Cornell Bowers C·IS
Computer Science

CS3410: Computer Systems and Organization

LEC25: Parallel Programming

Dr. Kevin Laeufer
Monday, November 24, 2025

Credits: Alvisi, Bala, Bracy, Garcia, Guidi, Kao, Laeufer, Martin, McKee, Sampson, Sirer, Susag, Van Renesse, Weatherspoon

Plan for today.

- POSIX threads library
- Analyzing parallel workloads
- Sheduling work across threads
- Producer / consumer parallelism
- Circular ring buffer
- Condition variables
- Amdahl's law
- Deadlocks

POSIX Threads (pthreads)

- Implementations for thread management and synchronization operations
- pthread.h
- Must provide `-lpthread` option
- Works on Linux and MacOS
- Use C standard `threads.h` if you want to target Windows as well.

Spawn & Join Threads

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

Pointer to `pthread_t` struct

Function pointer to the code to run in the thread

Function argument

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Thread to wait for

Return value *location*

Demo #1: Spawn & Join Threads

```
#include <stdio.h>
#include <pthread.h>
void* my_thread(void* arg) {
    printf("A");
    return nullptr;
}
int main() {
    printf("B");
    pthread_t thread;
    pthread_create(&thread, nullptr,
                   my_thread, nullptr);
    pthread_join(thread, nullptr);
    printf("C");
    return 0;
}
```

PollEv.com/cs3410

What does the program print?

Demo #2: Spawn & Join Threads

```
#include <stdio.h>
#include <pthread.h>
void* my_thread(void* arg) {
    printf("A");
    return nullptr;
}
int main() {
    printf("B");
    pthread_t thread;
    pthread_create(&thread, nullptr,
                   my_thread, nullptr);
    // pthread_join(thread, nullptr)
    printf("C");
    return 0;
}
```

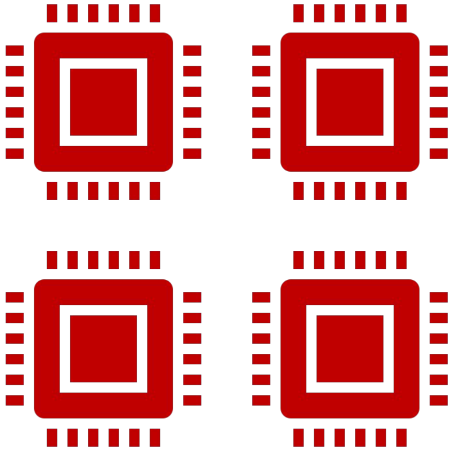
PollEv.com/cs3410

What does the program print?

Lec 23: Threads for Compute Performance

- Without threads, a single process can only run on a single CPU core.
- Each thread can run on a different CPU core.
- If compute and not I/O is the main concern, this is the rule of thumb: Use **as many threads as CPU cores** to optimally balance thread overhead and compute throughput.

Case Study: Computing Primes



```
bool is_prime(int n) {
 for (int i = 2; i < n; ++i) {
   if (n % i == 0) { return false; }
 }
 return true;
}

// ...
static const int NUMBERS = 1024;

for(int i = 1; i < NUMBERS; i += 1) {
 is_prime(i);
}
```

- `is_prime` does not need to access memory → fully *compute bound*
- no dependencies across loop iterations → *embarrassingly parallel*
- Compute on 4 CPU cores → How should work be distributed for maximum speedup?

One Thread per Number

```
// We'll set `prime[i]` to true iff `i` is prime.
bool prime[NUMBERS];
// Launch a thread to check every number.
pthread_t threads[NUMBERS]; ta t_args[NUMBERS];
for(int i = 1; i < NUMBERS; i += 1) {
    t_args[i] = (ta){.i = i };
    pthread_create(&threads[i], nullptr,
                   prime_thread, &t_args[i]);
}
for (int i = 1; i < NUMBERS; ++i) {
 pthread_join(threads[i], NULL);
}
```

- Each thread is assigned a number `i` and stores the result in the global array `prime`
- Performance for `NUMBERS = 1024 * 64`
- `hyperfine -N ./a.out`

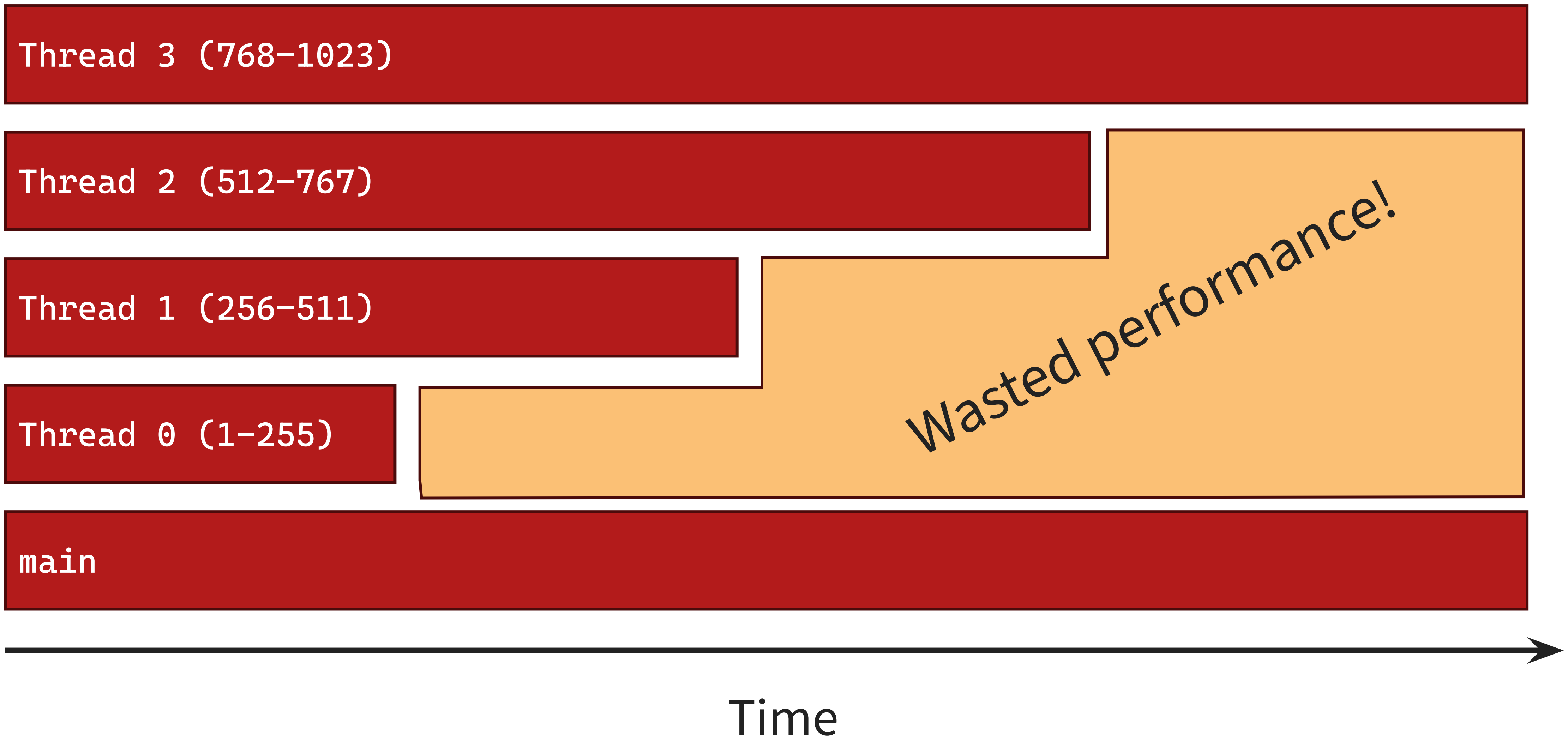
Equal Split with POSIX Threads

```
int numbers_per_thread = NUMBERS / THREADS;
for (int i = 0; i < THREADS; i += 1) {
 t_args[i] = (ta){
    .start = i == 0 ? 1 : i * numbers_per_thread,
    .end = (i + 1) * numbers_per_thread,
  };
  pthread_create(&threads[i], nullptr,
                 prime_thread, &t_args[i]);
}
for (int i = 1; i < THREADS; i += 1) {
 pthread_join(threads[i], NULL);
}
```

