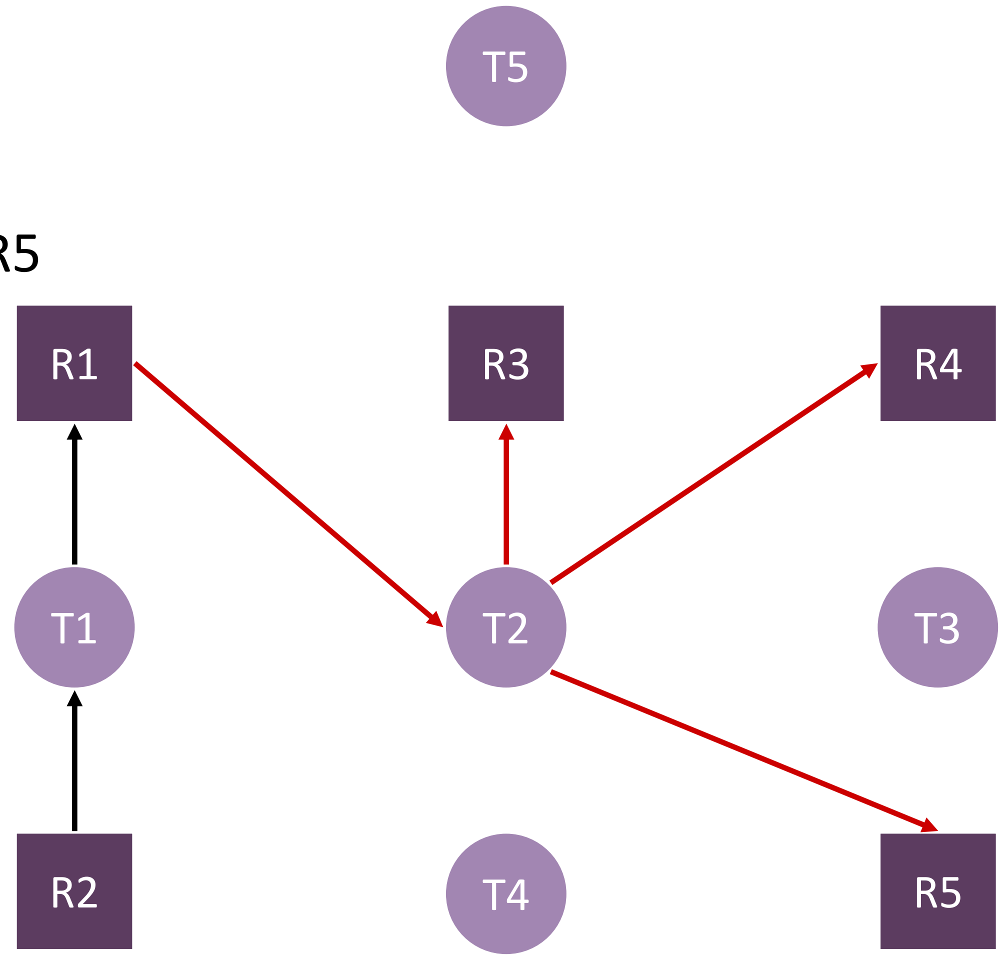
- Thread 1 has R2 but wants R1
- Thread 2 has R1 but wants R3, R4 and R5
- Thread 3 has R4 but wants R5
- Thread 4 has R5 but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

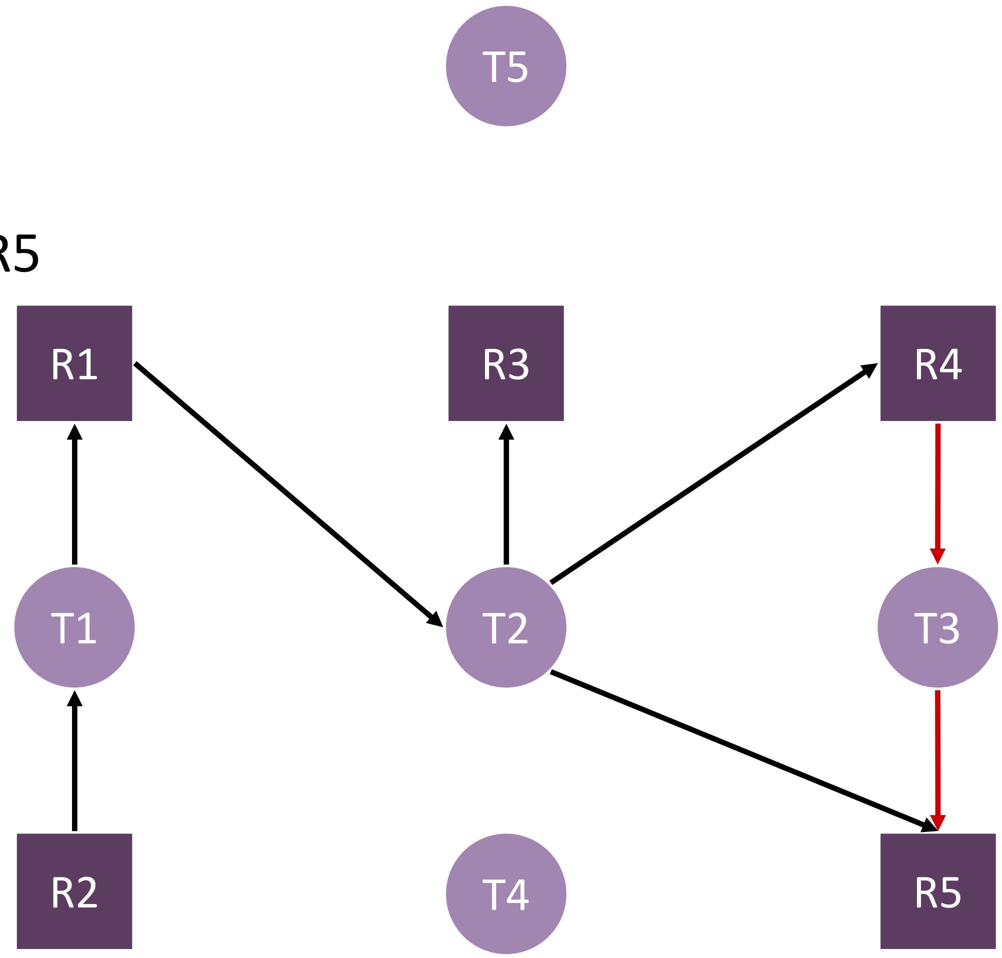
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

