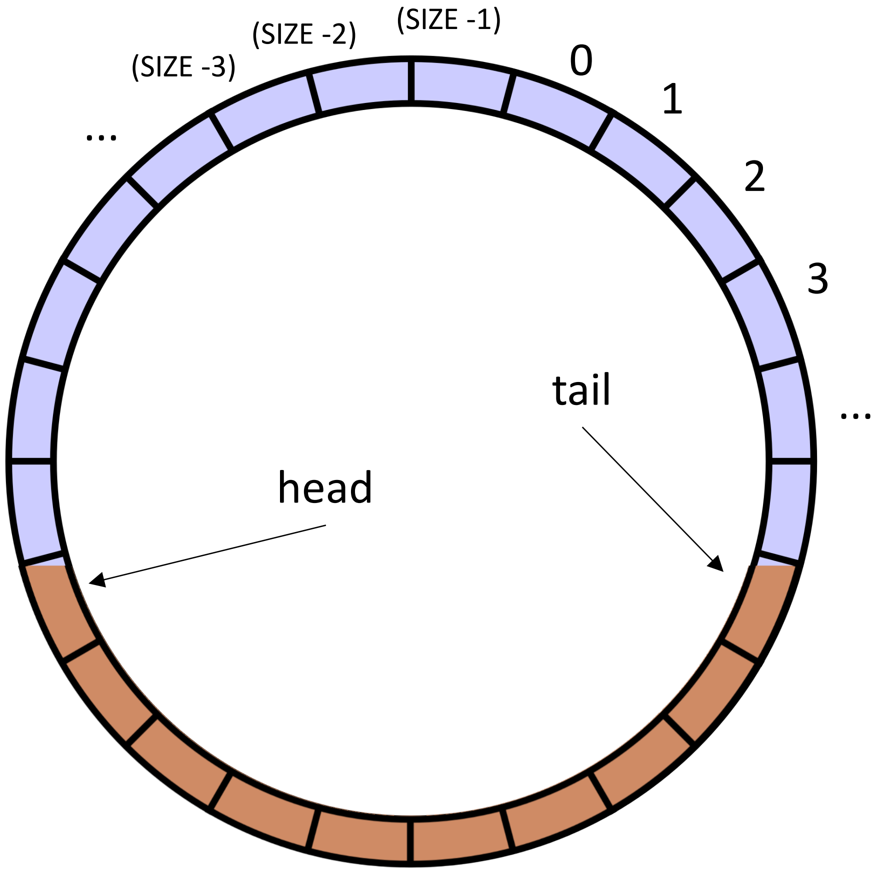
CSE113: Parallel Programming

Feb. 15, 2023

- **Topics**:
  - Producer consumer queues
    - Circular buffer



Announcements

- HW1 grades are out!
  - Please let us know if there are issues

- Homework 2 was due on Monday
  - We will start grading and try to get grades in 2 weeks

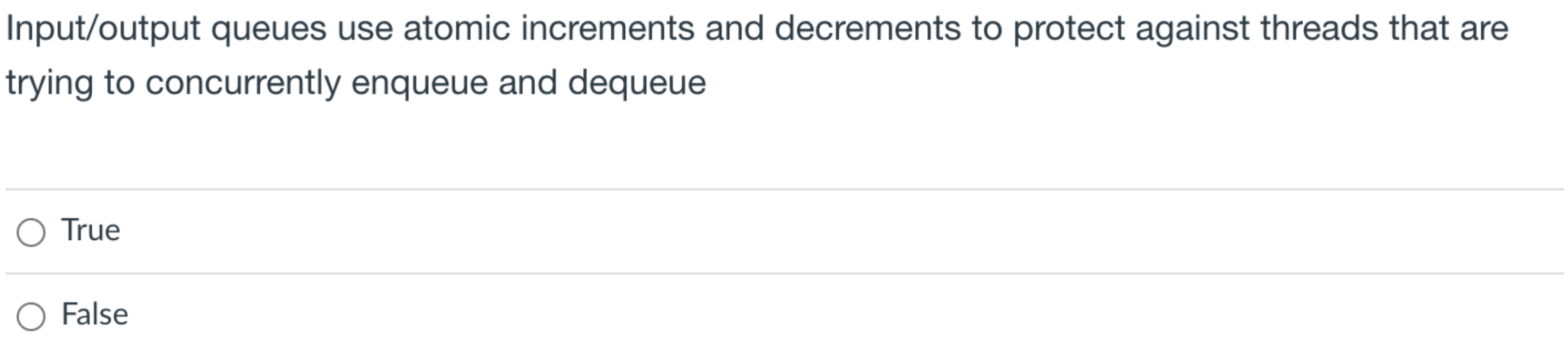
- Homework 3 is released
  - You can finish part 1 after today
  - Part 2 may need to wait until Friday
  - Due Feb 23 + 4 days = Feb 27

Announcements

- Midterm out!
  - asynchronous, 1 work week; Monday through Friday; no time limit
  - Open note, open internet (to a reasonable extent: no googling exact questions or asking questions on forums or ChatGPT)
  - do not discuss with classmates AT ALL while the test is active
  - **No late tests will be accepted.**

- You can ask clarifying questions about the midterm **(as private Piazza posts).** We will not comment on your answers or give any hints.

Previous quiz

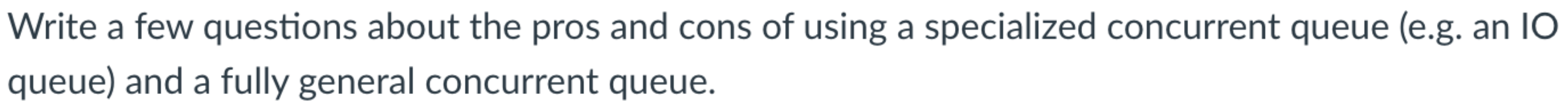


Input/output queues use atomic increments and decrements to protect against threads that are trying to concurrently enqueue and dequeue

○ True

○ False

Previous quiz

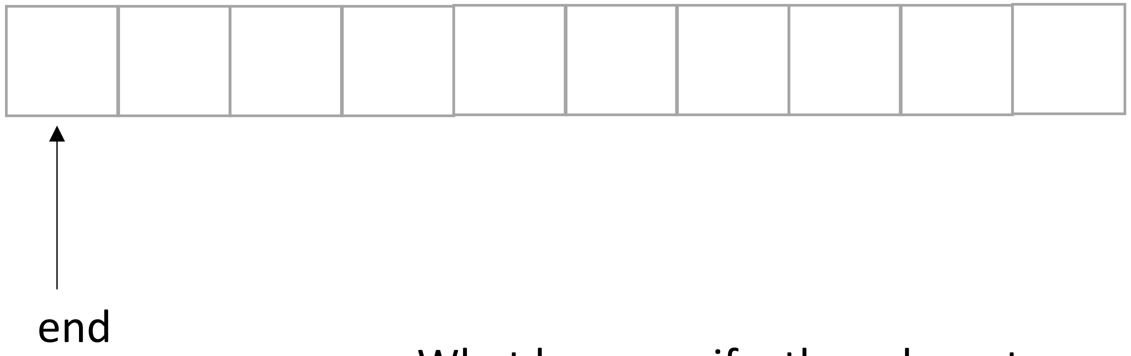


Write a few questions about the pros and cons of using a specialized concurrent queue (e.g. an IO queue) and a fully general concurrent queue.

Review

Input/Output Queues

Implementation

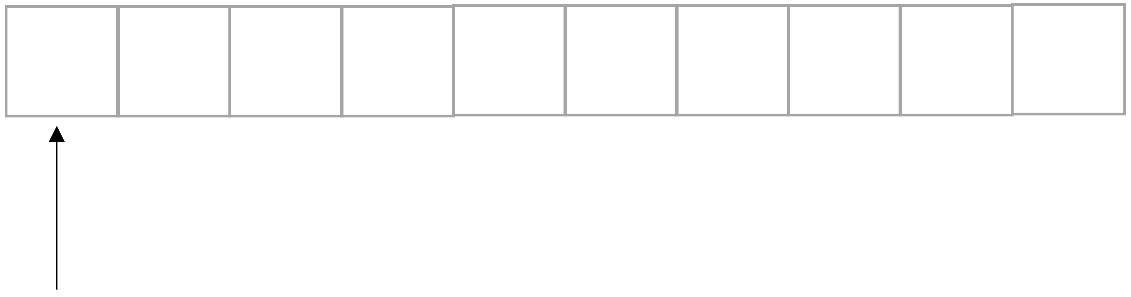


What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation



<u>Thread 0:</u>
enq(6);

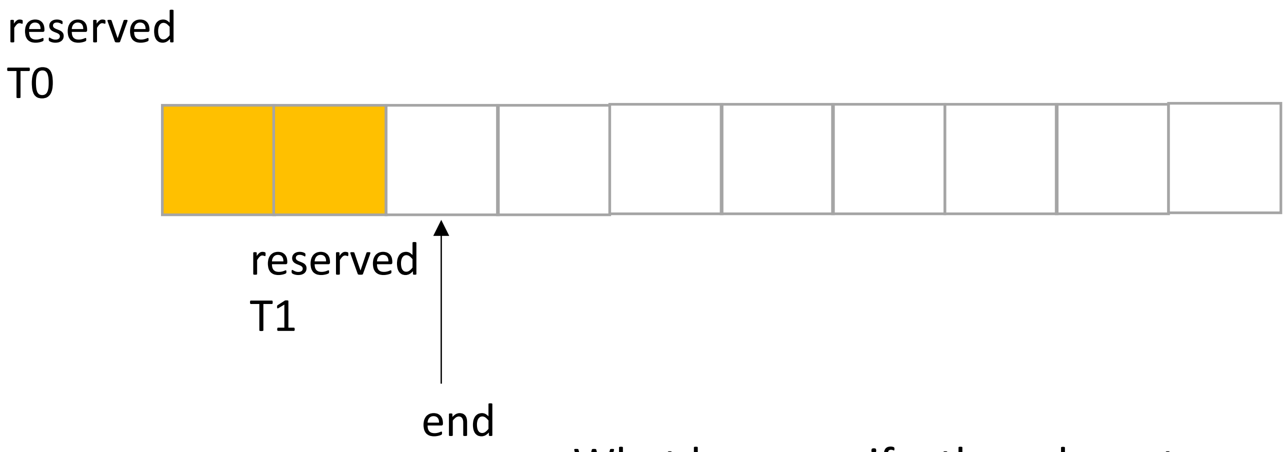
<u>Thread 1:</u>
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

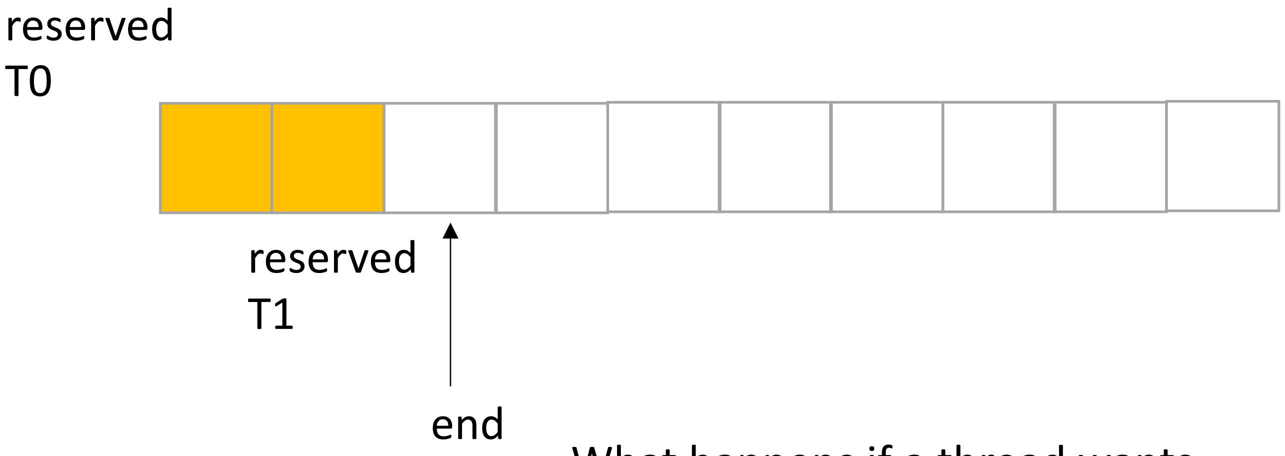
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data?



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

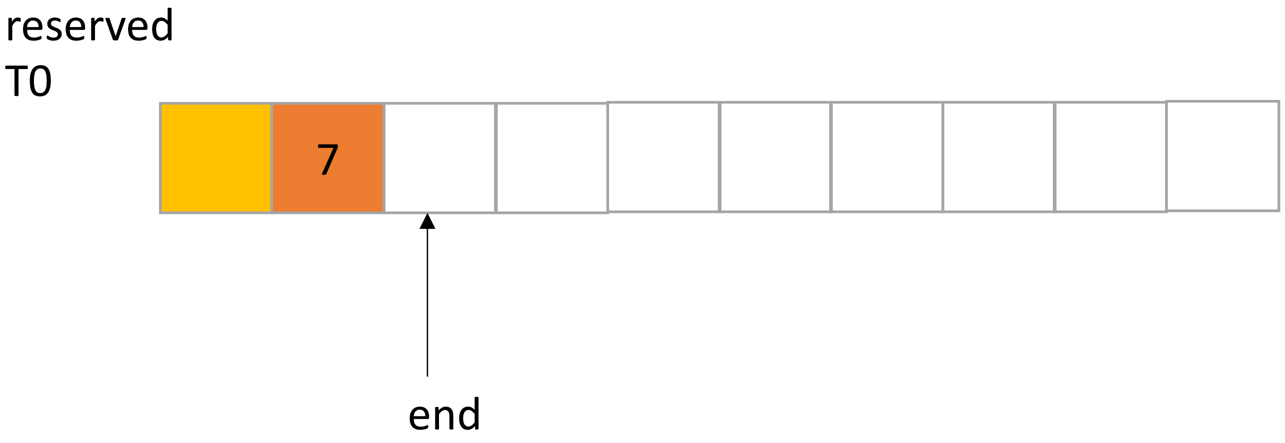
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data?



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

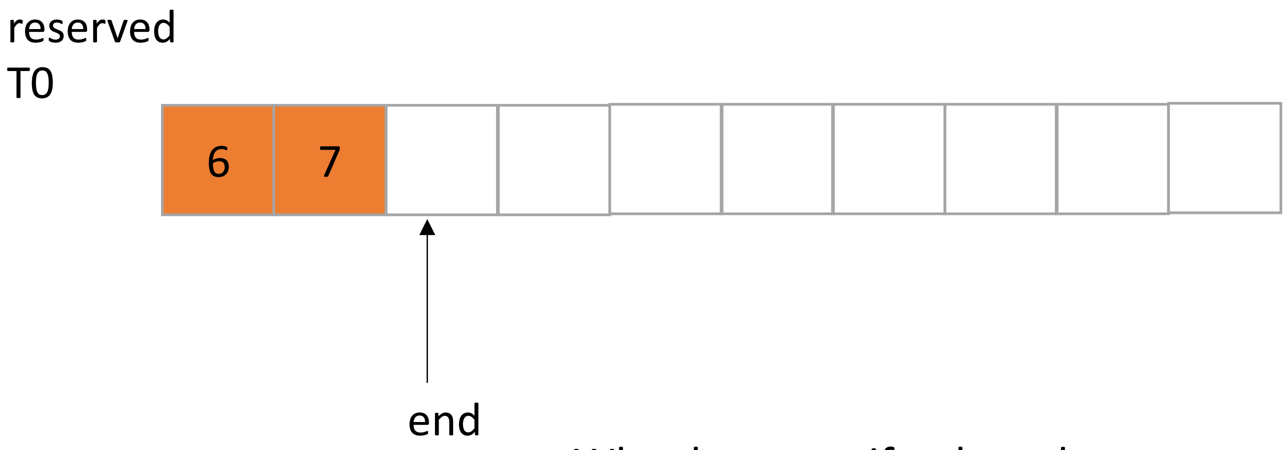
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data? No! Because there are no deqs!