- <u>One thread per core</u>.
- Each thread is assigned a range `start` to `end` and stores the result in the global array `prime`
- Performance for `NUMBERS = 1024 * 64`
- `hyperfine -N ./a.out`

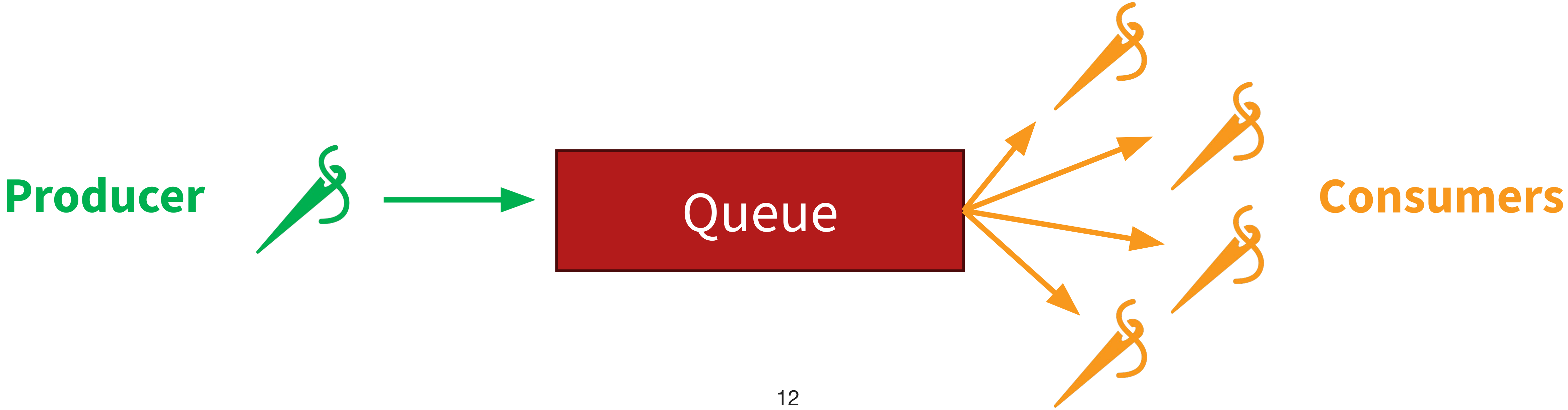
*not to scale

Equal Split Leads to *Thread Imbalance*



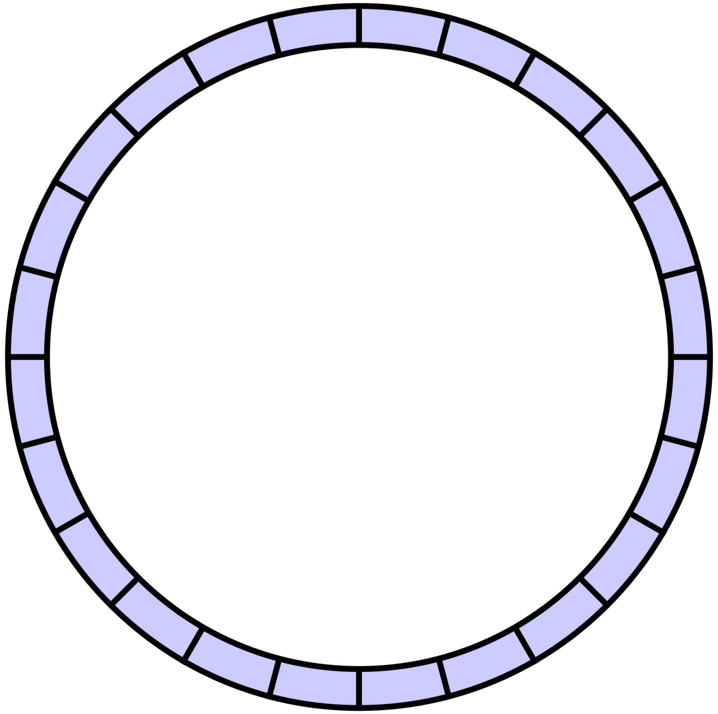
Producer/Consumer Technique

- Automatic technique to deal with imbalances
- **Key Idea:** One thread ***produces*** work to do while *n* parallel threads **consume** the work items (and do the work)
- **Better Balancing:** Threads dynamically acquire pieces of work
- **Downside:** Higher synchronization overhead.



