Optimus Prime

Number of lines comparison:

Jesse's ***over-the-top*** implementation: 761 lines

Student's excellent implementation: 166 lines

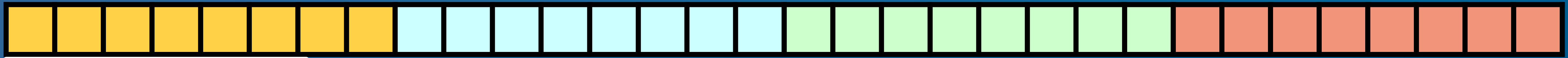
So, student's code, which fully meets all requirements, is 22% the line count of my code!

Did I mention that I like this assignment and have played with it too much?

This is our data structure.

The data member `bits` is the data type `uint32_t`, which has 32 bits, i.e. 4 bytes.

```
typedef struct BitBlock_s {
    uint32_t bits;
    pthread_mutex_t mutex;
} BitBlock_t;
```



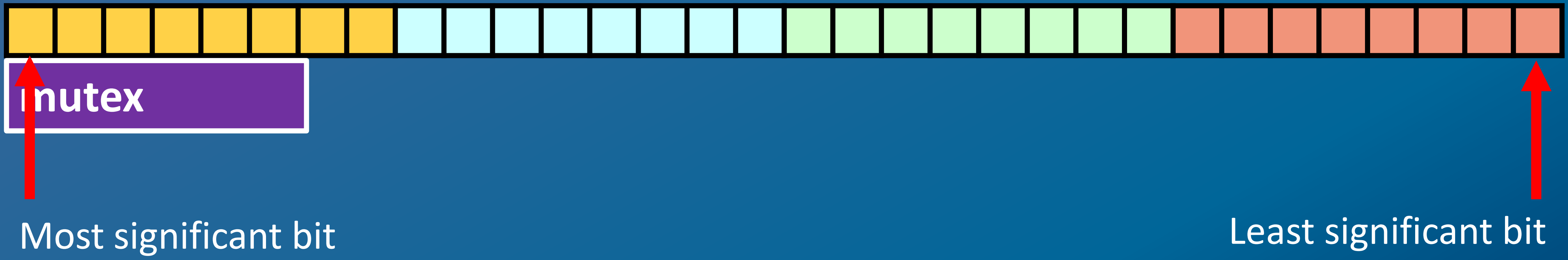
mutex

I like to think of this as 32 bits, with a mutex sidecar attached.

No doggles required!

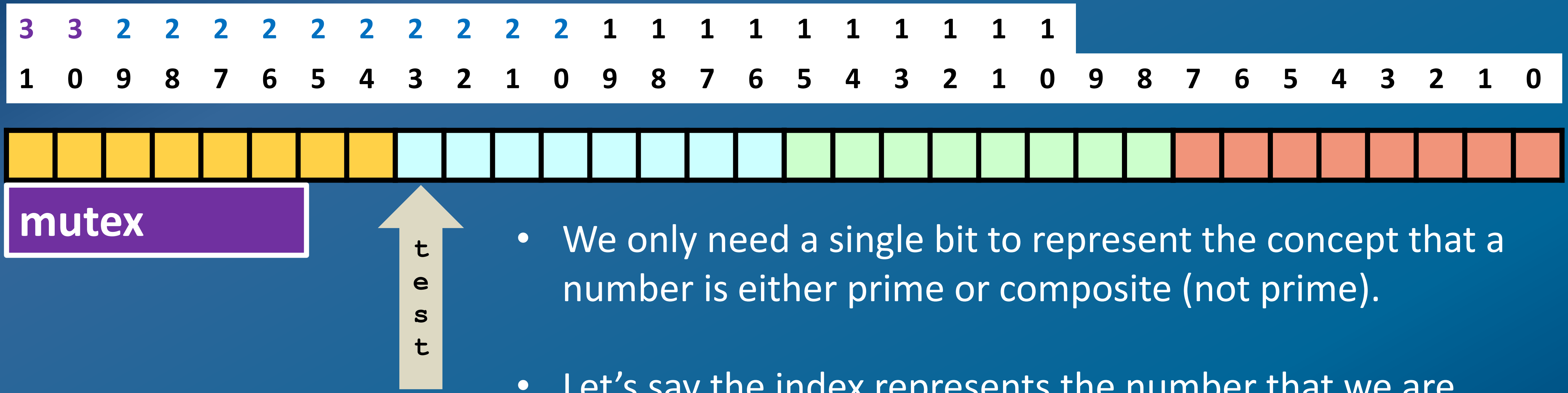
Let’s label the index of bits in the 32 bit variable.

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0



This is the normal way we label the bits in a word, when not talking about the endian-ness of the architecture.

