Cond & Threads Wrap-up

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

- What is love?

Administrivia

- PennOS
  - Milestone 1 & Extra Credit posted!
  - Need to meet with your TA next week before end of Friday
    - Late tokens possible to use
  - Rough list of allowed functions on Ed

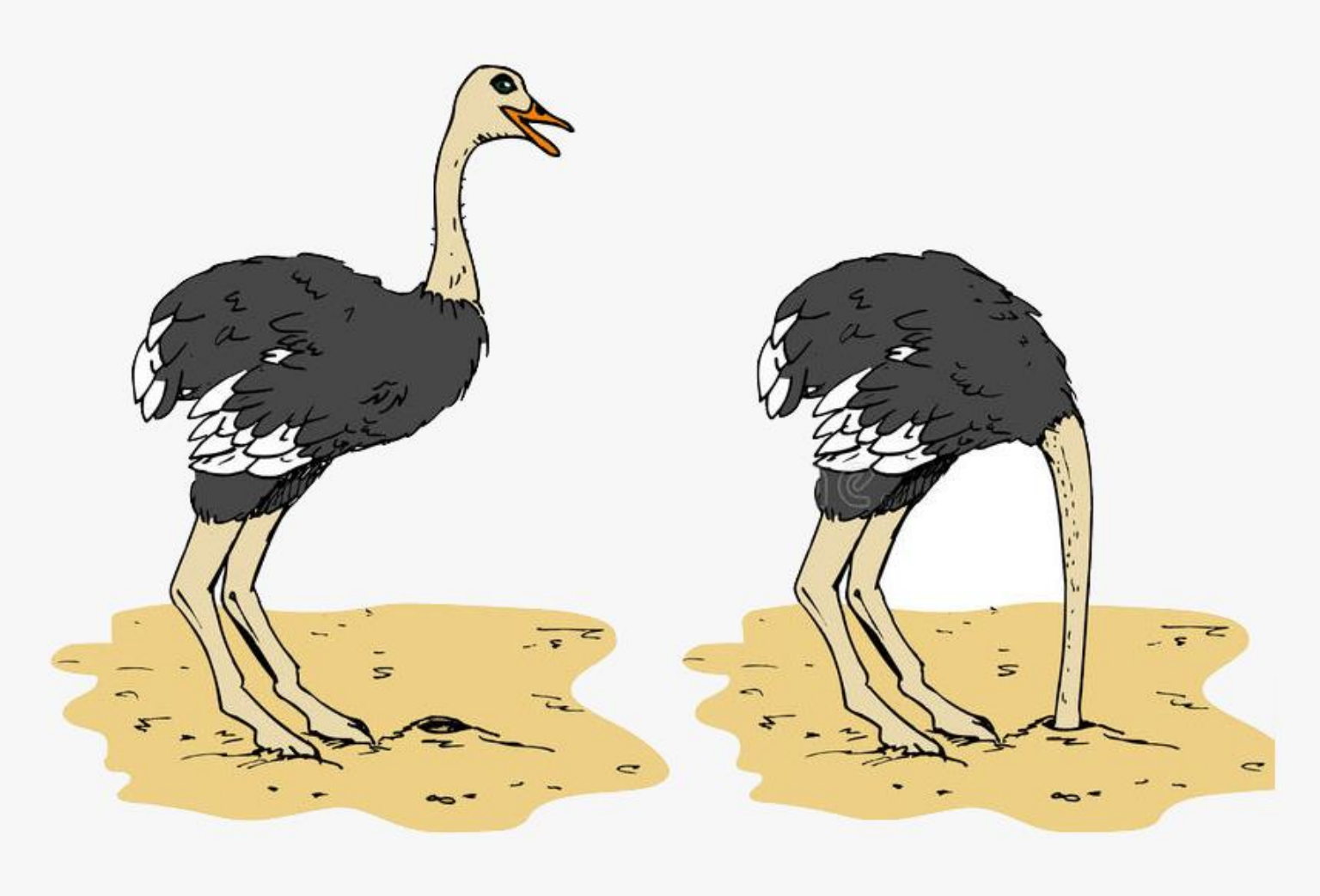
- Reading Assignment Posted!
  - Shouldn't take too long, please just do it
  - We are checking for AI
  - Grading isn't harsh
  - Due Friday @ midnight
    - Late tokens possible to use

Lecture Outline

- **Deadlock Handling**
- Data Races vs Race Conditions
- Producer/Consumer & Condition Variables
- 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

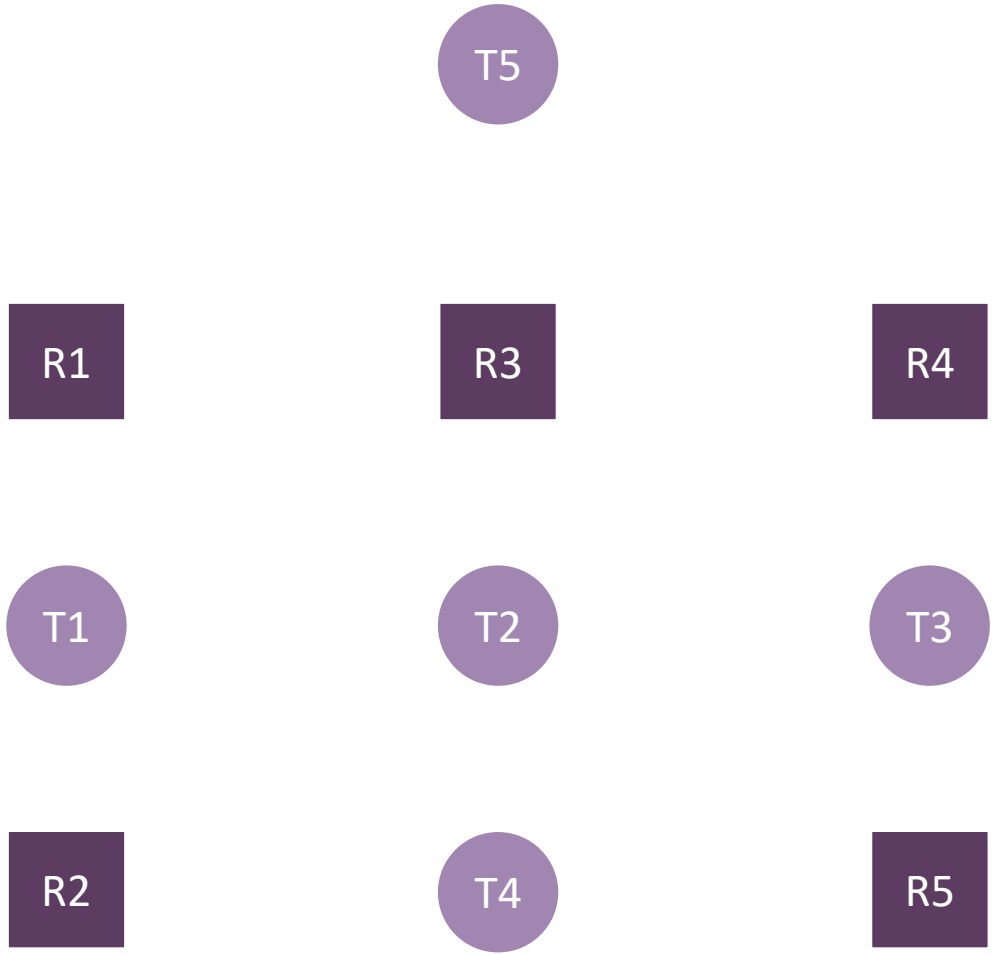
- Consider the following example with 5 threads and 5 resources that require mutual exclusion is this a deadlock?
  - Thread 1 has R2 but wants R1
  - Thread 2 has R1 but wants R3, R4 and R5
  - Thread 3 has R4 but wants R5
  - Thread 4 has R5 but wants R2
  - Thread 5 has R3

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

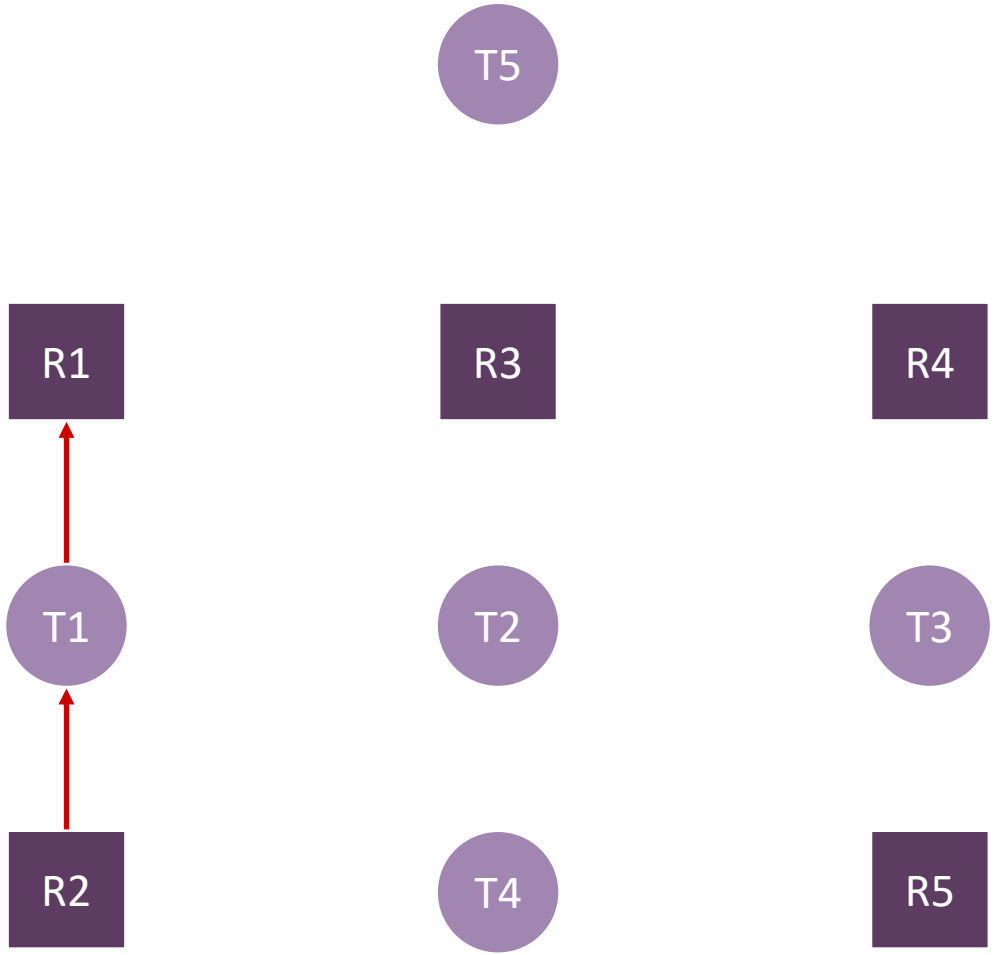
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