Thread 0:
enq(6);

Thread 1:
enq(7);

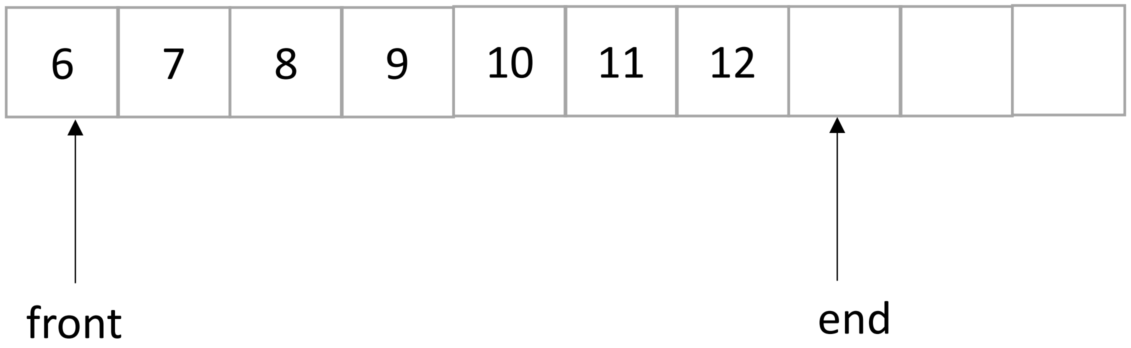
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

What about Input?

- Now we only do deqs



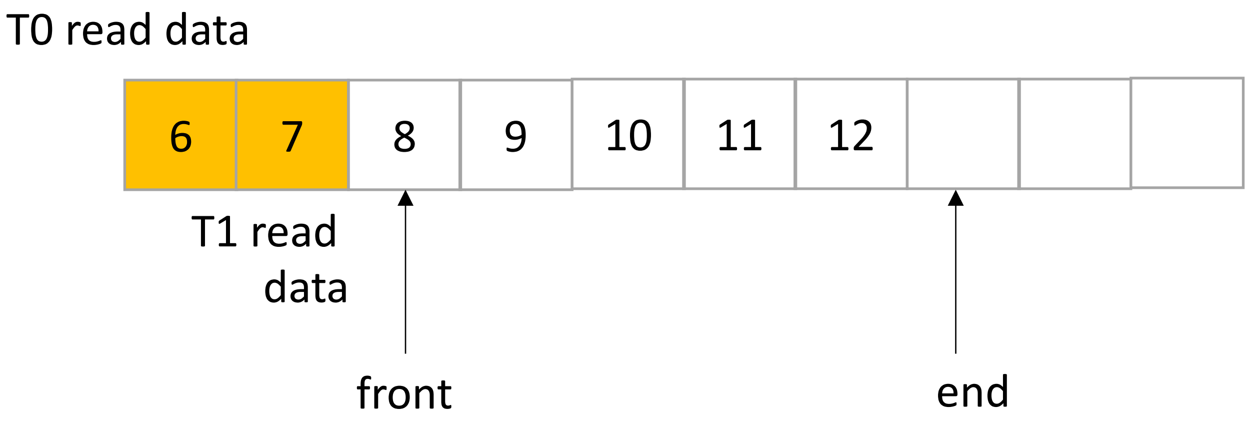
6 7 8 9 10 11 12

front

end

What about Input?

- Now we only do deqs



How to implement
a stack?

Thread 0:
deq(); // reads 6

Thread 1:
deq(); // reads 7

What happens if a thread wants
to deq an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     int deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return end.load() - front.load();
     }
 }
```

does the list need
to be atomic?

Is this queue thread safe?

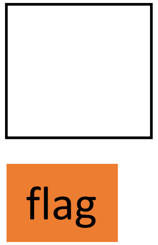
Is this queue lock free?

Synchronous Producer Consumer Queues

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



<u>Consumer Thread</u>

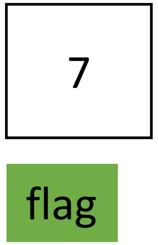
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



7

flag

Consumer Thread

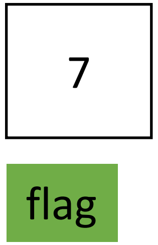
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



7

flag

Consumer Thread

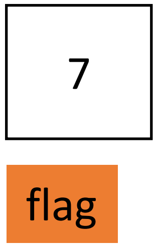
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



7

flag

Consumer Thread

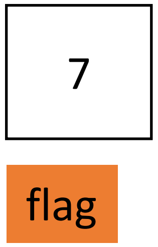
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```


7

flag

Consumer Thread

```
deq();
deq();
```

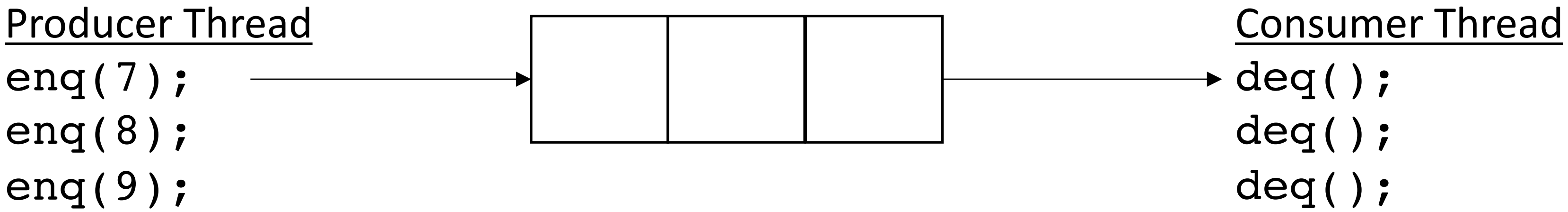
```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Schedule

- Producer Consumer Queues
  - Synchronous
  - Circular buffer

Producer Consumer Queues

- Asynchronous:



Producer Thread

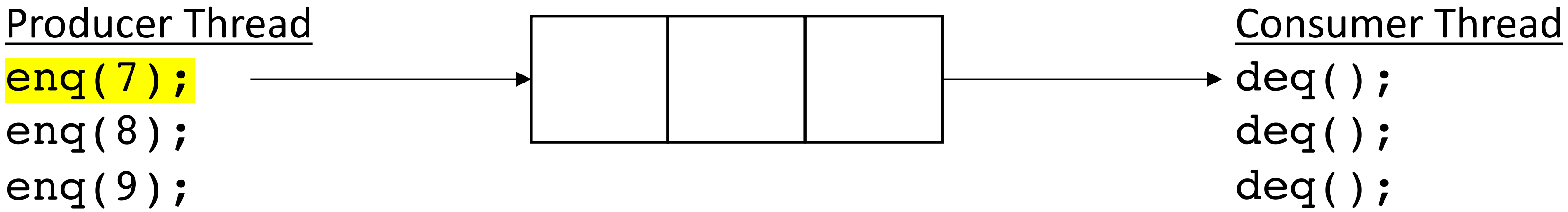
```
enq(7);
enq(8);
enq(9);
```

Consumer Thread

```
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:



Producer Thread

```
enq(7);
enq(8);
enq(9);
```

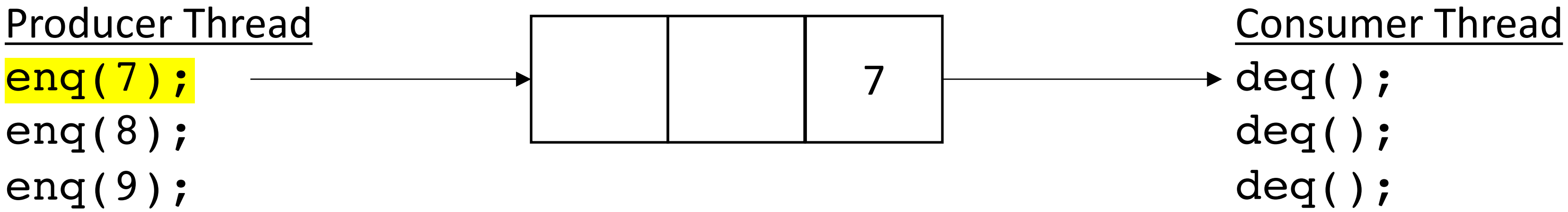
Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:



Producer Thread

```
enq(7);
enq(8);
enq(9);
```

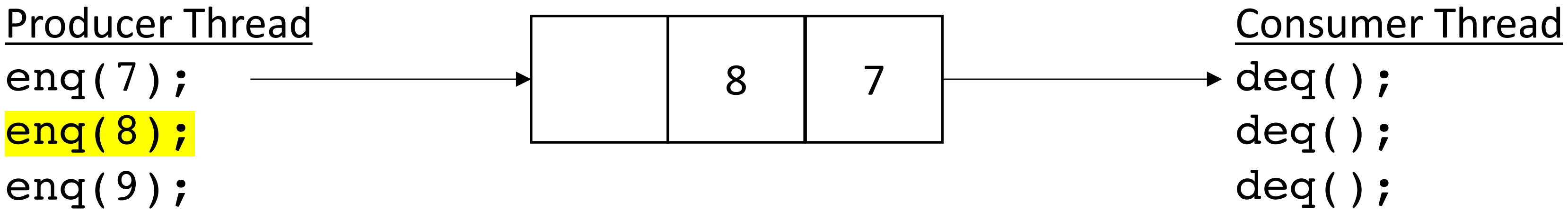
Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:



Producer Thread

```
enq(7);
enq(8);
enq(9);
```

Consumer Thread

```
deq();
deq();
deq();
```

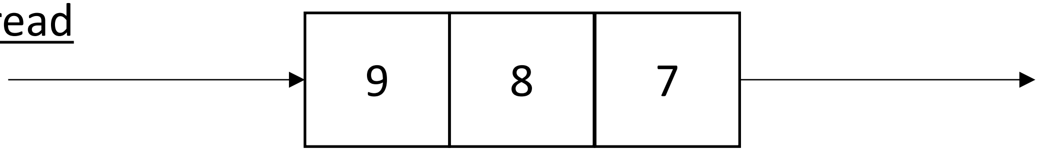
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
```



<u>Consumer Thread</u>

```
deq();
deq();
deq();
```

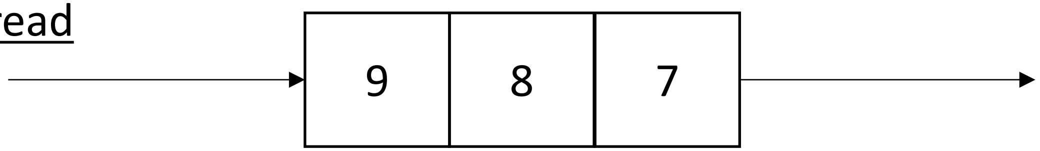
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

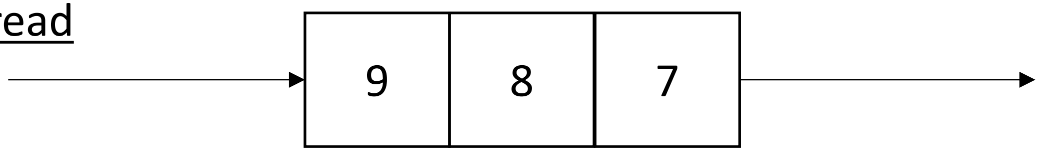
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

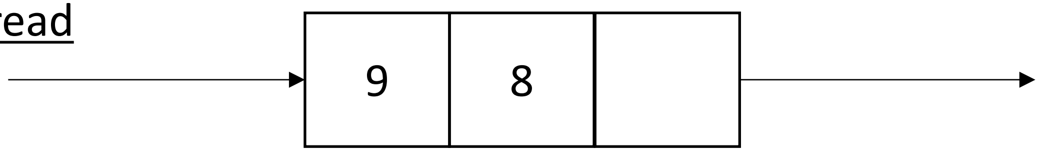
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

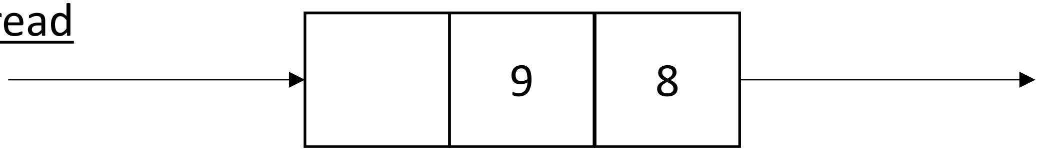
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

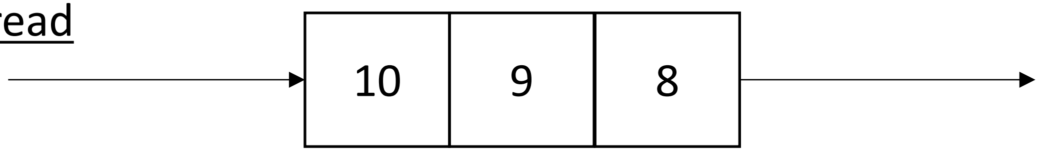
Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```

finishes



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

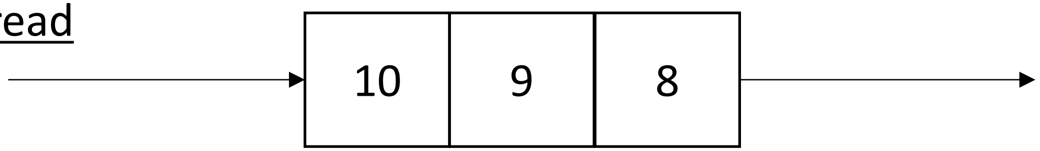
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

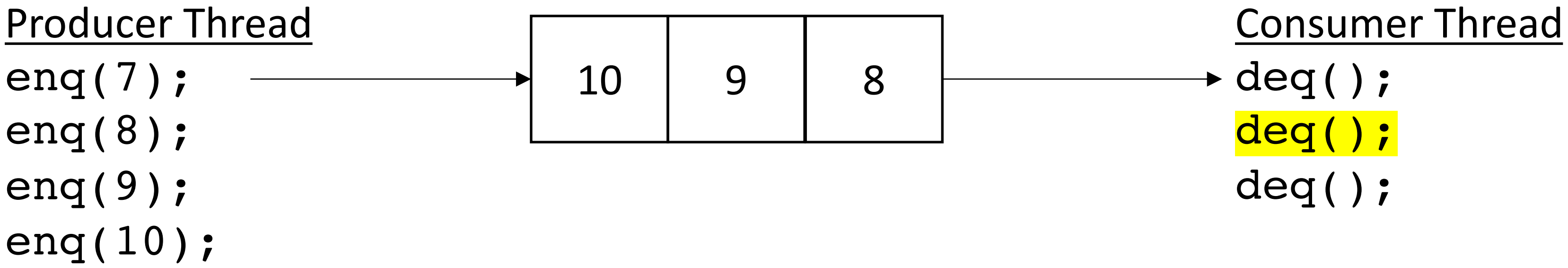
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```

Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

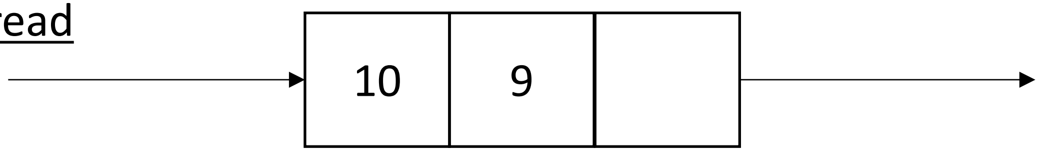
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

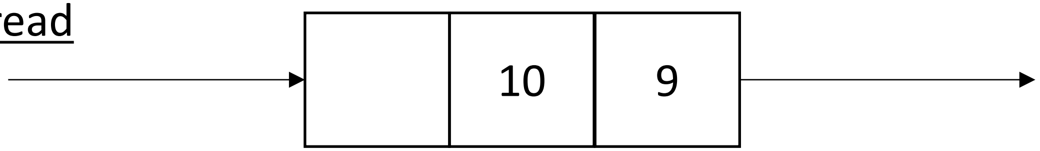
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

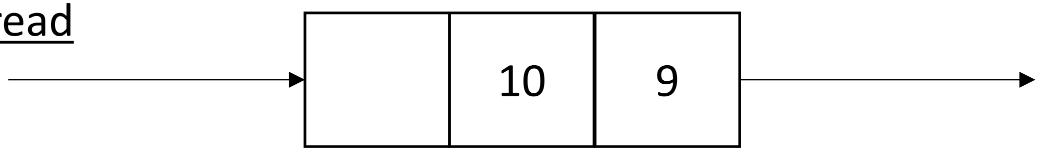
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

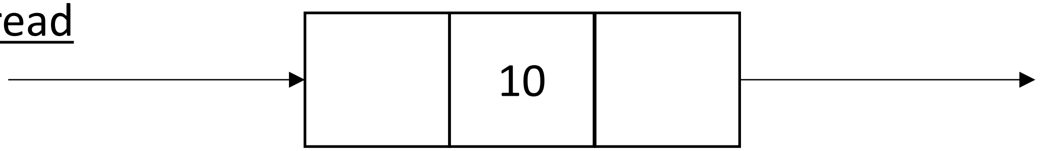
returns 9

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

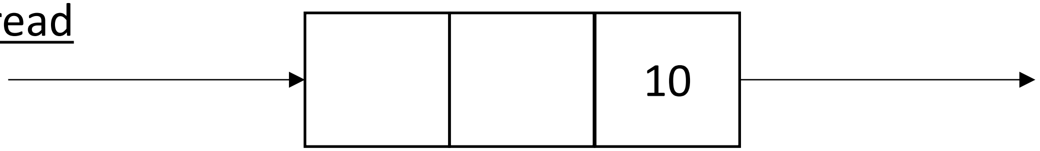
```
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

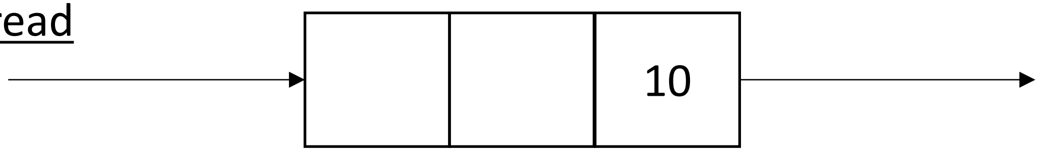
```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

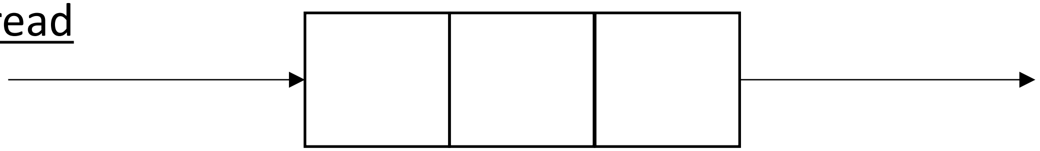
```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
deq();
deq();
```

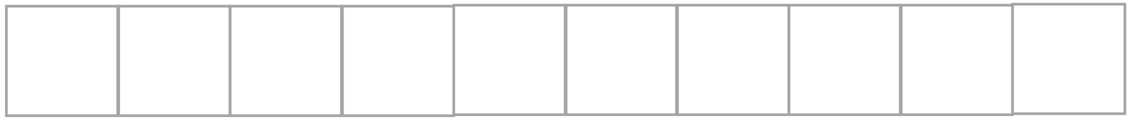
blocks when there is nothing in the queue

Producer Consumer Queues

- How do we implement it?