Portland State
Computer Science

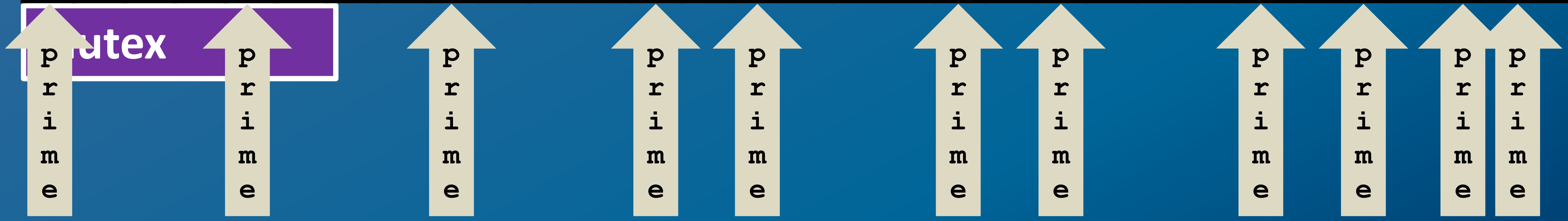


- We only need a single bit to represent the concept that a number is either prime or composite (not prime).
- Let's say the index represents the number that we are going to test for prime-ness.
- Then, index 23 represents whether the number 23 is prime or composite.

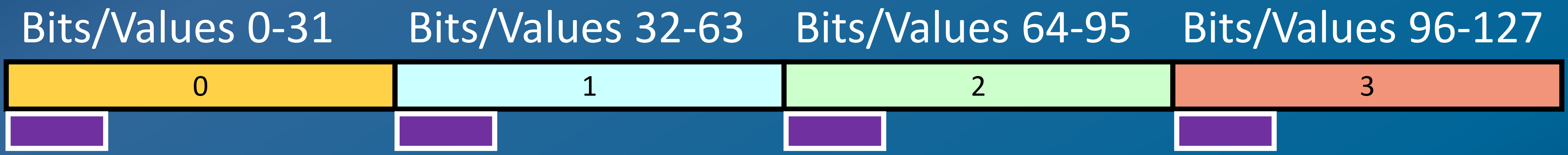
- If we fill in the prime-ness/compositeness of the values 0-31, we’d get something like below.

- The values 2, 3, 5, 7, 11, 13, 17, 19, 23, 27, and 31 are all prime. The others are composite.

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	1	0	0



- If we create an array of 4 of the data structures, we can represent the values 0 - 127 for prime-ness/composite-ness.



- If we create an array of 100 of the data structures, we can represent the values 0 - 3,199 for prime-ness/composite-ness.
- If we create an array of 10,000 of the data structures, we can represent the values 0 - 319,999 for prime-ness/composite-ness.
- If we create an array of 1 million of the data structures, we can represent the values 0 - 31,999,999 for prime-ness/composite-ness.

- How do we know how many of the data structures we must allocate to represent the prime numbers up to N?

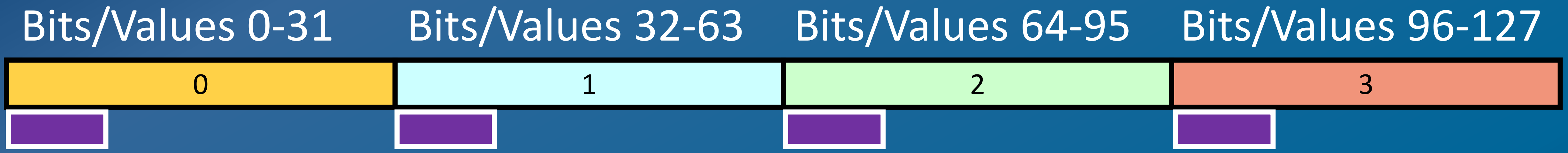
- Assuming there are 32 bits in the bits data member:

  - $\lceil N / 32 \rceil$ (ceiling of N / 32)

  - Using the integer arithmetic of C (which truncates), we can (**roughly**) make this:
    - (N / 32) + 1

If we want to generate prime numbers up to 10,000,000, we need 312,501 data structures.
(10,000,000 / 32) + 1 = 312,501
In this case we actually waste 1 data structure.