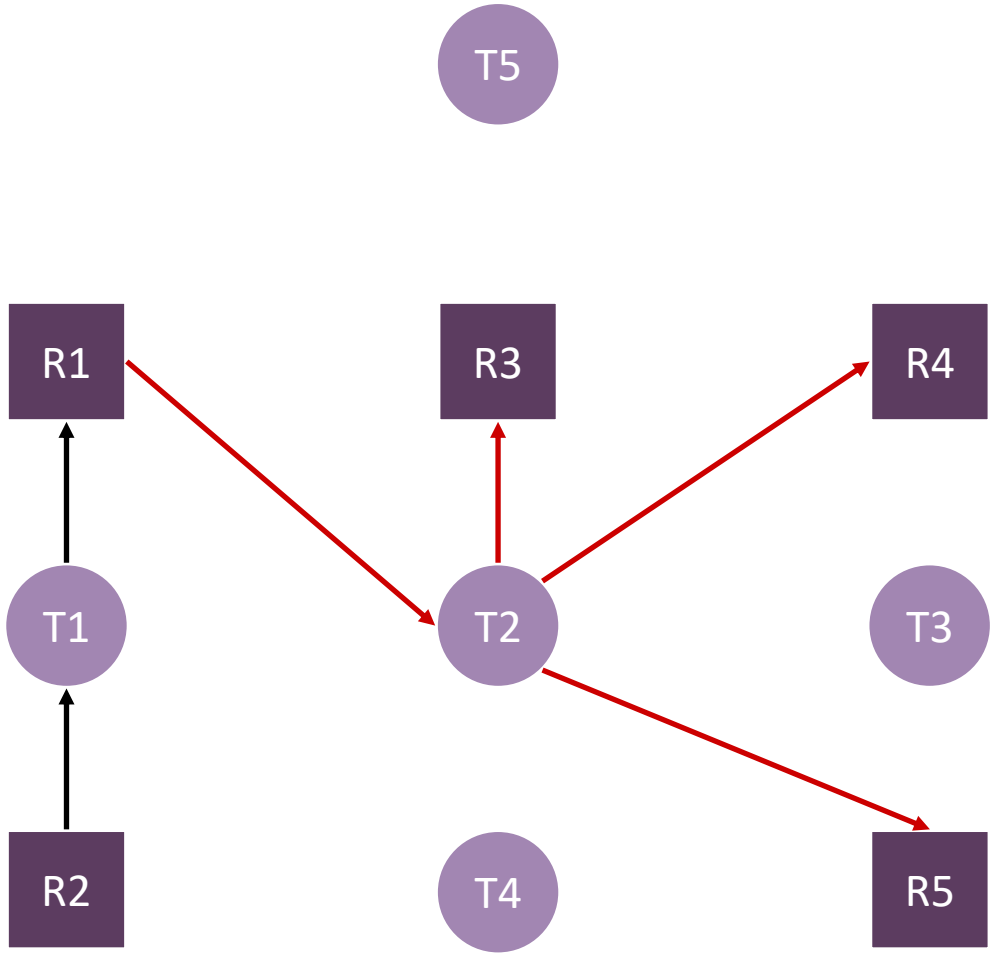
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

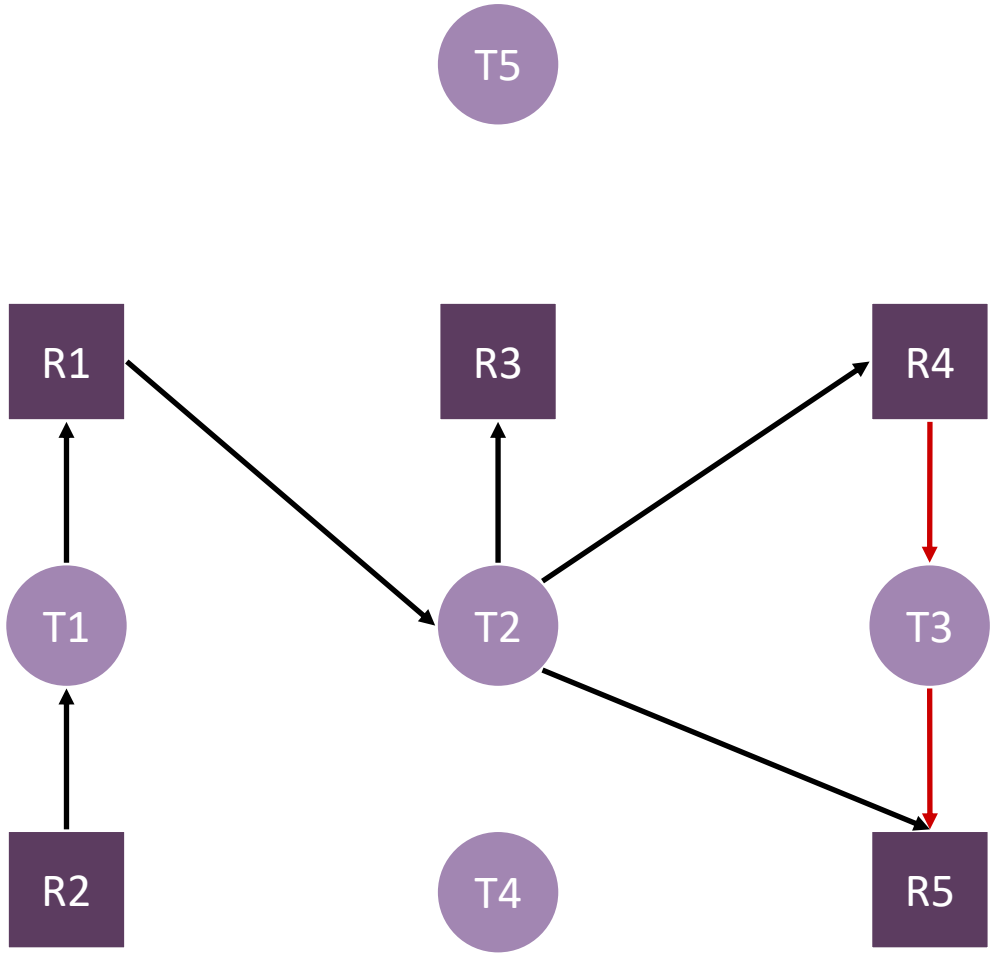
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

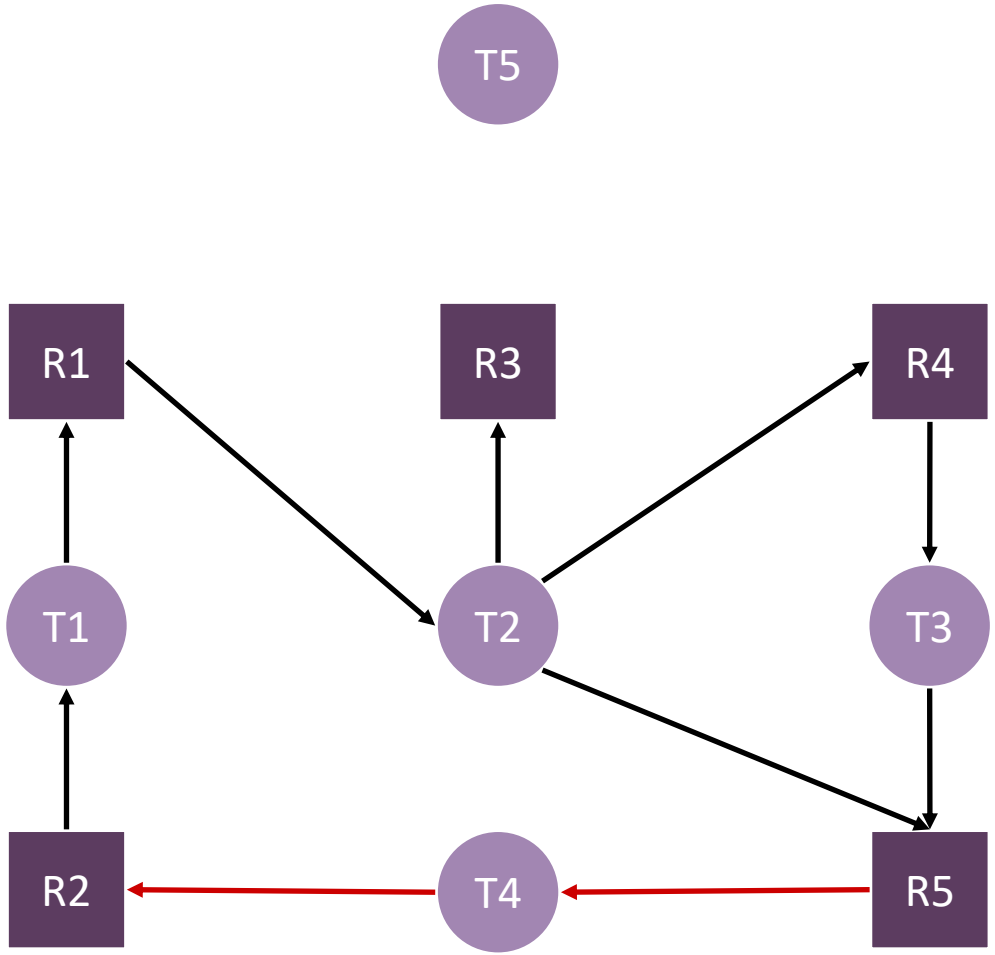
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