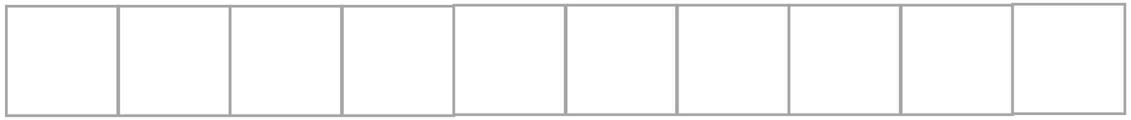
Producer Consumer Queues

- Start with a fixed size array



Producer Consumer Queues

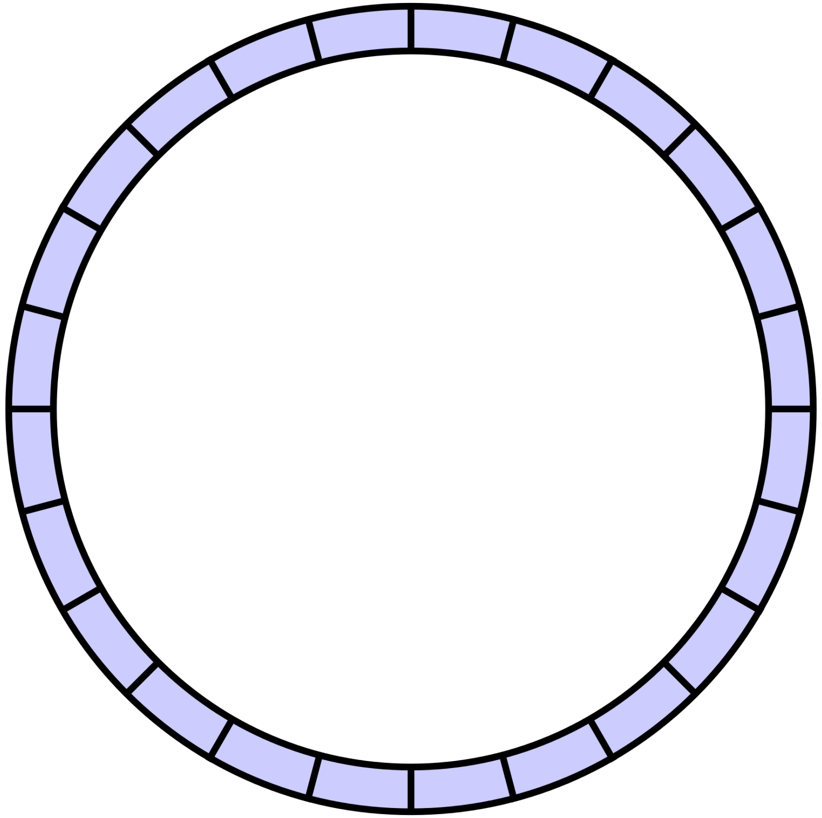
- Start with a fixed size array



We will use what is called a *circular buffer method*

Producer Consumer Queues

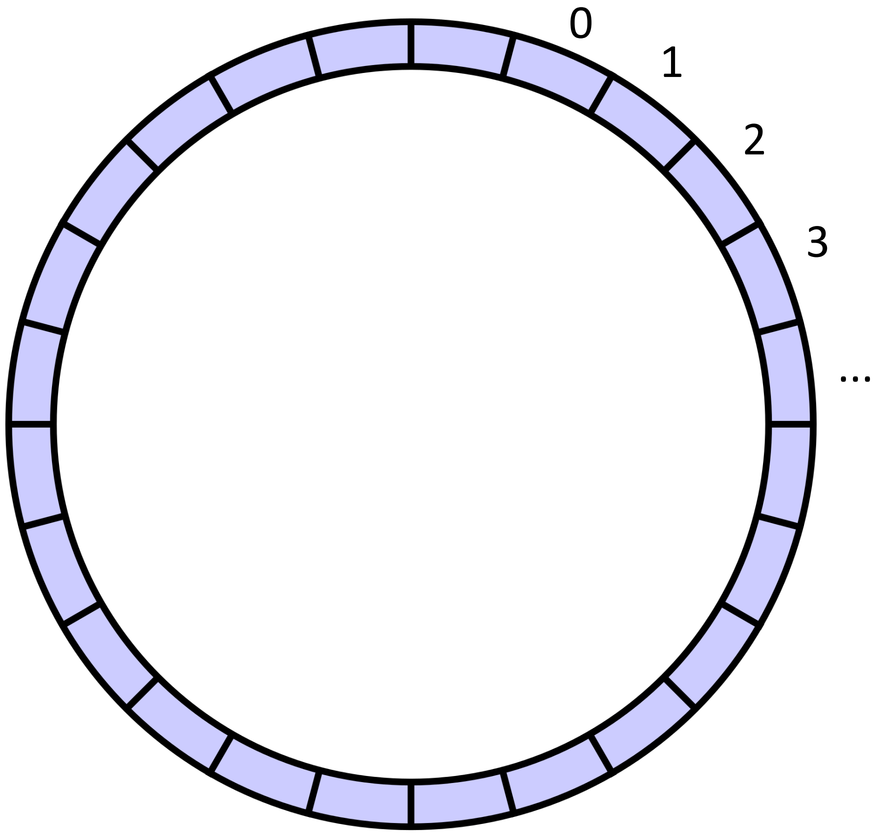
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

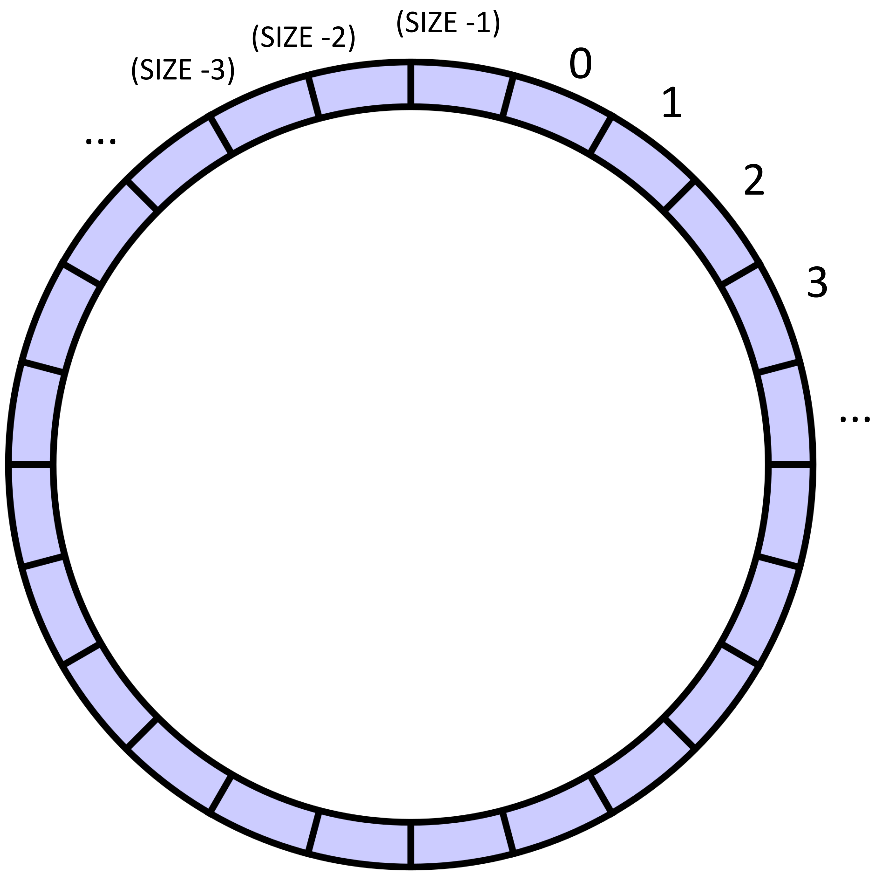
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

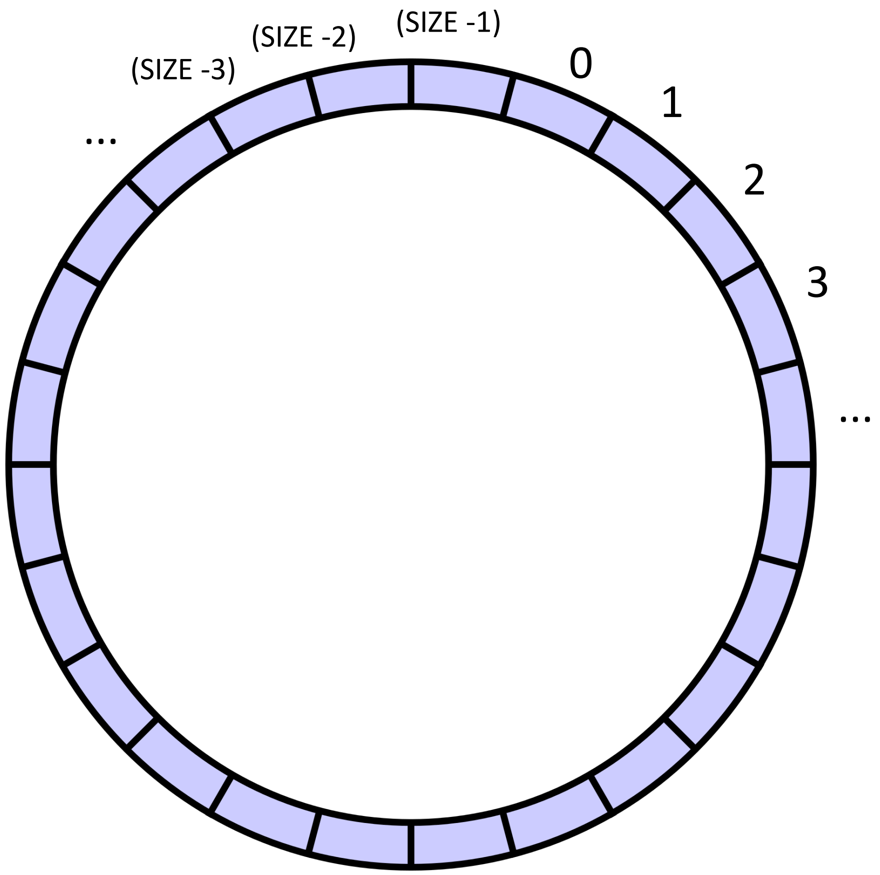


indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

we will assume modular arithmetic:

if x = (SIZE - 1) then
x + 1 == 0;

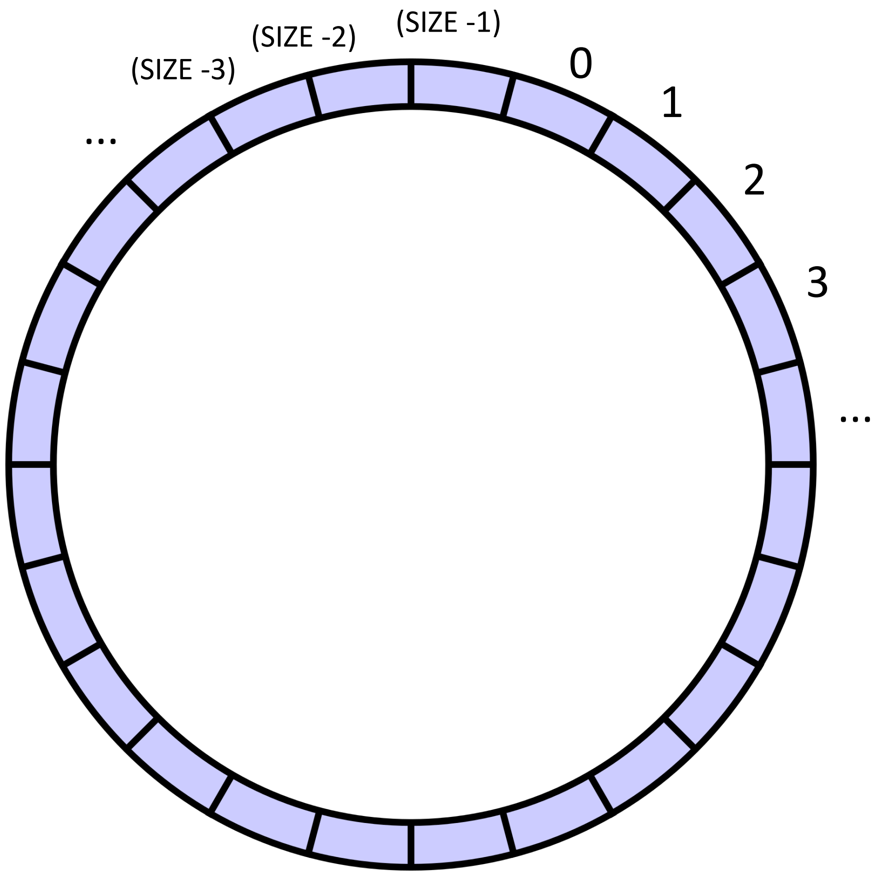
conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