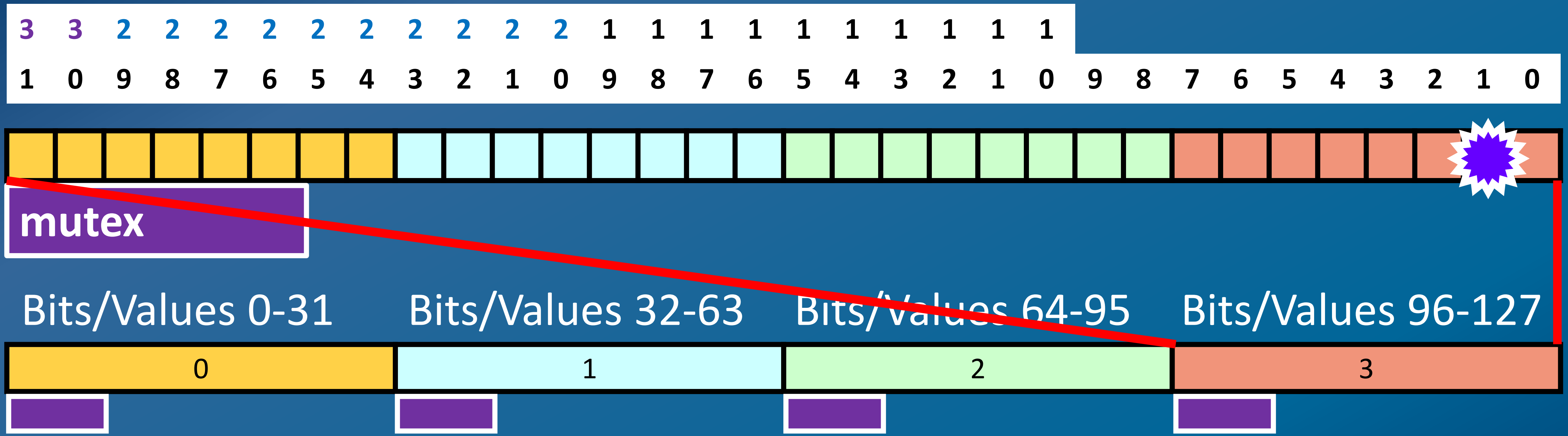
- How do we access the kth bit of the array of data structures?



- We have an array of 4 data structures (indexed 0-3), representing prime numbers up to 127.

- We want to access bit 97 (k == 97) in the array:
  - Index = 97 / 32 = **3** (using integer arithmetic)
  - Bit number 97 % 32 = **1**

  - So, it's index 3 bit 1.



The kth bit in our array of 4 data structures.

- k = 97
- Index = 97 / 32 = 3 (using integer arithmetic)
- Bit number 97 % 32 = 1

Portland State
Computer Science

Great, we now know that kth value is bit 1 in a specific `uint32_t` data type, but **HOW** do we access just that single bit???

Be shifty.

```
uint32_t mask = 0x0;
mask = 0x1 << (k % 32); // 1 << 1 in this case
```

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0

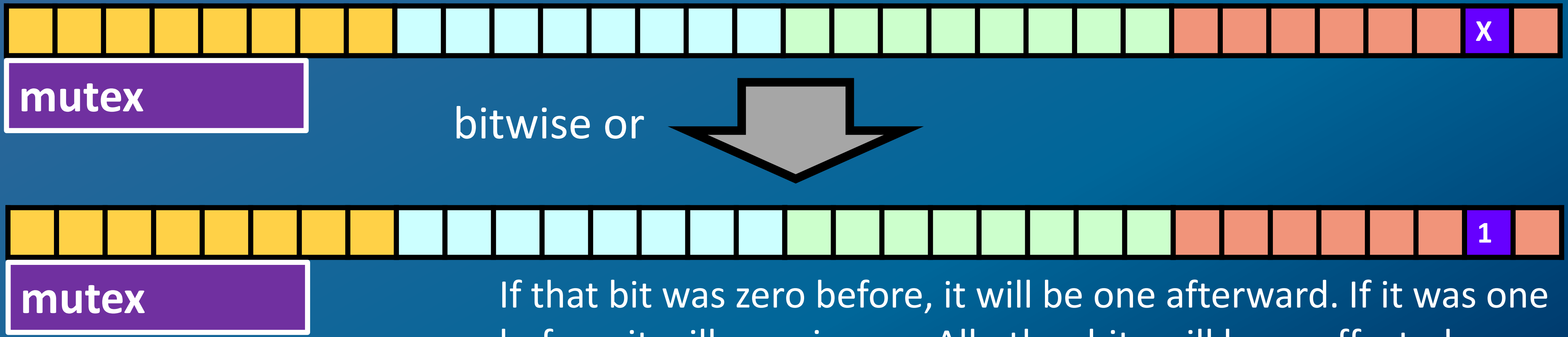
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

$\underbrace{\hspace{10em}}_{\texttt{mask}}$

Portland State
Computer Science

- If we do a bitwise ***or*** (the single `|` character) with the `mask` and our `bits` data member, we have just **set** that bit in the `bits` data member.

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0



If that bit was zero before, it will be one afterward. If it was one before, it will remain one. All other bits will be unaffected.