Queue Implementation: Circular (Ring) Buffer

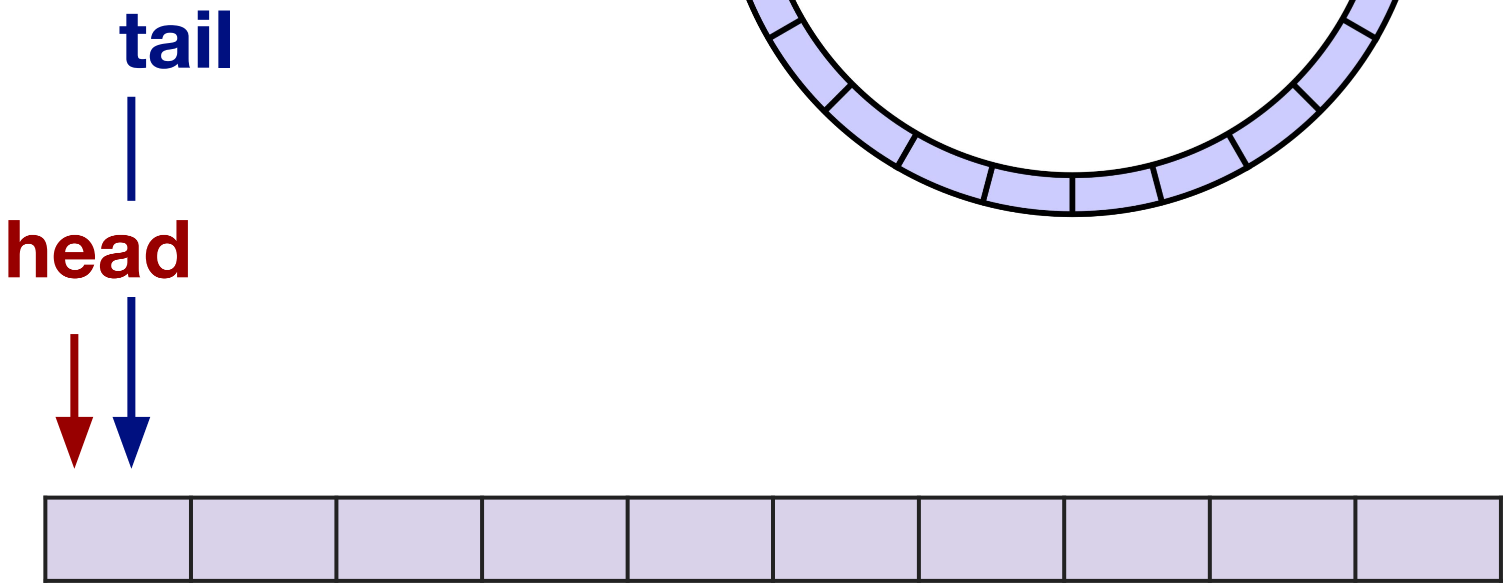
Allocate an array of *n* elements



Keep track of two indices: head & tail

Producers **push** to the tail

Consumers **pop** from the head



Queue Implementation: Circular (Ring) Buffer

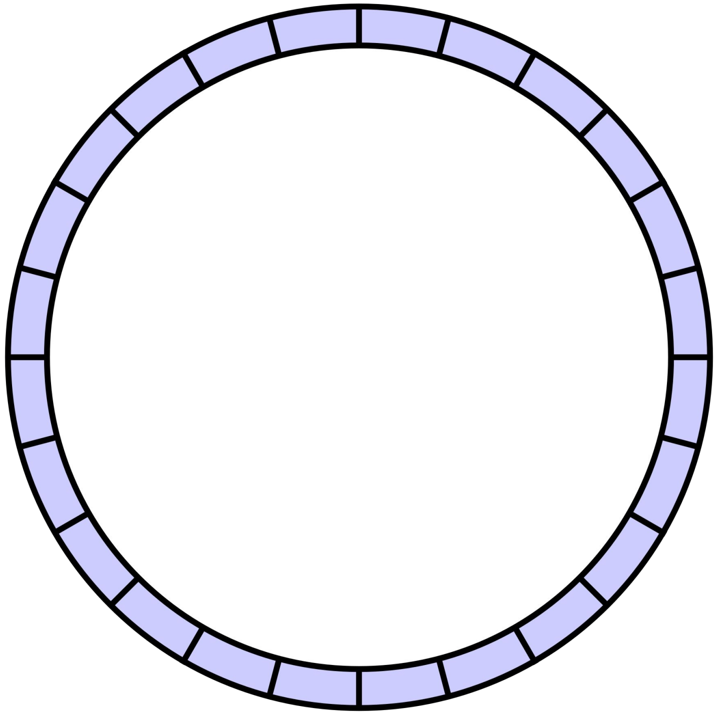
Allocate an array of *n* elements

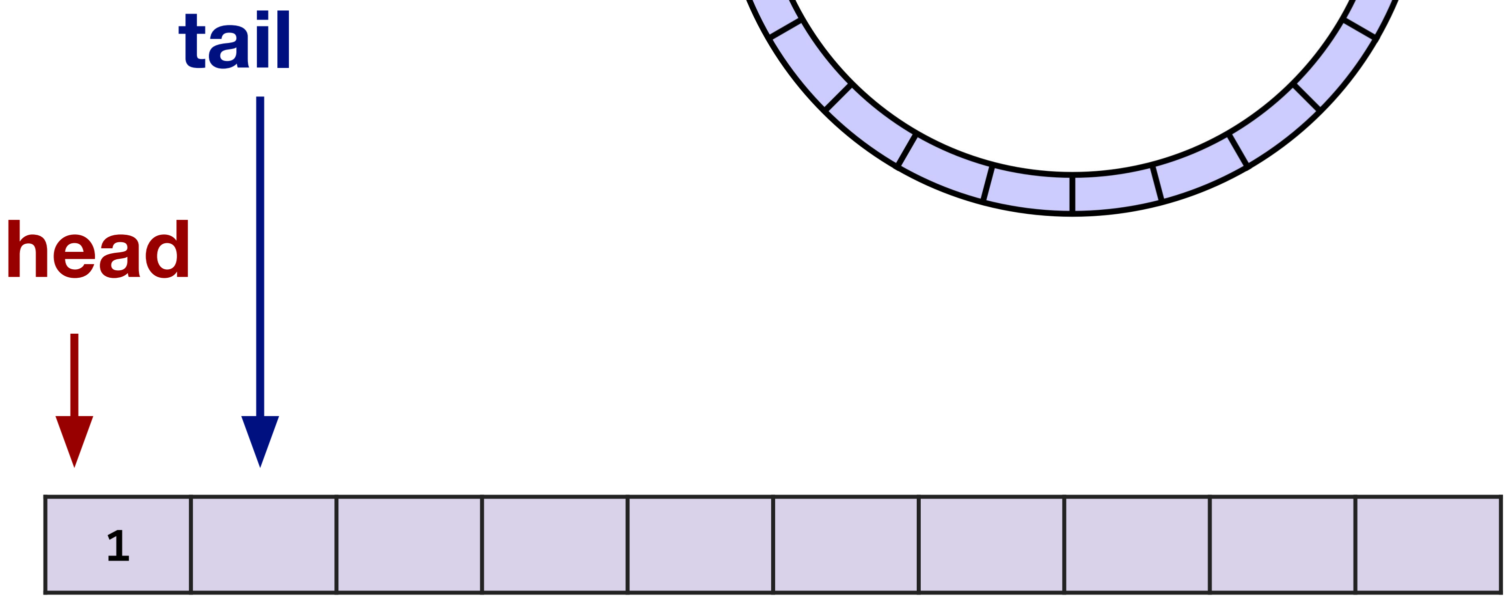
`push(1)`

Keep track of two indices: head & tail

Producers **push** to the tail

Consumers **pop** from the head





Queue Implementation: Circular (Ring) Buffer

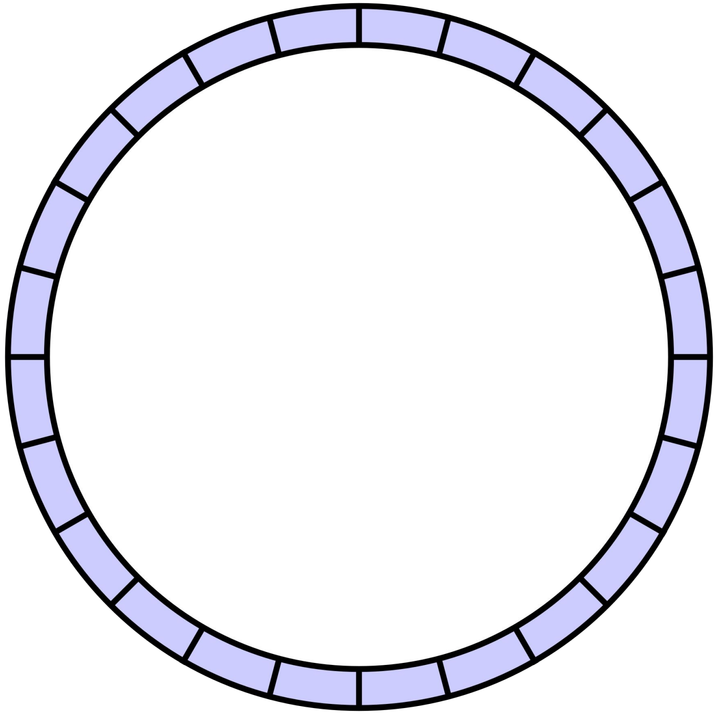
Allocate an array of *n* elements

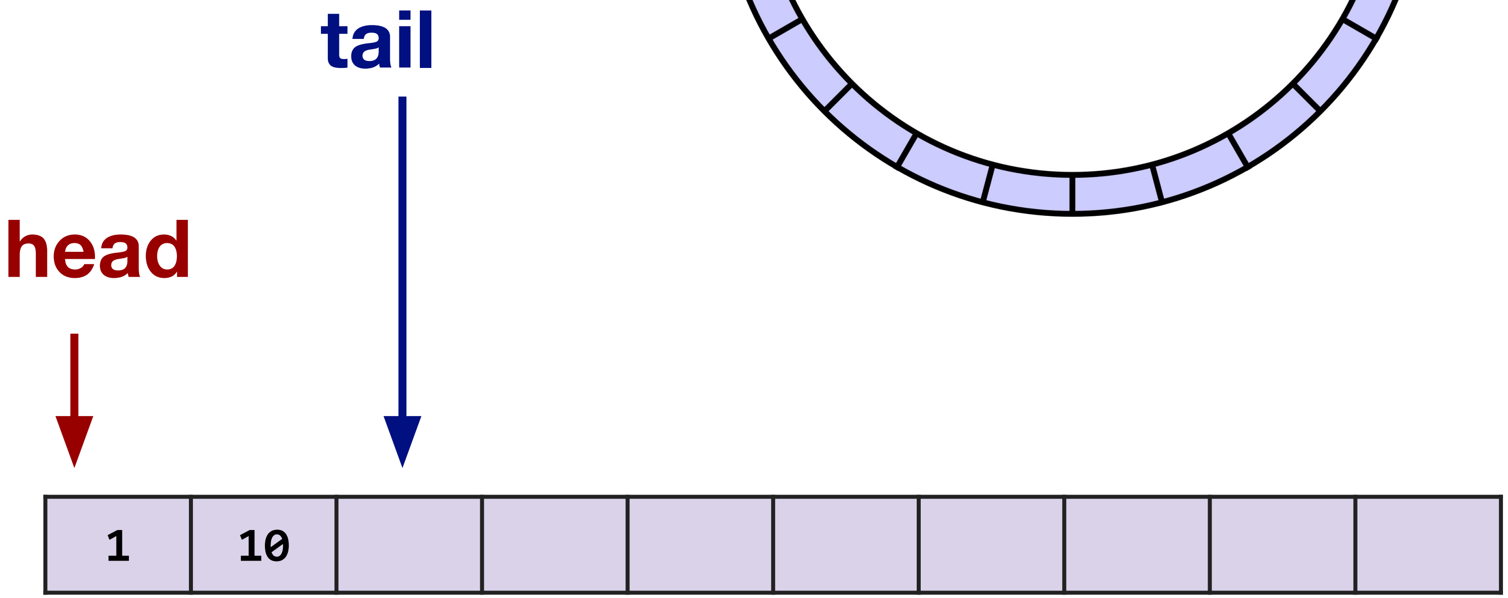
```
push(1)
push(10)
```

Keep track of two indices: head & tail

Producers **push** to the tail

Consumers **pop** from the head





Queue Implementation: Circular (Ring) Buffer

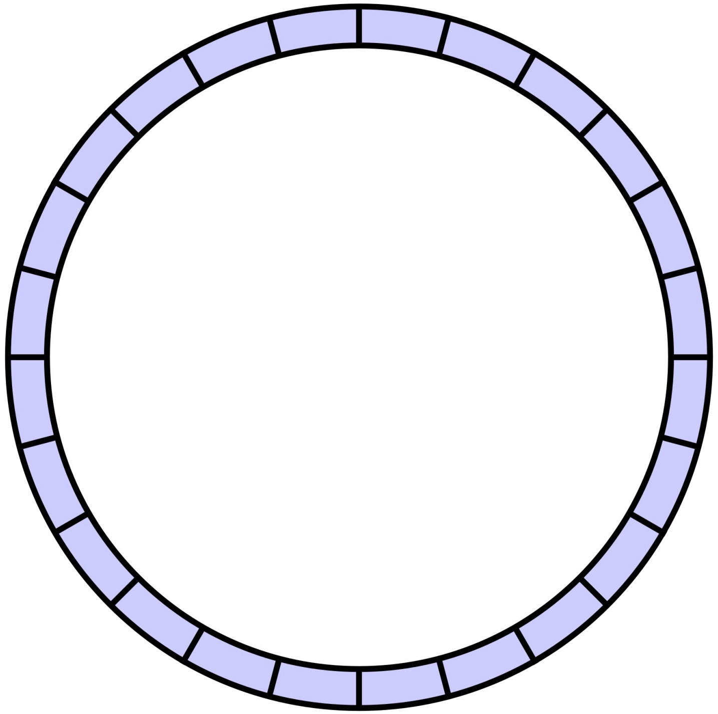