Two variables to keep track of where to deq and enq:

head and tail



indexes will circulate in order and wrap around

conceptually it is a circle

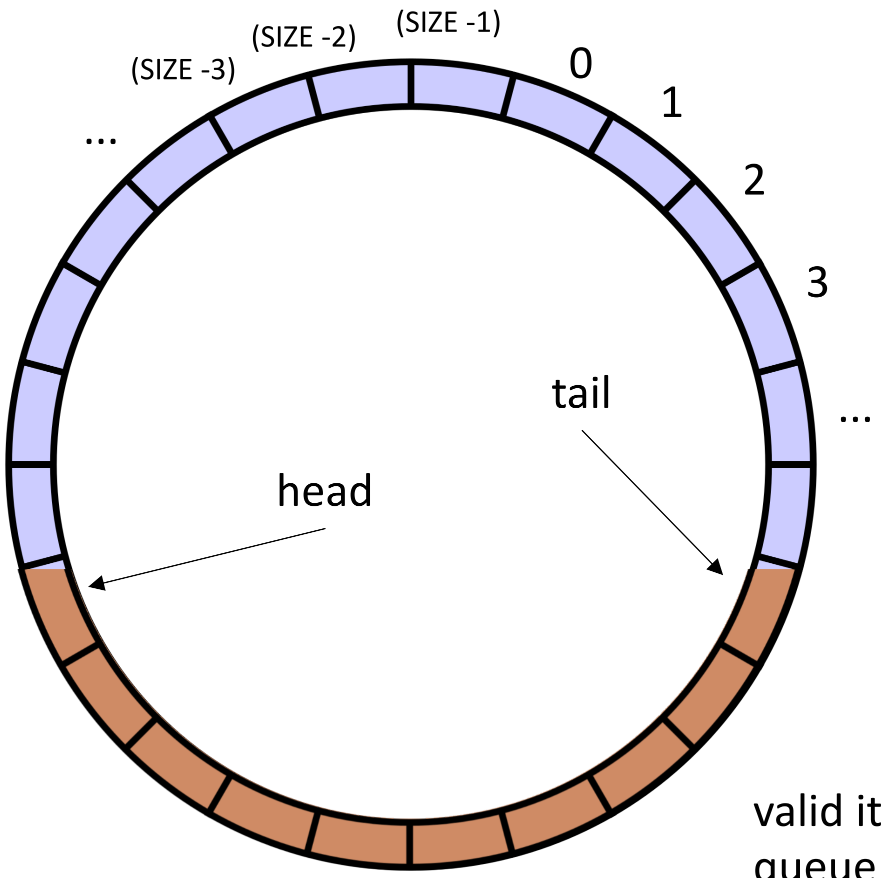
Producer Consumer Queues

- Start with a fixed size array

Two variables to keep track of where to deq and enq:

head and tail:

enq to the head, deq from the tail



indexes will circulate in order and wrap around

valid items in the queue

conceptually it is a circle

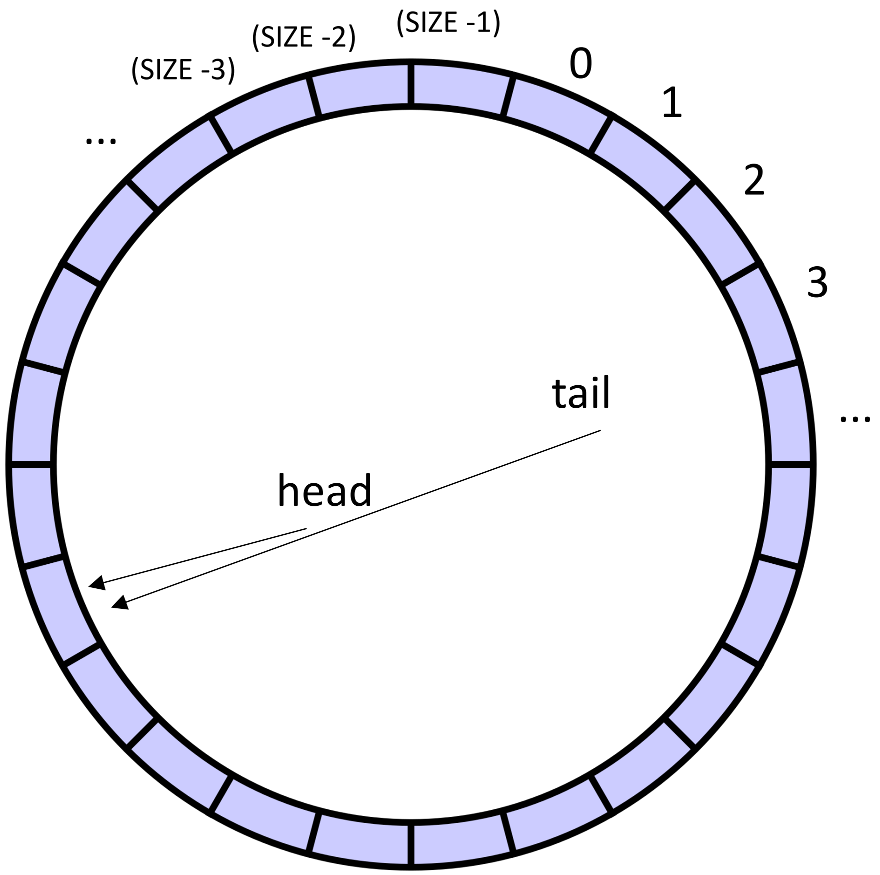
Producer Consumer Queues

- Start with a fixed size array

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

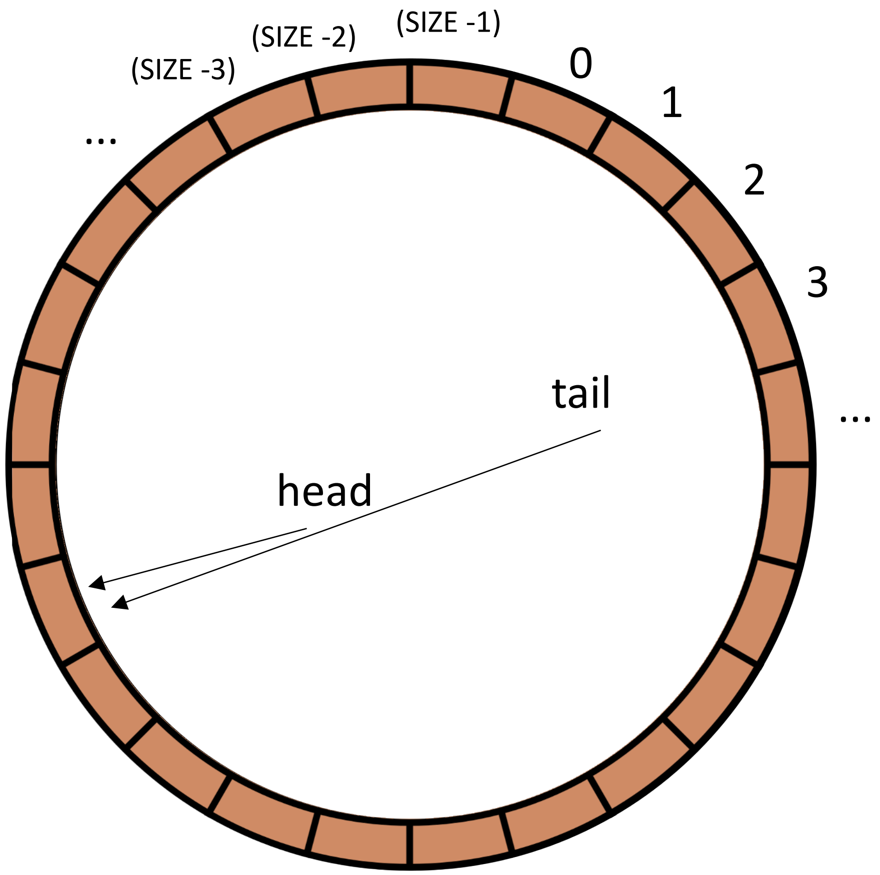


indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

Full queue is when head == tail?

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

Two variables to keep track of where to deq and enq:

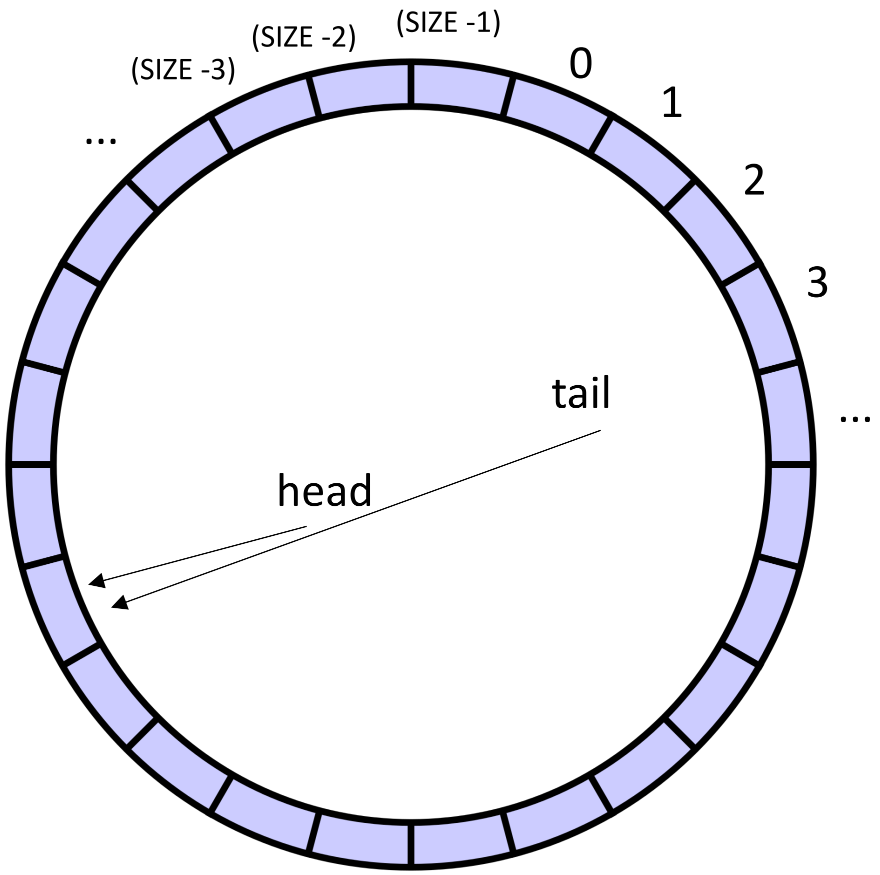
head and tail

Empty queue is when head == tail

Full queue is when head == tail?

conceptually it is a circle

but then
how to tell
full queue from
empty?



indexes will circulate in order and wrap around

Producer Consumer Queues

- Start with a fixed size array

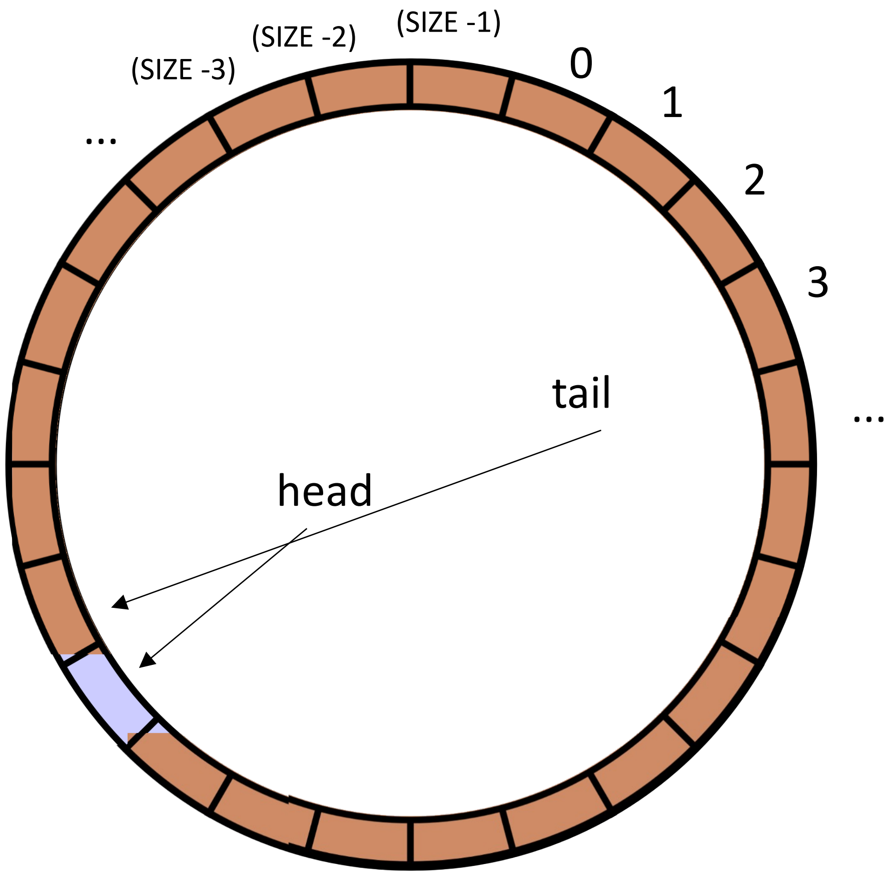
Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

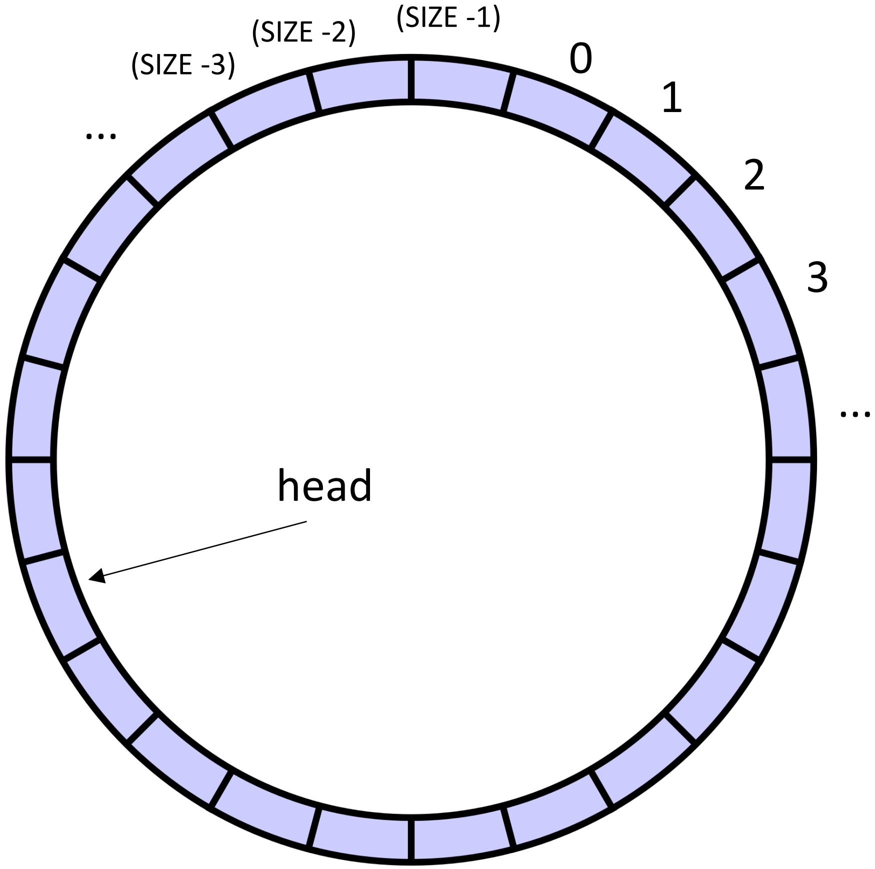
Full queue is when head + 1 == tail

conceptually it is a circle

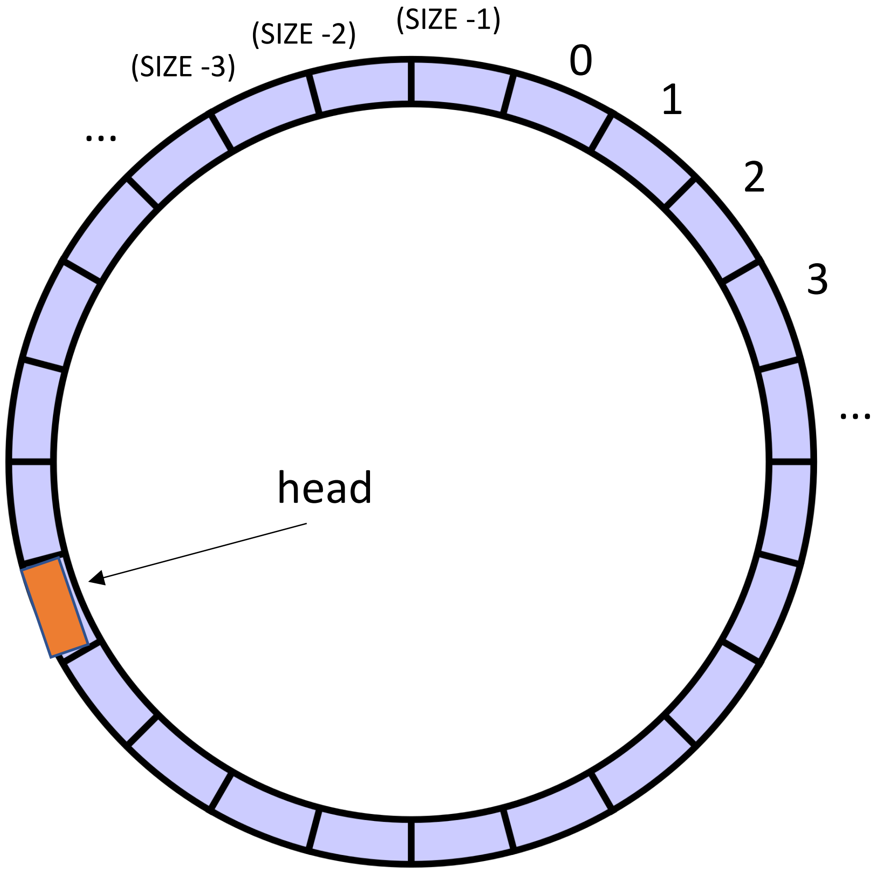


wasting one location, but its okay...