- If we do a bitwise ***and*** (the single `&` character) with the `mask` and our `bits` data member, we **test** the bits data member to see if that bit is set.

3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1										
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
																														x	

mutex

bitwise and

result

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0

If that bit was one before the `&`, it will still be one. If the bit was zero, it will continue to zero. In either case, all other bits will be zero. We can test the result as either zero or non-zero.

Portland State
Computer Science

- If need to **clear** a bit takes a little bit more work. Use the ~ operator (the tilde is a bit wise negate, aka 1's complement).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

bitwise negate

All the bits are flipped

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1

																														X	

mutex

bitwise and

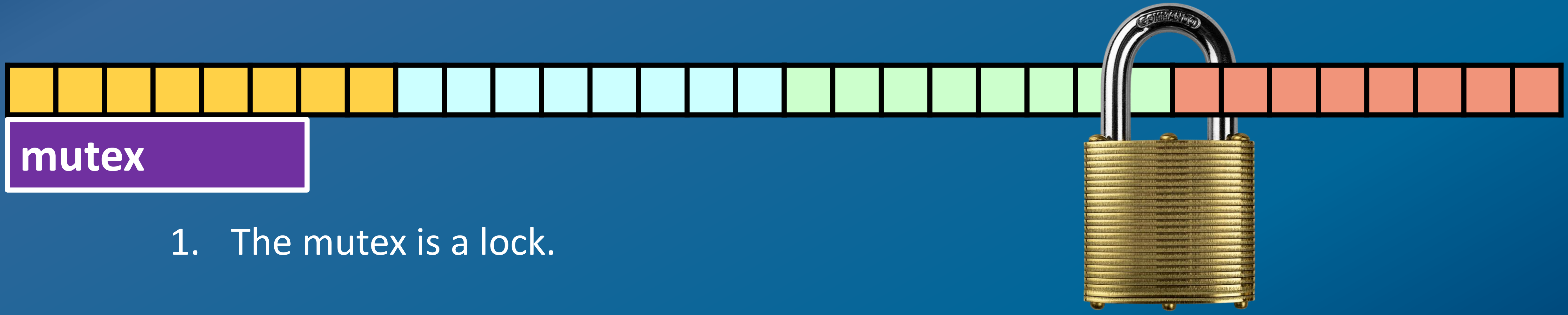
Now, do a bitwise **and**

																														0	

mutex

Okay, got it. We can set/test/clear the kth bit of our bit vector.

But still, what is the mutex … thingie?



1. The mutex is a lock.
2. It it used to control access to the bits data member.
3. You don’t want 2 (or more) threads accessing the bits data member at the same time.

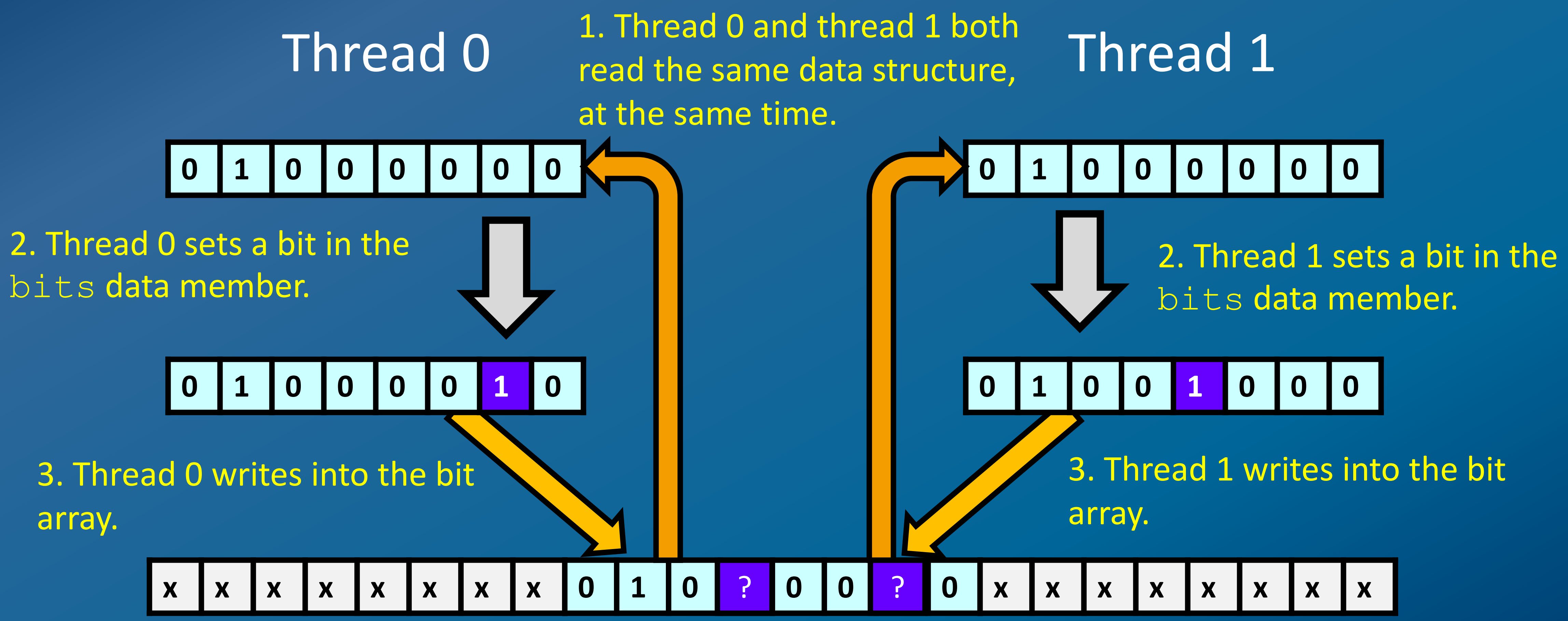
You don’t want to have multiple threads reading from or writing to a `bits` data member at the same time.