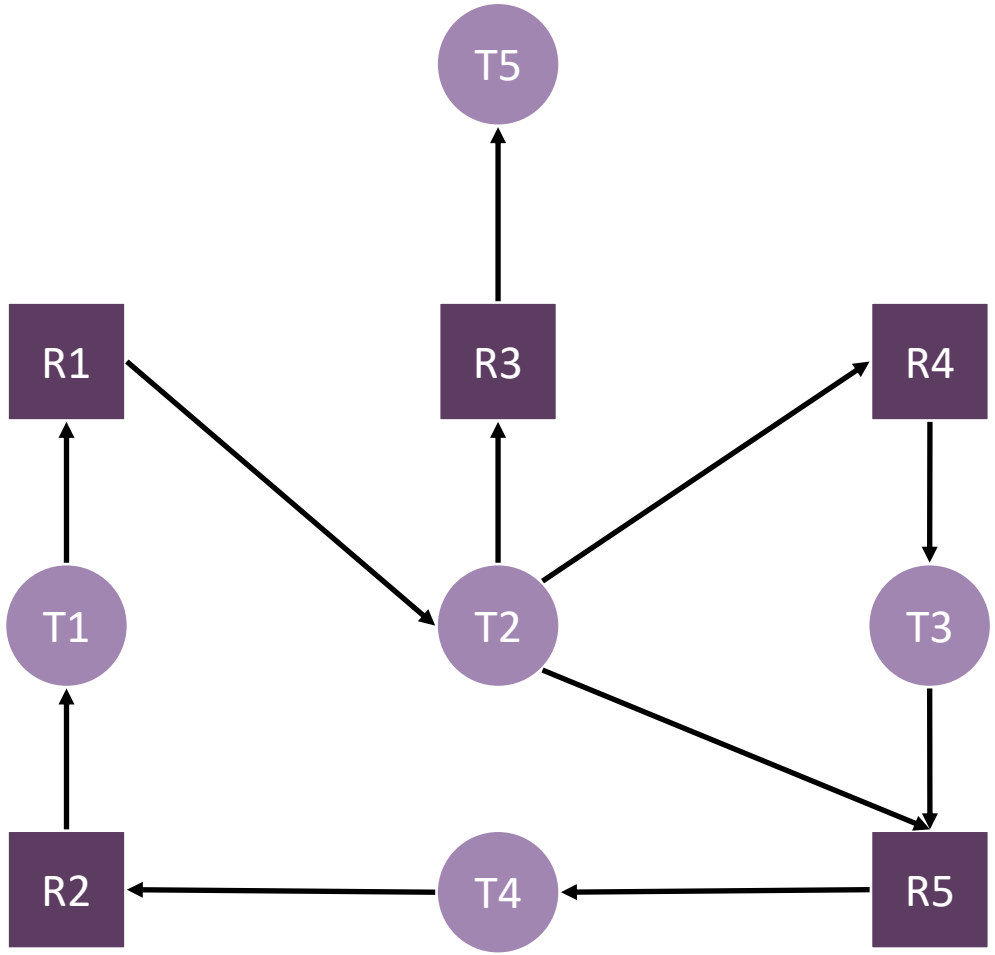
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

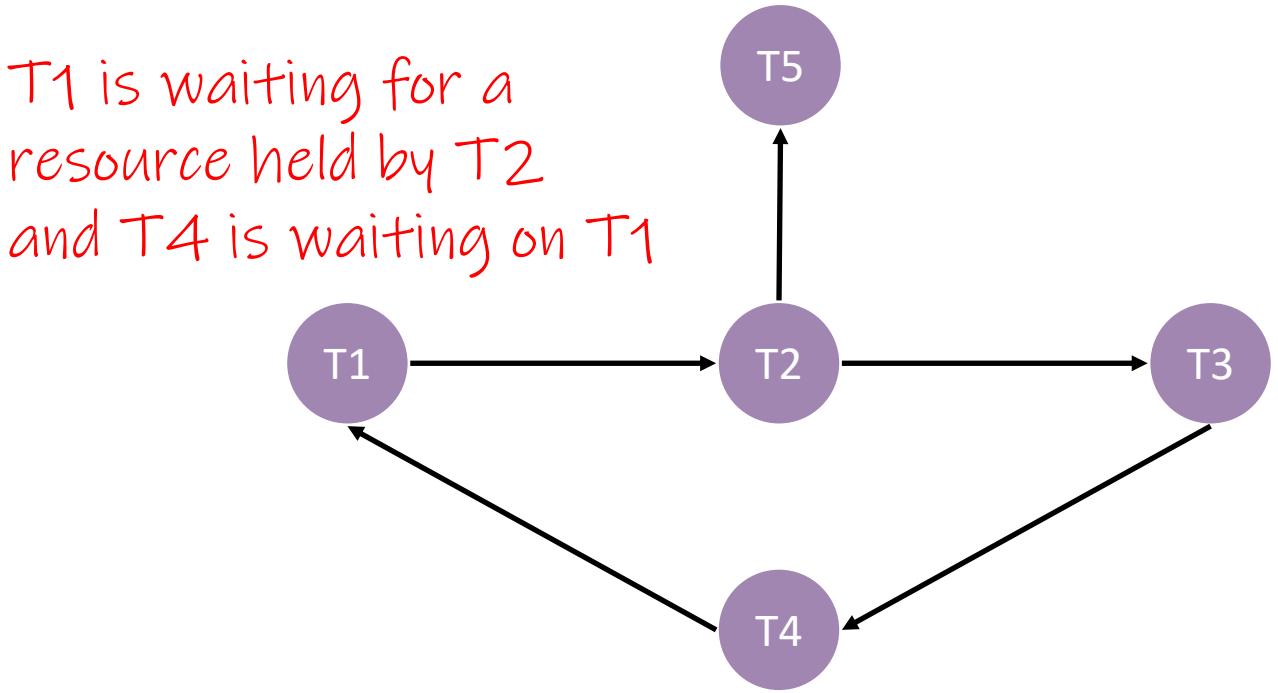
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



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



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
- **Data Races vs Race Conditions**
- Producer/Consumer & Condition Variables
- Amdahl's Law

pollev.com/tqm

```
pthread_mutex_t lock;
string data;

void* produce(void* arg) {
  int fd = open(some_file, O_RDONLY);
  char buf[1024];
  ssize_t res = read(fd, buf, 1023);
  buf[res] = '\0';

  pthread_mutex_lock(&lock);
  data = string(buf);
  pthread_mutex_lock(&lock);

  pthread_exit(NULL);
}
```

```
void* consume(void* arg) {
  pthread_mutex_lock(&lock);
  if (!data.empty()) {
    print(data);
  }
  thread_mutex_unlock(&lock);
  pthread_exit(NULL);
}
```

Does this code have a data race?

- Assume that there is one thread running produce() and another thread running consume()
- Can this program enter an “invalid” (unexpected or error) state from having concurrent memory accesses?
- Assume lock initialized and funcs don’t fail

❖ Any issues with this code?

Race Condition vs Data Race

- Data-Race: when there are concurrent accesses to a shared resource, with at least one write, that can cause the shared resource to enter an invalid or "unexpected" state.

- Race-Condition: Where the program has different behaviour depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are "atomic" or "locked"

- The previous example has no data-race, but it does have a race condition

Thread Communication

- Sometimes threads may need to communicate with each other to know when they can perform operations

- Example: Producer and consumer threads
  - One thread creates tasks/data
  - One thread consumes the produced tasks/data to perform some operation
  - The consumer thread can only consume things once the producer has produced them

- Need to make sure this communication has no data race or race condition

Lecture Outline

- Deadlock Handling
- Data Races vs Race Conditions
- **Producer/Consumer & Condition Variables**
- Amdahl's Law

Producer & Consumer Problem

- Common design pattern in concurrent programming.
  - There are at least two threads, at least one producer and at least one consumer.
  - The producer threads create some data that is then added to a shared data structure
  - Consumers will process and remove data from the shared data structure

- We need to make sure that the threads play nice

Aside: C++ deque

- I am using a c++ **deque** for this example so that we don't have to write our own data structure. This is not legal C

- Deque is a double ended queue, you can push to the front or back and pop from the front or back

```
// global deque of integers
// will be initialized to be empty
deque<int> dq {};

int main() {
  dq.push_back(3);       // adds 3
  int val = dq.at(0);    // access index 0
  dq.pop_front()         // delete first element
  printf("%d\n", val);   // should print 3
}
```

Producer Consumer Example

- Does this work?
- Assume that two threads are created, one assigned to each function

```
deque<int> dq {};

void* producer_thread(void* arg) {
  while (true) {
    dq.push_back(long_computation());
  }
}

void* consumer_thread(void* arg) {
  while (true) {
    while (dq.size() == 0) {
      // do nothing
    }
    int val = dq.at(0);
    dq.pop_front();
    do_something(val);
  }
}
```

Poll Everywhere

pollev.com/tqm

- How do we use mutex to fix this? To make sure that the threads access dq safely.
  - You are only allowed to add calls to pthread_mutex_lock and pthread_mutex_unlock
  - Can add other mutexes if needed
- Similar code: no_sync.cpp

```
deque<int> dq {};
pthread_mutex_t dq_lock;

void* producer_thread(void* arg) {
  while (true) {
    dq.push_back(long_computation());
  }
}

void* consumer_thread(void* arg) {
  while (true) {
    while (dq.size() == 0) {
      // do nothing
    }
    int val = dq.at(0);
    dq.pop_front();
    do_something(val);
  }
}
```

Any issue?