Allocate an array of *n* elements

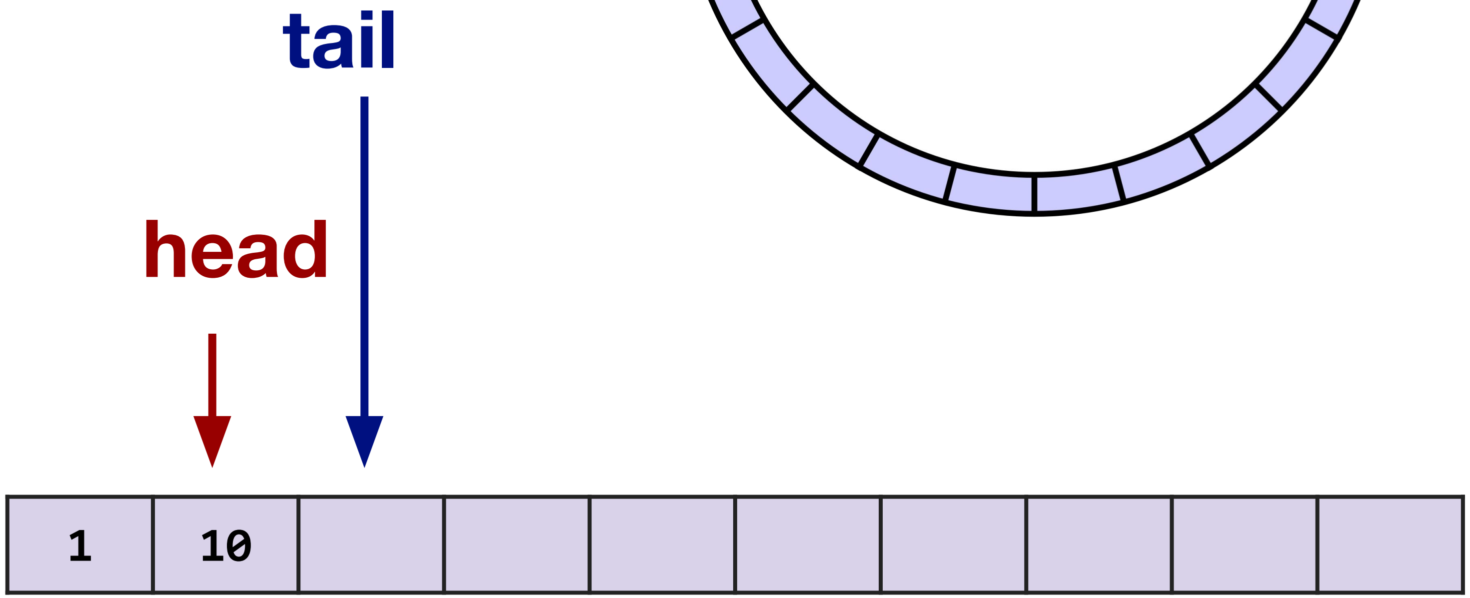
Keep track of two indices: head & tail

Producers **push** to the tail

Consumers **pop** from the head

```
push(1)
push(10)
pop() → 1
```





Queue Implementation: Circular (Ring) Buffer

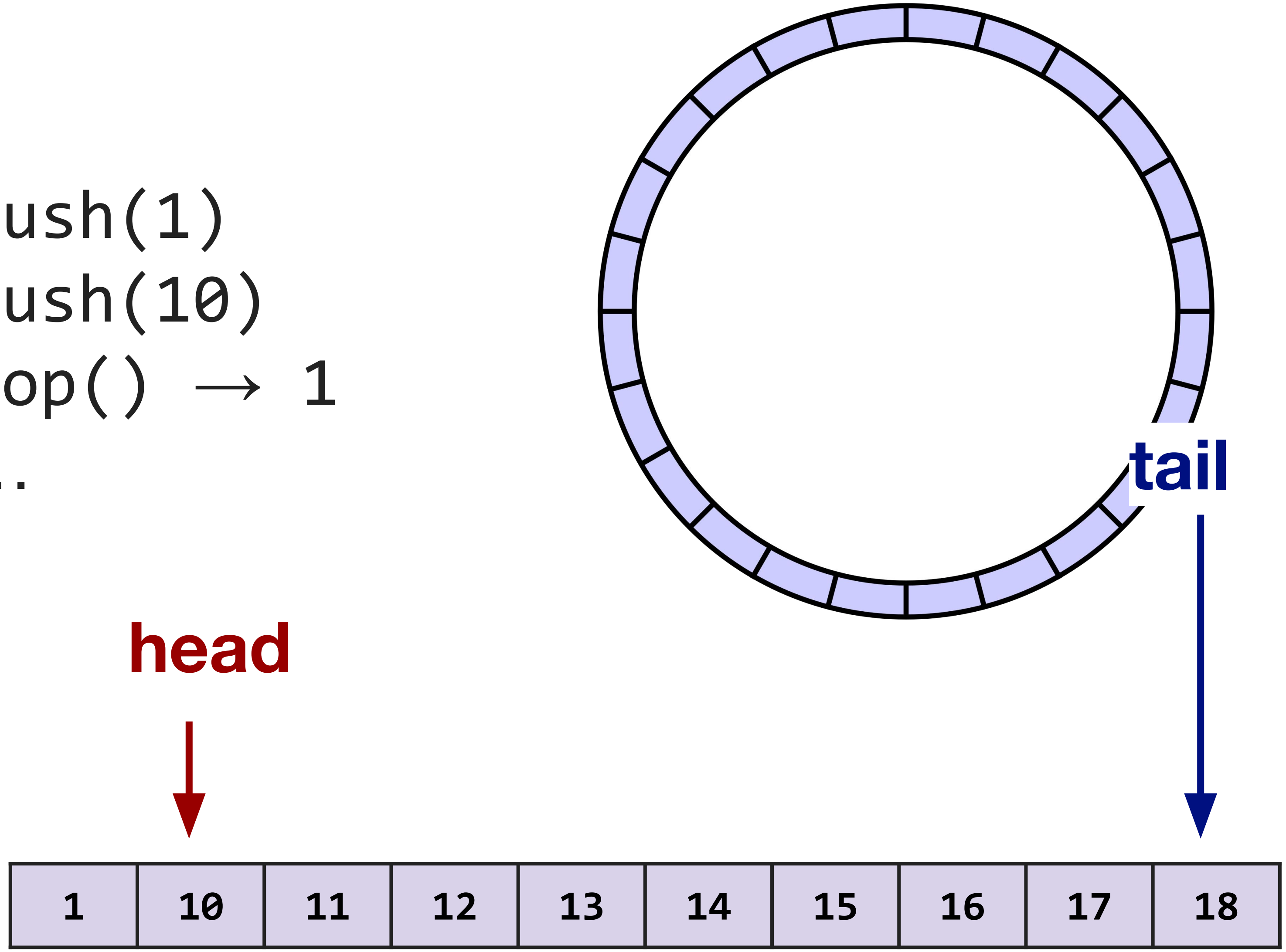
Allocate an array of *n* elements

Keep track of two indices: head & tail

```
push(1)
push(10)
pop() → 1
…
```

Producers **push** to the tail

Consumers **pop** from the head



Queue Implementation: Circular (Ring) Buffer

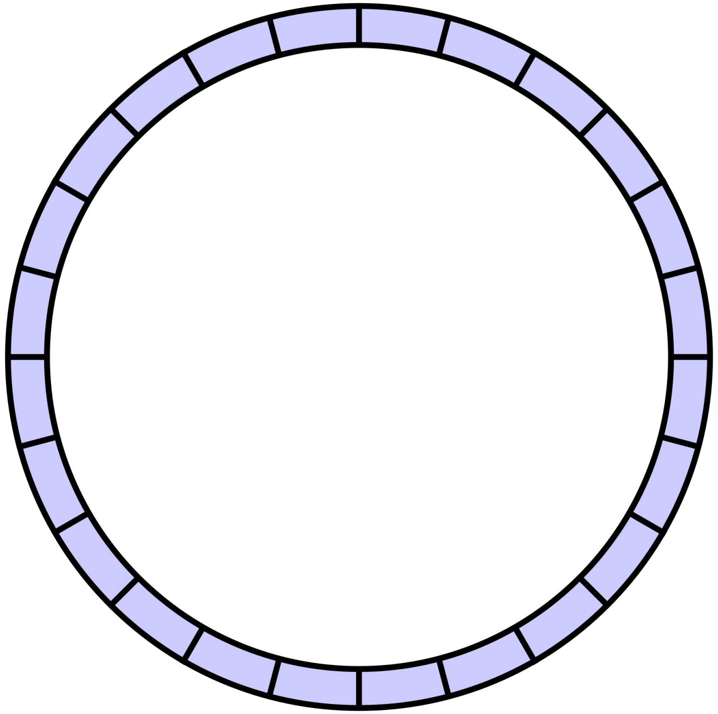
Allocate an array of *n* elements

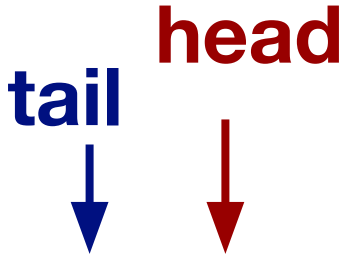
Keep track of two indices: head & tail

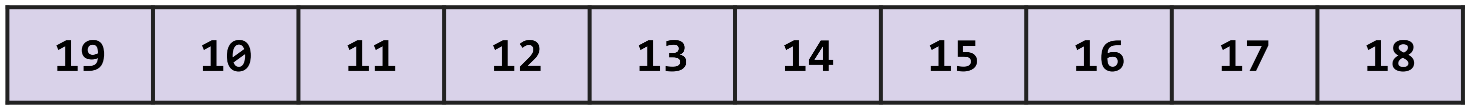
Producers **push** to the tail

Consumers **pop** from the head

```
push(1)
push(10)
pop() → 1
...
push(19)
```