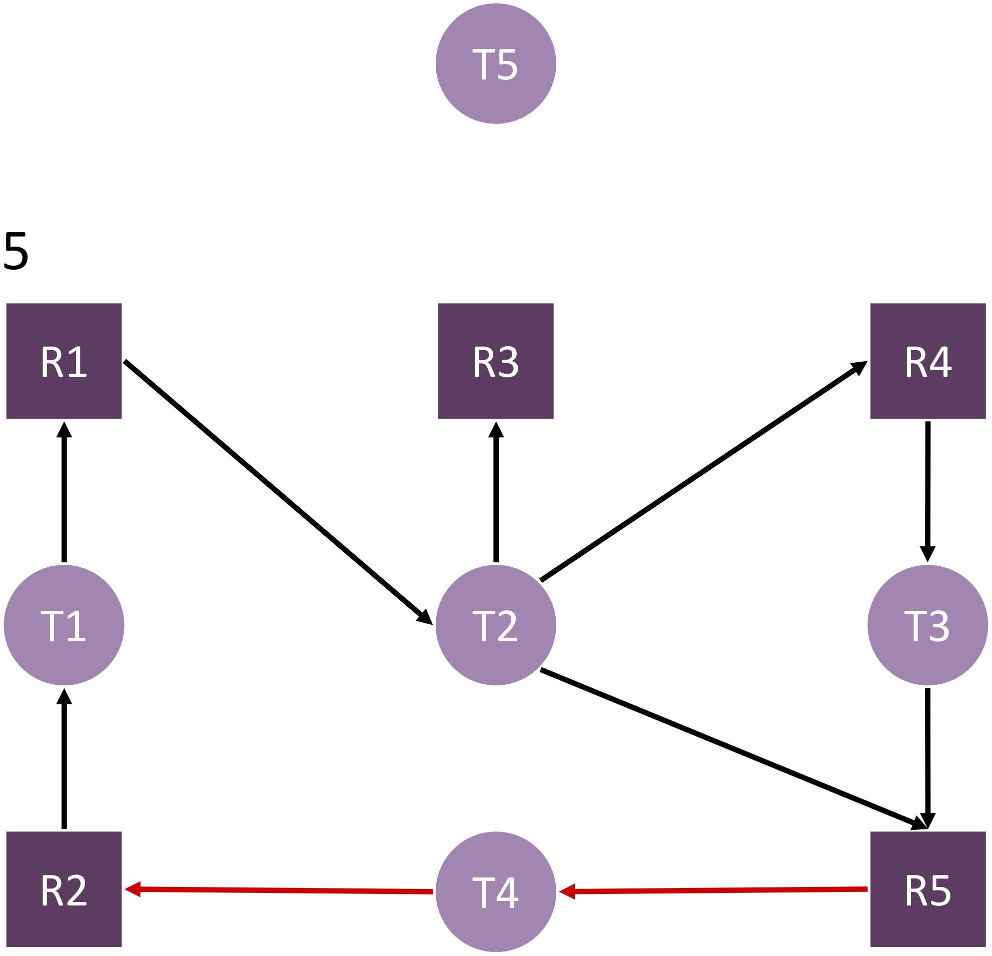
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

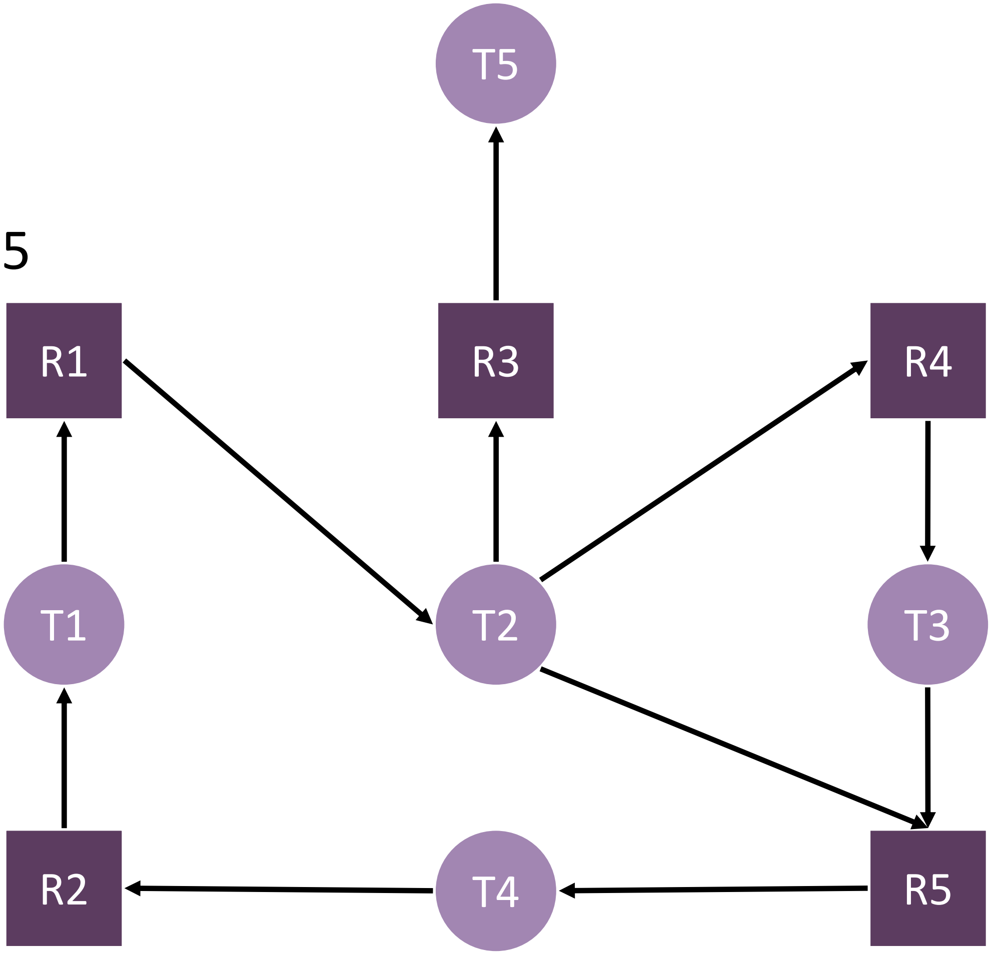
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



