Concurrency & Parallel Analysis

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Sid Sannapareddy Maya Huizar

Poll Everywhere

pollev.com/ashfujiyama

- How is PennOS going?

Administrivia

- PennOS
  - Milestone 1 posted!
    - Demo materials online (check MS1 section of pennos assignment)
  - Need to meet with your TAs next week before end of next week (7/18)
    - No late tokens – try to contact your TAs early to ensure you have a time to demo
    - Give us >= 24hr notice for any changes in meeting time

Lecture Outline

- **Producer/Consumer**
- Condition Variables
- Parallel Analysis & Amdahl's Law
- Parallel Algorithms

Synchronization So Far

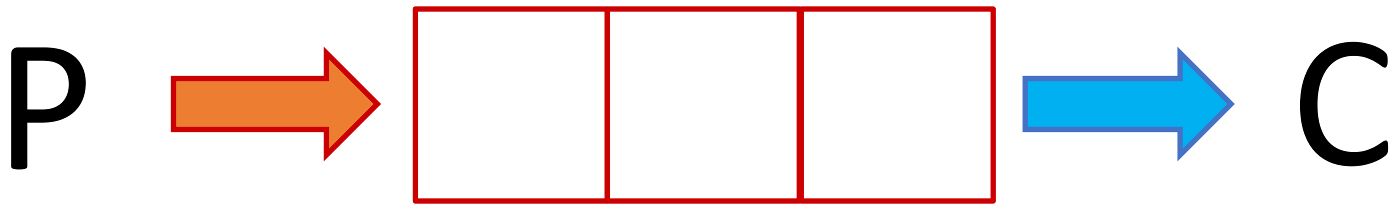
- Before, we used mutexes or disabled interrupts to make accesses to a shared data structure *indivisible*

- Example: Adding all values in an array of ints using 2 threads
  - Divide the array in half
  - First thread adds the first half of array
  - Second thread adds the second half of array
  - As long as we protect the global variable (sum), it doesn't matter which thread accesses *sum* first.

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 remove data from the shared data structure and process it

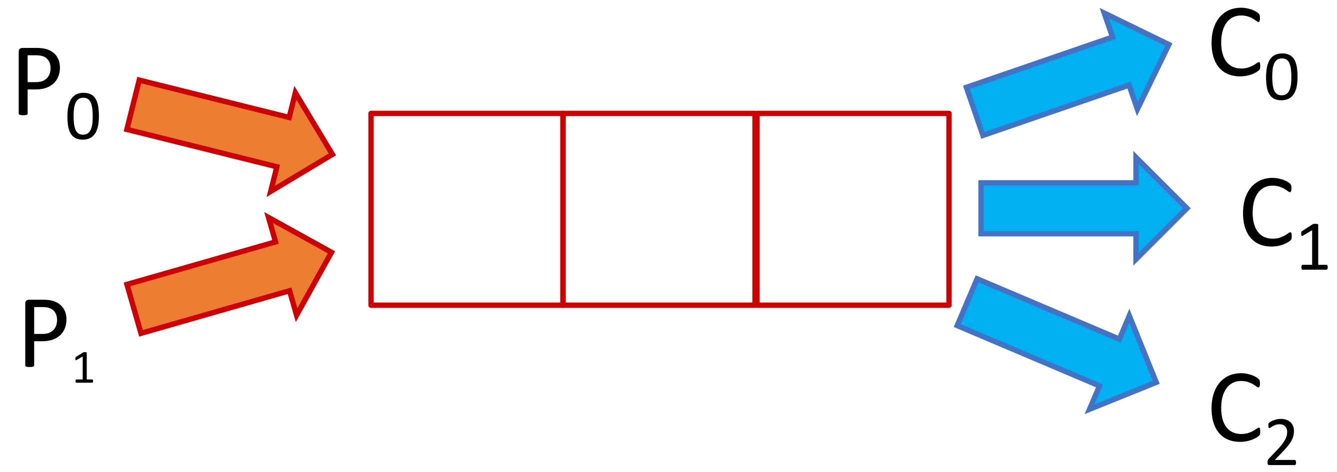
- We need to make sure that the threads play nice



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 remove data from the shared data structure and process it

- We need to make sure that the threads play nice



Poll Everywhere

pollev.com/ashfujiyama

- Does this work?
- Assume that two threads are created, one assigned to produce_thread and one assigned to consume_thread
- Assume that Vec *buf was properly initialized in `main()`

```
Vec *buf;

void* producer_thread(void *arg) {
    while (true) {
        int *random = malloc(sizeof(int));
        *random = rand();
        usleep(10000);
        vec_push_back(buf, random);
    }
}

void* consumer_thread(void *arg) {
    while (true) {
        printf("%d\n", vec_get(buf, 0));
        vec_erase(buf, 0);
    }
}
```

Poll Everywhere

pollev.com/ashfujiyama

- We now added a mutex to protect access to buf

- What's wrong? How do we fix it?
- Assume that buf and the mutex vec_lock was properly initialized in `main()`

```
Vec *buf;
pthread_mutex_t vec_lock;

void* producer_thread(void *arg) {
	while (true) {
		int *random = malloc(sizeof(int));
		*random = rand();
		pthread_mutex_lock(&vec_lock);
		vec_push_back(buf, random);
		pthread_mutex_unlock(&vec_lock);
		usleep(10000);
	}
}
void* consumer_thread(void *arg) {
	while (true) {
		pthread_mutex_lock(&vec_lock);
		while(vec_is_empty(buf)) { // do nothing
		}
		printf("%d\n", vec_get(buf, 0));
		vec_erase(buf, 0);
		pthread_mutex_unlock(&vec_lock);
	}
}
```

Poll Everywhere

discuss

- Our code is officially working, but I think there's an issue that needs to be addressed.
- What might be not ideal about this code? (Hint: inefficiency)

```
void* producer_thread(void *arg) {
    while (true) {
        int *random = malloc(sizeof(int));
        *random = rand();
        pthread_mutex_lock(&vec_lock);
        vec_push_back(buf, random);
        pthread_mutex_unlock(&vec_lock);
        usleep(10000);
    }
}
void* consumer_thread(void *arg) {
    while (true) {
        pthread_mutex_lock(&vec_lock);
        while(vec_is_empty(buf)) {
            pthread_mutex_unlock(&vec_lock);
             pthread_mutex_lock(&vec_lock);
        }
        printf("%d\n", vec_get(buf, 0));
        vec_erase(buf, 0);
        pthread_mutex_unlock(&vec_lock);
    }
}
```

Thread Communication: Naïve Solution

- In the Producer-Consumer problem, the consumer must wait for the producer to add something to the buffer

- How does the Producer Thread alert the Consumer Thread?

- Possible solution: "Spinning"
  - Infinitely loop until the producer thread notifies that the consumer thread can print
  - Use **`top`** to check CPU usage (Helpful for PennOS!)