- The code is correct, but do we notice anything wrong with this code?

- Maybe a common inefficiency that I have told you about several times before (just in other contexts?)

- The consumer code “busy waits” when there is nothing for it to consume.
  - It is particularly bad if we have multiple consumers, the locks make the busy waiting of the consumers sequential and use more CPU resources.

Thread Communication: Naïve Solution

- Consider the example where a thread must wait to be notified before it can print something out and terminate

- Possible solution: "Spinning"
  - Infinitely loop until the producer thread notifies that the consumer thread can print

- See `spinning.cpp`
  - The thread in the loop uses A LOT of cpu just checking until the value is safe
  - Use **`top`** to see CPU util

- Alternative: Condition variables

Condition Variables

- Variables that allow for a thread to wait until they are notified to resume
- Avoids waiting clock cycles “spinning”
- Done in the context of mutual exclusion
  - a thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_init(pthread_cond_t* cond,
                        const pthread_condattr_t* attr);
  ```
  - Initializes a condition variable with specified attributes
- ```
  int pthread_cond_destroy(pthread_cond_t* cond);
  ```
  - "Uninitializes" a condition variable – clean up when done

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_wait(pthread_cond_t* cond,
                        pthread_mutex_t* mutex);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked
- ```
  int pthread_cond_signal(pthread_cond_t* cond);
  ```
  - Unblock at least one of the threads on the specified condition
- ```
  int pthread_cond_broadcast(pthread_cond_t* cond);
  ```
  - Unblock all threads blocked on the specified condition
  ```

pthread_cond_t Internal Pseudo-Code

- Here is some pseudo code to help understand condition variables

```
int pthread_cond_wait(pthread_cond_t* cond, pthead_mutex_t* mutex) {
  pthread_mutex_unlock(&lock);
  sleep_on_cond(cond);  // sleeps till cond wakes them up
  pthread_mutex_lock(&lock);
  return 0;
}
```

```
int pthread_cond_signal(pthread_cond_t* cond) {
  wakeup_a_thread(cond);  // wake's up a thread sleeping on the cond
  return 0;
}
```

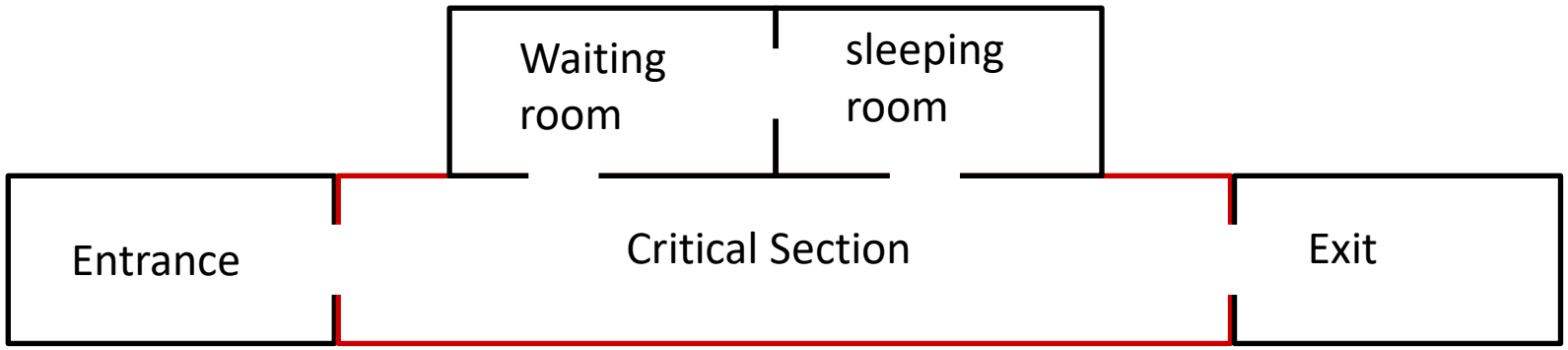
```
int pthread_cond_broadcast(pthread_cond_t* cond) {
  for (thread_sleeping : cond->asleep) { // wake's up all threads
    wakeup(thread_sleeping);
  }
  return 0;
}
```

Demo: `cond.cpp`

- See `cond.cpp`
  - Changes our spinning code to use a condition variable properly
  - No issues with cpu utilization!

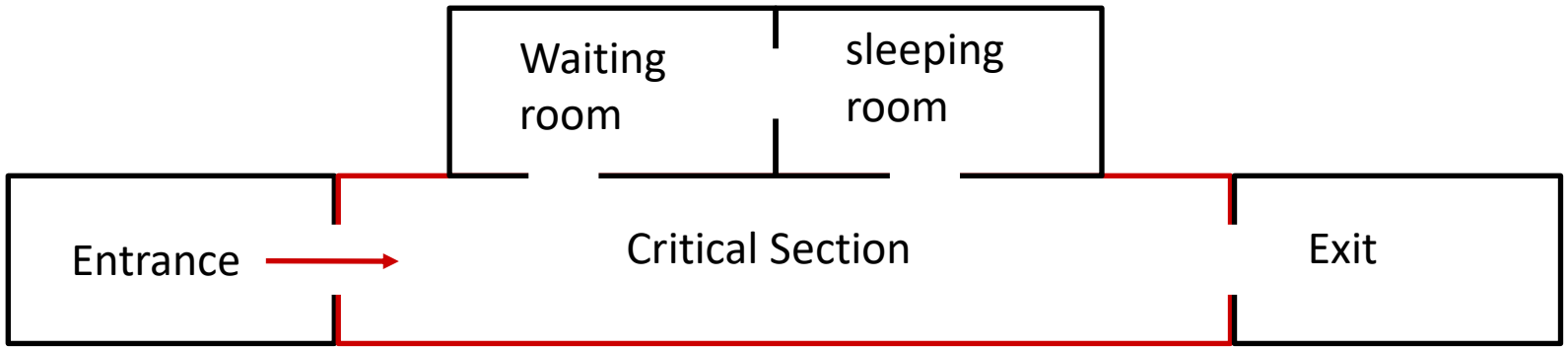
Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example