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

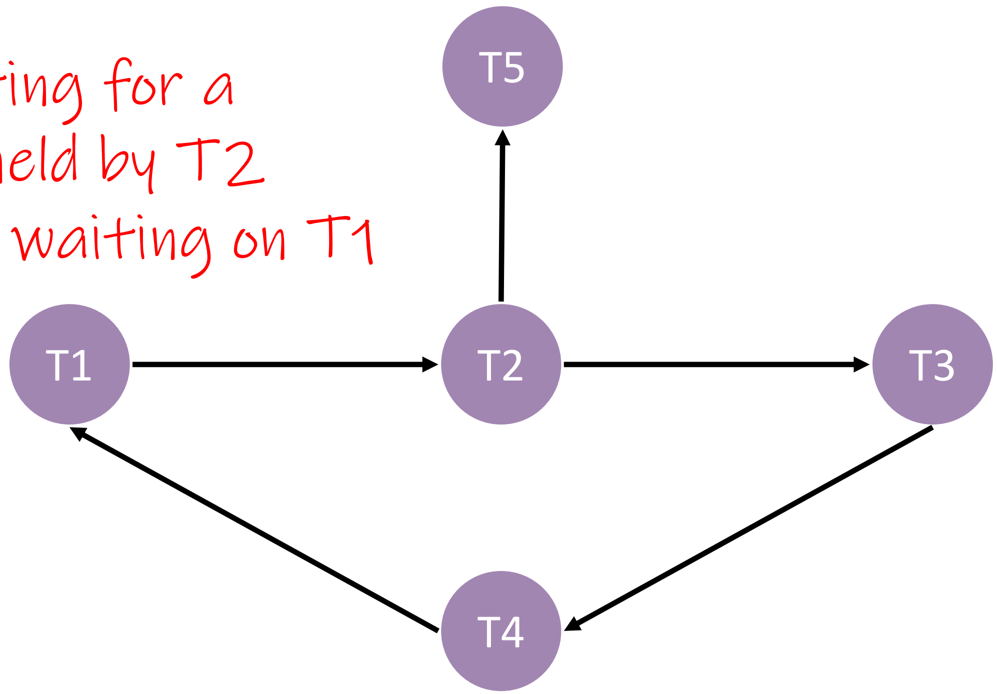


Resource Allocation Graph

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

- Deadlock Handling (start)
  - Ostrich
  - Prevention
  - Detection
  - Avoidance
- **Parallel Analysis**
  - **Recurrences**
  - **Amdahl's Law**

Parallel Algorithms

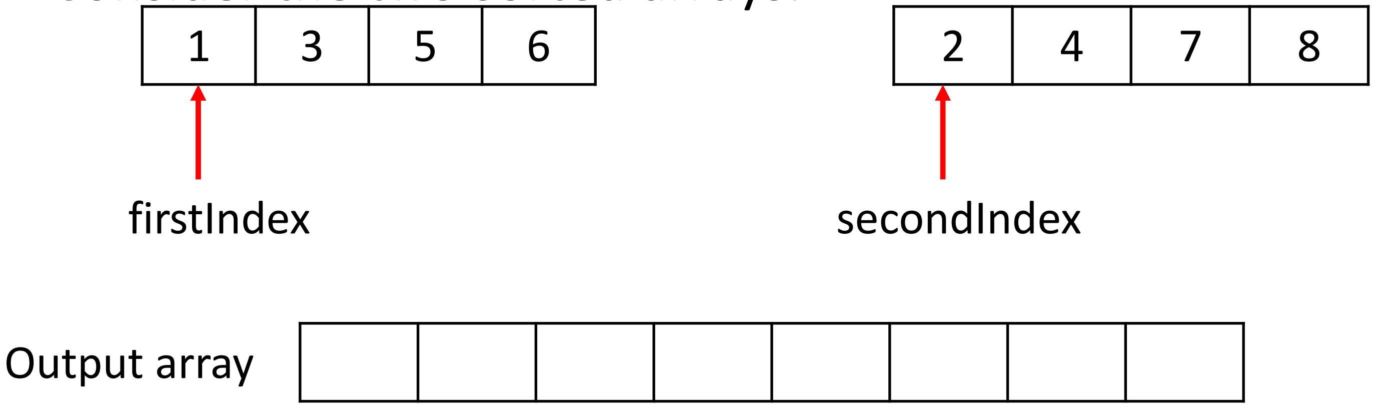
- One interesting applications of threads is for faster algorithms