- Alternative: Condition variables

Lecture Outline

- Producer/Consumer
- **Condition Variables**
- Parallel Analysis & Amdahl's Law
- Parallel Algorithms

Condition Variables

- Variables that allow for a thread to wait until they are notified to resume
- Avoids spinning by blocking/suspending the waiting thread
- 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

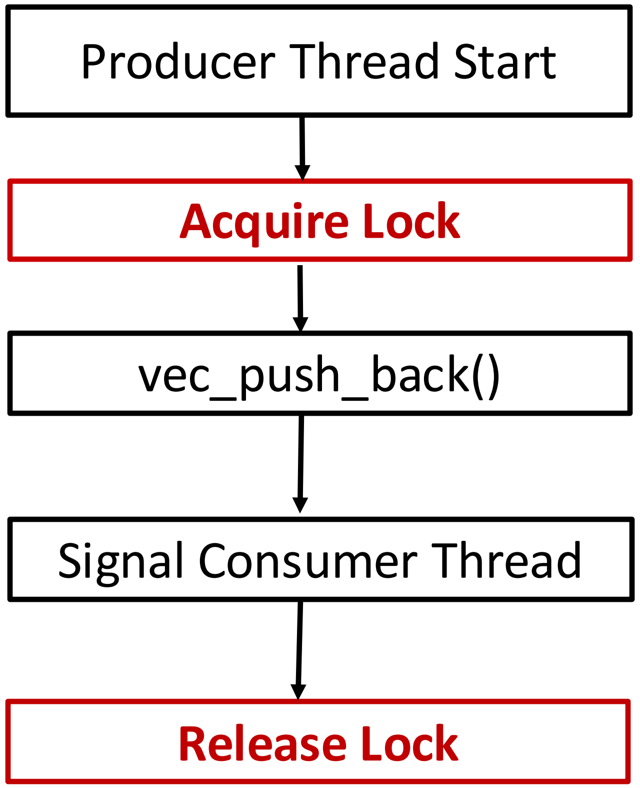
- `pthread.h` defines datatype `pthread_cond_t`

- ```
  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

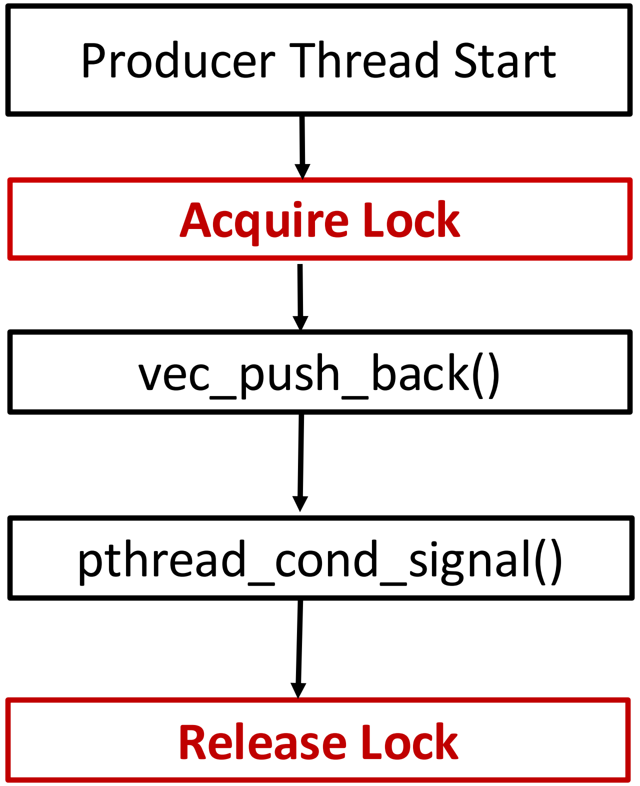
- ```
  int pthread_cond_signal(pthread_cond_t* cond);
  ```
  - Unblock at least one of the threads on the specified condition

- ```
  int pthread_cond_broadcast(pthread_cond_t* cond);
  ```
  - Unblock all threads blocked on the specified condition
  ```

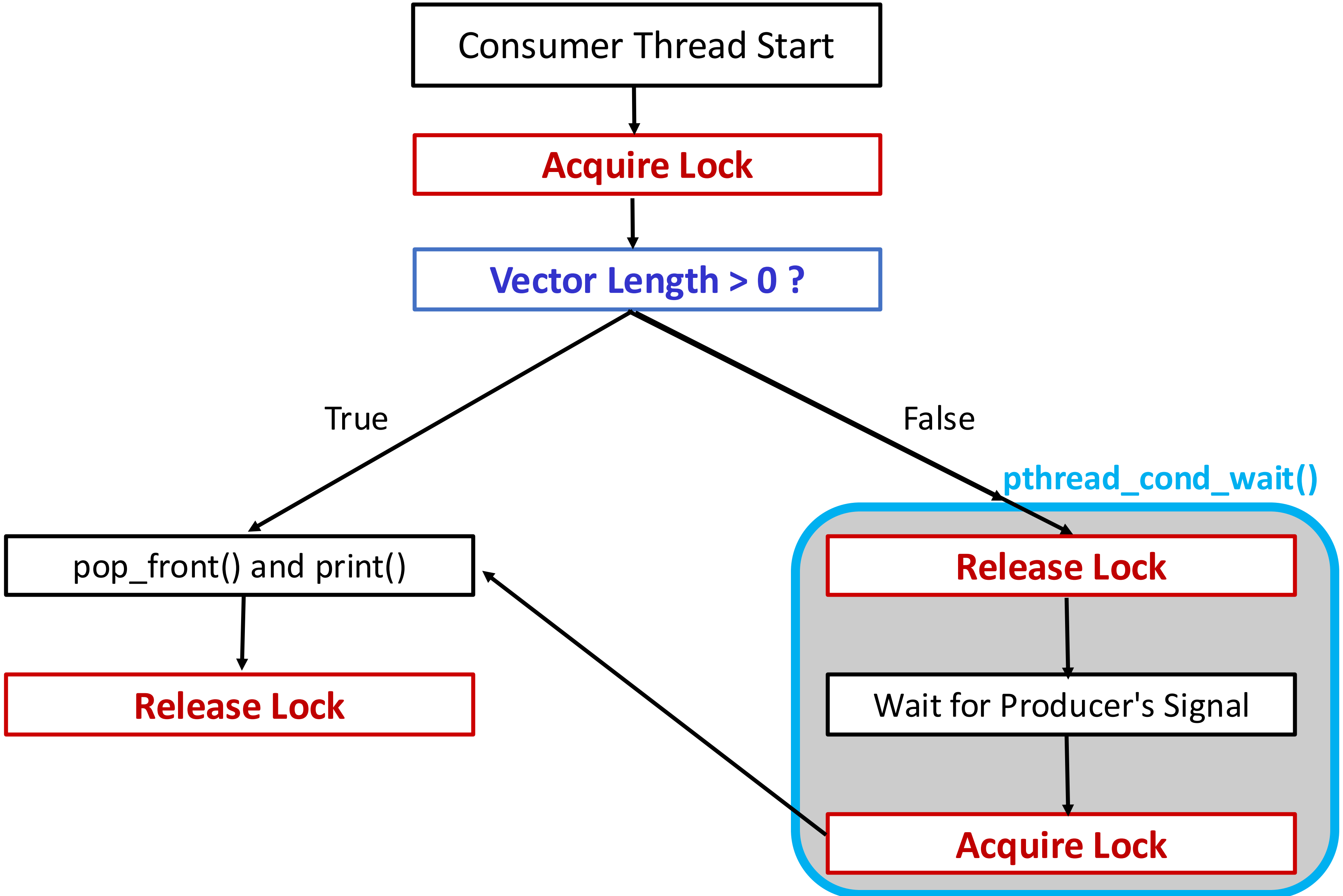
Condition Variables and the Producer (our example)



Condition Variables and the Producer(our example)



Condition Variables and the Consumer (our example)



Condition Variables: Other Considerations

- In our example, we had an "unlimited buffer"
- What else would we need to handle if our buffer was an array (fixed size)?
- What if we had multiple producers and/or consumers?