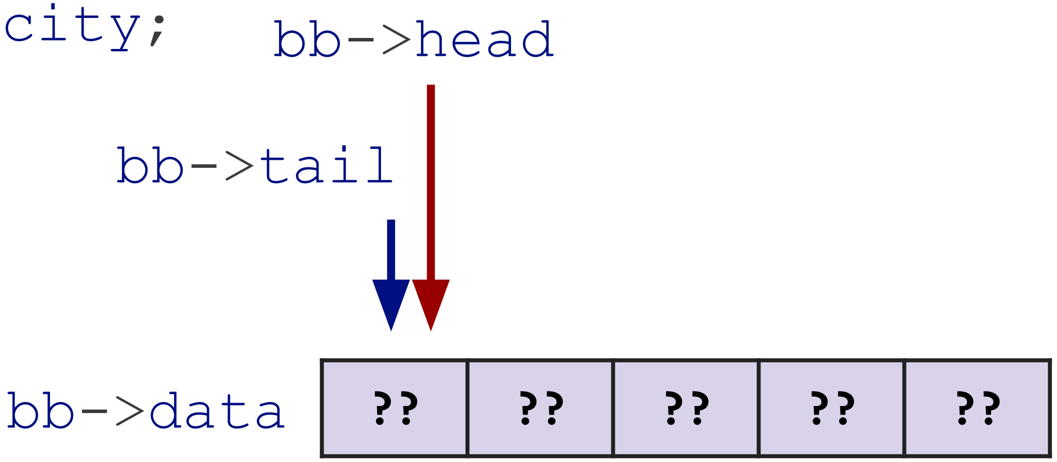
pointers wrap around using modulo (%)

Let’s Implement a Circular Buffer!

Push to Buffer

```
bool bb_full(buf_t* bb) {
    return bb_size(bb) >= bb->capacity - 1;
}
void bb_push(buf_t* bb, int value) {
    assert(!bb_full(bb));
    bb->data[bb->tail] = value;
    bb->tail = (bb->tail + 1) % bb->capacity;
}
```

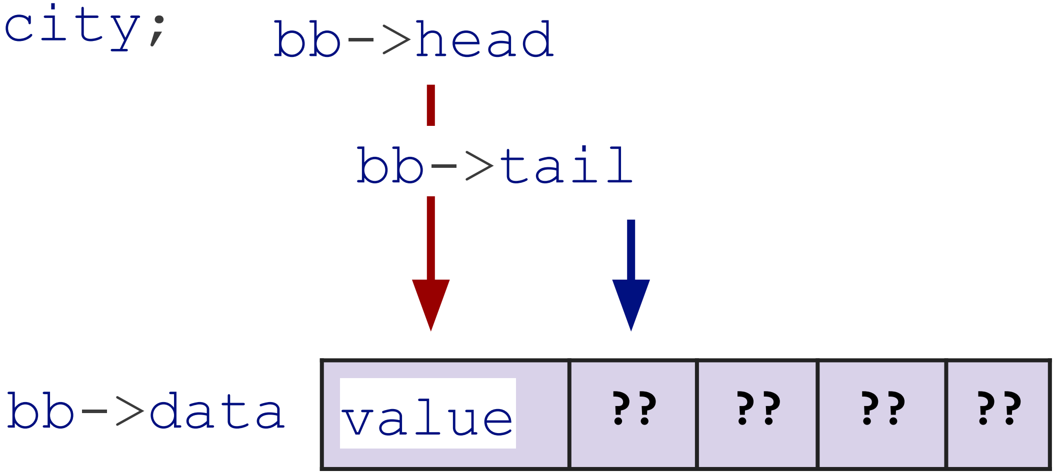
- `bb->capacity` is the length of `bb->data`



Push to Buffer

```
bool bb_full(buf_t* bb) {
    return bb_size(bb) >= bb->capacity - 1;
}
void bb_push(buf_t* bb, int value) {
    assert(!bb_full(bb));
    bb->data[bb->tail] = value;
    bb->tail = (bb->tail + 1) % bb->capacity;
}
```

- `bb->capacity` is the length of `bb->data`



Thread-safe Push

```
void bb_block_push(bounded_buffer_t* bb, int value) {
   pthread_mutex_lock(bb->mutex);









   pthread_mutex_unlock(bb->mutex);
}
```