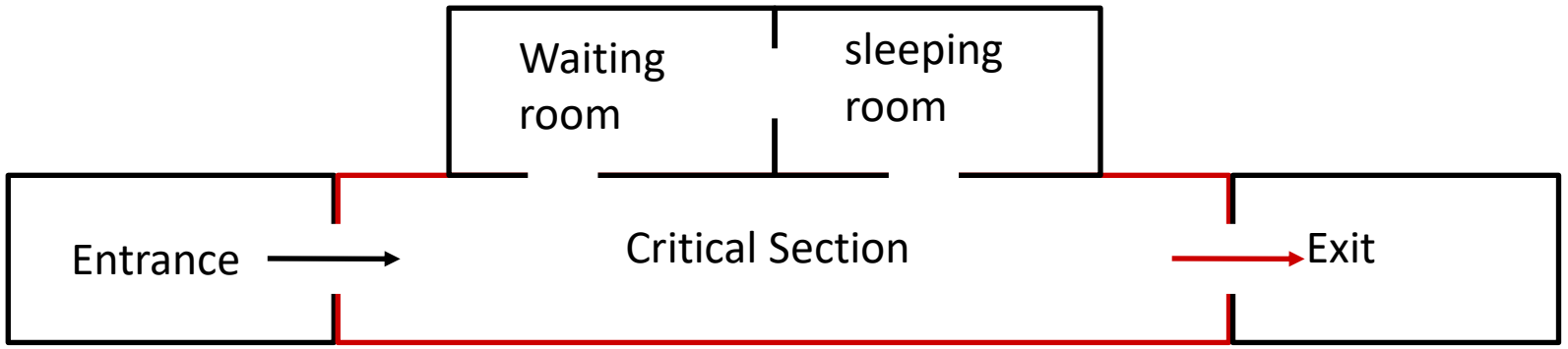


`pthread_mutex_lock`

A thread enters the critical section by acquiring a lock

Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



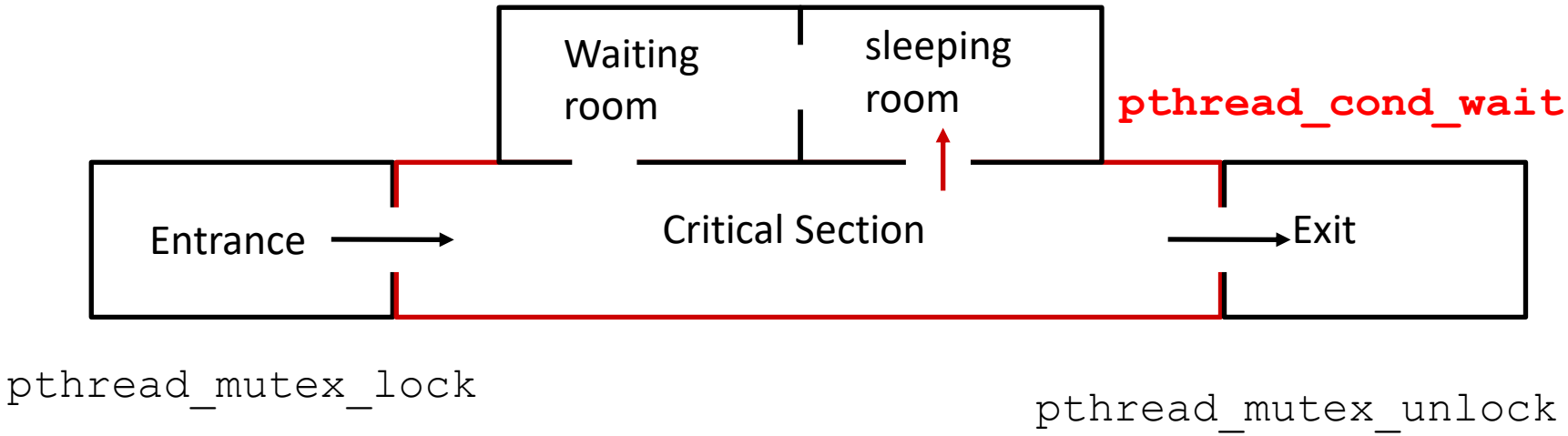
`pthread_mutex_lock`

`pthread_mutex_unlock`

A thread can exit the critical section by acquiring a lock

Condition Variable & Mutex Visualization

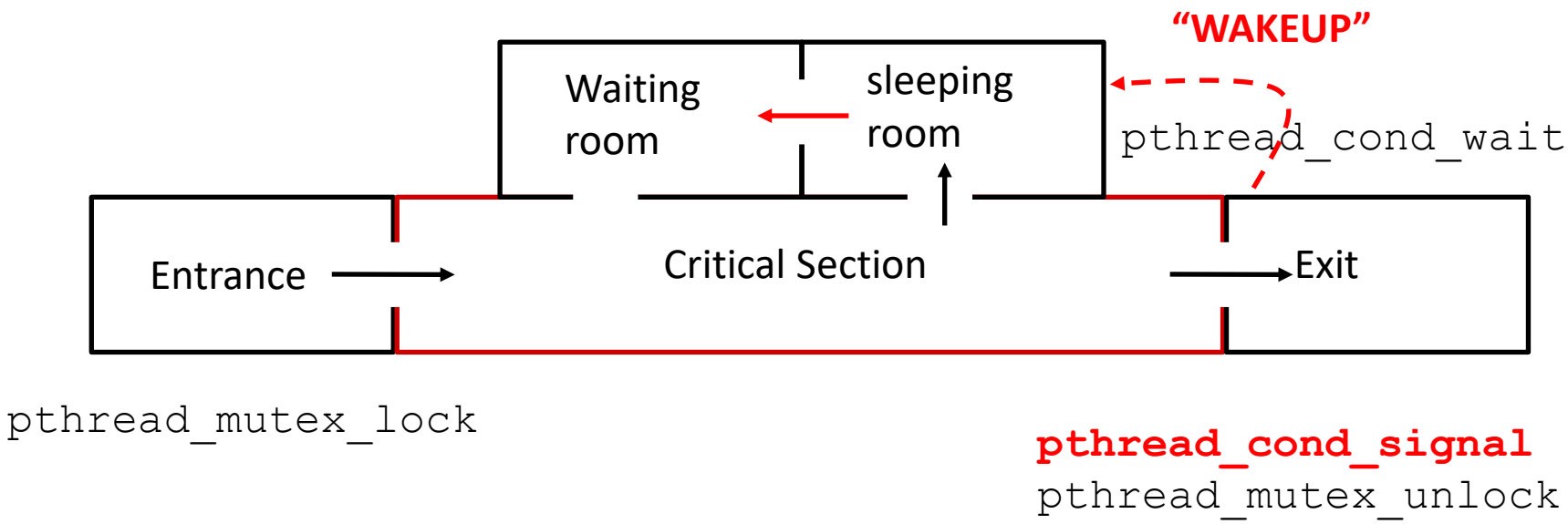
- This is to visualize how we are using condition variables in this example



If a thread can't complete its action, or must wait for some change in state, it can "go to sleep" until someone wakes it up later.
It will release the lock implicitly when it goes to sleep

Condition Variable & Mutex Visualization

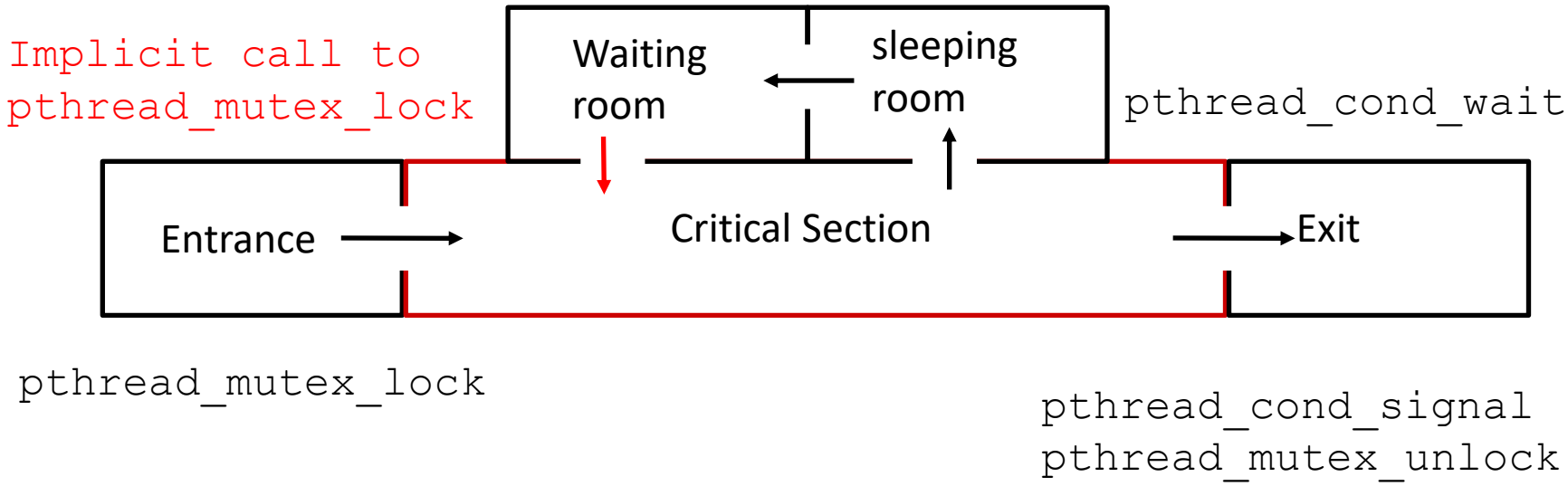
- This is to visualize how we are using condition variables in this example



When a thread modifies state and then leaves the critical section, it can also call pthread_cond_signal to wake up threads sleeping on that condition variable

Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



One or more sleeping threads wake up and attempt to acquire the lock.
Like a normal call to pthread_mutex_lock the thread will block until it can acquire the lock

Lecture Outline

- Deadlock Handling
- Data Races vs Race Conditions
- Producer/Consumer & Condition Variables
- **Parallel Analysis & Amdahl's Law**