Doing so is likely to cause invalid reads or data corruption on writes.

Before you read/test: **lock**
    read
Unlock

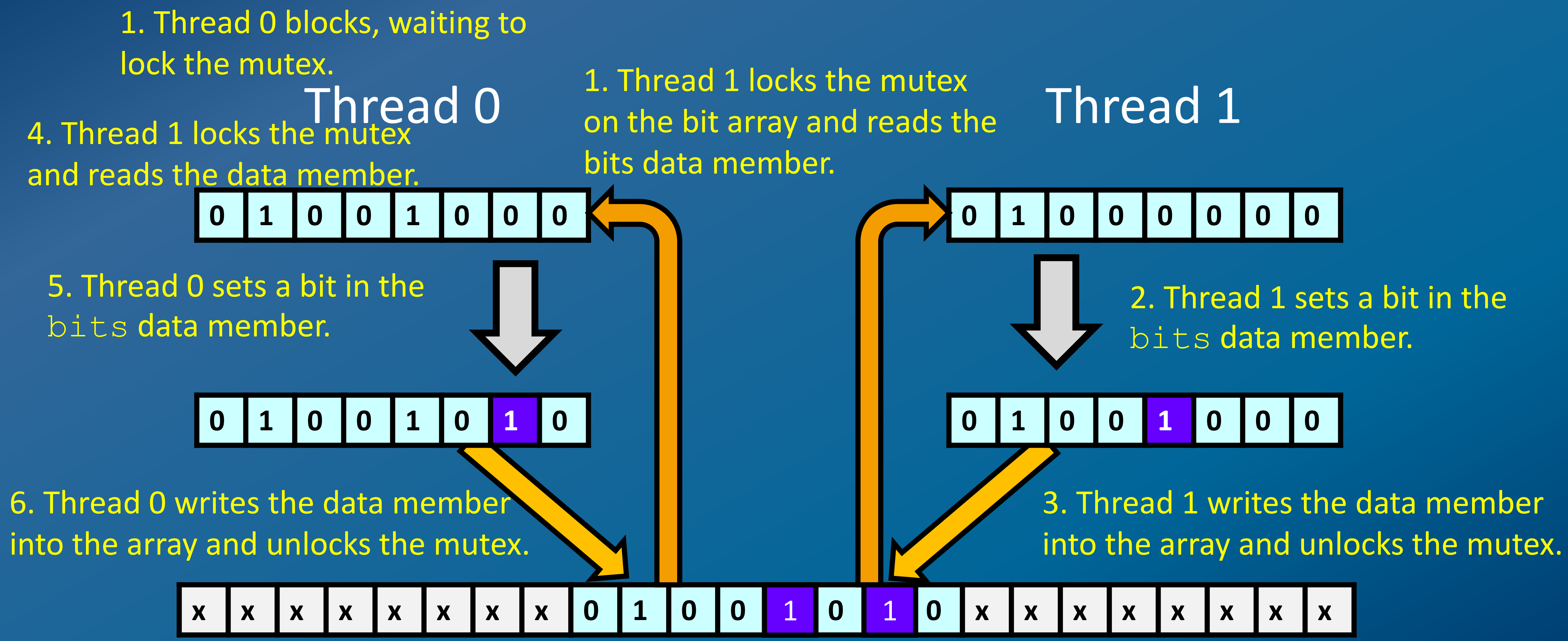
Before you modify: **lock**
    read
    make your changes
    write
Unlock

Without Locks (mutexes)



What value is stored in the array?
Can you say “Race condition”?