- Access to `bb` is guarded by `bb->mutex`

Thread-safe Push

```
void bb_block_push(bounded_buffer_t* bb, int value) {
    pthread_mutex_lock(bb->mutex);

    // Spin to wait until the queue has room to push.
    while (bb_full(bb)) {
        pthread_mutex_unlock(bb->mutex);
        pthread_mutex_lock(bb->mutex);
    }




    pthread_mutex_unlock(bb->mutex);
}
```

- Access to `bb` is guarded by `bb->mutex`
- Wait until there is space in the buffer

Thread-safe Push

```
void bb_block_push(bounded_buffer_t* bb, int value) {
   pthread_mutex_lock(bb->mutex);

   // Spin to wait until the queue has room to push.
   while (bb_full(bb)) {
       pthread_mutex_unlock(bb->mutex);
       pthread_mutex_lock(bb->mutex);
   }

   // Actually do the push.
   bb_push(bb, value);

   pthread_mutex_unlock(bb->mutex);
}
```

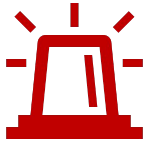
- Access to `bb` is guarded by `bb->mutex`
- Wait until there is space in the buffer
- Then perform the push.
- Performance problems?

Condition Variables

Condition variables let you *temporarily* release a lock while you wait for some condition to be true.

`pthread_cond_wait(cond, mutex)`: Call when you hold `mutex`. Waits for a signal from another thread on condition variable `cond`, and then re-acquires `mutex`

`pthread_cond_signal(cond)`: Signal (i.e., wake up) *one* thread that is currently waiting on `cond`



`pthread_cond_broadcast(cond)`: Signal (i.e., wake up) *all* threads that are currently waiting on `cond`

Spurious Wakeups

Threads might wake up without being signaled!

The Correct Way:

```c
pthread_mutex_lock(mutex);
while (!check_your_condition()) {
  pthread_cond_wait(cond, mutex);
}
do_stuff(); // `check_your_condition()` returned
pthread_mutex_unlock(mutex);
```

Wrap your calls to `pthread_cond_wait` in a while loop!

Thread-safe Push with Condition Variables

```
void bb_block_push(bounded_buffer_t* bb, int value) {
   pthread_mutex_lock(bb->mutex);







   // Actually do the push.
   bb_push(bb, value);

   pthread_mutex_unlock(bb->mutex);

}
```

- We still need to acquire the mutex.

Thread-safe Push with Condition Variables

```
void bb_block_push(bounded_buffer_t* bb, int value) {
   pthread_mutex_lock(bb->mutex);

   while (bb_full(bb)) {
      pthread_cond_wait(bb->full_cv, bb->mutex);
   }

   // Actually do the push.
   bb_push(bb, value);

   pthread_mutex_unlock(bb->mutex);

}
```

- We still need to acquire the mutex.
- Signal to the OS that we are waiting for the buffer to be no longer full.

Thread-safe Push with Condition Variables

```
void bb_block_push(bounded_buffer_t* bb, int value) {
   pthread_mutex_lock(bb->mutex);

   while (bb_full(bb)) {
      pthread_cond_wait(bb->full_cv, bb->mutex);
   }


   // Actually do the push.
   bb_push(bb, value);

   pthread_mutex_unlock(bb->mutex);
   pthread_cond_signal(bb->empty_cv);
}
```

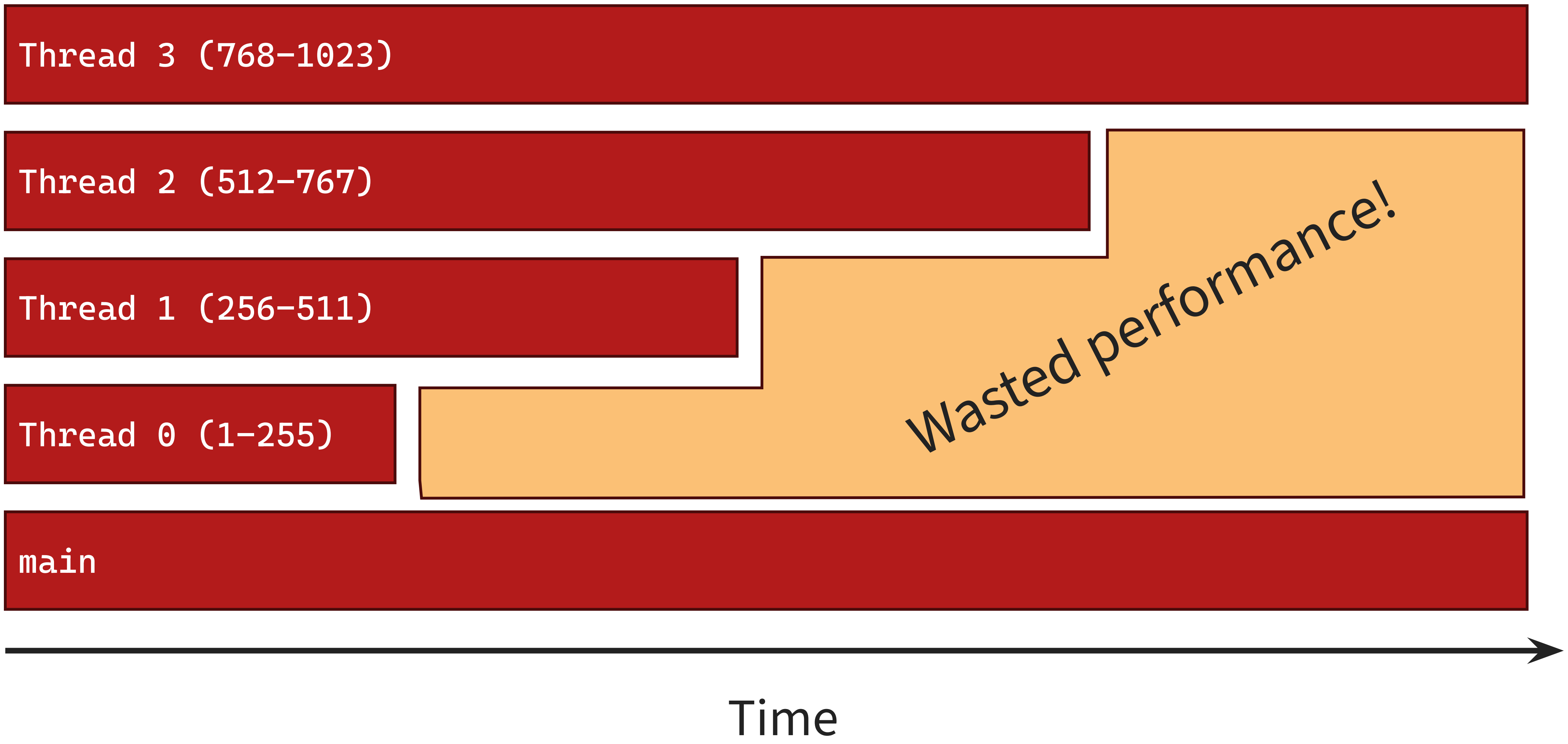
- We still need to acquire the mutex.
- Signal to the OS that we are waiting for the buffer to be no longer full.
- Signal to other threads that queue now has at least one entry.

No More Busy Waiting!