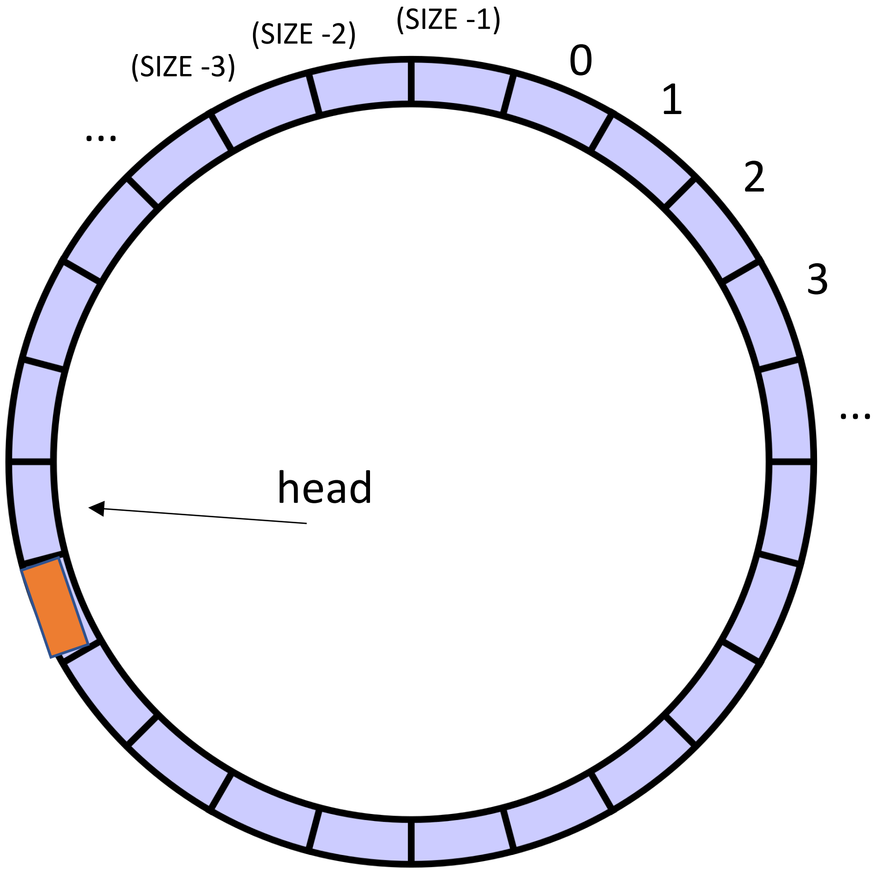
indexes will circulate in order and wrap around



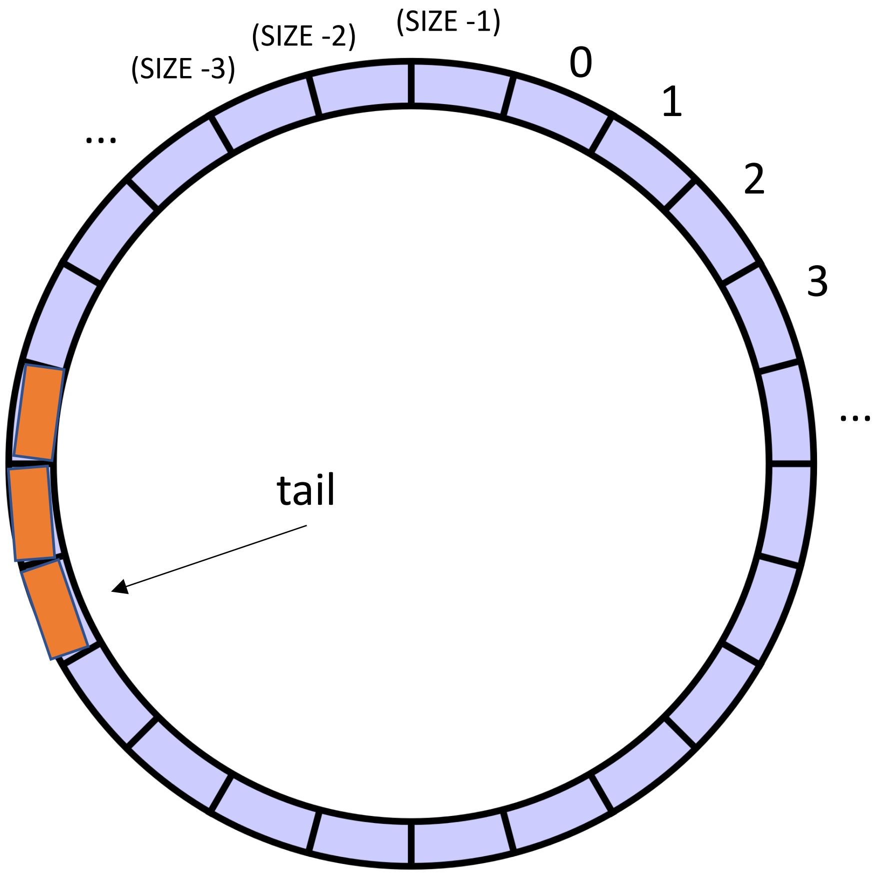
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```



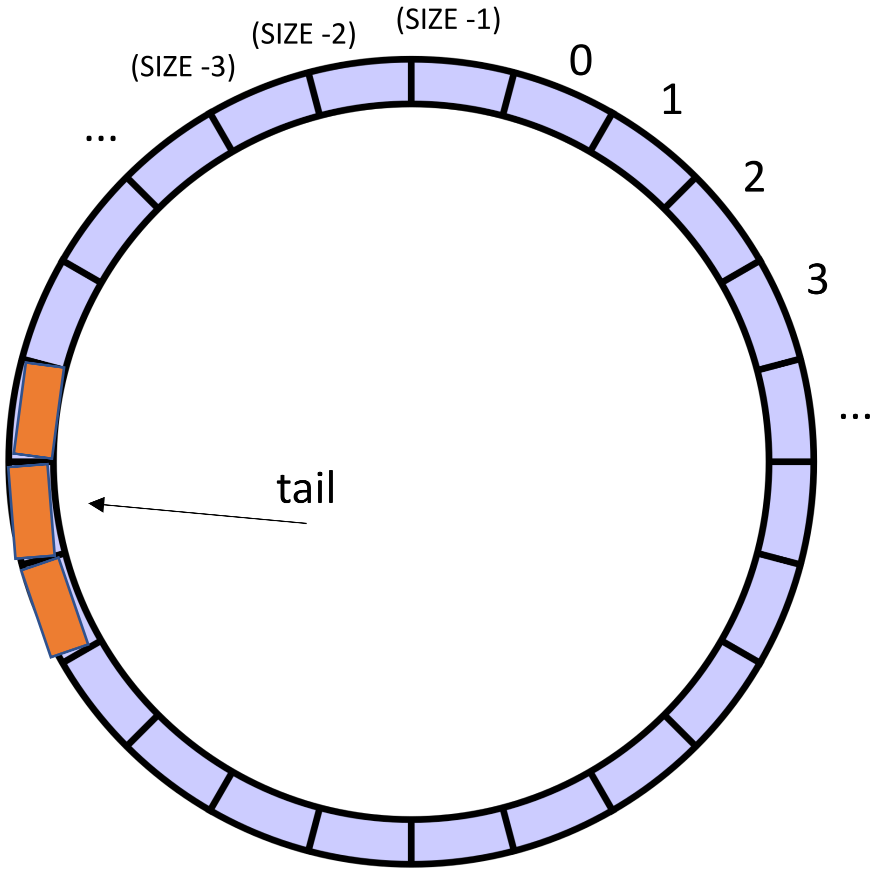
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```

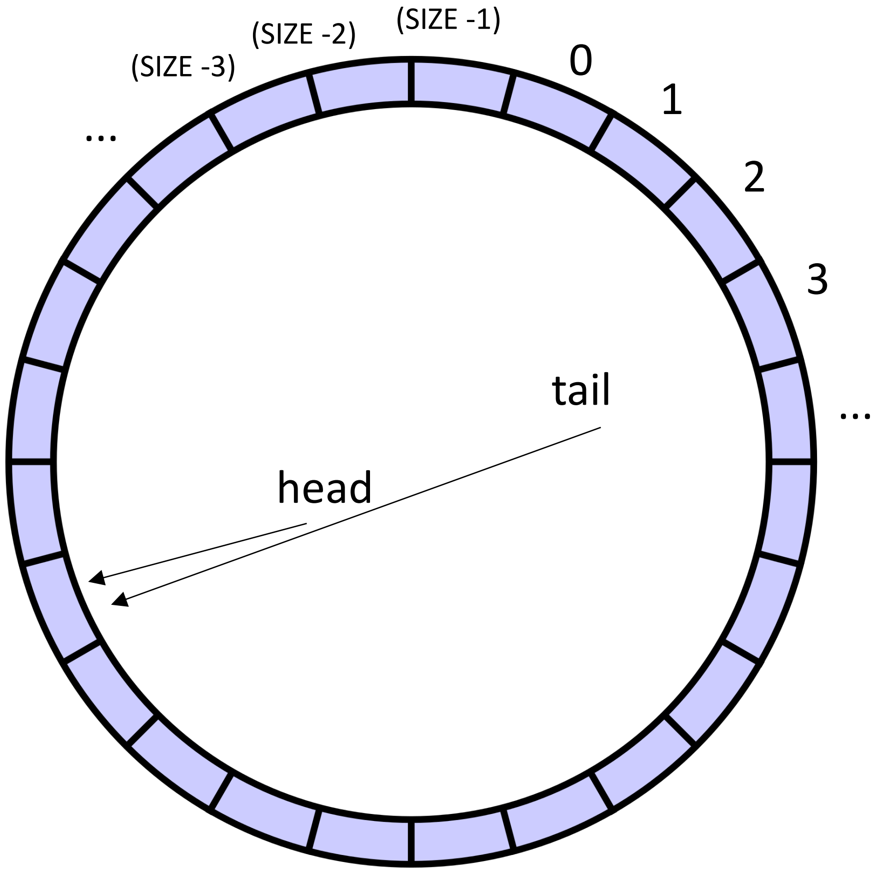


```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```



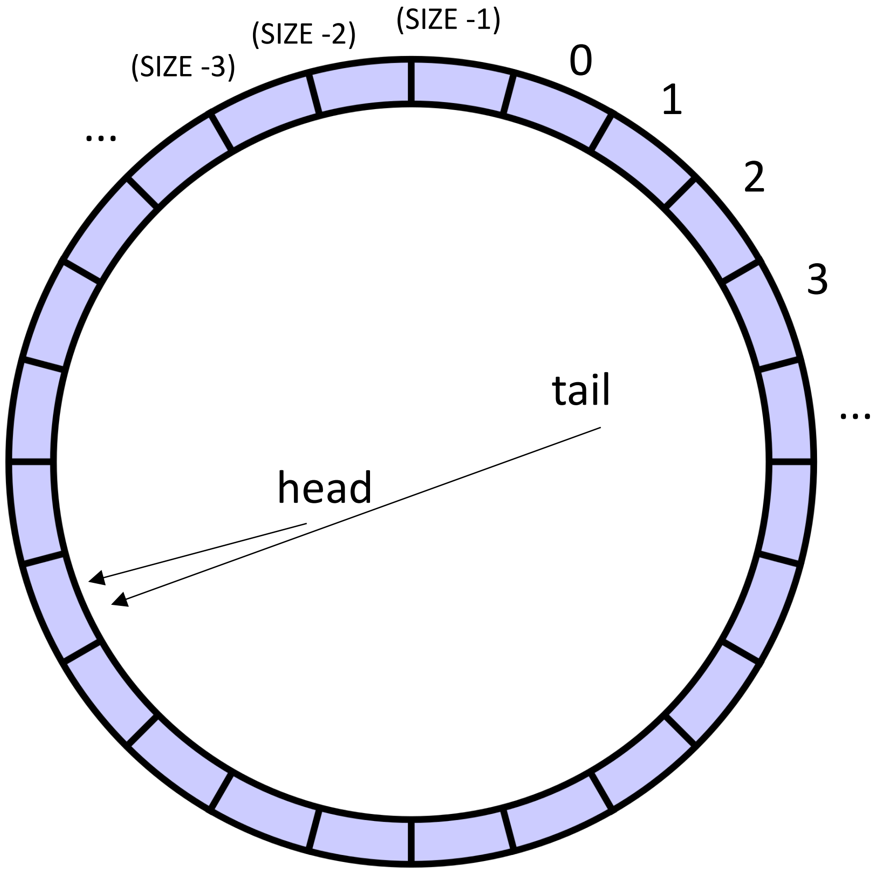
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

This looks like the two threads don't even share head and tail! What is missing?

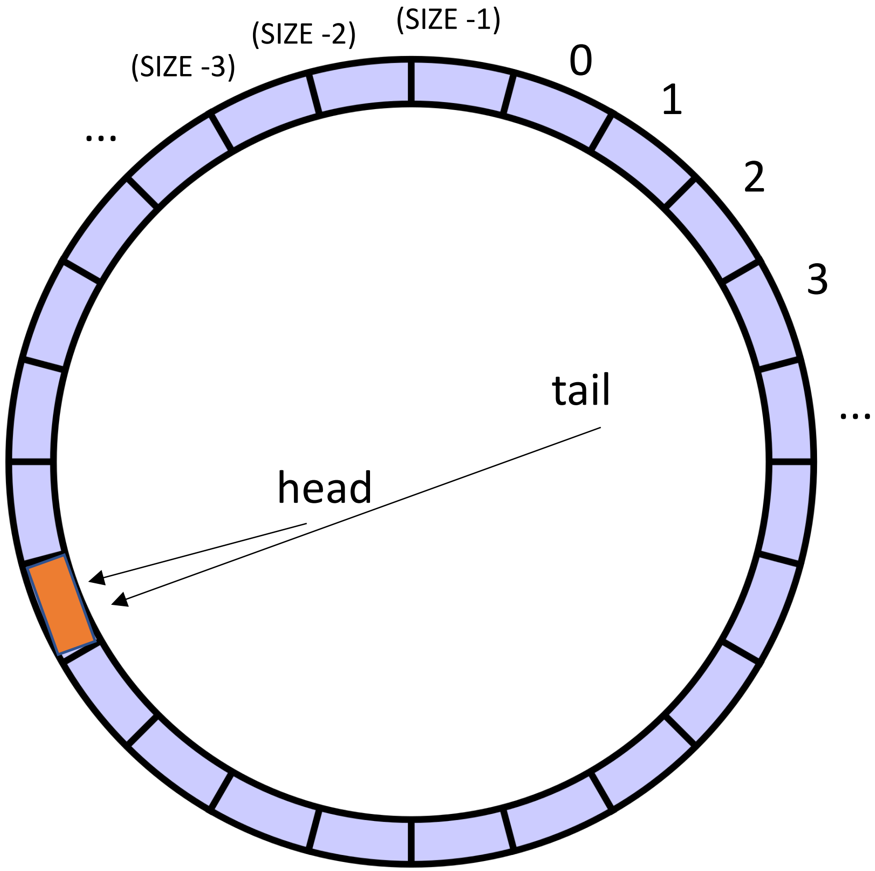


```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

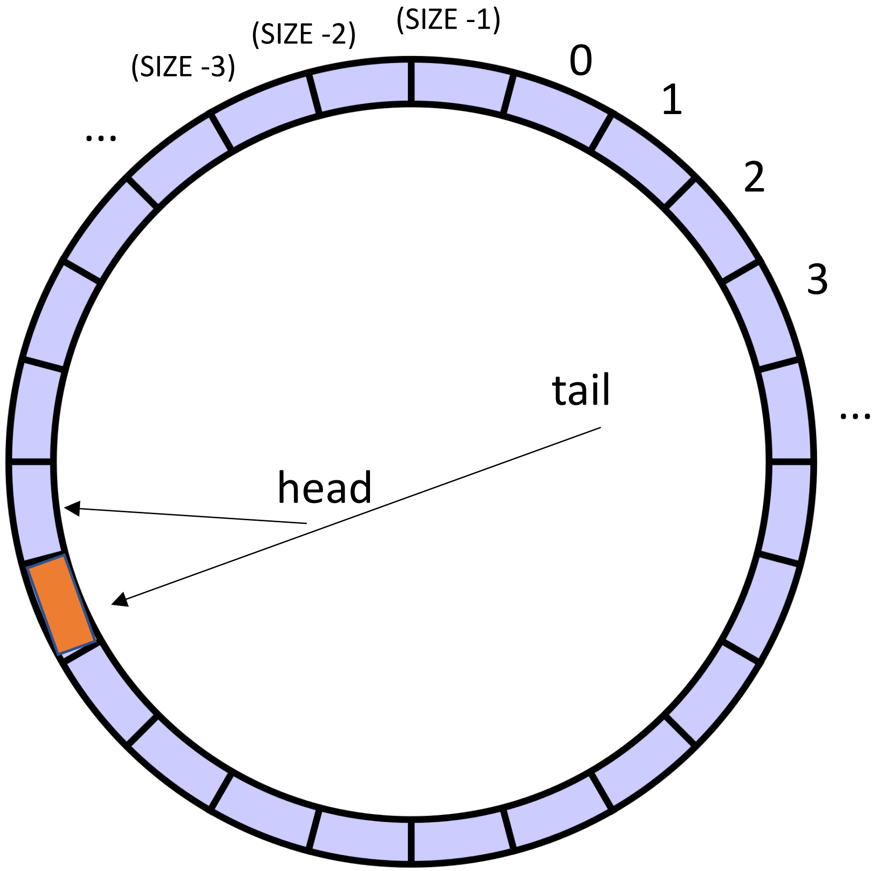
what happens if we try to dequeue here?