Multiple Consumers

- Situation: one producer and two consumers sharing 1 vector
  - Producer pushes values onto the vector
  - Consumer removes values from the vector

- Producer and Consumer function implementation is the same except we use pthread_cond_broadcast() to send a signal to wake up both consumer threads

Poll Everywhere

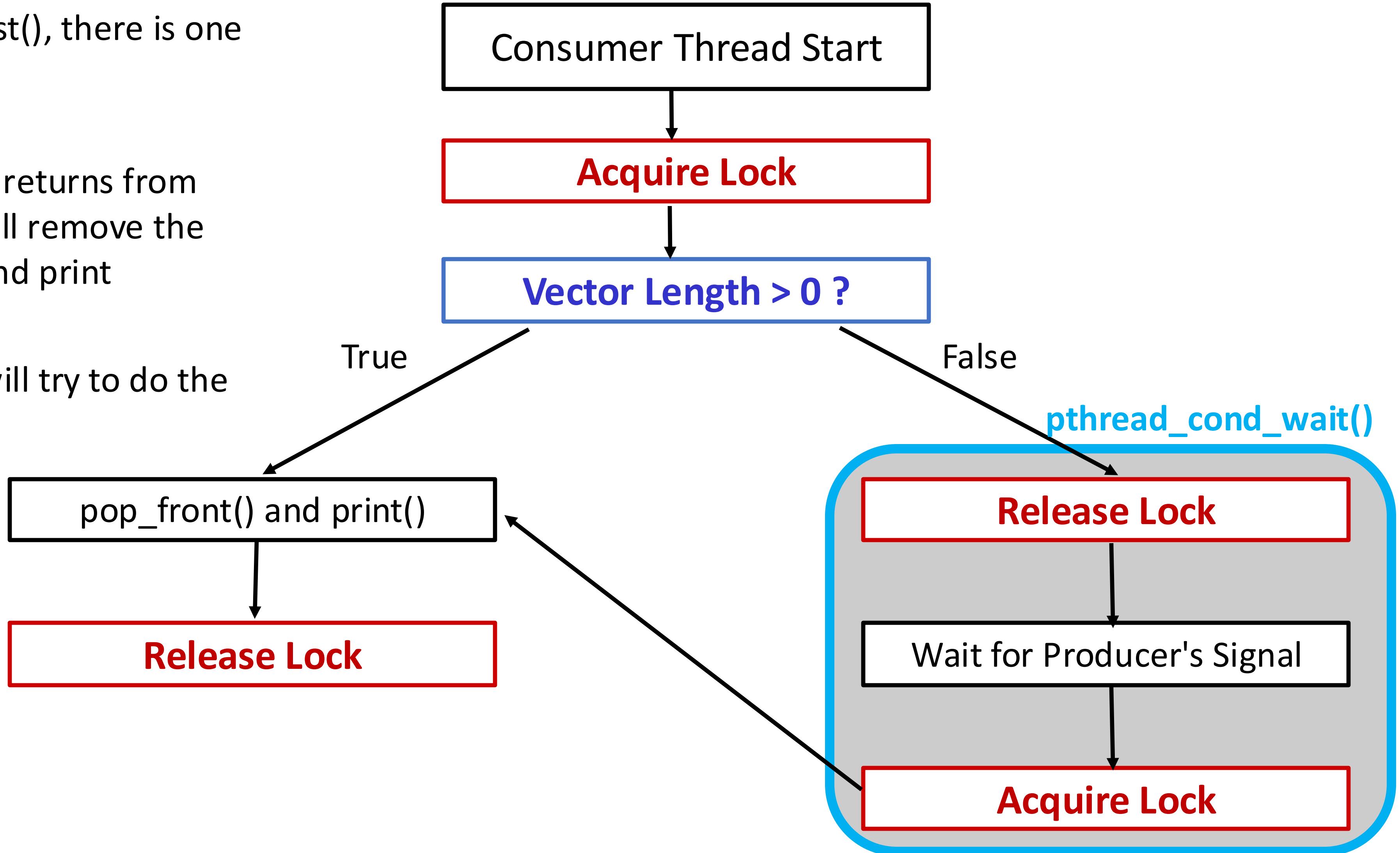
pollev.com/ashfujiyama

- Does this work if there was 1 producer and 2 consumer threads?

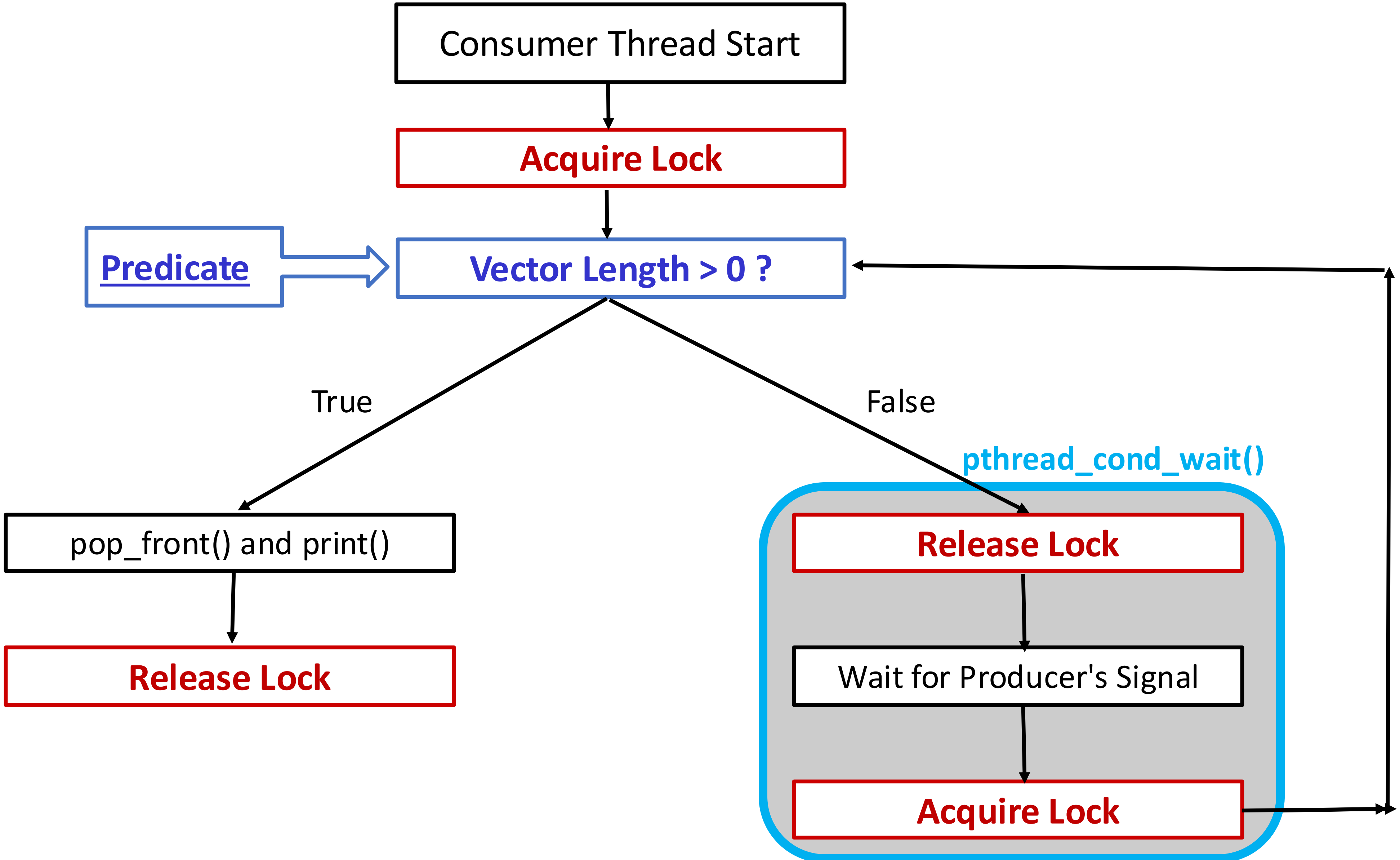
```
void* consumer_thread(void *arg) {
    while (true) {
        pthread_mutex_lock(&vec_lock);
        if(vec_is_empty(buf)) {
            pthread_cond_wait(&vec_cond, &vec_lock);
        }
        // at this point, we have the lock
        // print first element, then delete
        printf("%d\n", *(int*)vec_get(buf, 0));
        vec_erase(buf, 0);
        pthread_mutex_unlock(&vec_lock);
    }
    return NULL;
}
```