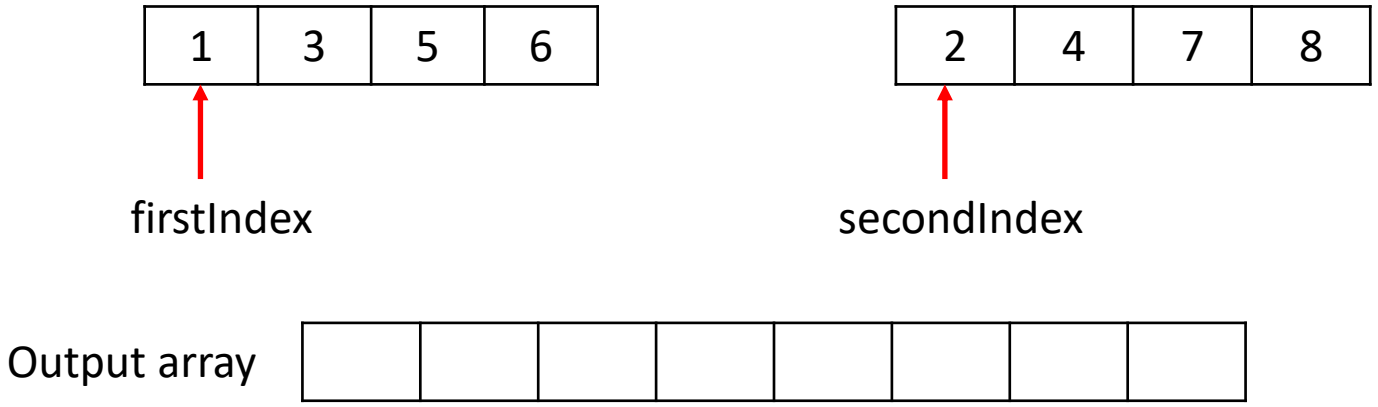
Parallel Algorithms

- One interesting applications of threads is for faster algorithms

- Common Example: Merge sort

Merge Sort: Core Ideas

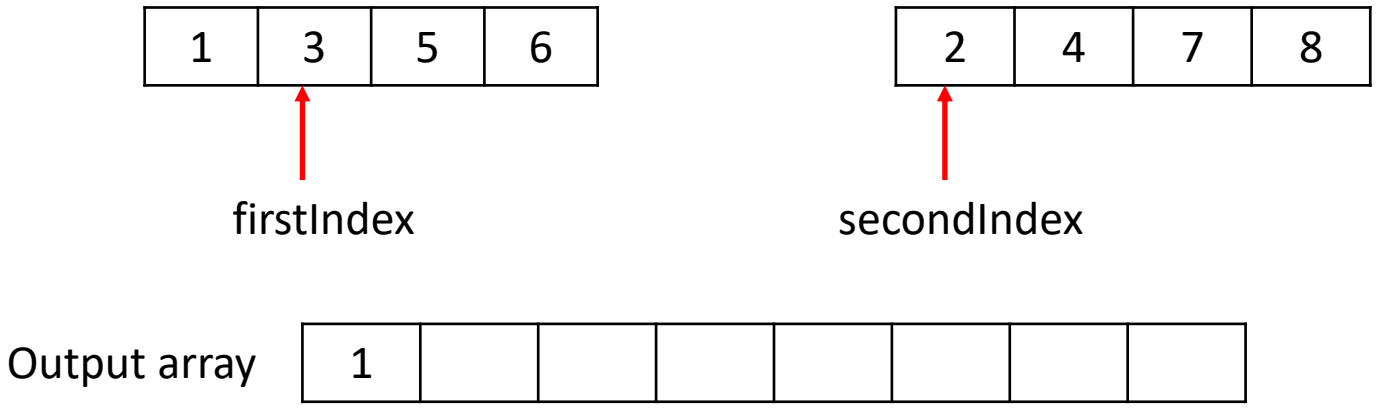
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

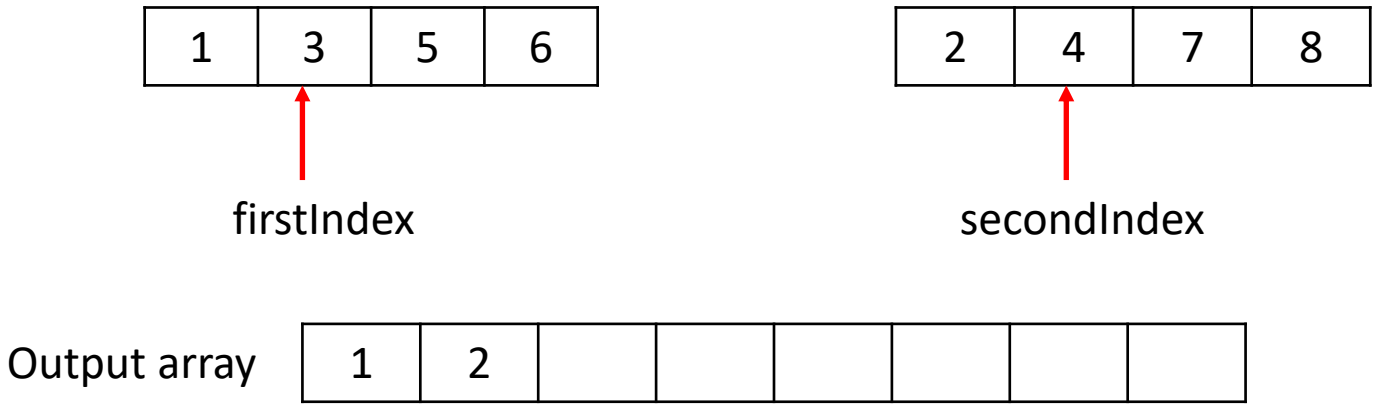
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



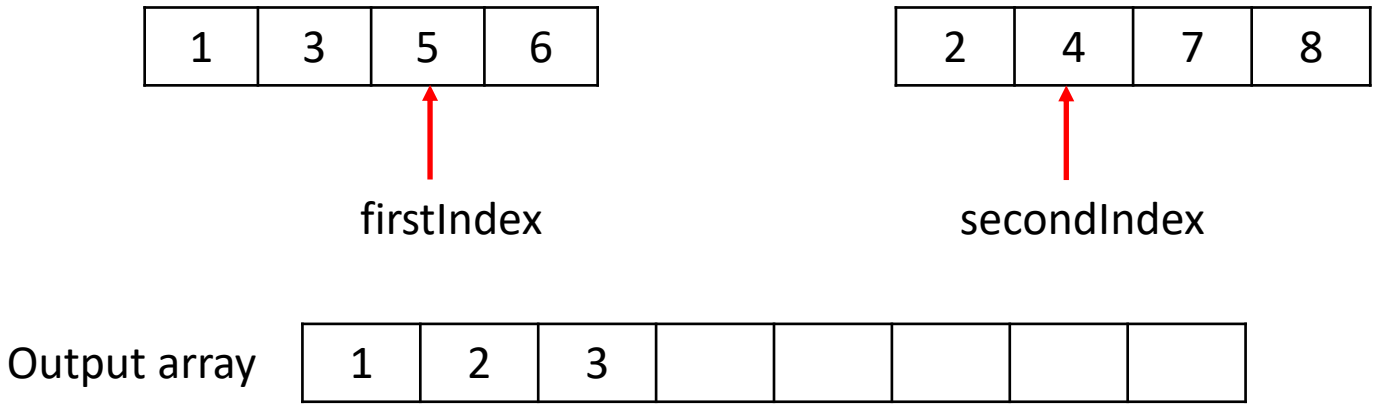
Merge Sort: Core Ideas

- 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:



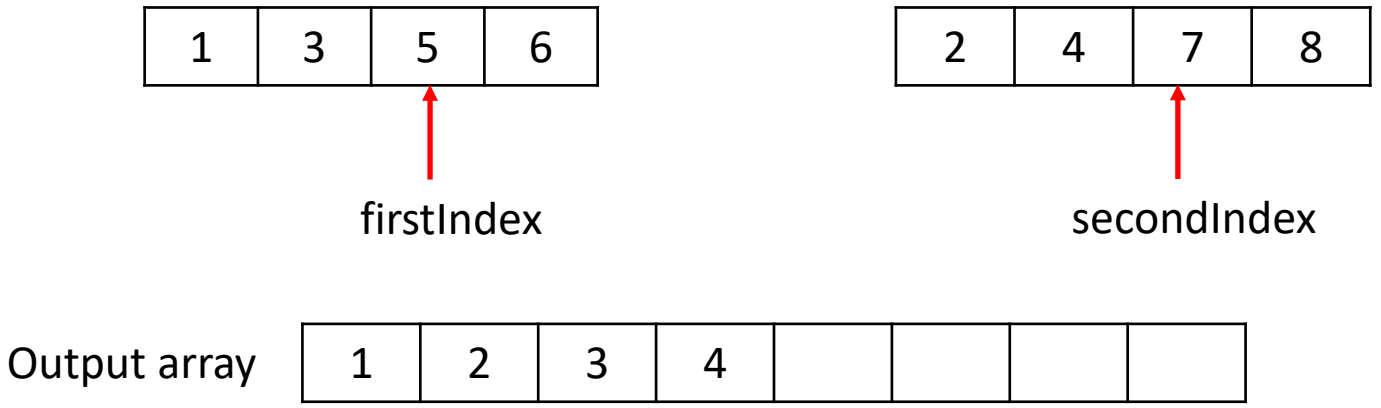
Merge Sort: Core Ideas

- 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:



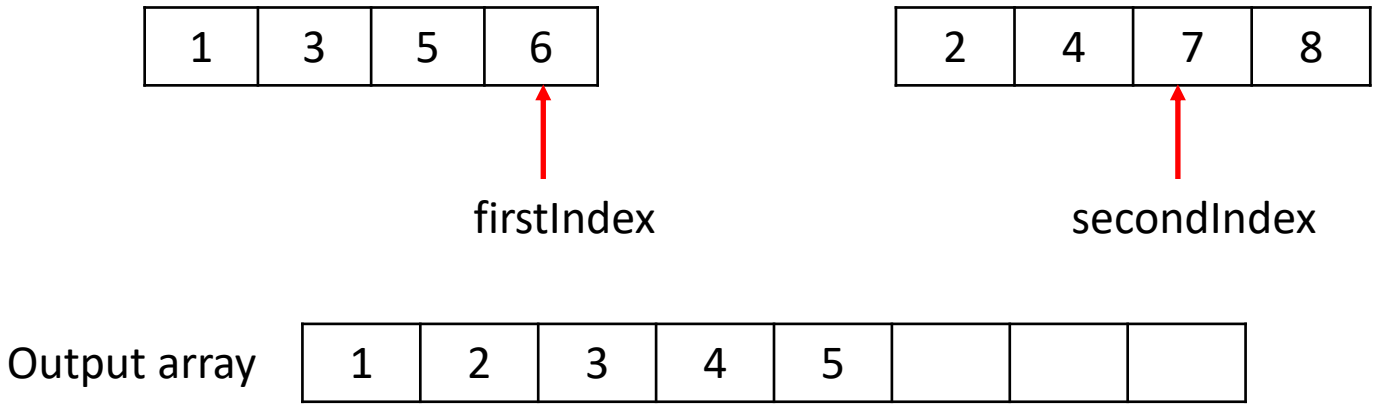
Merge Sort: Core Ideas

- 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:



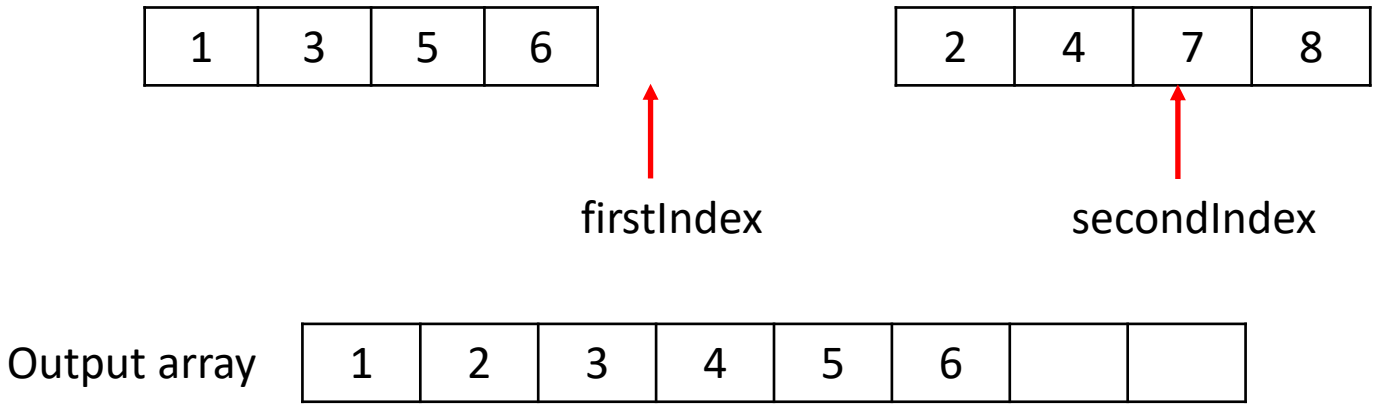
Merge Sort: Core Ideas

- 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:



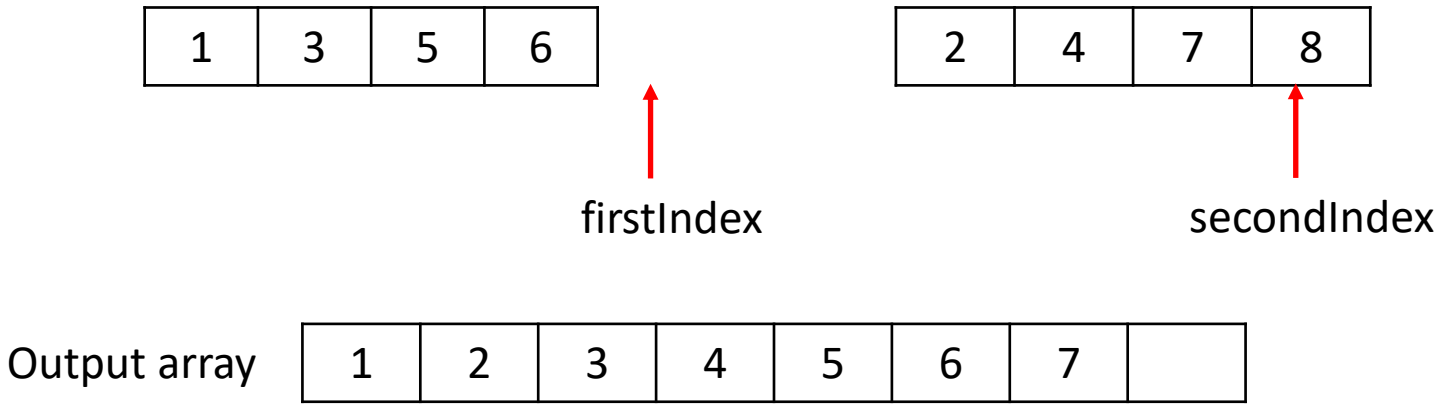
Merge Sort: Core Ideas

- 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:



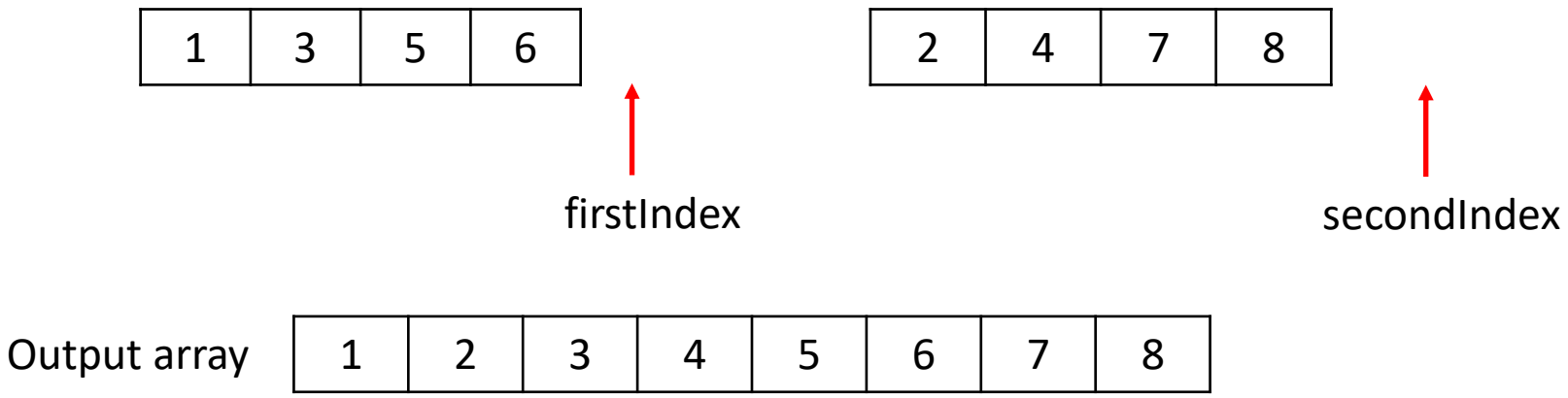
Merge Sort: Core Ideas

- 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

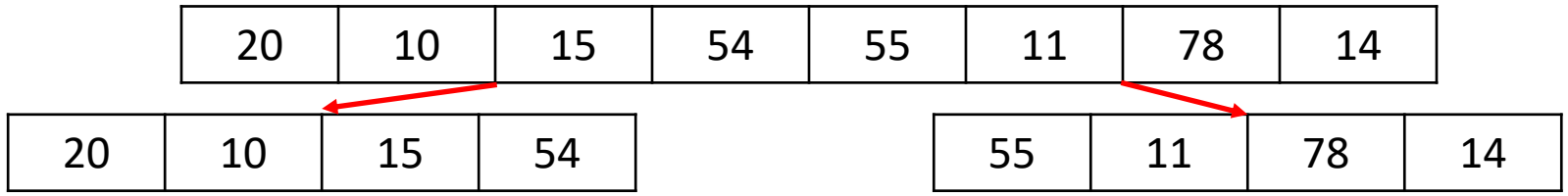
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