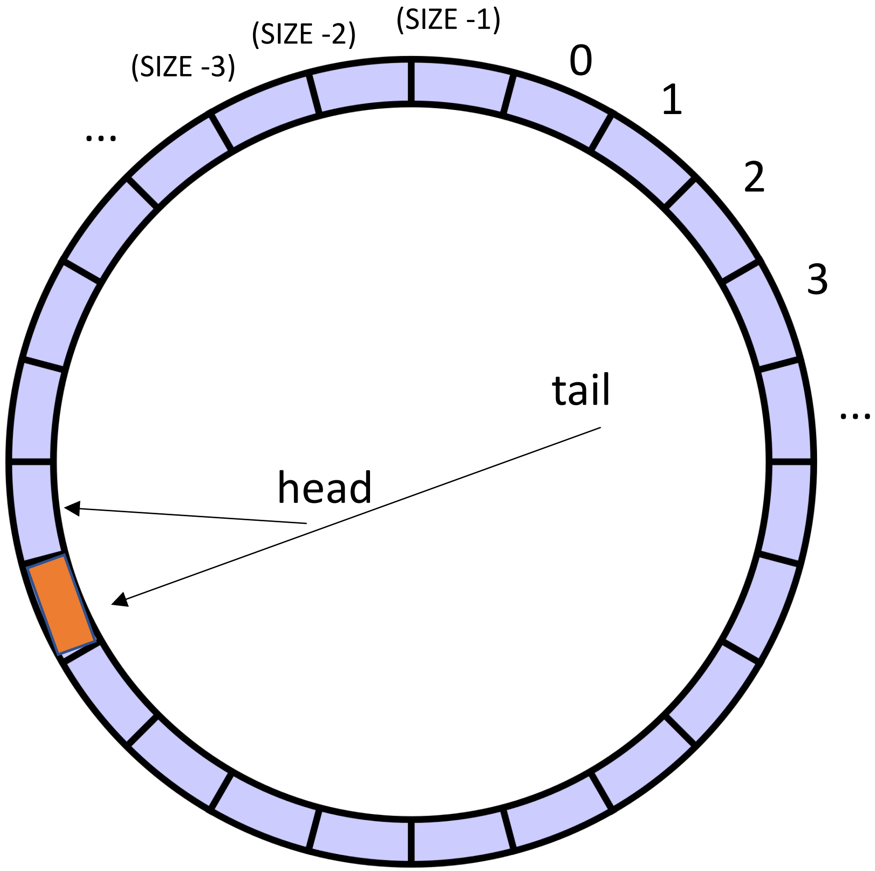
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



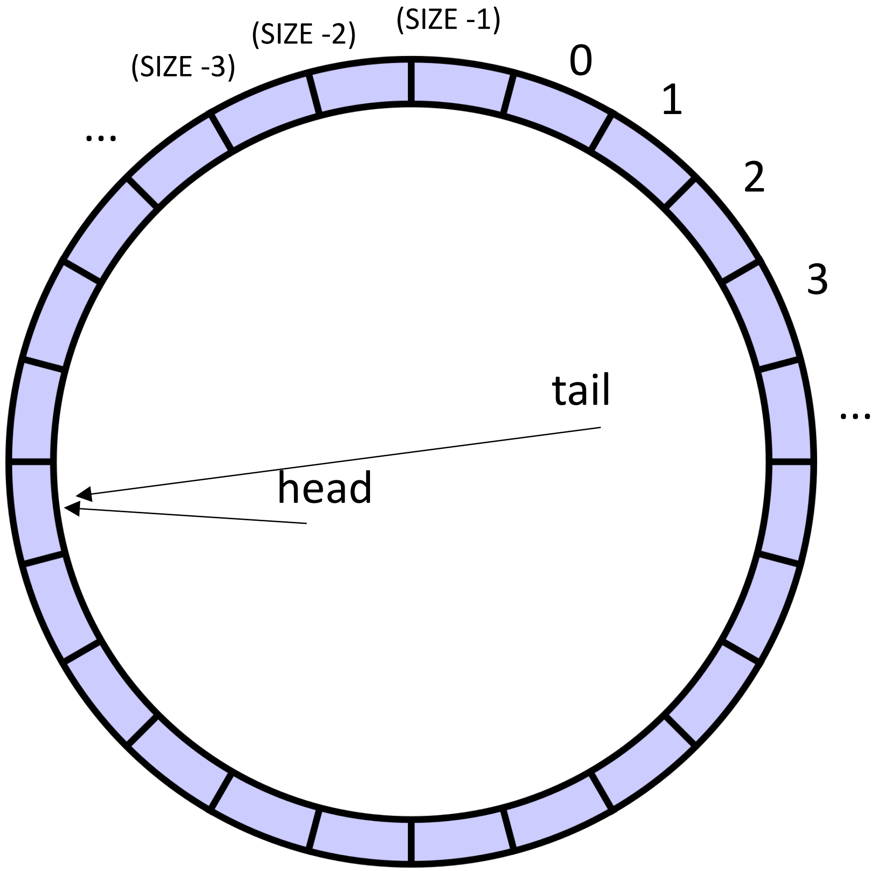
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



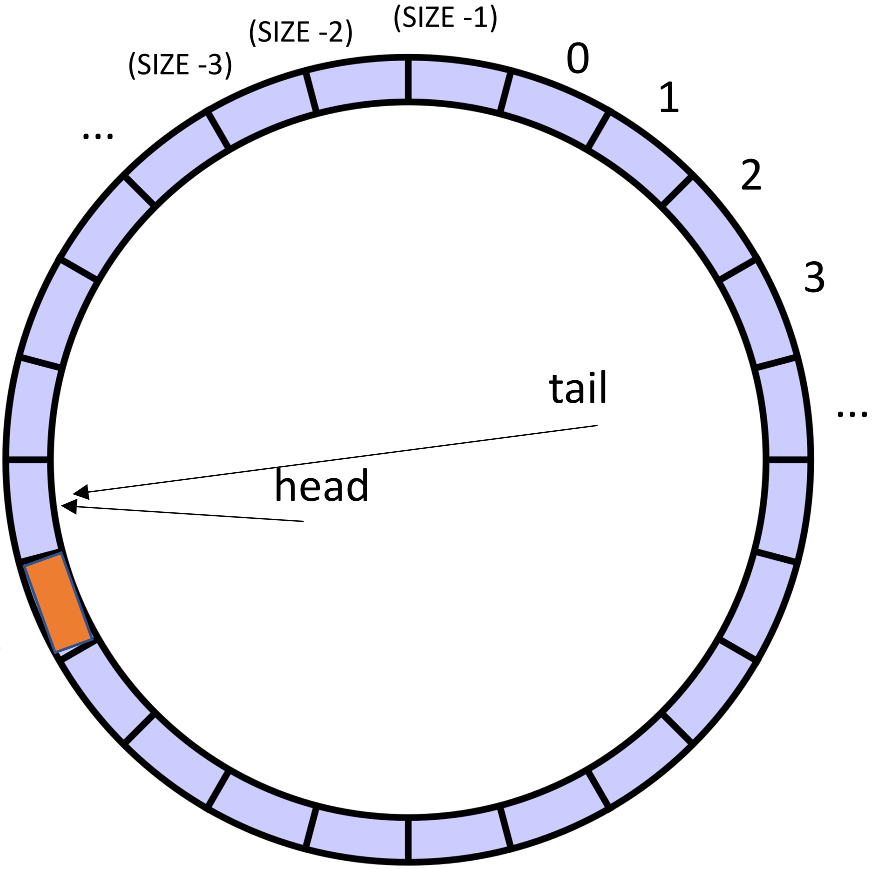
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

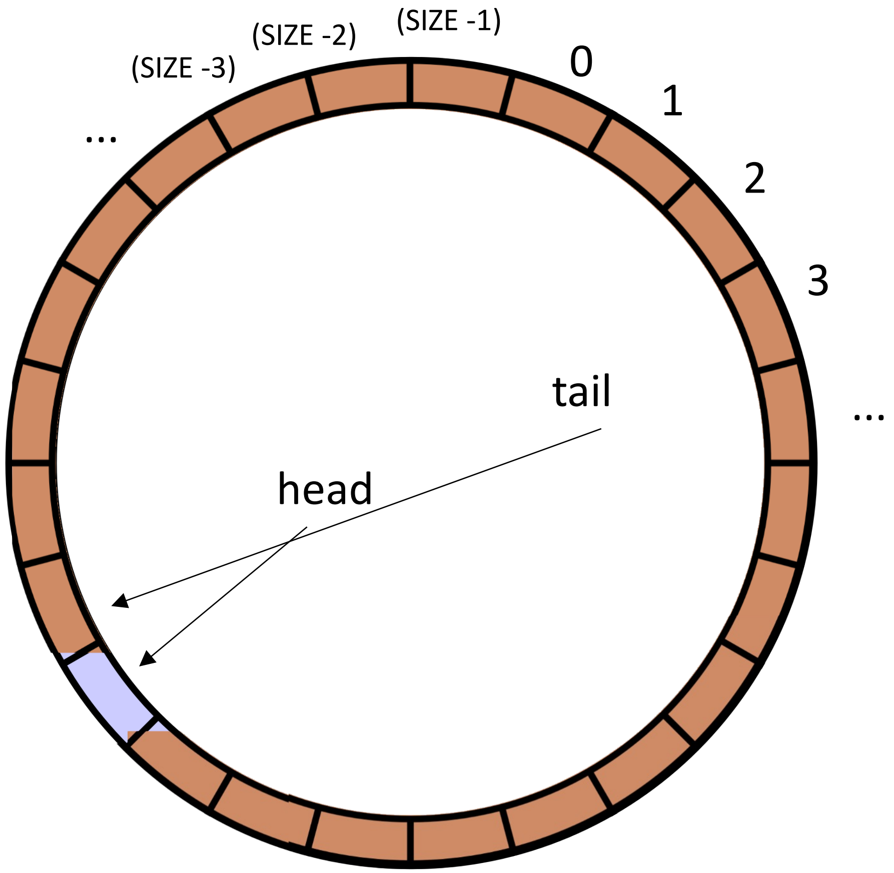


```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



What value is stored here?

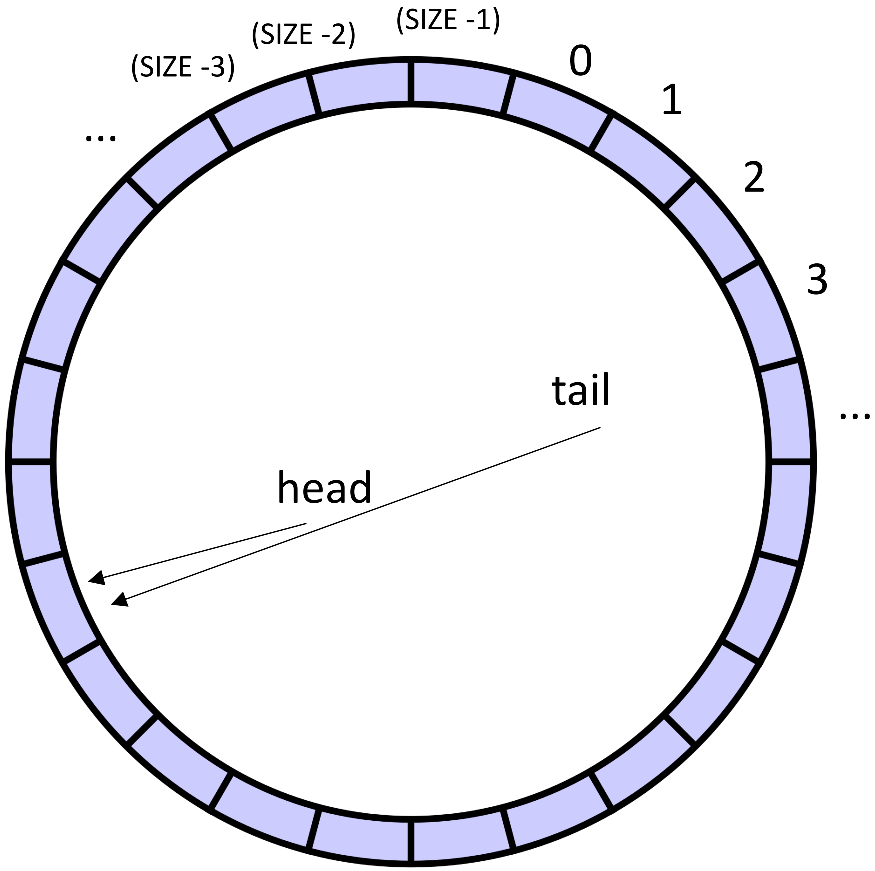
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

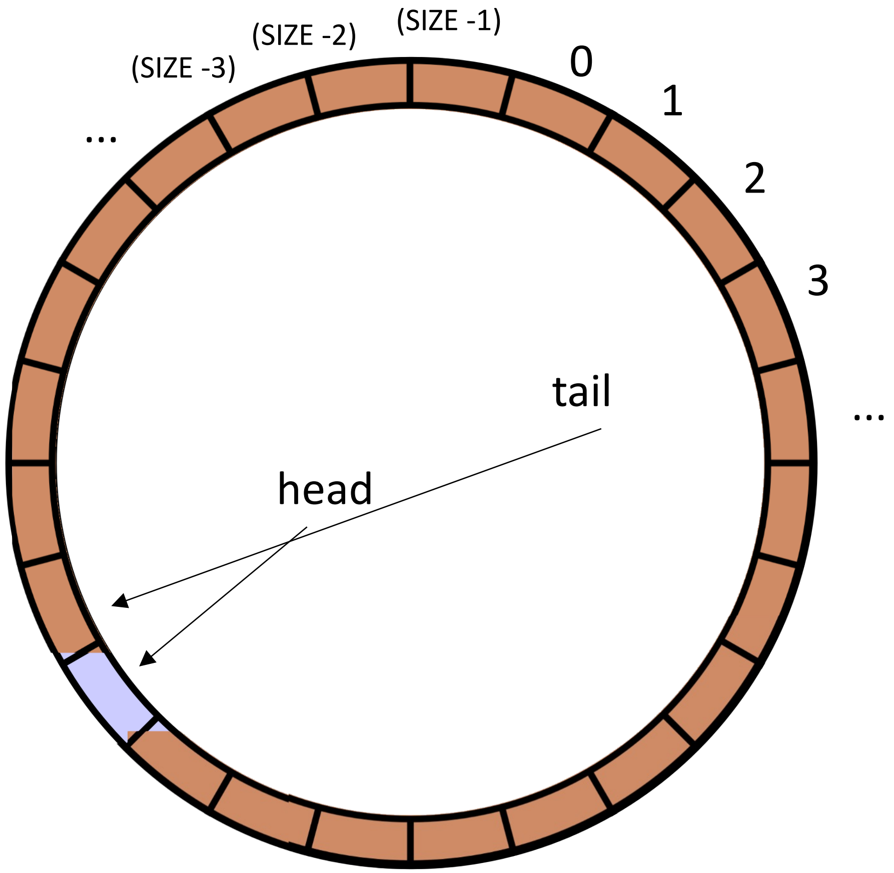
similarly for enqueue

but why can’t we enqueue?



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

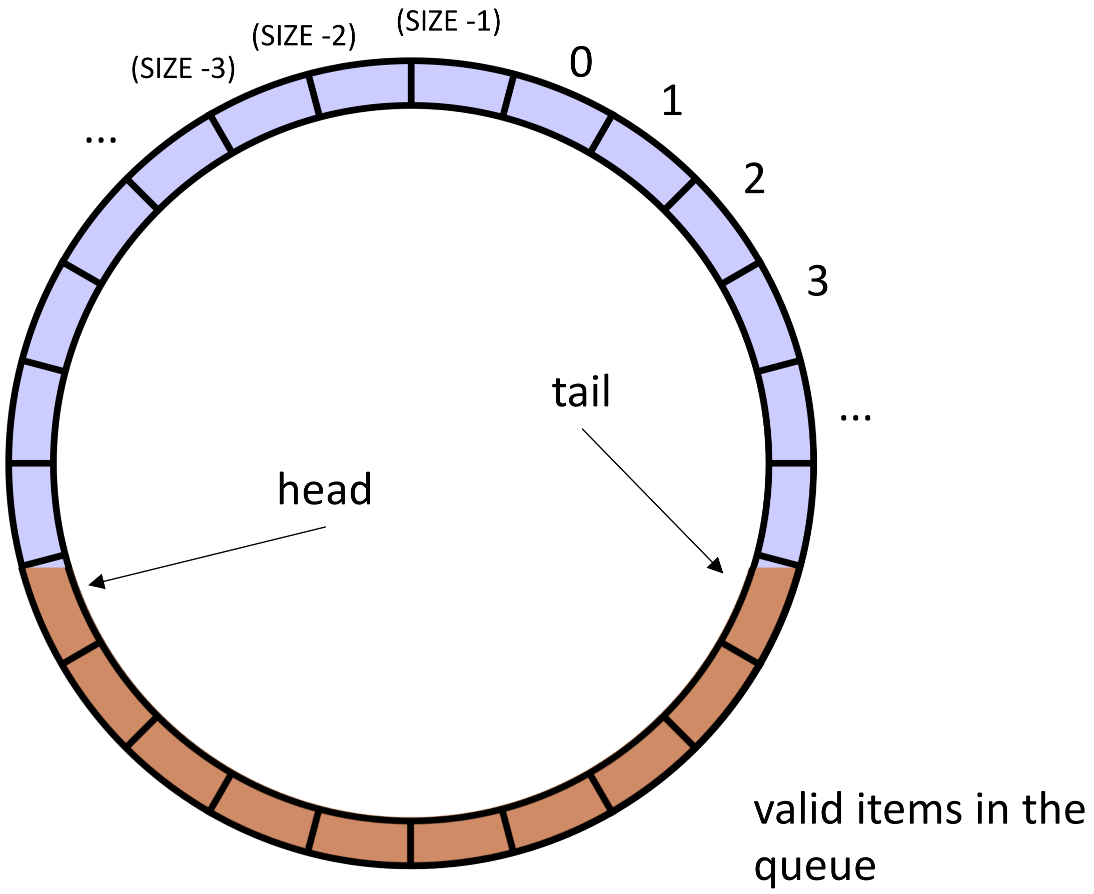
incrementing the head would make it empty!



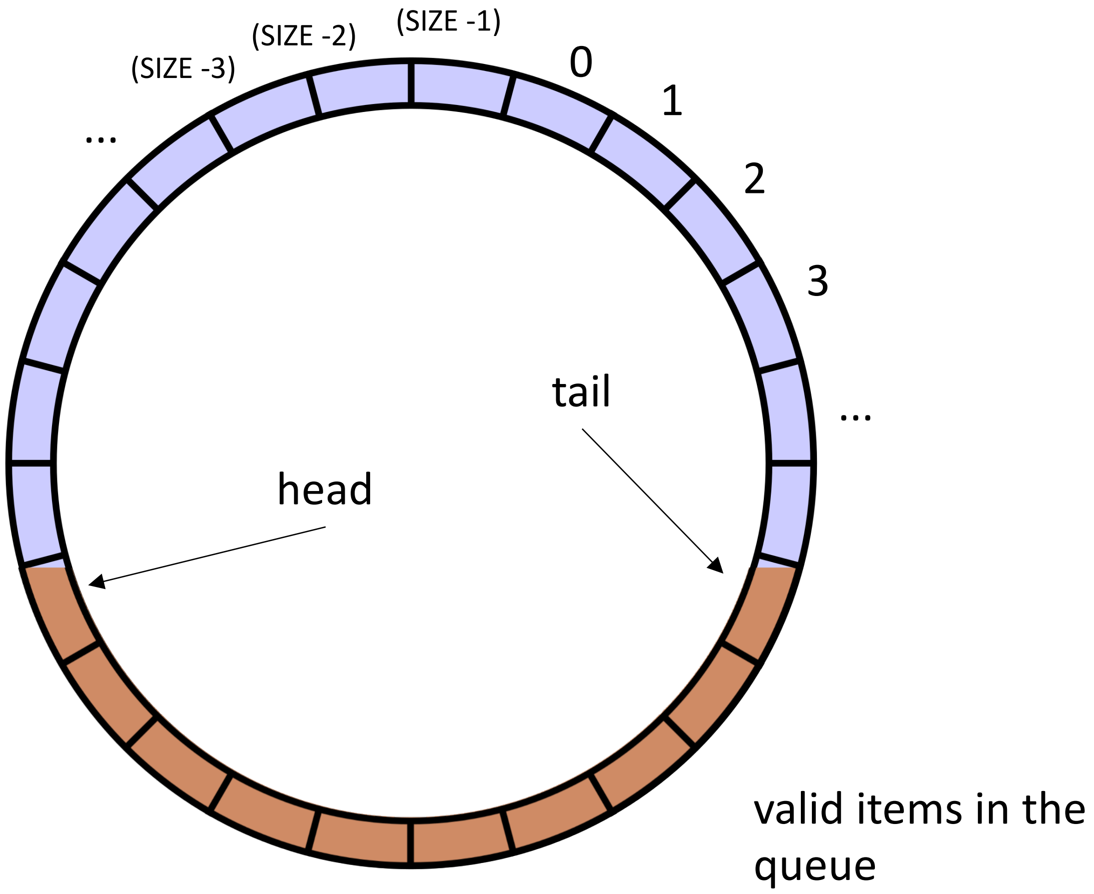
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

we need to wait for there
to be room

Other questions:



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



Other questions:

Do these need to be atomic RMWs?

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

Next topic

- Work stealing

Schedule

- Workstealing
    - **DOALL Loops**
    - Parallel Schedules
        - Static schedule
        - Global worklist
        - Local worklists

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

are they the same if you traverse them backwards?

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] += a[i+1]
}
```