With Locks (mutexes)



Use of the mutex serializes access to the array and maintains correctness.

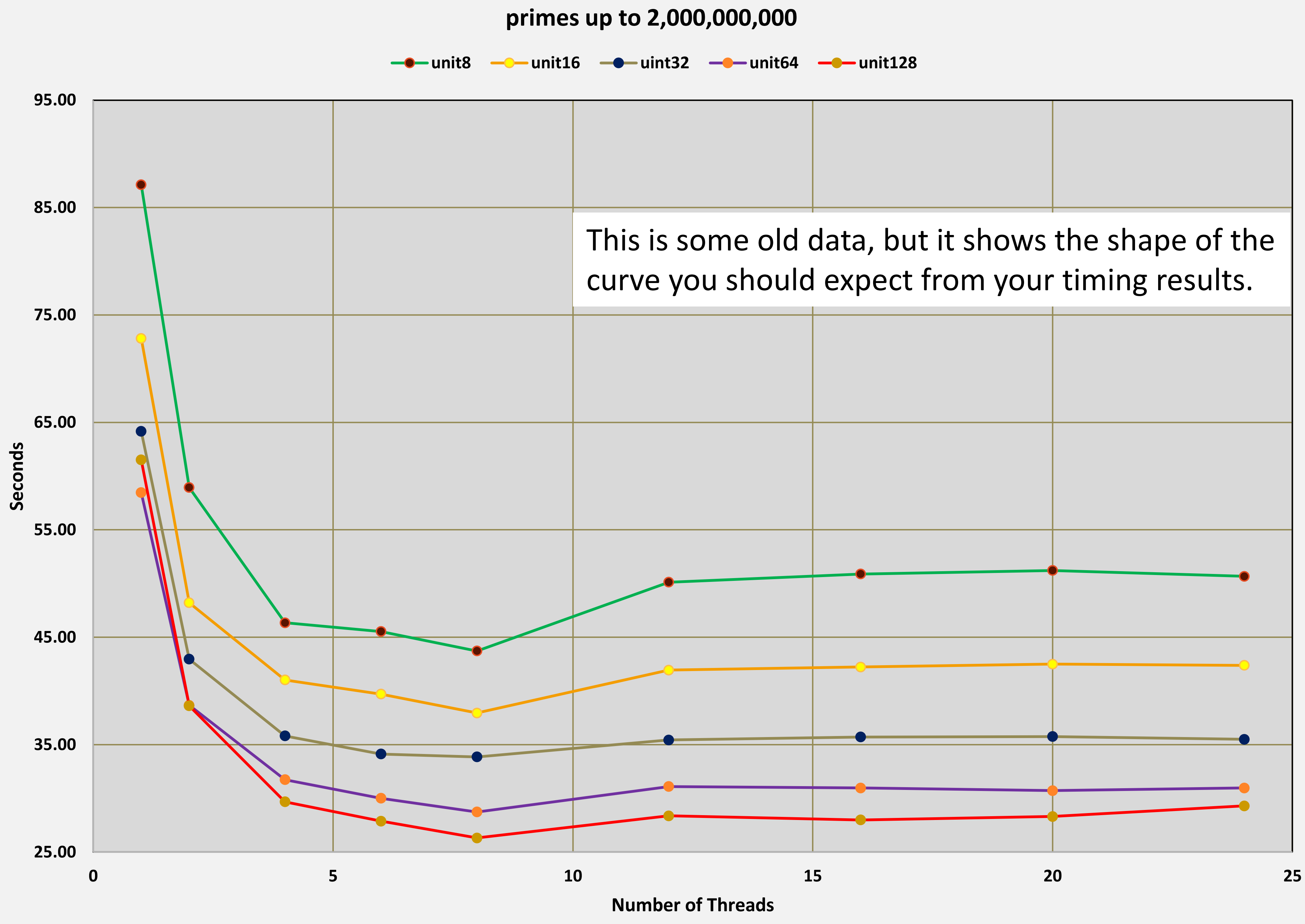
Portland State Computer Science

- Before you can use a mutex, you must initialize it.
  - You can use `PTHREAD_MUTEX_INITIALIZER` for statically allocated mutexes
  - Or, `pthread_mutex_init()` for dynamically allocated mutexes
- You will need to initialize all of the `bits` data members.
  - If you want to initialize all bits in a `bits` data member to zeroes, just assign zero to the data member.
  - If you want to initialize all `bits` in a bits data member to ones, assign `~0` to it.
- If you want to be really *coolieo* (and have spare time), use multiple threads to initialize the `bits` data member and mutexes with a barrier.

This project can be easily done using either static work allocation or dynamic work allocation.