Multiple Consumers – What Happens?

- When producer calls pthread_cond_broadcast(), there is one value inside the vector
- The consumer who first returns from pthread_cond_wait() will remove the value from the vector and print
- The second consumer will try to do the same, and then panic!



Multiple Consumers Solution



Spurious Wakeups

- It's possible that when a thread wakes up due to pthread_cond_signal() or pthread_cond_broadcast(), that the condition it originally waited for is not satisfied at the time of wakeup

- This is known as a "spurious wakeup," and it creates a race condition
  - If you have two threads that received the broadcast signal, one thread "wins" and the other experiences the spurious wakeup

- This is why we have to check the predicate condition after pthread_cond_wait() returns

Lecture Outline

- Producer/Consumer
- Condition Variables
- **Parallel Analysis & Amdahl's Law**
- Parallel Algorithms

Why Would We Write Multithreaded Code?

- Make the program run faster
- Handle multiple tasks at the same time
- That's it.

Why Wouldn't We Want to Write Multithreaded Code?

- Guaranteed complexity
  - Takes longer to develop than single-thread code
  - Difficult to read and maintain

- May not give us the speedup we desire
  - Speedup could be a negligible difference (or sometimes slower!)
  - Cost benefit analysis: development time versus running time

- Especially not worth it when:
  - Functions are fast (light computation)
  - Data structures are not big

Limitations of Parallelization

- Hardware limits:
  - Number of hardware threads
  - Number of cores

- Memory layout may be bad -> frequent cache misses
  - Runtime more dependent on I/O than CPU

- Thread overhead contributes to the percentage of sequential code
  - More sequential code runtime = less time spent running parallel code

Good Practice: make it work first, figure out optimizations later

Amdahl's Law

- How much speedup if we optimize a portion of the code?

- Speedup = $\dfrac{1}{(1-P) + \frac{P}{N}}$

  - P = percent of *runtime* spent on parallelized code
  - N = number of threads
  - If speedup = 2, then the parallelized version of the code is 2x faster than the original code

Amdahl's Law

- Total runtime of program = 1
- Total runtime of program = Parallel + Sequential = P + (1 – P)
- On a single thread: Speedup = 1 / ((1 – 0) + 0 / 1) = 1

How Fast?

- How much speedup will we experience when we use
  - 4 threads on 50% parallelized code? **1.6 times**
  - 1,000,000 threads on 50% parallelized code? **1.999998 times**

How Fast?

- How much speedup will we experience when we use
  - 4 threads on 50% parallelized code? **1.6 times**
  - 1,000,000 threads on 50% parallelized code? **1.999998 times**
  - 4 threads on 90% parallelized code? **3.1 times**
  - 1,000,000 threads on 90% parallelized code? **9.9999 times**

Amdahl's Limit

- Recall: Speedup = $\frac{1}{(1-P) + \frac{P}{N}}$
- As the number of threads (N) goes up, P/N approaches 0
- Then, speedup becomes dependent on the percentage of sequential execution
- Impossible to have 100% parallelized code

Lecture Outline

- Producer/Consumer
- Condition Variables
- Parallel Analysis & Amdahl's Law
- **Parallel Algorithms**

Parallel Algorithms

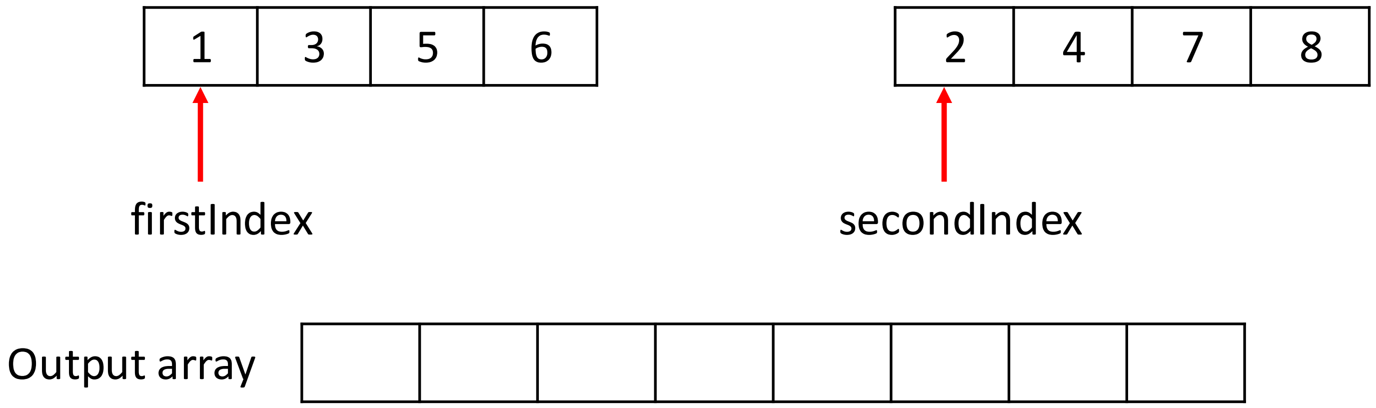
- One interesting applications of threads is for faster algorithms