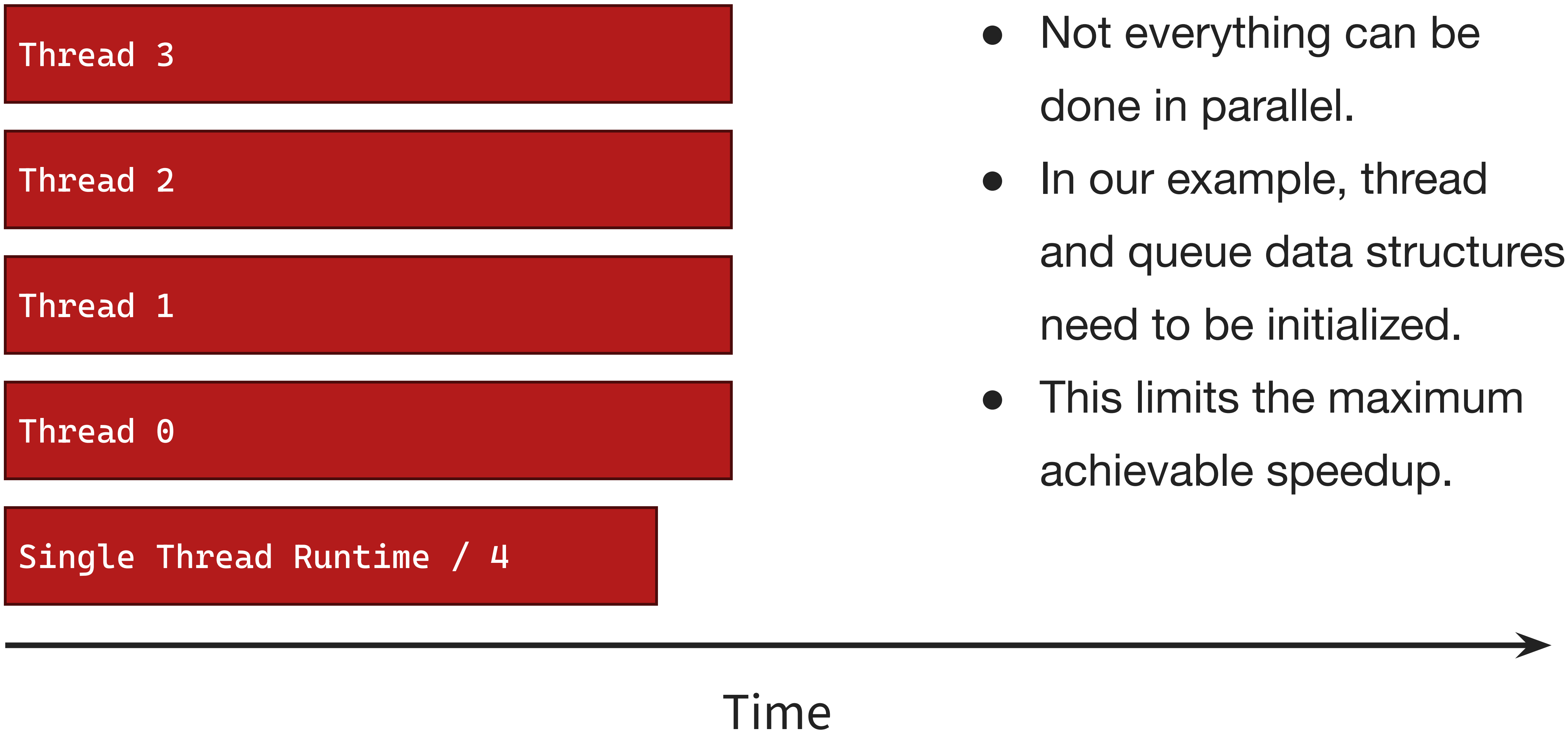
Limits on Speedups through Parallelism: Amdahl's Law

*not to scale

Remember: *Thread Imbalance*

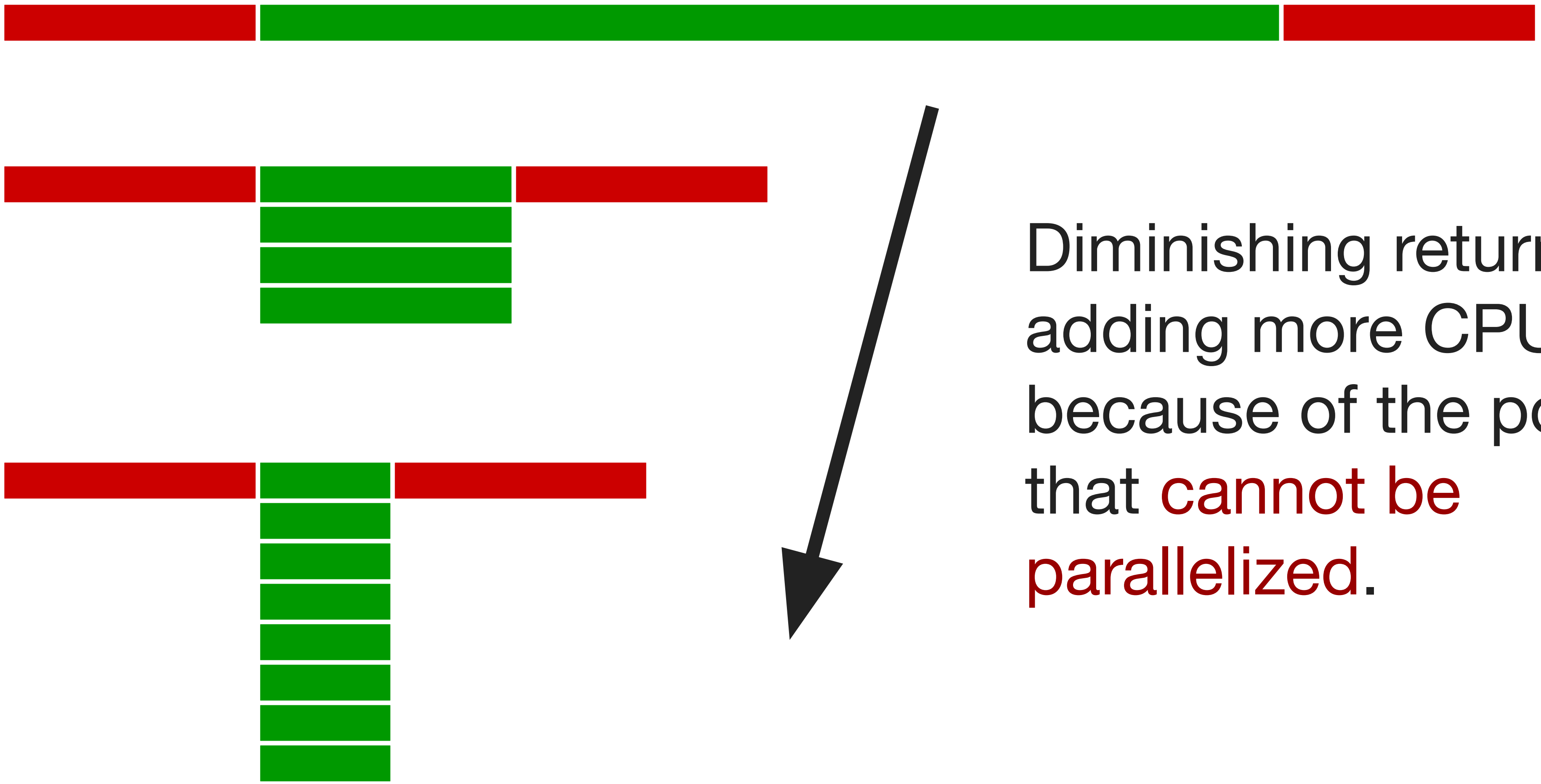


Balanced with Producer / Consumer?