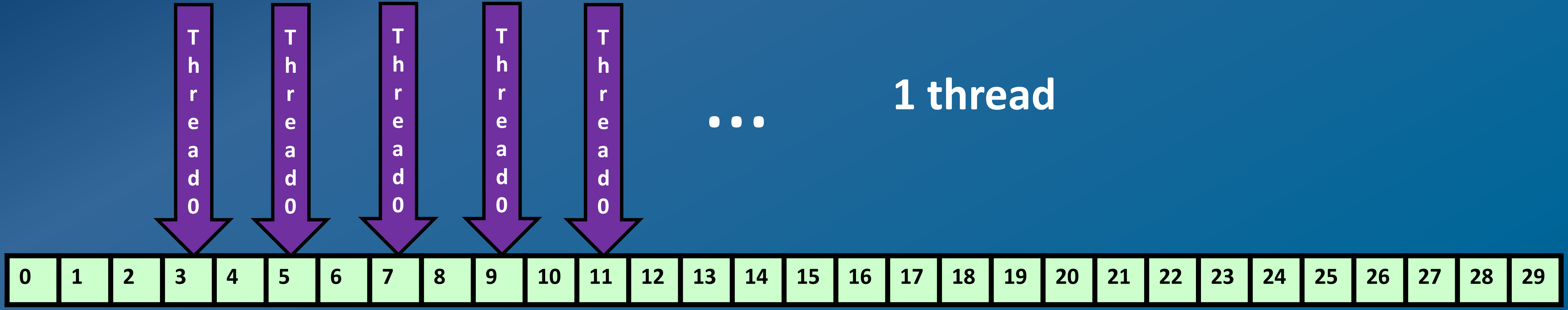
Static work allocation has the threads stride through the bit array, based on a thread identifier.

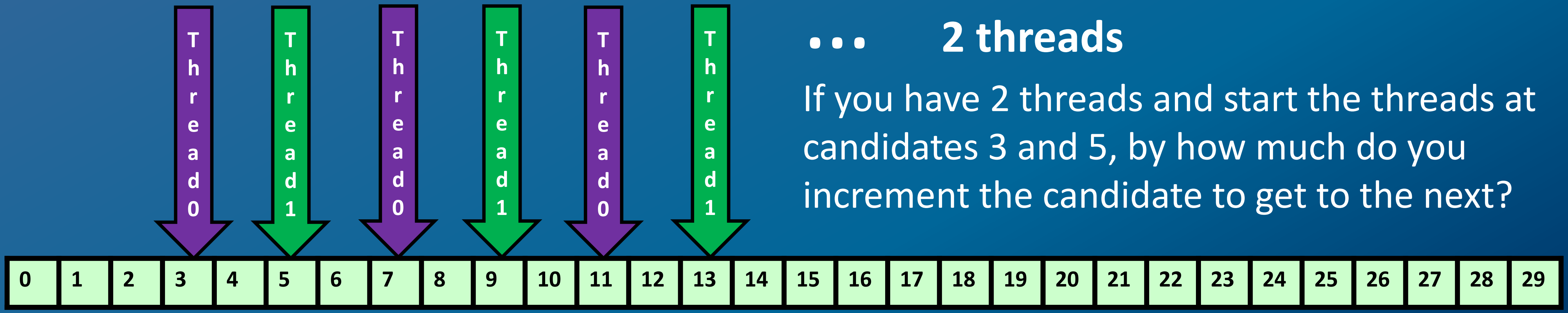
Dynamic work allocation uses a function to generate the next prime candidate for a thread to test, very much like the mm4 matrix multiplication uses a function to generate the next row to compute in the matrix multiplication InClass assignment.



This is some old data, but it shows the shape of the curve you should expect from your timing results.