- Common Example: Merge sort

Merge Sort: Core Ideas

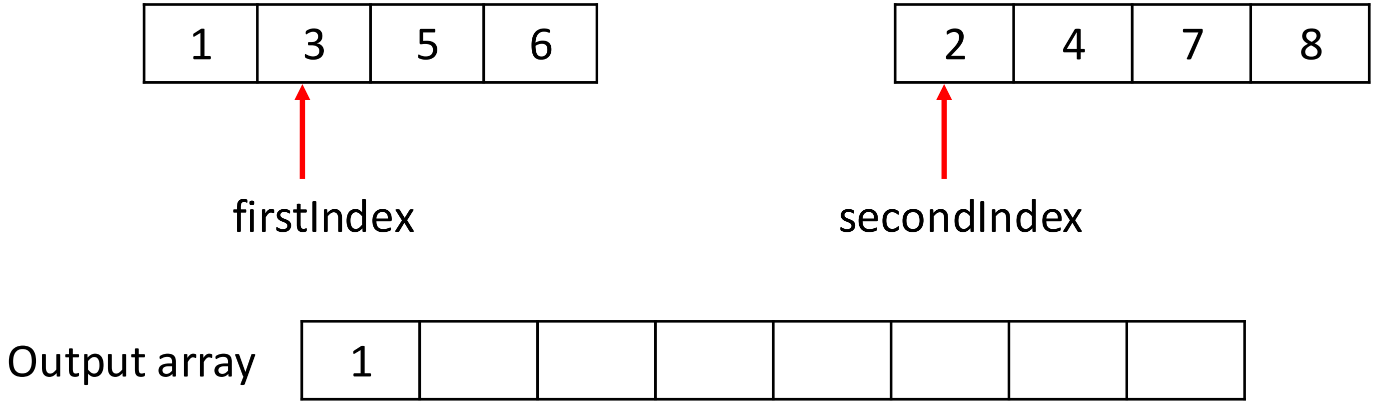
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



Merge Sort: Core Ideas

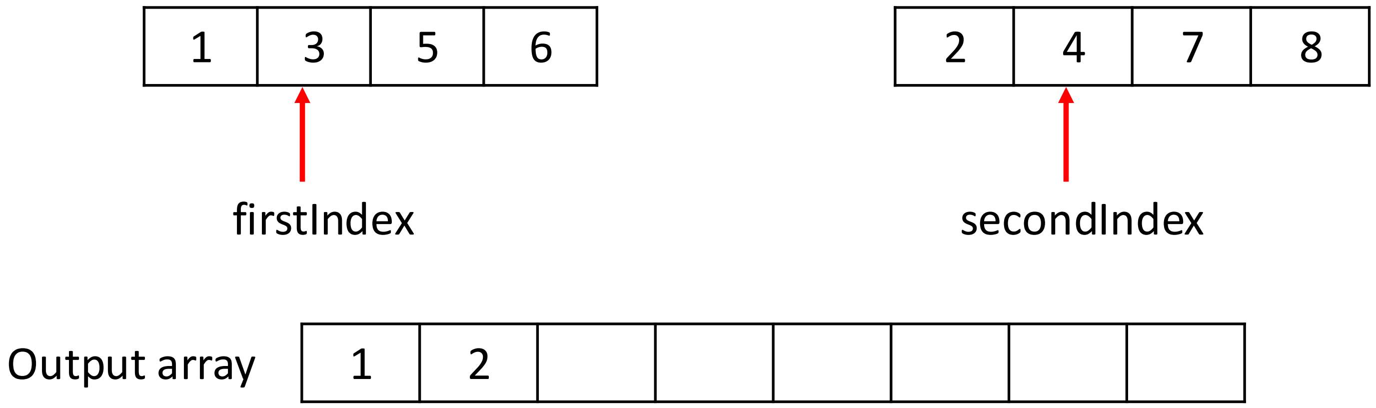
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



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

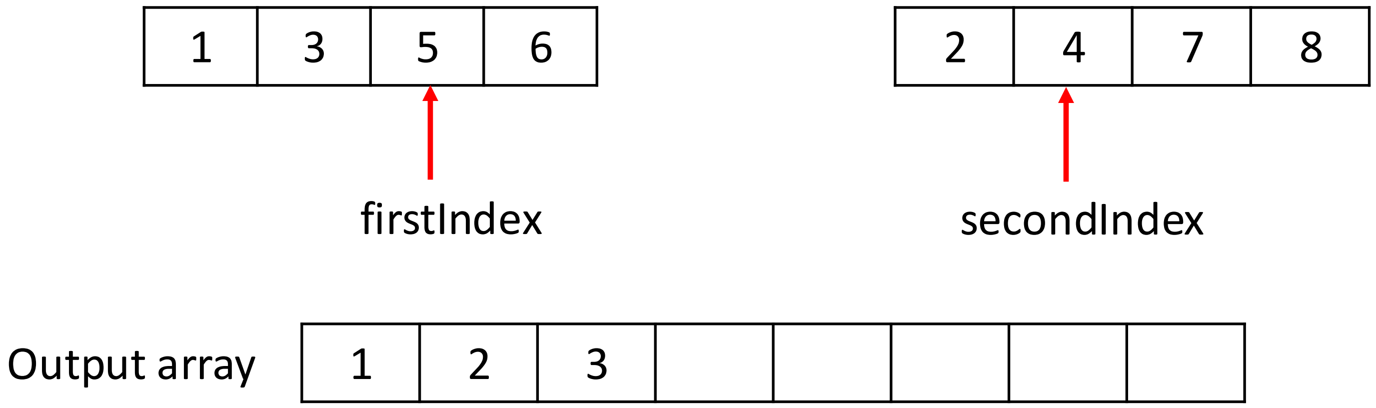
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

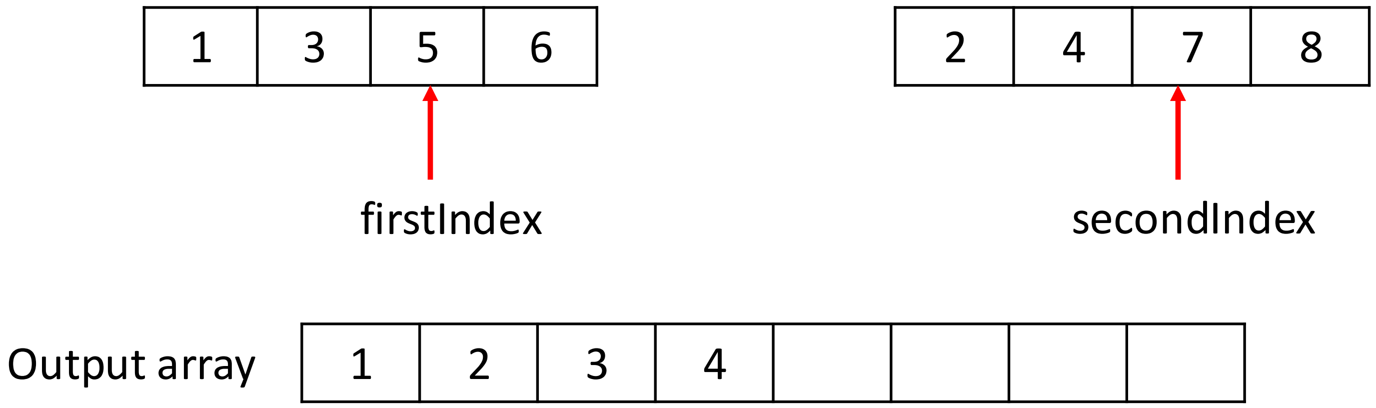
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