are they the same if you traverse them backwards?

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] += a[i+1]
}
```

No!

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

what about a random order?

```
for (pick i randomly) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (pick i randomly) {
  a[i] += a[i+1]
}
```

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

what about a random order?

```
for (pick i randomly) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (pick i randomly) {
  a[i] += a[i+1]
}
```

No!

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

These are **DOALL** loops:

- Loop iterations are independent
- You can do them in ANY order and get the same results

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

These are **DOALL** loops:

- Loop iterations are independent
- You can do them in ANY order and get the same results

- Most importantly: you can do the iterations in parallel!
- Assign each thread a set of indices to compute

DOALL Loops

- Given a nest of For loops, can we make the outer-most loop parallel?
    - Safely
    - Efficiently

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
for (int i = 0; i < dim1; i++) {
  for (int j = 0; j < dim3; j++) {
    for (int k = 0; k < dim2; k++) {
      a[i][j] += b[i][k] * c[k][j];
    }
  }
}
```

matrix multiplication
example

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

DOALL Loops

- Given a nest of ***candidate*** *For* loops, determine if we can we make the outer-most loop parallel?
    - Safely
    - efficiently

- Criteria: every iteration of the outer-most loop must be *independent*
    - The loop can execute in any order, and produce the same result

Safety Criteria

- How do we check this?
  - If the property doesn’t hold then there exists 2 iterations, such that if they are re-ordered, it causes different outcomes for the loop.
  - **Write-Write conflicts**: two distinct iterations write different values to the same location
  - **Read-Write conflicts**: two distinct iterations where one iteration reads from the location written to by another iteration.

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable
Computation to store in the memory location

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

index(i_x) != index(i_y)

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:

$\mathtt{i_x\ !=\ i_y}$

Check:

$\mathtt{index(i_x)\ !=\ index(i_y)}$

Why?

Because if

$\mathtt{index(i_x)\ ==\ index(i_y)}$

then:

$\mathtt{a[index(i_x)]}$ will equal either $\mathtt{loop(i_x)}$ or $\mathtt{loop(i_y)}$ depending on the order

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
   a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

write_index(i_x) != read_index(i_y)

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
   a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:

`i`$_x$ `!=` `i`$_y$

Check:

`write_index(i`$_x$`) != read_index(i`$_y$`)`

Why?

if `i`$_x$ iteration happens first, then iteration `i`$_y$ reads an updated value.

if `i`$_y$ happens first, then it reads the original value

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

Schedule

- DOALL Loops

- **Parallel Schedules:**
  - Static
  - Global Worklists
  - Local Worklists

Parallel Schedules

- Consider the following program:

There are 3 arrays: `a, b, c.`

We want to compute

```
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
```

Is this a DOALL loop?

Parallel Schedules

- Consider the following program:

There are 3 arrays: `a, b, c.`

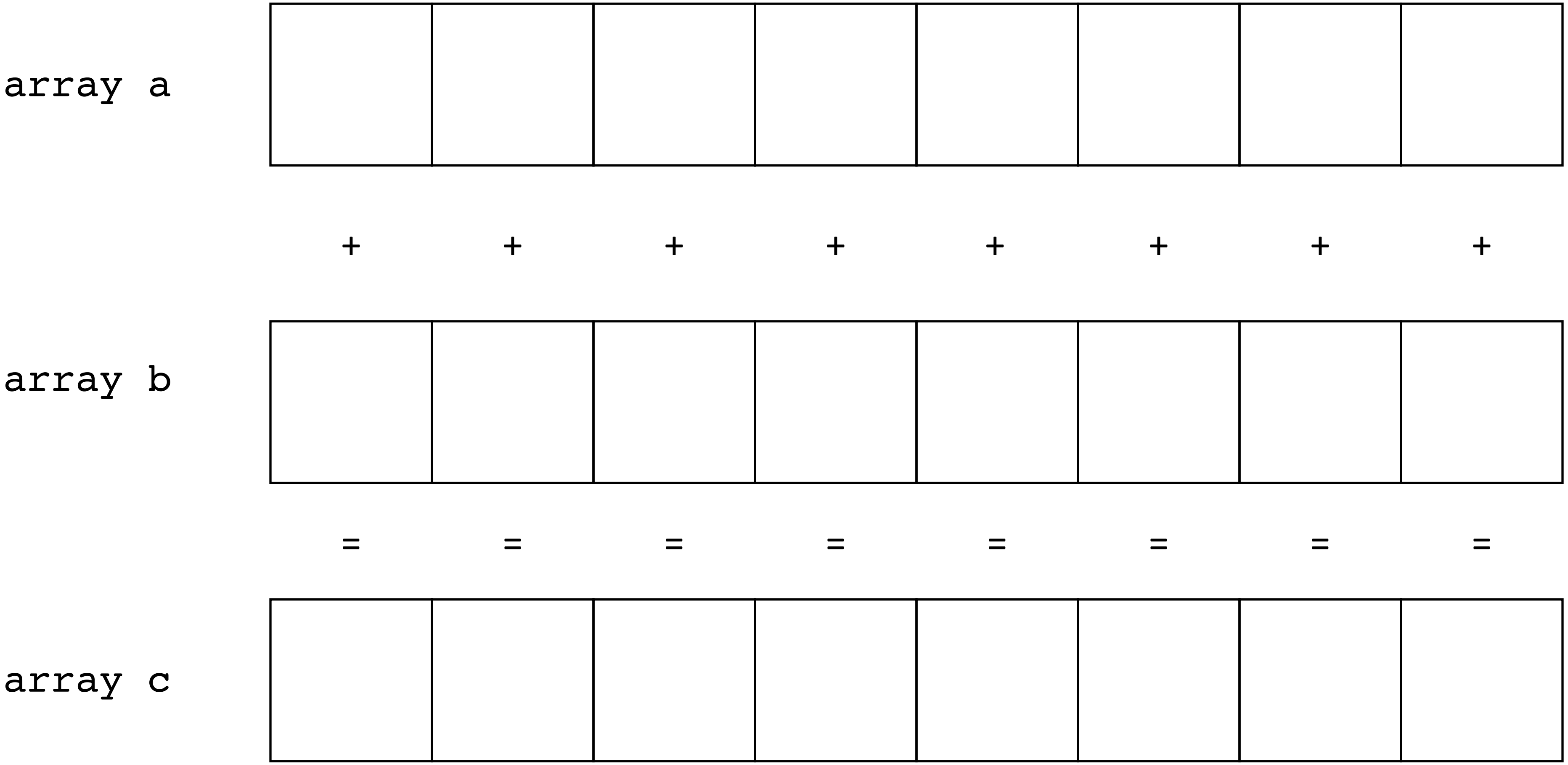
We want to compute

```
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
```

Is this a DOALL loop?

How should we parallelize it?