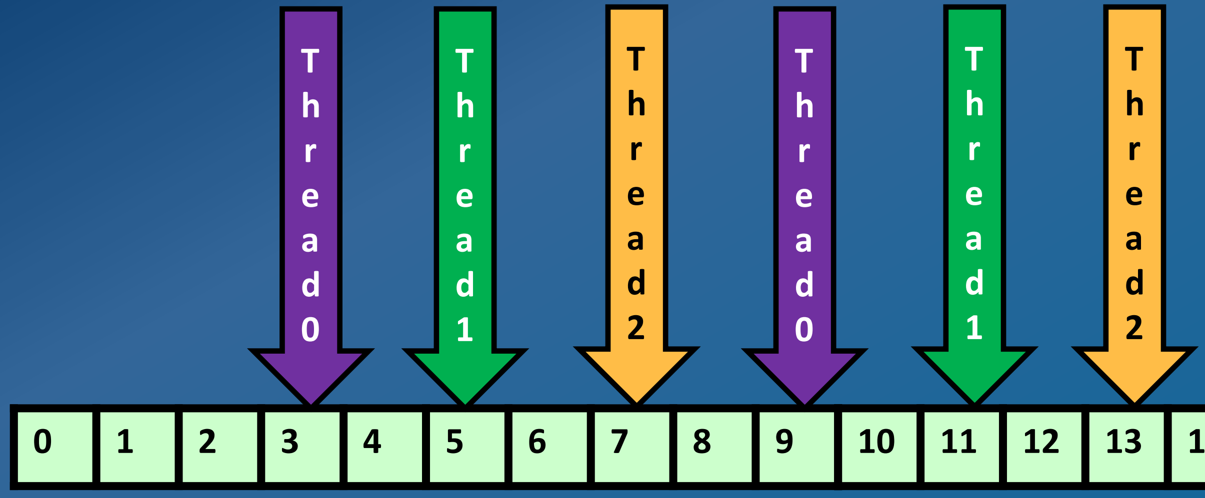
Static Work Allocation





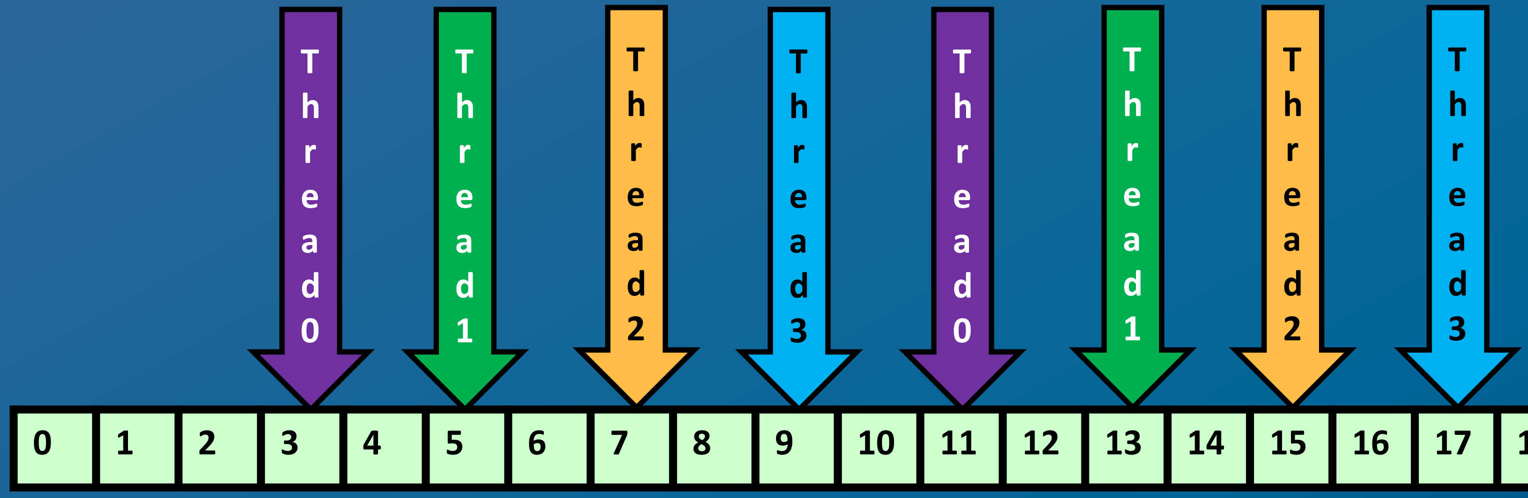
If you have 2 threads and start the threads at candidates 3 and 5, by how much do you increment the candidate to get to the next?

Static Work Allocation



- … **3 threads**

If you have 3 threads and start the threads at candidates 3, 5, and 7, by how much do you increment the candidate to get to the next?



- … **4 threads**

If you have 4 threads and start the threads at candidates 3, 5, 7, and 9, by how much do you increment the candidate to get to the next?

That Pointer for `pthread_create()`

You get to send 1 single pointer to the pthread_create() function. Make it a good one.

As helpful as the statement may be, exactly what is a “good” one?

Let’s break down the configuration of the threads a little. Let’s categorize configuration into 2 buckets:

- Things that are common to all threads
- Things that are unique to each thread.

That Pointer for `pthread_create()`

First, what are the variables that constitute, the “configuration” for threads?

1. The number of threads used
2. The upper bound
3. The starting location for each thread in the bit array of candidates.

That Pointer for `pthread_create()`

- Things that are common to all threads:
  - The number of threads
  - The upper bound on prime numbers to compute
- Things that are unique to each thread:
  - The starting location in the bit vector of candidates

Sooooo….

- Things that are in common to all threads could be (gasp) global variables.
- Things/Thing unique to each thread could be passed to the thread when it is created.