Merge Sort: Core Ideas

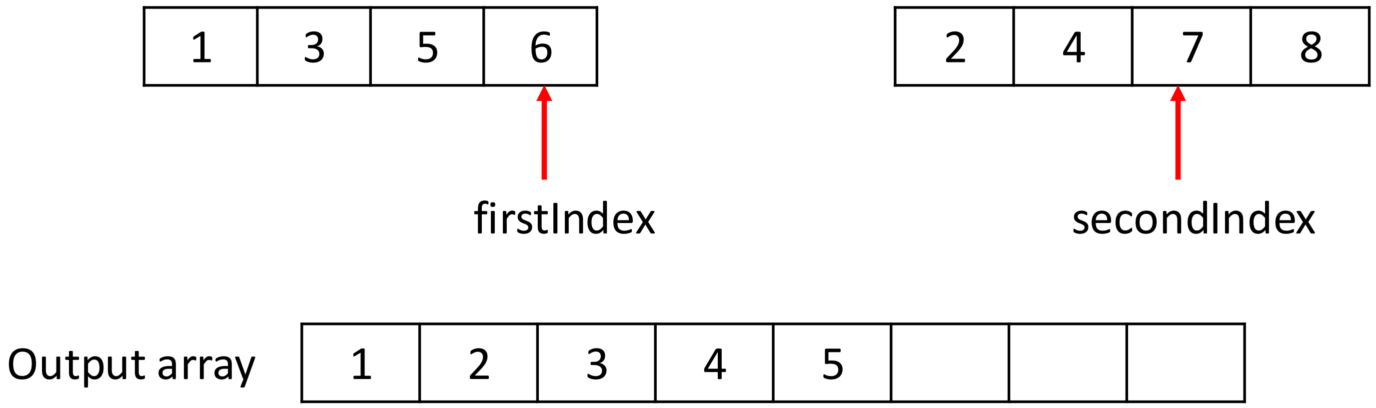
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



Merge Sort: Core Ideas

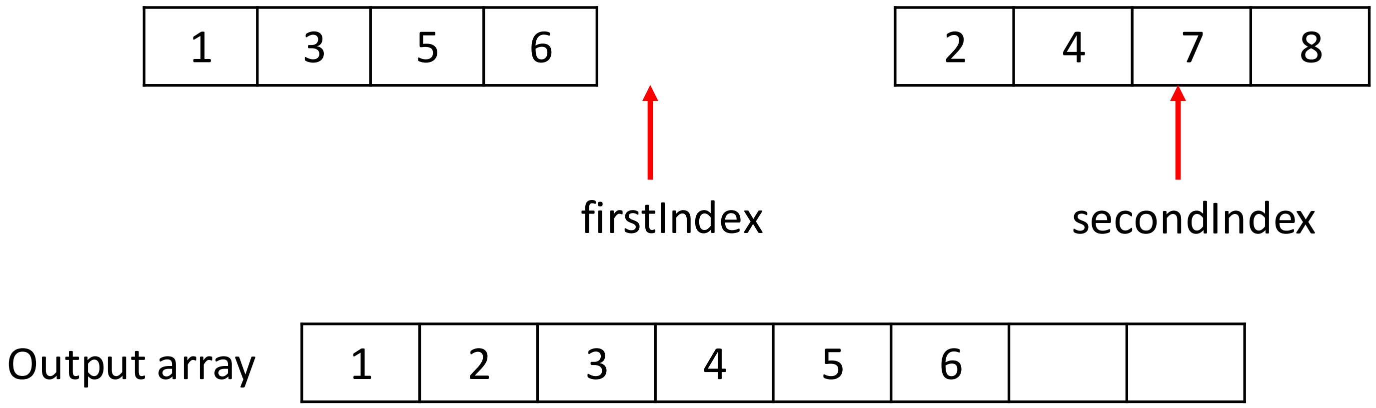
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array

  - Consider the two sorted arrays:



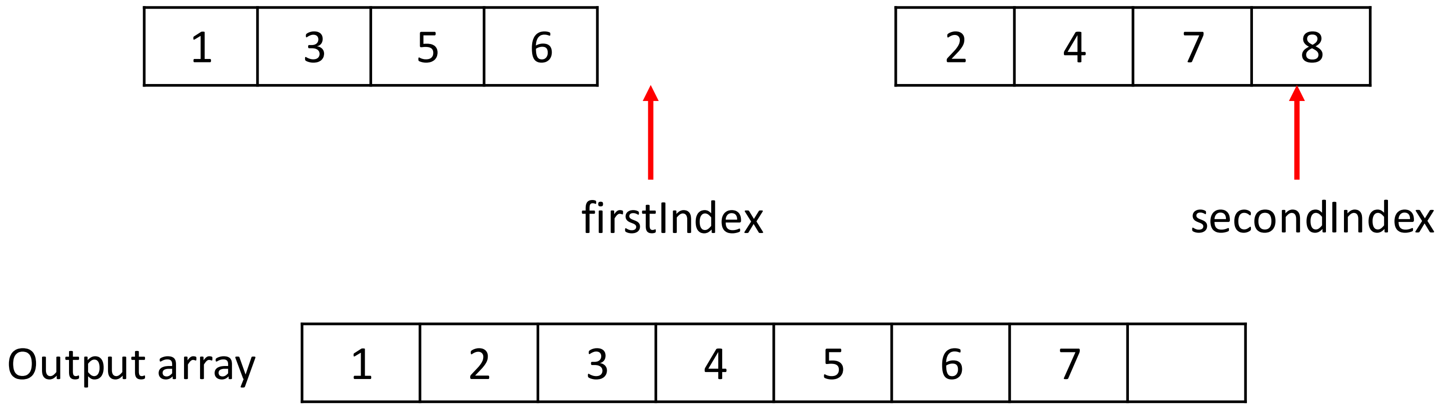
Merge Sort: Core Ideas

- 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

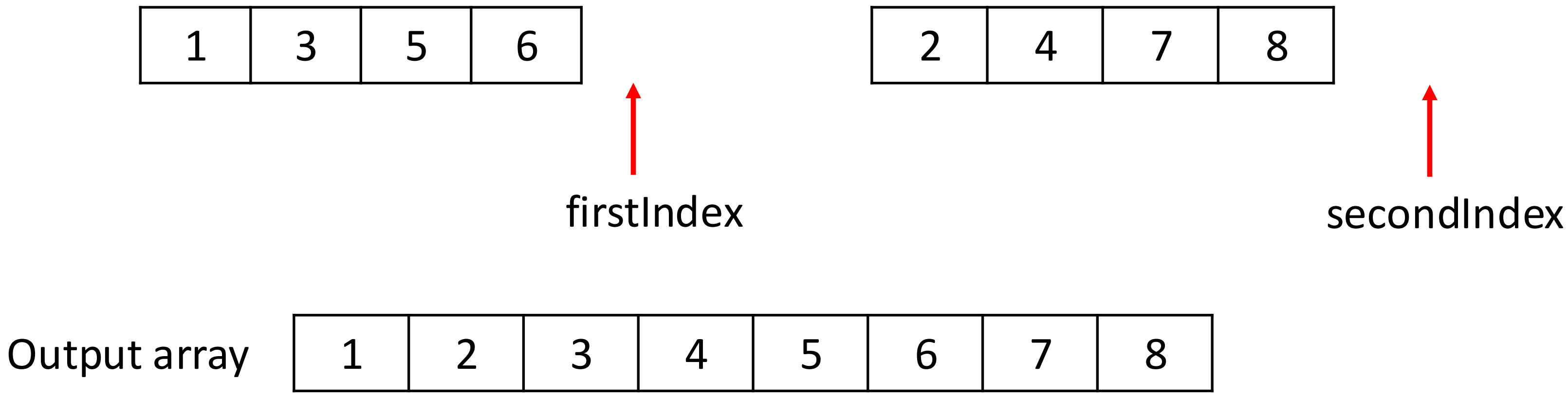
- It is easier to sort small arrays than big arrays
- It is quicker to merge two sorted arrays than sort an unsorted array
  - Consider the two sorted arrays:



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

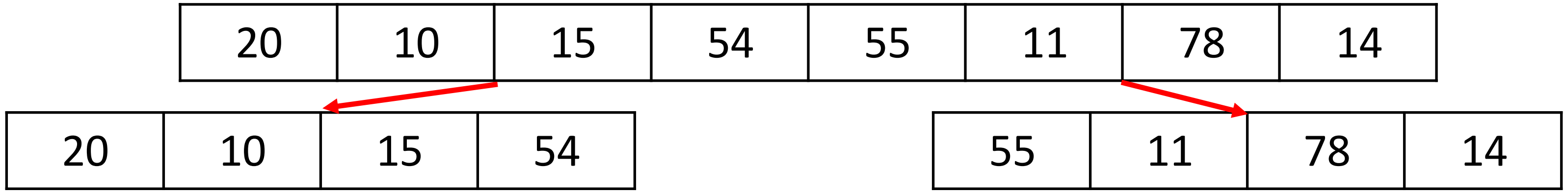
  - Consider the two sorted arrays:



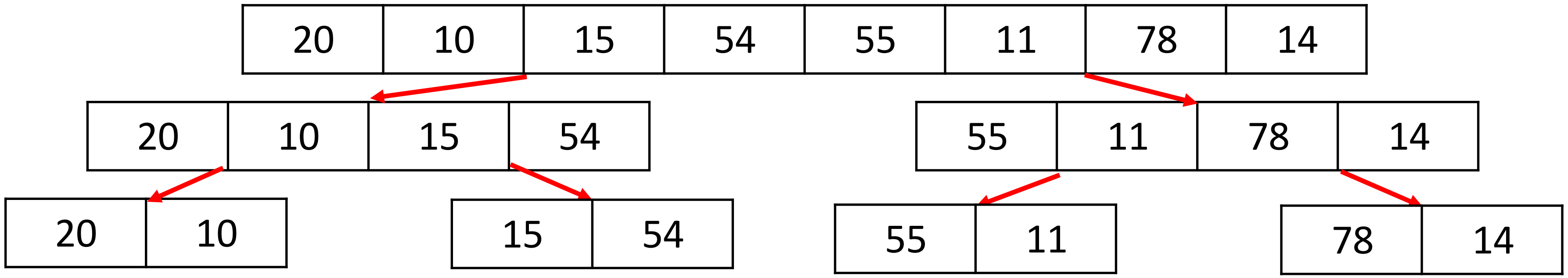
Merge Sort: High Level Example

20	10	15	54	55	11	78	14

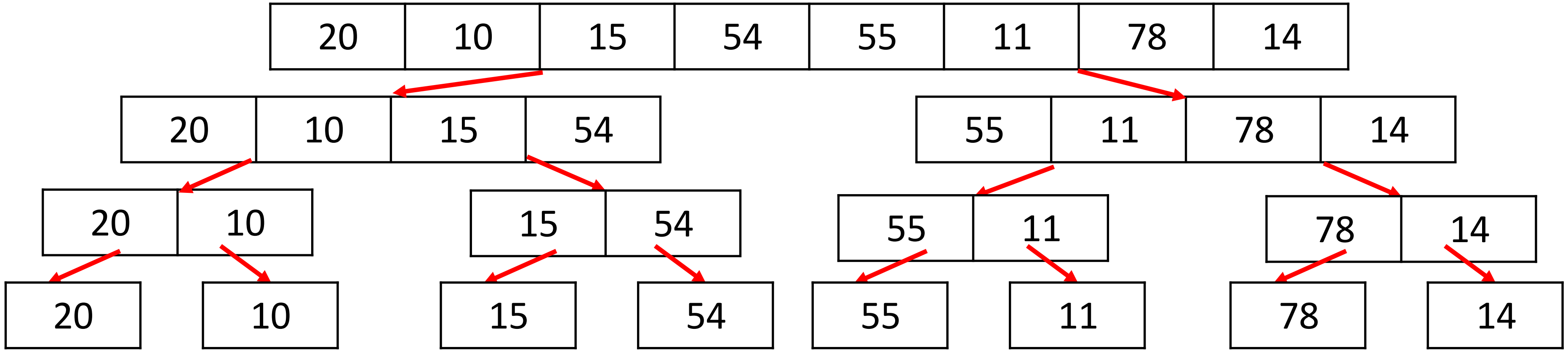
Merge Sort: High Level Example



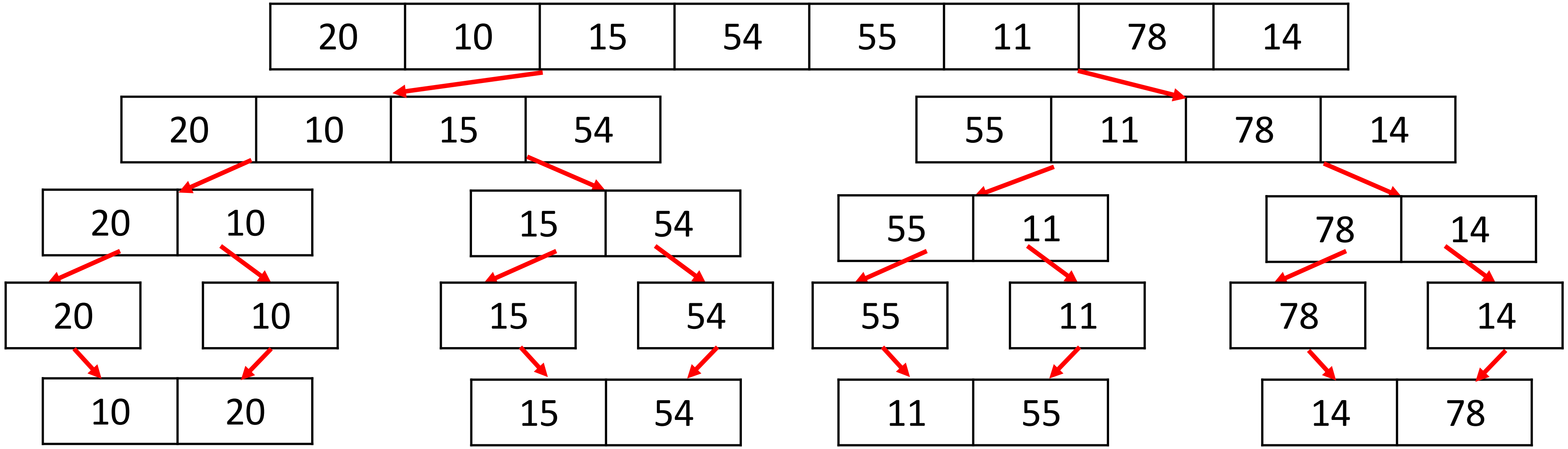
Merge Sort: High Level Example



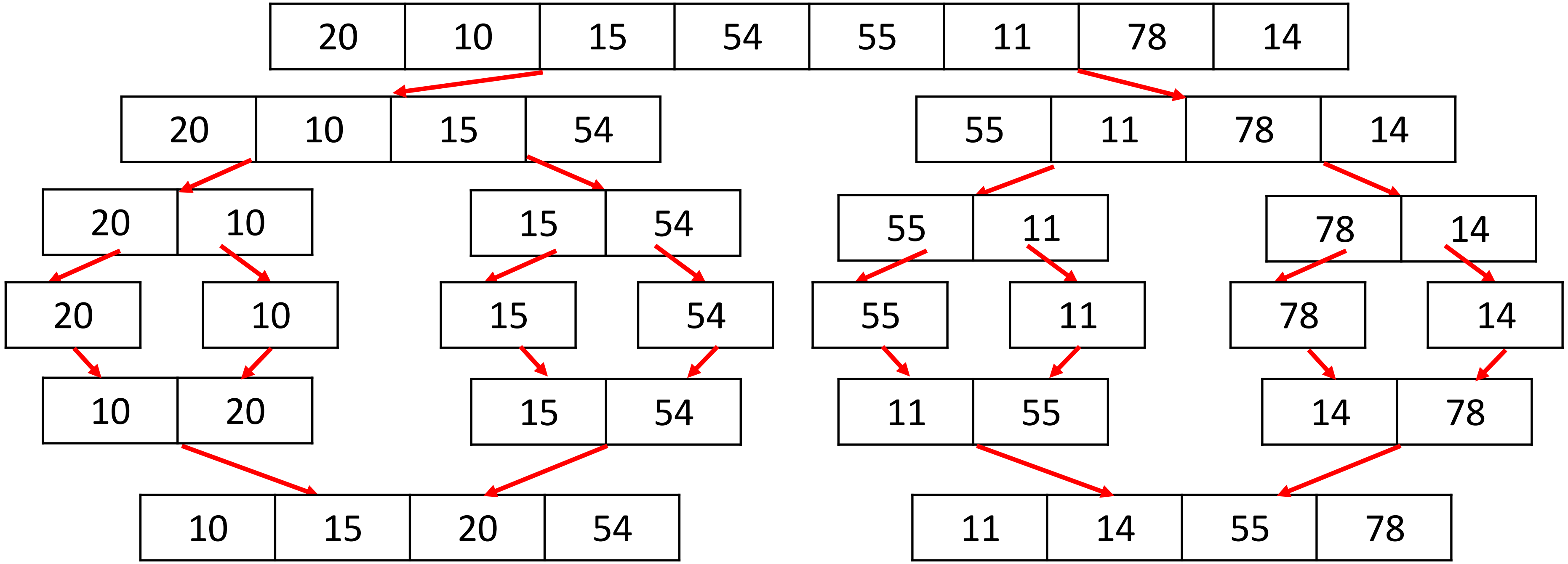