Parallel Schedules



array a

+ + + + + + + +

array b

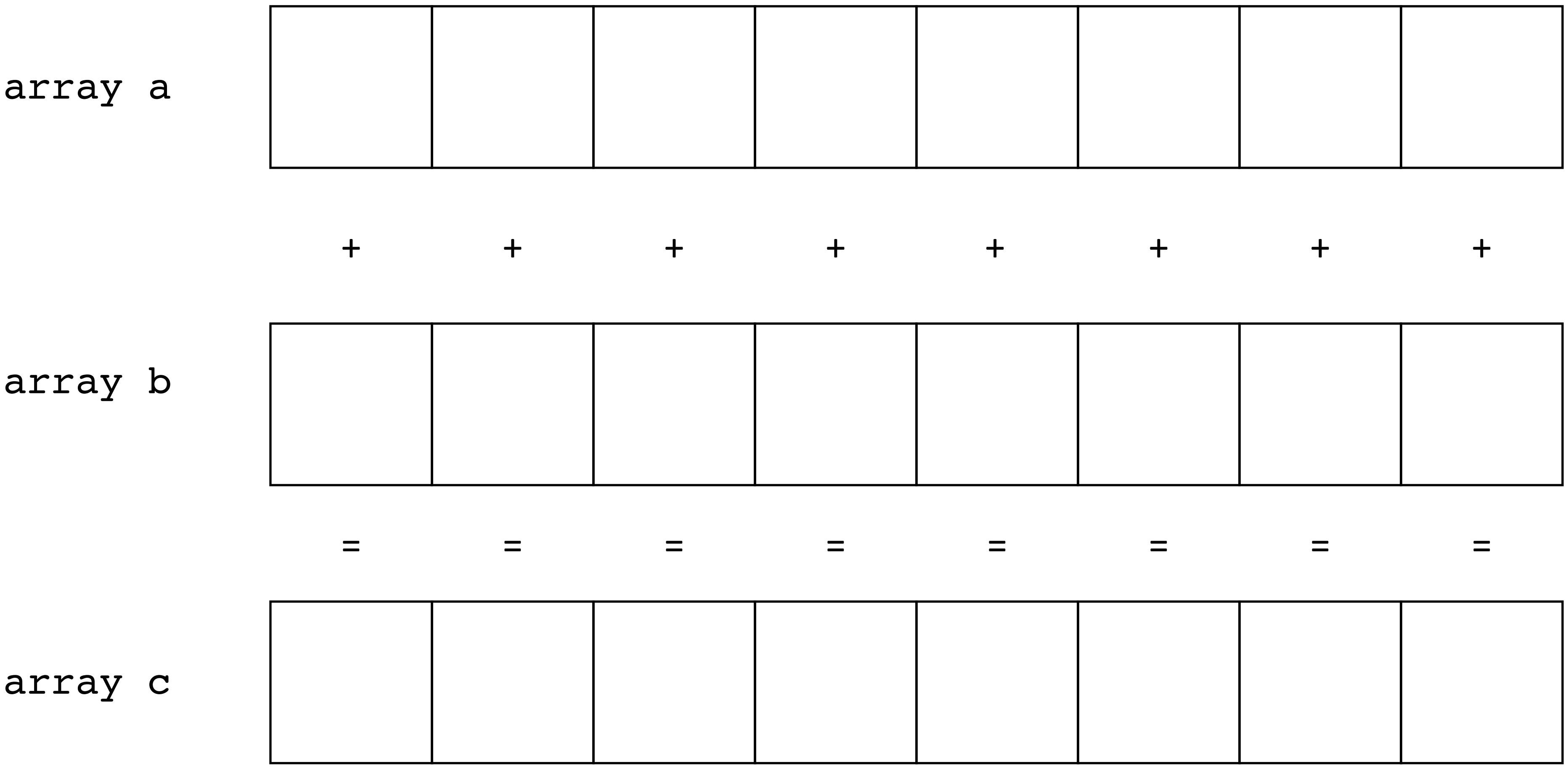
= = = = = = = =

array c

Parallel Schedules

Computation can easily be divided into threads

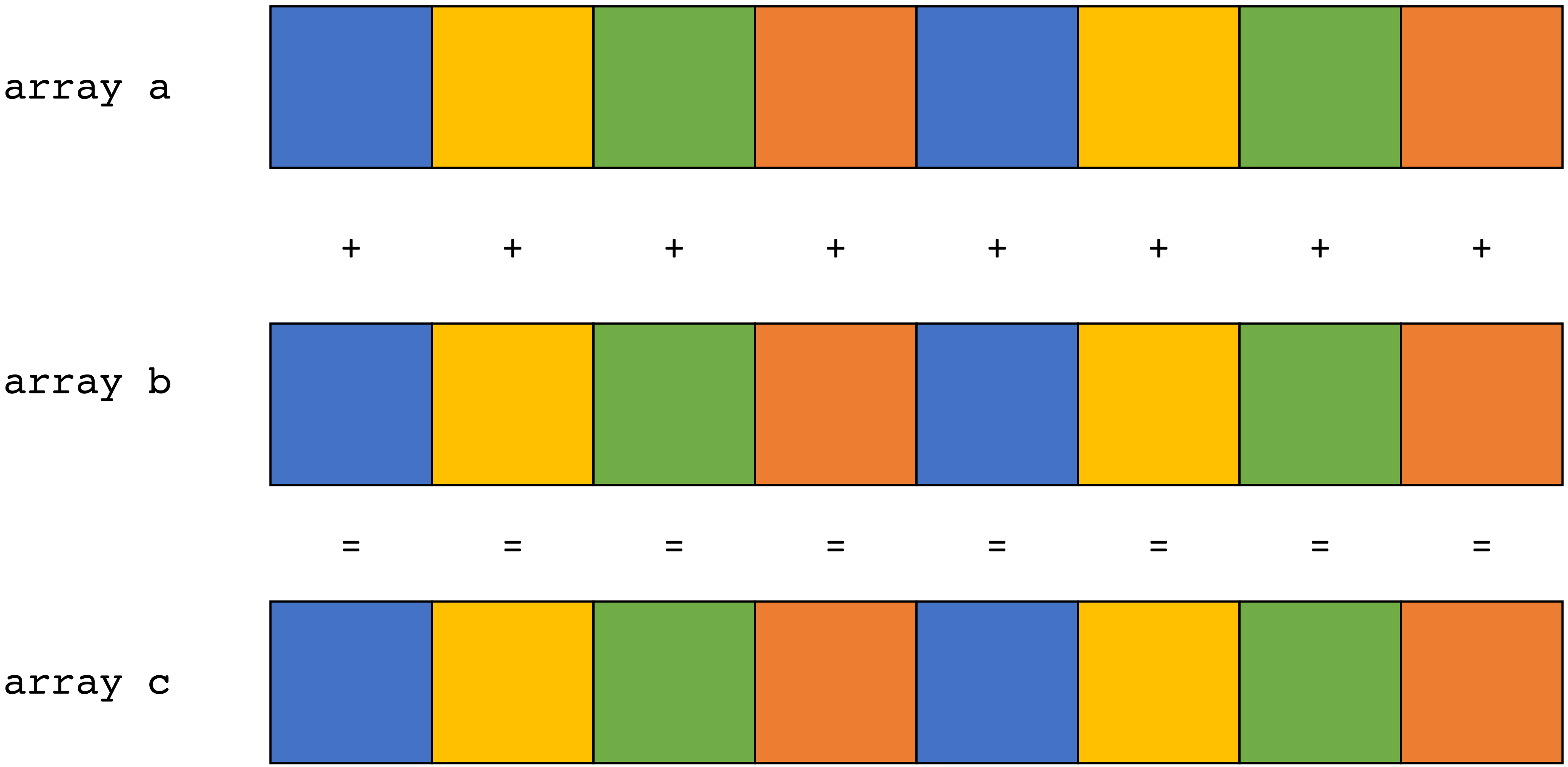
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



array a

+ + + + + + + +

array b

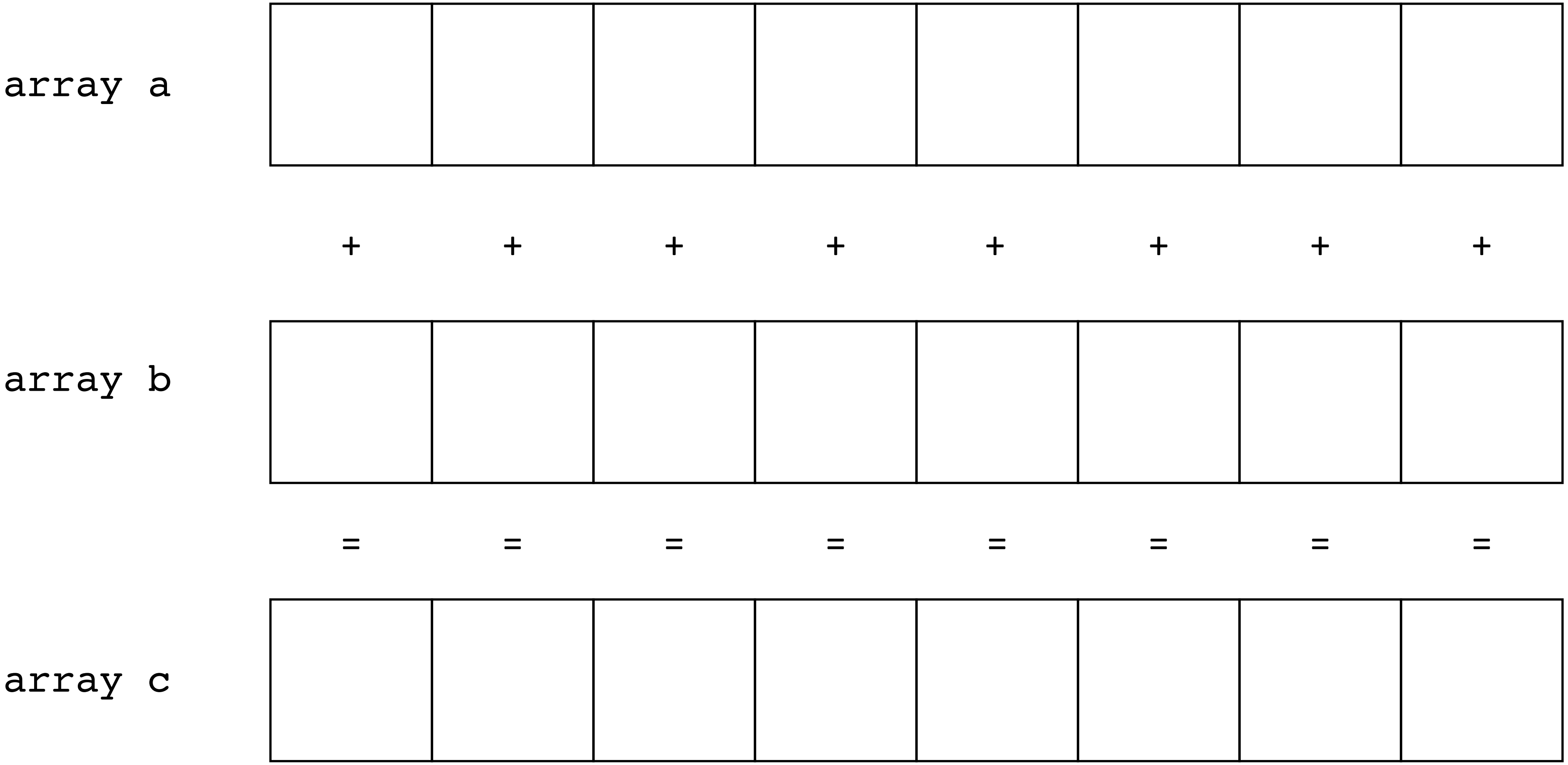
= = = = = = = =

array c

Parallel Schedules

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



`array a`

+ \+ \+ \+ \+ \+ \+ \+

`array b`

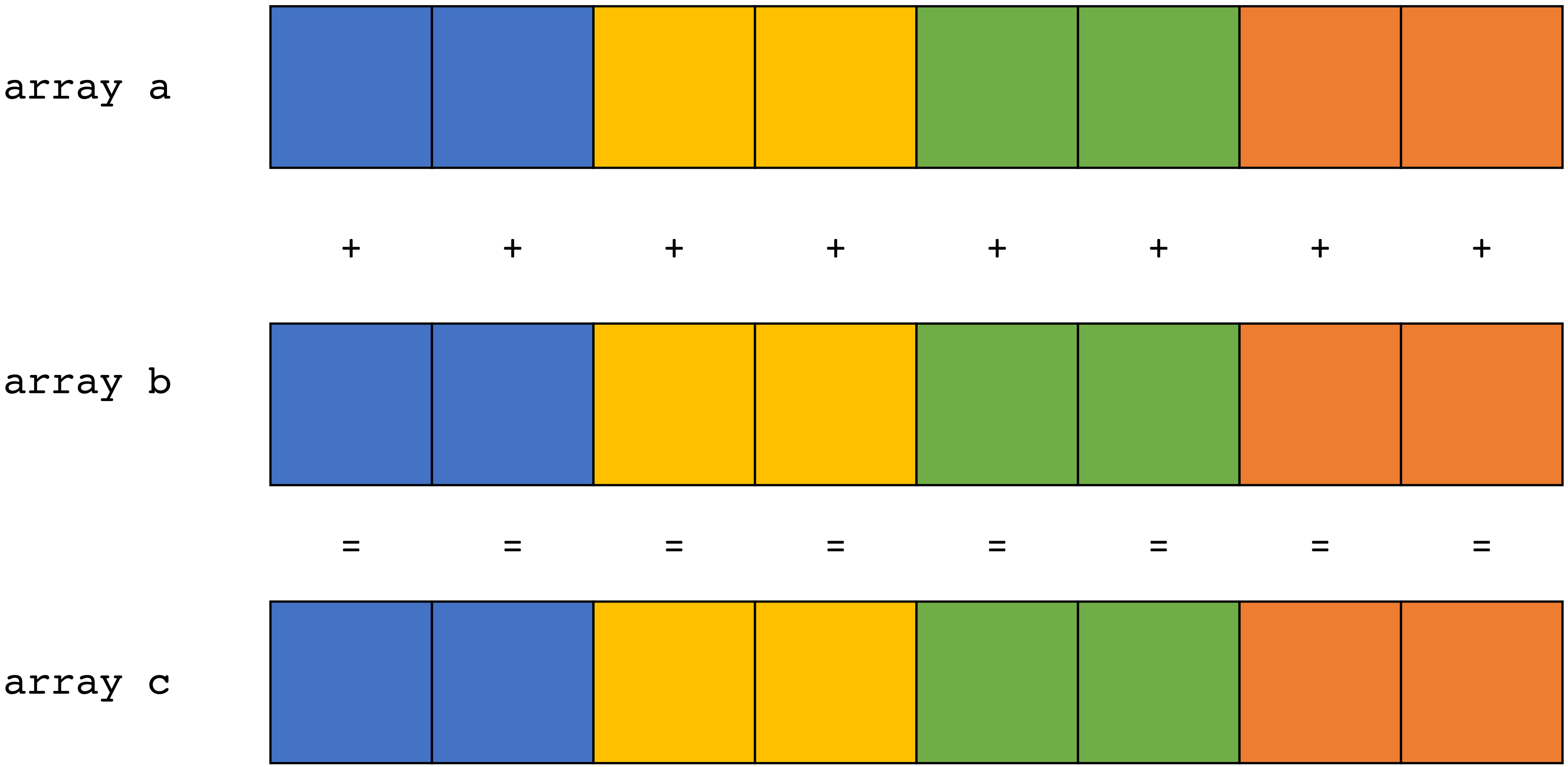
= = = = = = = =

`array c`

Parallel Schedules

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

- Which one is more efficient?

Parallel Schedules

- Which one is more efficient?

- These are called Parallel Schedules for DOALL Loops
- We will discuss several of them.

Schedule

- DOALL Loops

- **Parallel Schedules:**
  - **Static**
  - Global Worklists
  - Local Worklists

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

0	1	2	3	4	5	6	7		SIZE -1

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

say SIZE / NUM_THREADS = 4

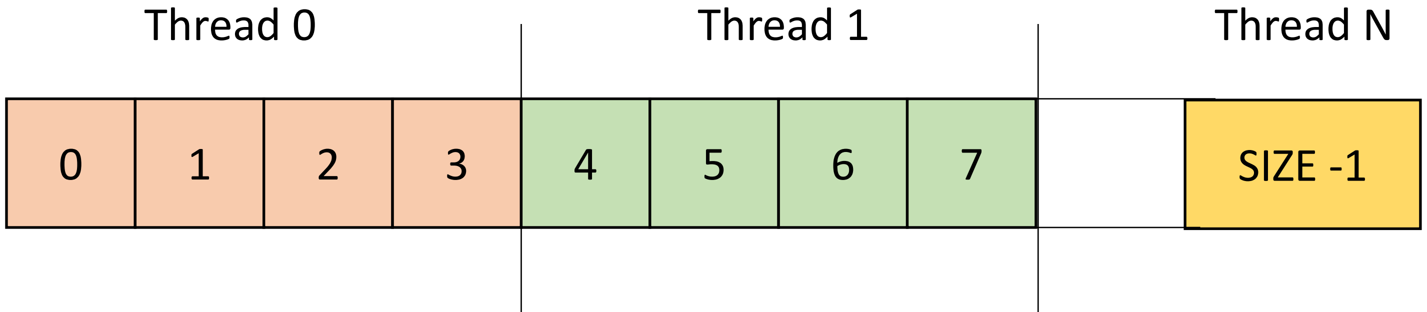
0	1	2	3	4	5	6	7		SIZE -1

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

say SIZE / NUM_THREADS = 4



Thread 0 | Thread 1 | Thread N

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | SIZE -1

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

make a new function with the for loop inside. Pass all needed variables as arguments. Take an extra argument for a thread id

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
~~  for (int x = 0; x < SIZE; x++) {~~
~~  // Each iteration takes roughly~~
~~  // equal time~~
~~  }~~
...
}
```

```
void parallel_loop(..., int tid, int num_threads)
{

  for (int x = 0; x < SIZE; x++) {
    // work based on x
  }
}
```

make a new function with the for loop inside. Pass all needed variables as arguments. Take an extra argument for a thread id

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
~~  for (int x = 0; x < SIZE; x++) {~~
~~  // Each iteration takes roughly~~
~~  // equal time~~
~~  }~~
...
}
```

```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  for (int x = 0; x < SIZE; x++) {
    // work based on x
  }
}
```

determine chunk size in new function

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  ~~for (int x = 0; x < SIZE; x++) {~~
  ~~// Each iteration takes roughly~~
  ~~// equal time~~
  ~~}~~
...
}
```

```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

Set new loop bounds

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int t = 0; t < NUM_THREADS; t++) {
    spawn(parallel_loop(..., t, NUM_THREADS))
  }
  join();
...
}
```

```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

You will need to adapt the thread spawn, join
to C++

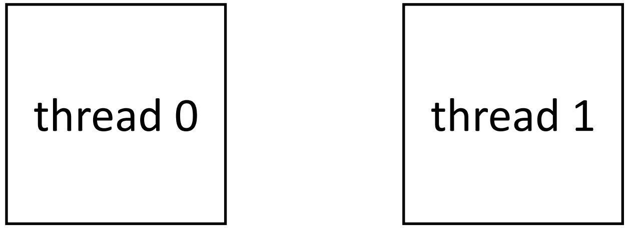
Spawn threads

Static schedule

- Example, 2 threads/cores, array of size 8

0	1	2	3	4	5	6	7

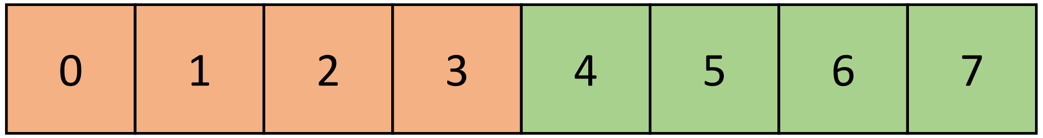
```
      chunk_size = ?

0: start = ?     1: start = ?

0: end = ?       1: end = ?
```



```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

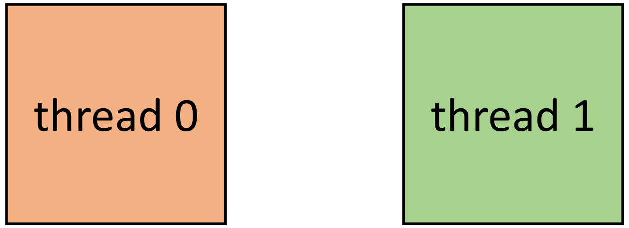
Static schedule

- Example, 2 threads/cores, array of size 8



```
chunk_size = 4
```

```
0: start = 0    1: start = 4
0: end = 4      1: end = 8
```



```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

End example

Next lecture

- Work stealing and generalized concurrent objects

- Get HW 2 turned in today!

- HW 3 is out today. You can get started on Part 1

- Work on midterm