- 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

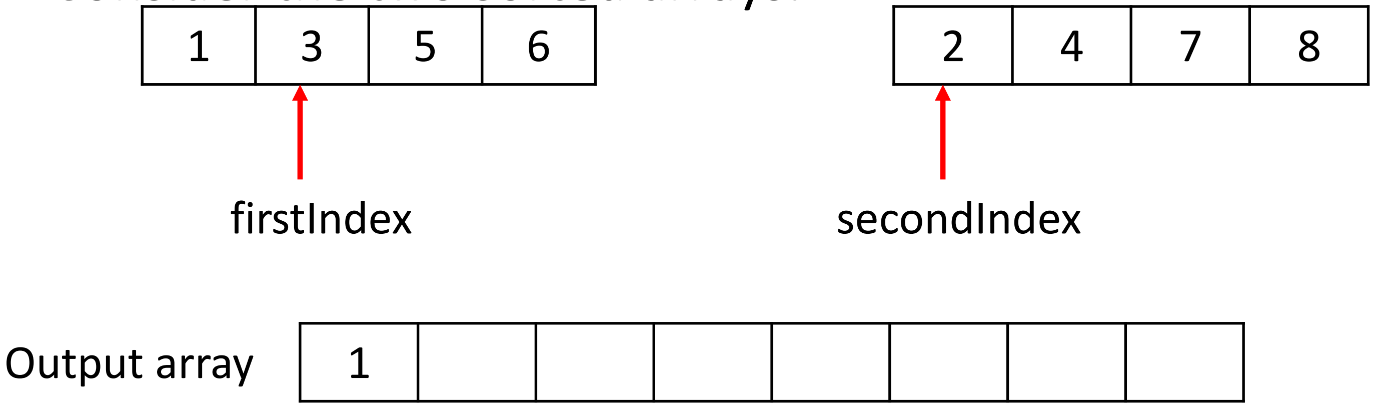
  - Consider the two sorted arrays:



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

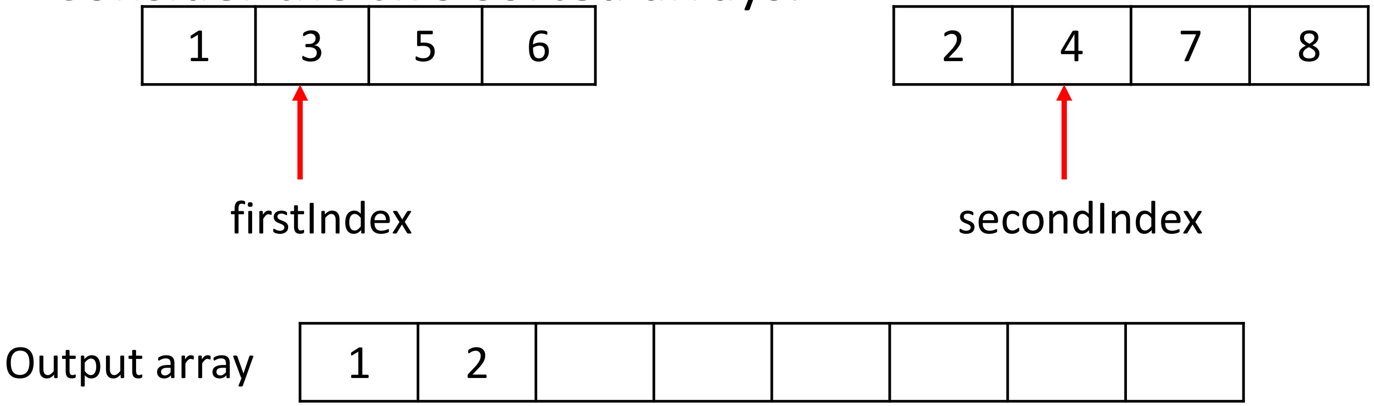
  - Consider the two sorted arrays:



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

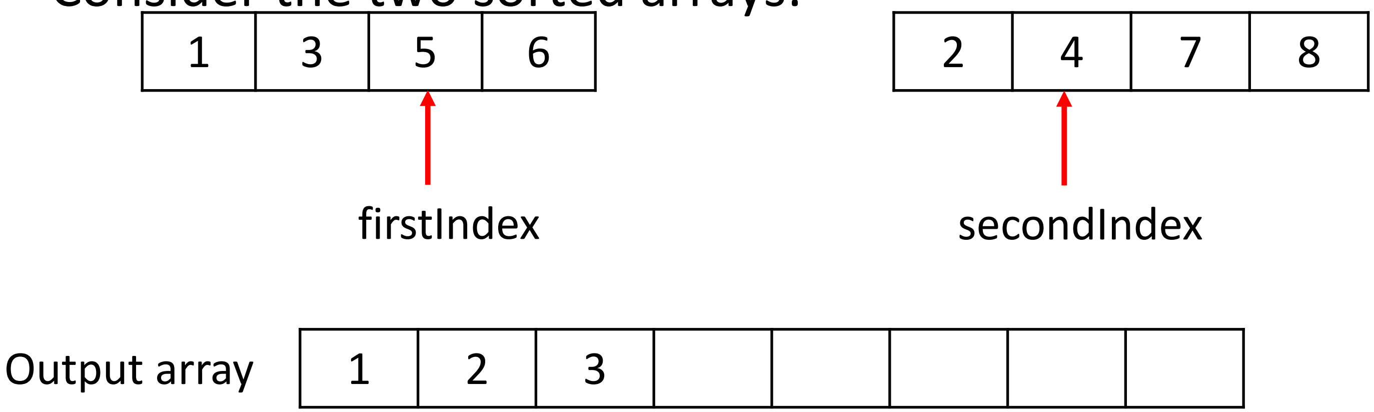
  - Consider the two sorted arrays:



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

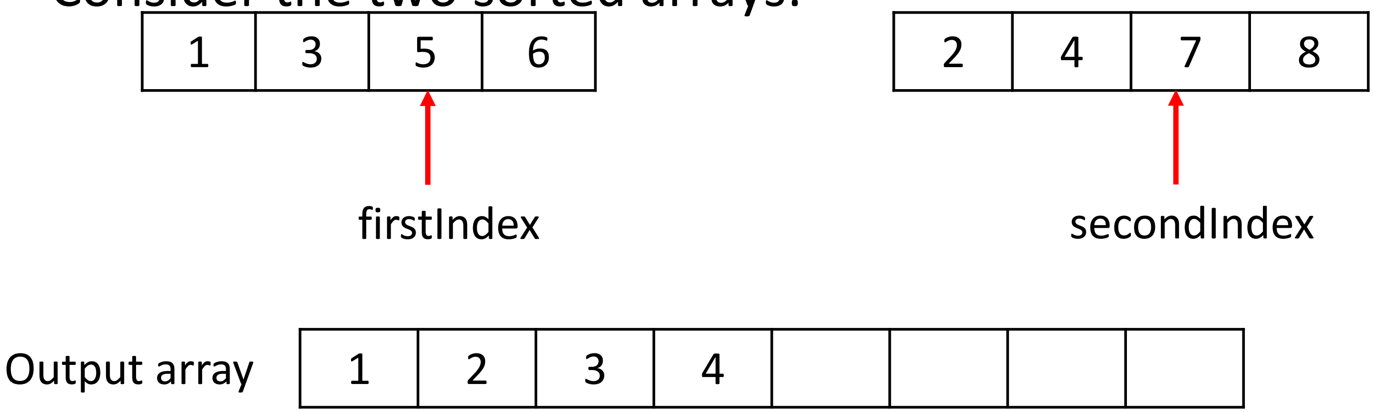
  - Consider the two sorted arrays:



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

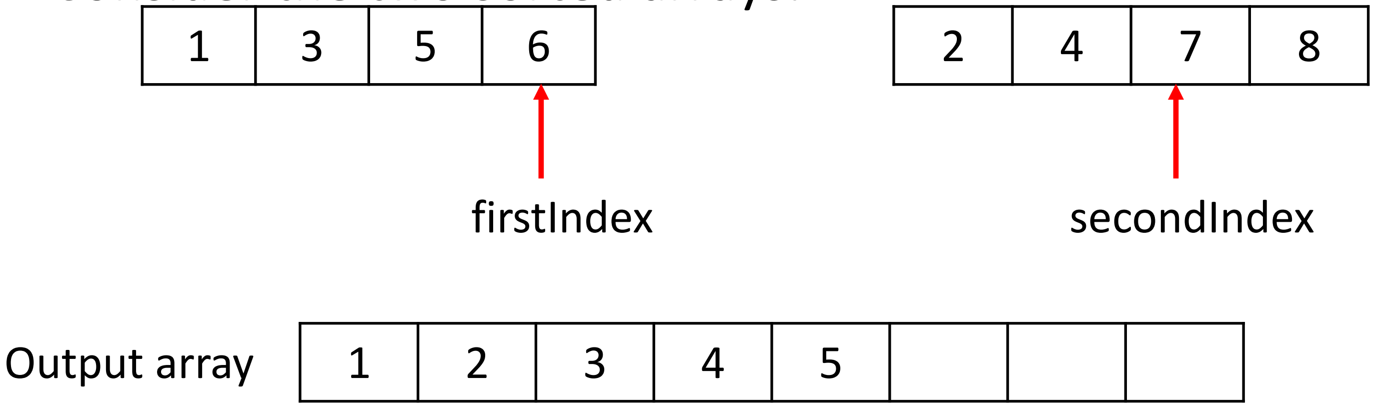
  - Consider the two sorted arrays:



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

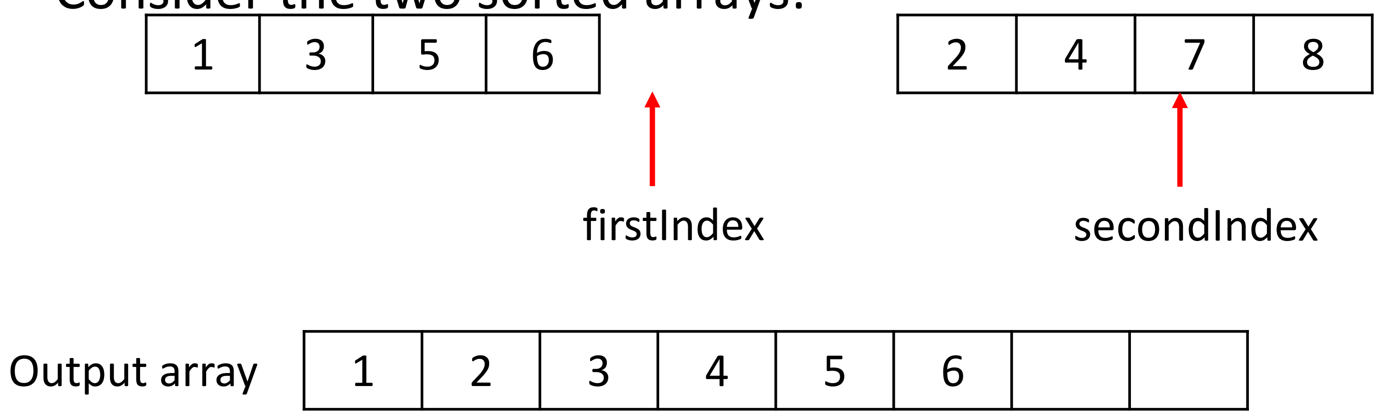
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