Merge Sort: High Level Example



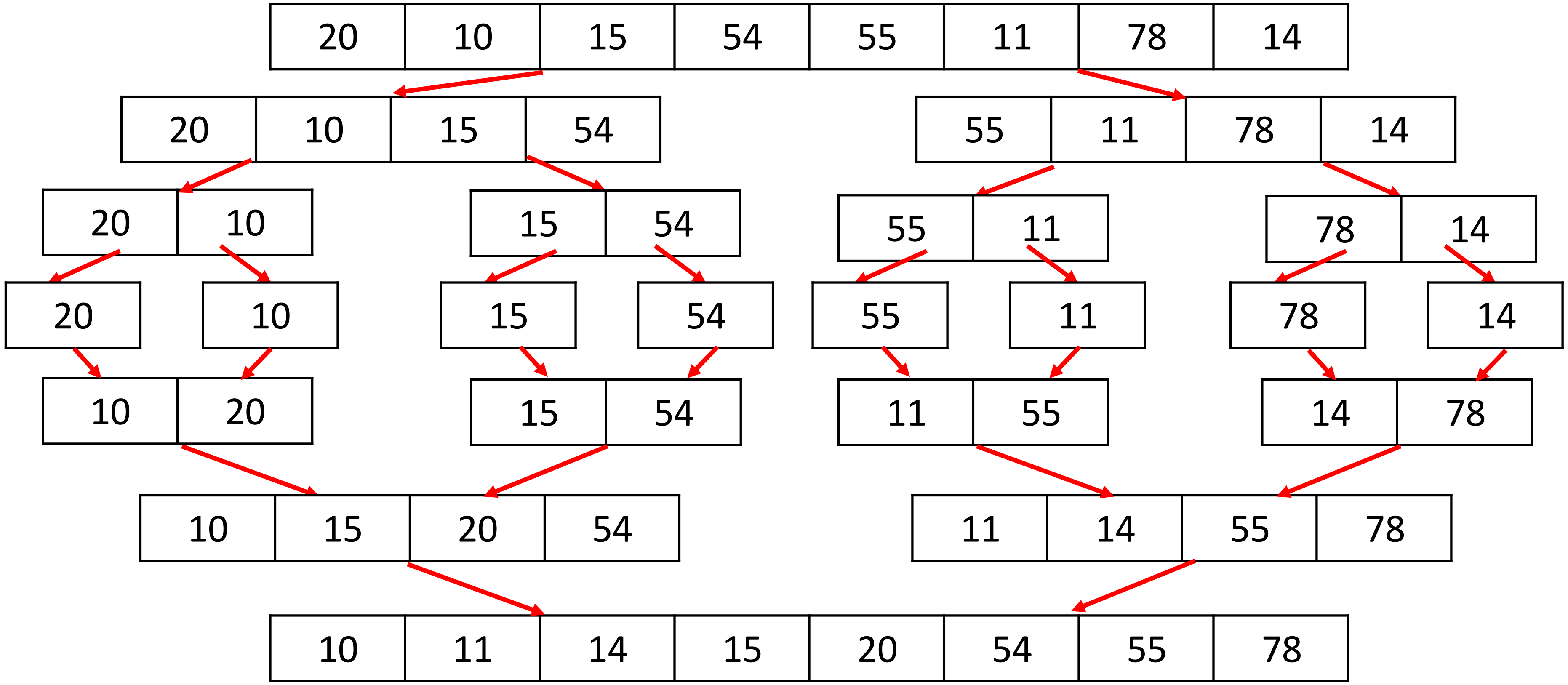
Merge Sort: High Level Example



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 $T(n) = \sim 2 * O(n) = O(n)$
  - This parallel merge sort is $O(n)$, but there are further optimizations that can be done to reach $\sim O(\log(n))$
- There is a lot more to parallel algo analysis than just this, I am just giving you a sneak peek