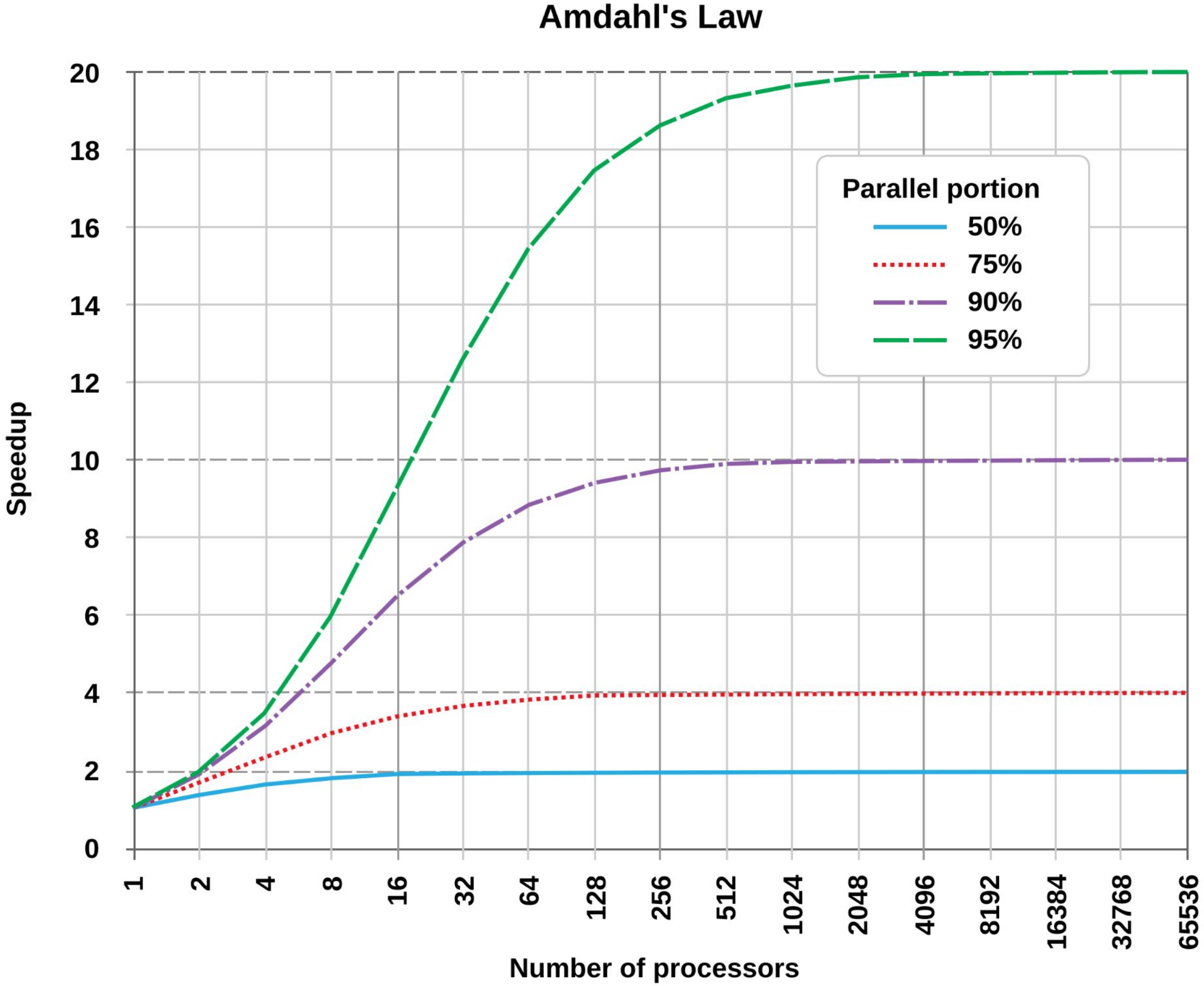
- Not everything can be done in parallel.
- In our example, thread and queue data structures need to be initialized.
- This limits the maximum achievable speedup.

Amdahl's Law

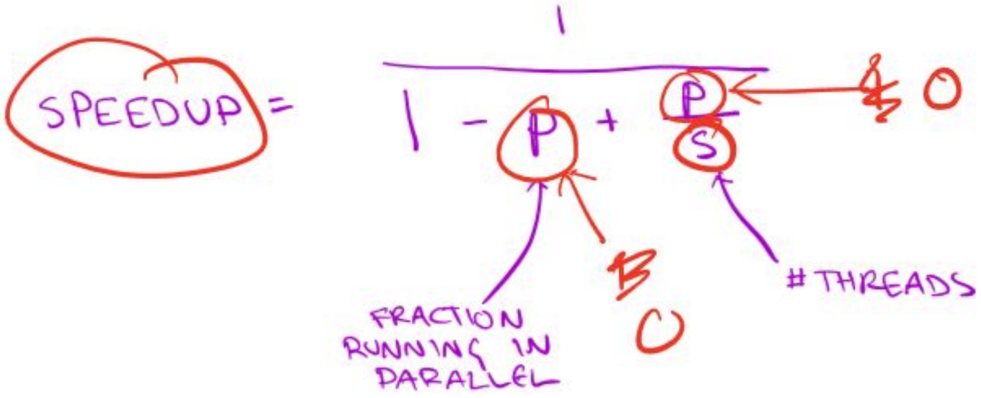


Diminishing return on adding more CPU cores because of the portion that cannot be parallelized.

Amdahl's Law



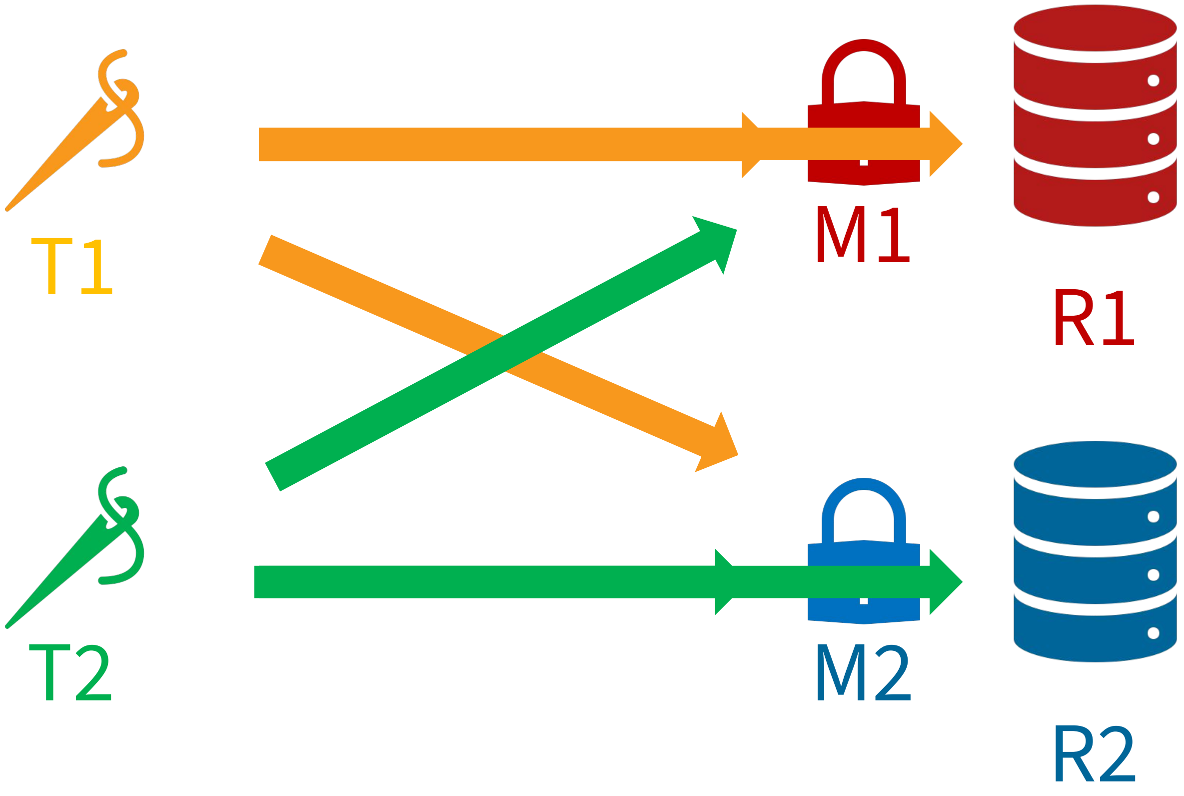
- **Parallel portion p** as a percentage of total sequential runtime.
- p=0: Nothing can be parallelized.
- p=1.0: Perfectly (embarrassingly) parallel.



Deadlock

Deadlock occurs when two different threads get stuck waiting for each other.

Both T1 and T2 need to access R1 and R2.



Deadlock Demo

```
void* thread1(void* arg) {
    pthread_mutex_lock(&lock1);
```

```
    pthread_mutex_lock(&lock2);
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return nullptr;
}
```

```
void* thread2(void* arg) {
    pthread_mutex_lock(&lock2);
```

```
    pthread_mutex_lock(&lock1);
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    return nullptr;
}
```

A deadlock occurs of both threads reach this point concurrently.

Avoiding Deadlock

Key Problem: threads acquire locks in different orders

Three Easy Steps to Avoid Deadlock with Mutexes:

1. Decide on a **total order** among all your mutexes
2. Always acquire the mutexes in that order.
3. Always release the mutexes in the *opposite* order.

Think of locks as curly brackets!