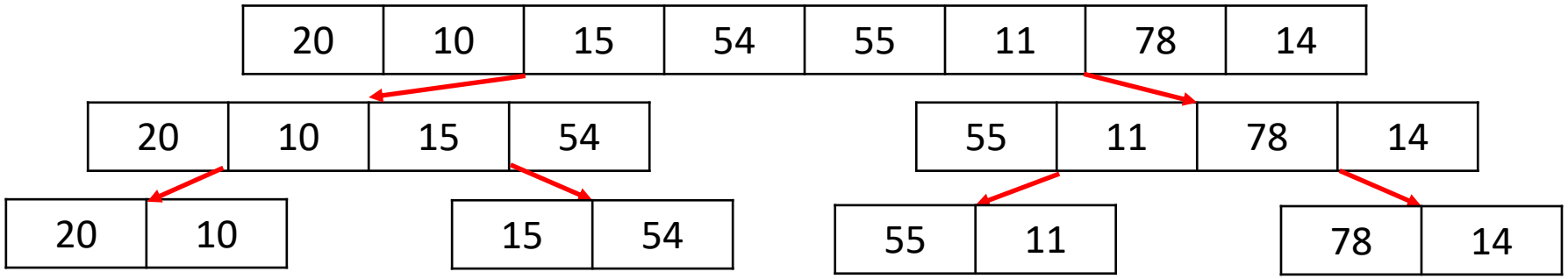
Merge Sort: High Level Example

20	10	15	54	55	11	78	14

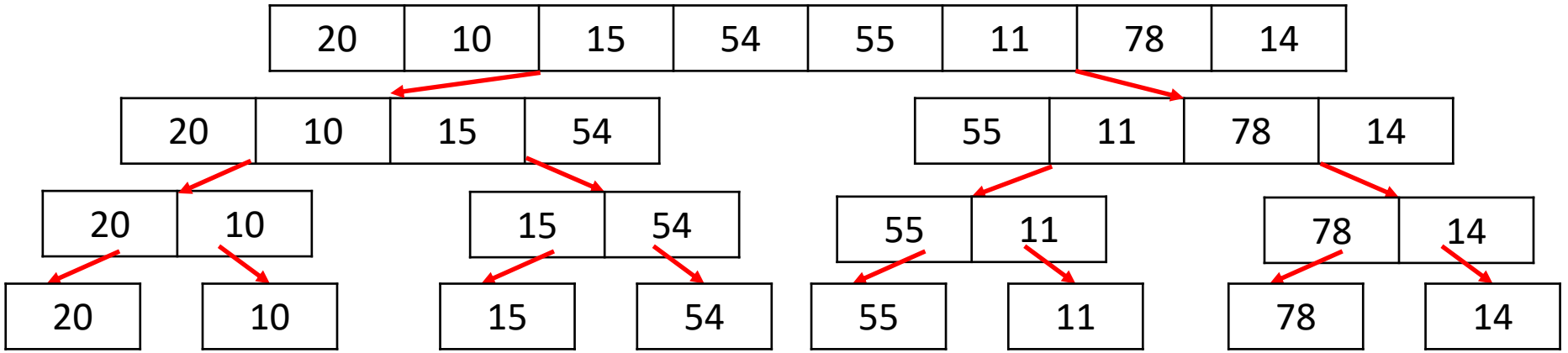
Merge Sort: High Level Example



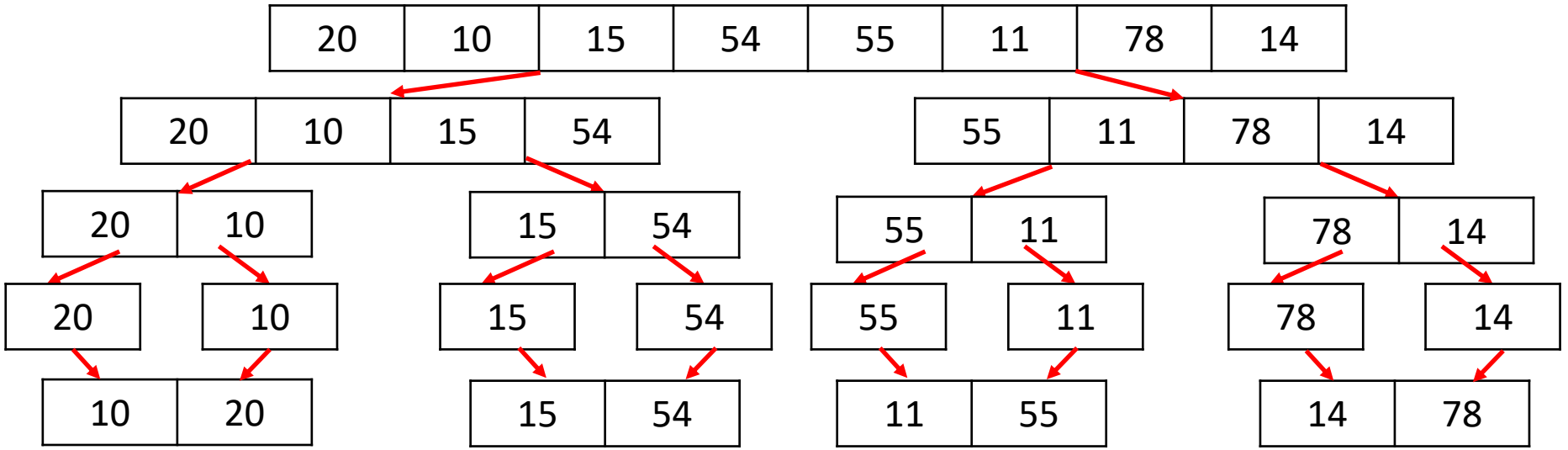
Merge Sort: High Level Example



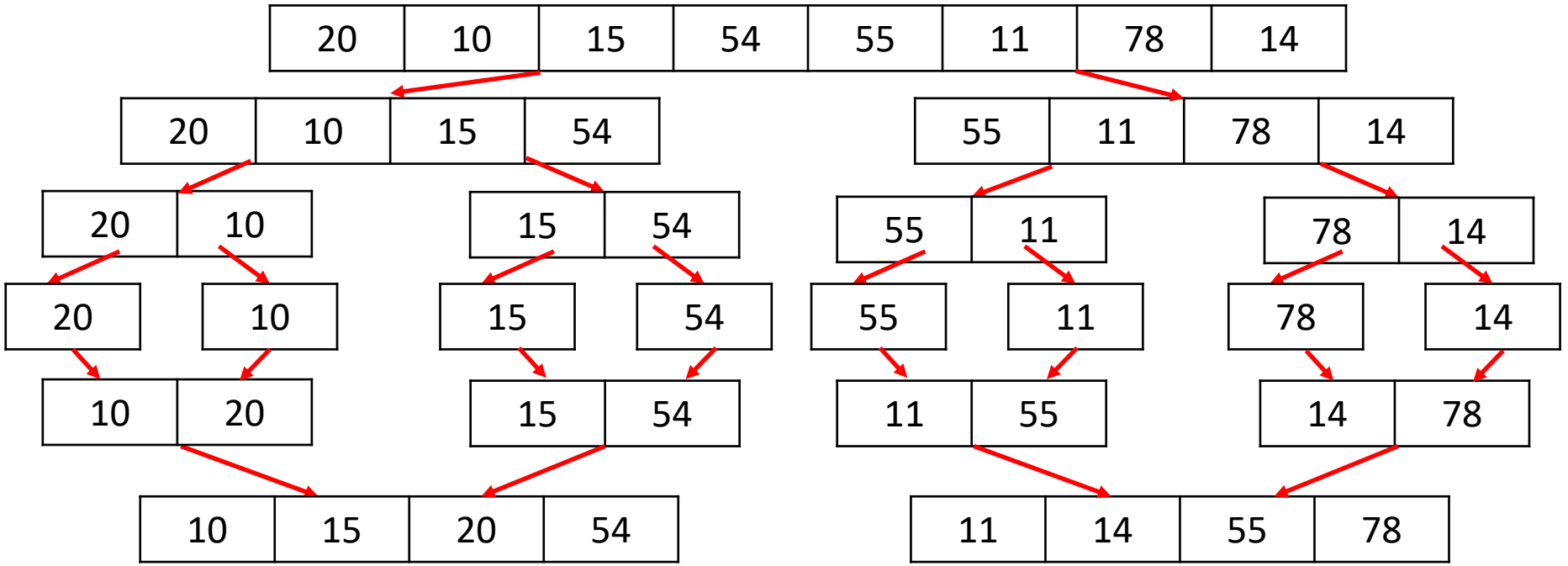
Merge Sort: High Level Example



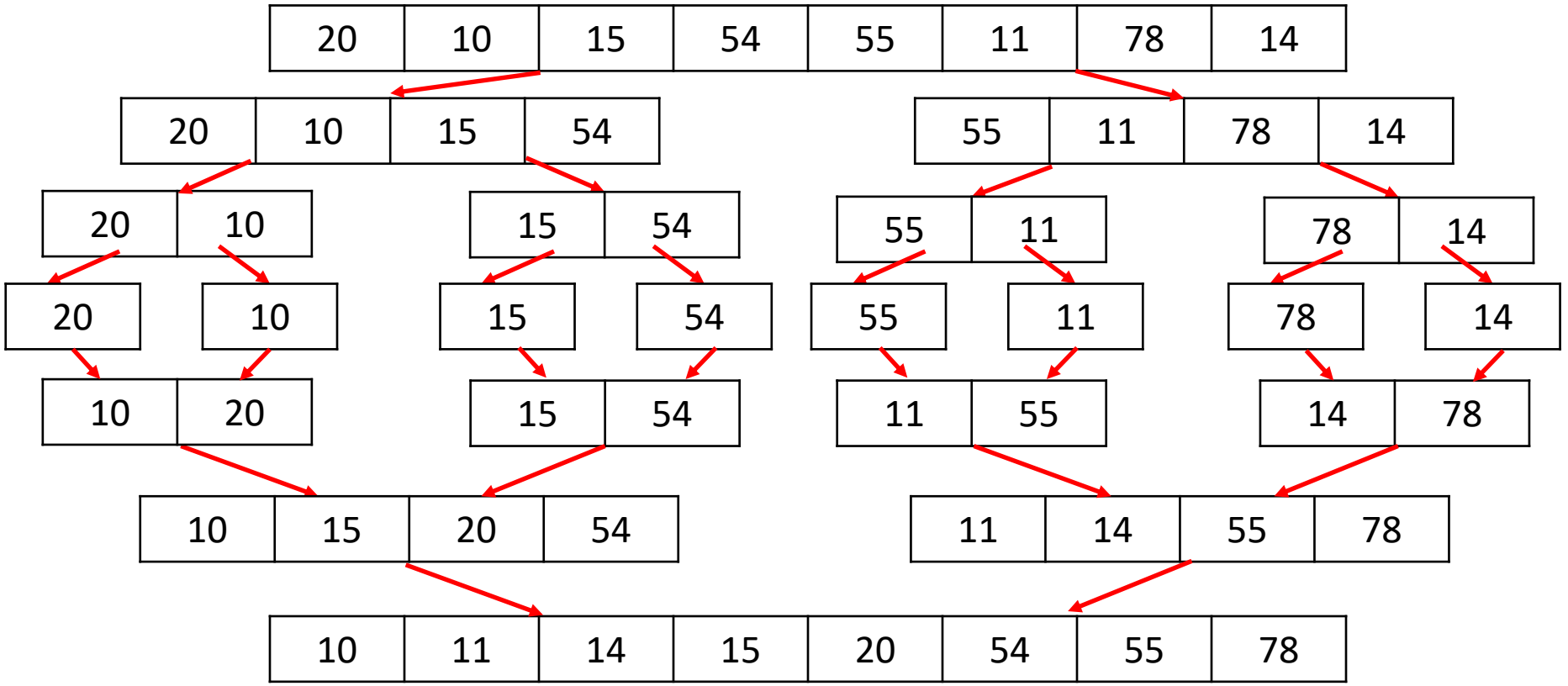
Merge Sort: High Level Example



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