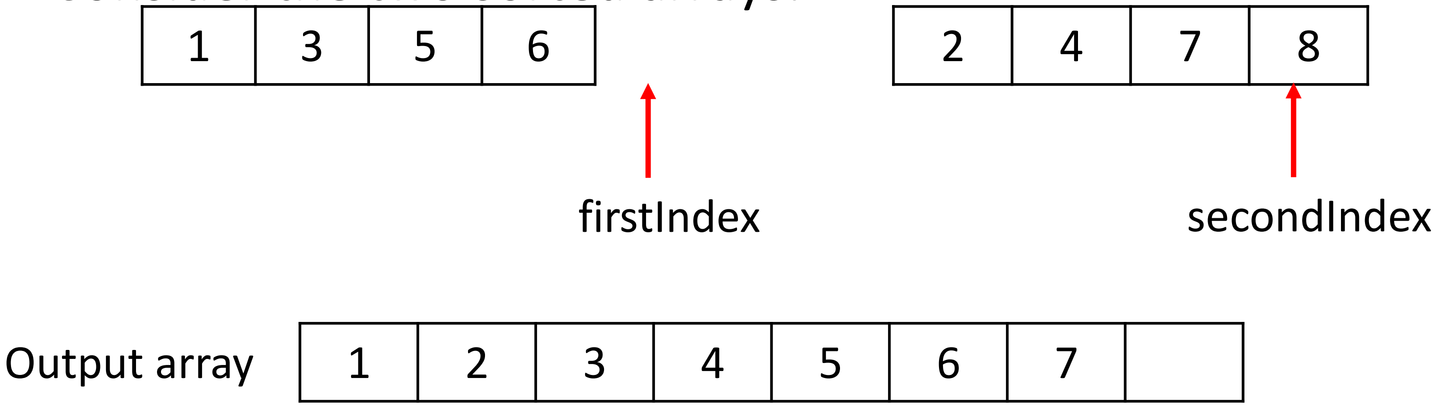
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



Merge Sort: Core Ideas

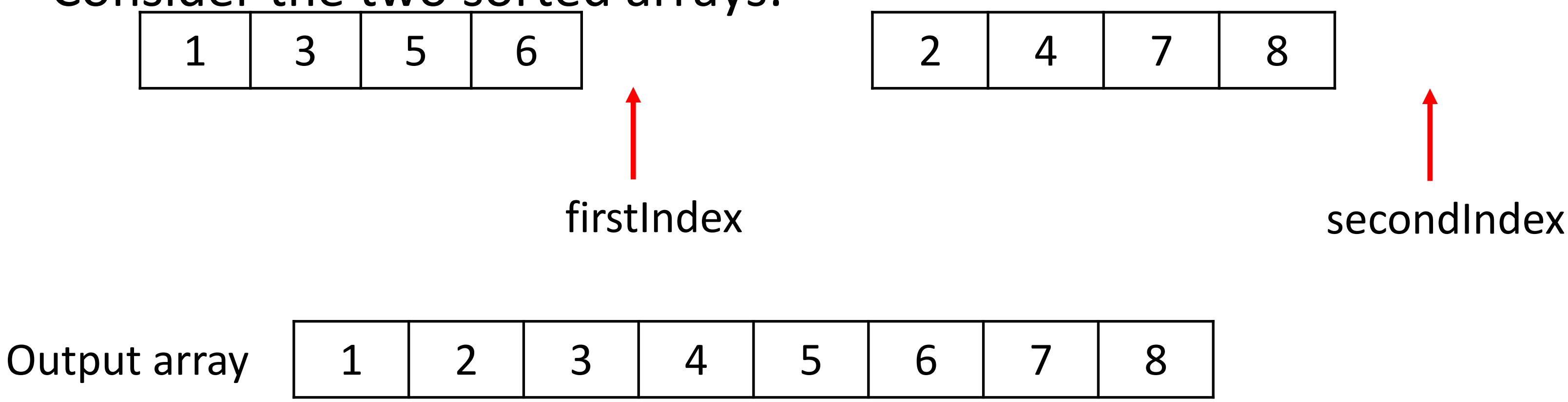
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



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:



Merge Sort: High Level Example

20	10	15	54	55	11	78	14

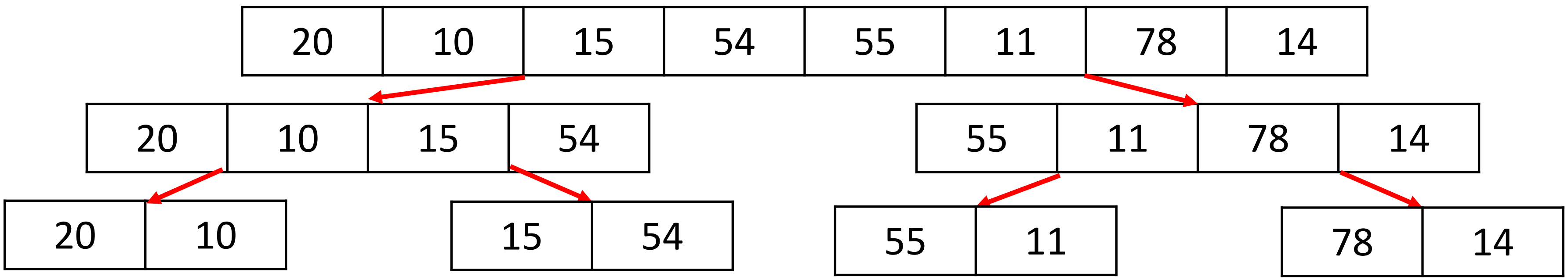
Merge Sort: High Level Example

20	10	15	54	55	11	78	14

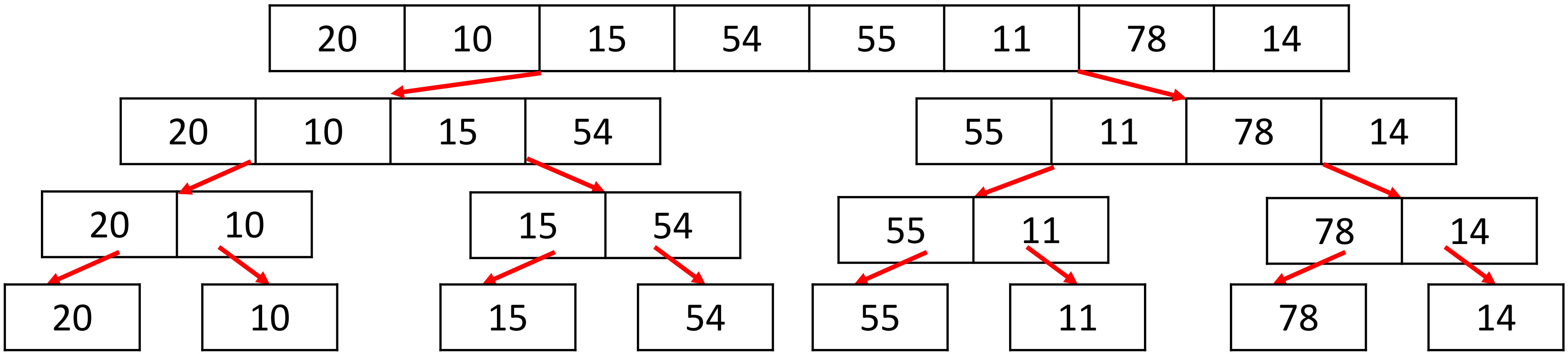
20	10	15	54

55	11	78	14

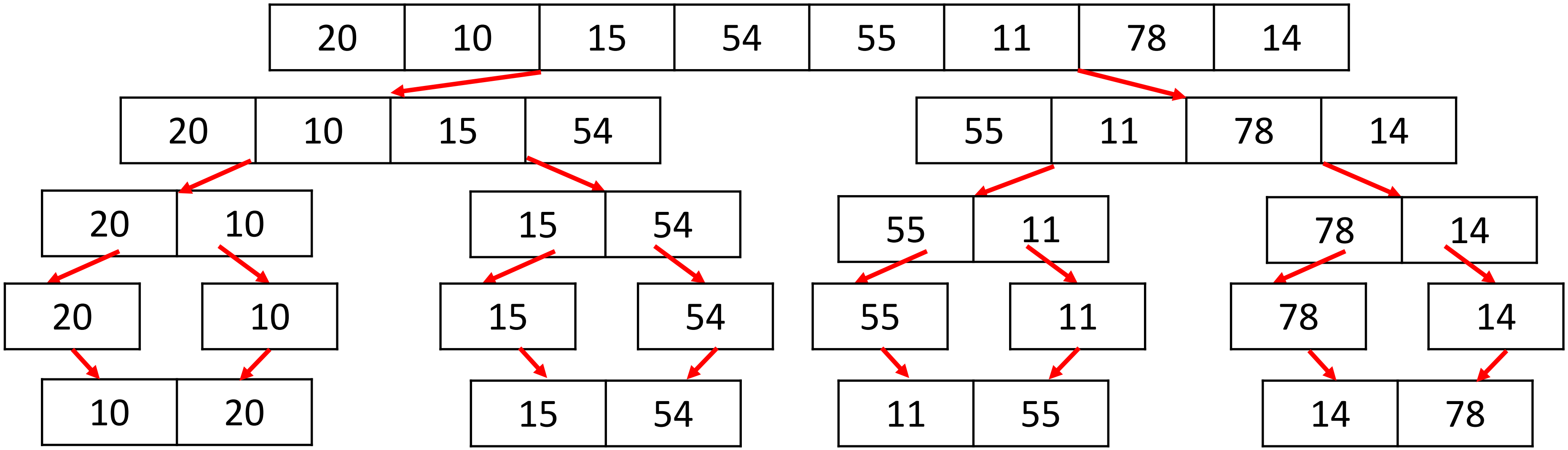
Merge Sort: High Level Example



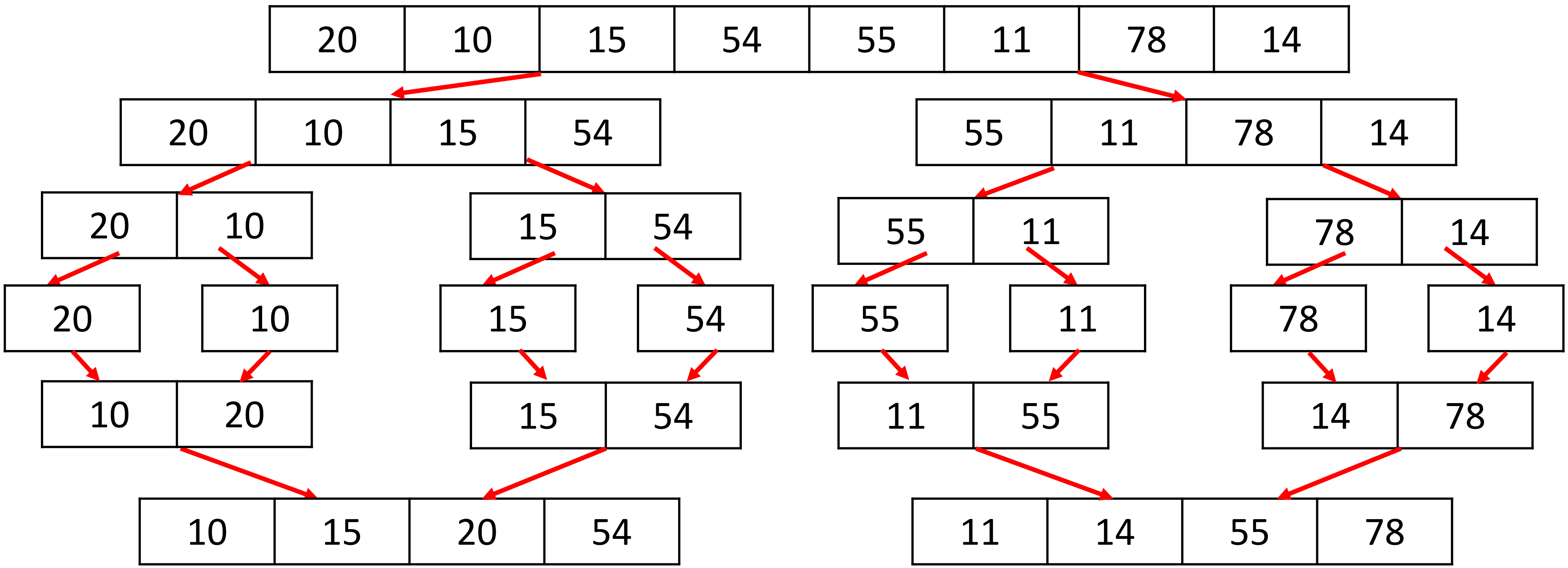
Merge Sort: High Level Example



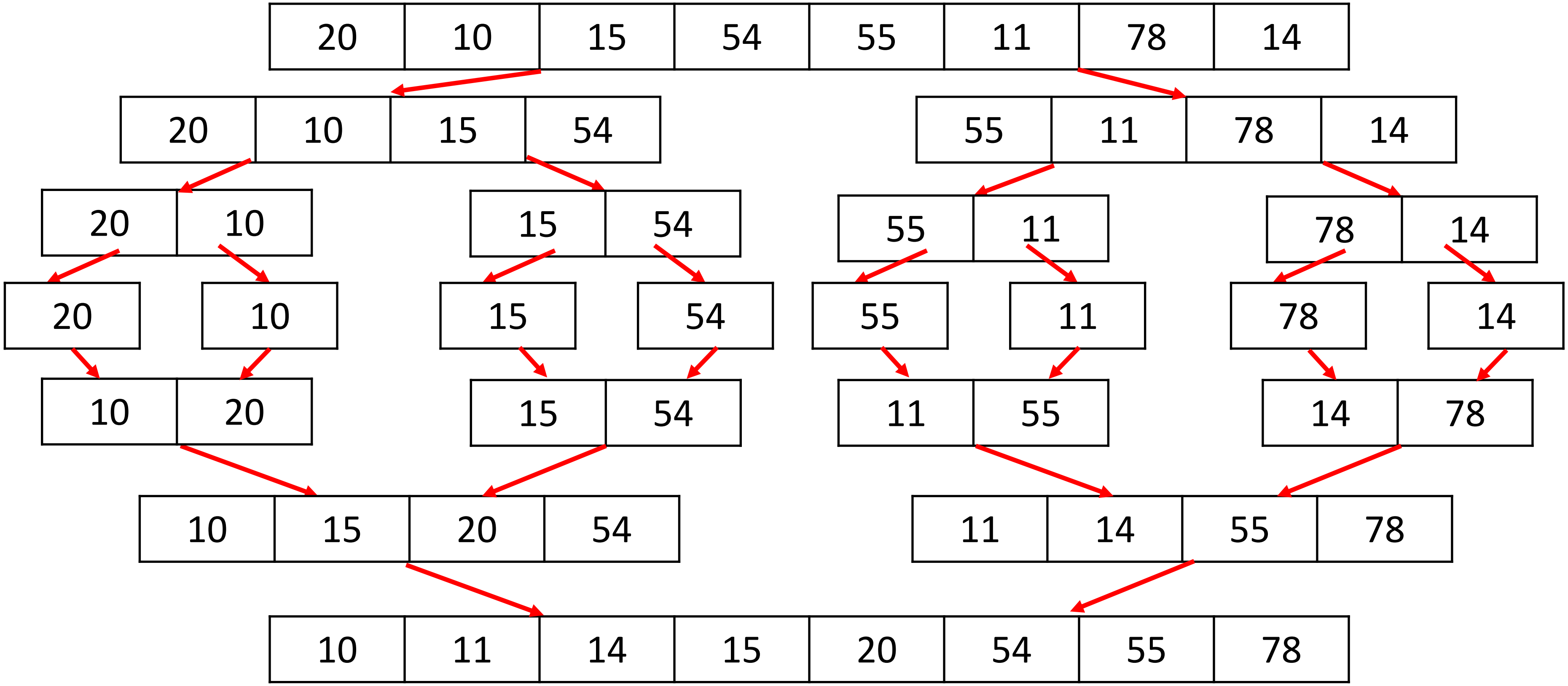
Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort 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..)
  - $\frac{T_1}{T_p} = \frac{1}{1-0.6666+\frac{0.6666}{100000}} = 2.9999 \text{ times faster than sequential}$

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

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

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