Week 11 – Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Semaphores
  - Signaling
  - Mutual exclusion
  - Multiplexing

Questions?

Project 3

Barriers

Barriers

- Sometimes a bunch of threads are working on a task that has phases
- We want to guarantee that all threads have finished Phase 1 before moving on to Phase 2
- To guarantee this, we can use **barriers**
- A barrier prevents threads from continuing unless ***k*** threads have reached it
  - It's common for ***k*** to be the total number of threads
  - Sometimes, however, the calculation is fine as long as at least ***k*** are done
- It's possible to do this kind of coordination with semaphores, but it's hard to get it exactly right

Barrier example

- Self-driving cars solve a very difficult problem
  - Adjustments have to be made based on sensor data like cameras and GPS
  - Planning has to be done based on internally stored map data
  - Marrying together the planning with ever-changing data is something that computers are not very good at
- We might need data from several different threads to be gathered before we're ready to do the next phase of planning
- A barrier might be the right tool to make sure that enough threads are ready
- The barrier might not even require all the threads, since it might be better to make decisions *now* based on sensor data from 5 out of 7 sensors than to wait

Barrier functions

```
int pthread_barrier_init (pthread_barrier_t *barrier, const
  pthread_barrierattr_t *attr, unsigned count);
```

- Create a barrier with the attributes given (often NULL) and the count of threads blocked

```
int pthread_barrier_destroy (pthread_barrier_t *barrier);
```

- Free up the resources associated with a barrier

```
int pthread_barrier_wait (pthread_barrier_t *barrier);
```

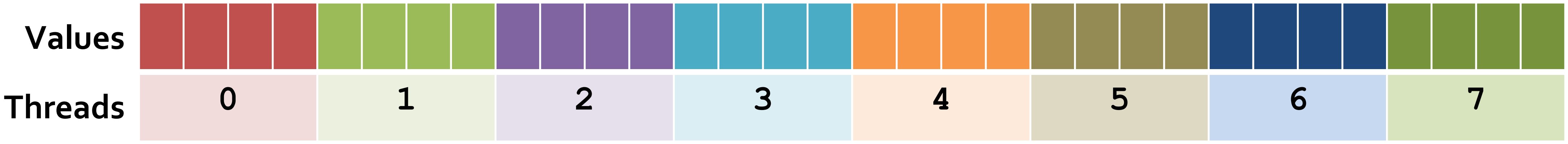
- Wait on a barrier until enough threads reach it

Merge sort

- We can imagine a threaded merge sort that works in this way:
  - Each thread is assigned a section of the array to sort
  - Each thread uses merge sort to sort that part of the array
  - All threads wait on a barrier
- Then
  - Even numbered threads merge together their section with the neighboring section
  - Threads that are multiples of four merge together double sections with other double sections
  - Threads that are multiples of eight merge together quadruple sections with other quadruple sections
  - ...

Threaded merge sort visualized

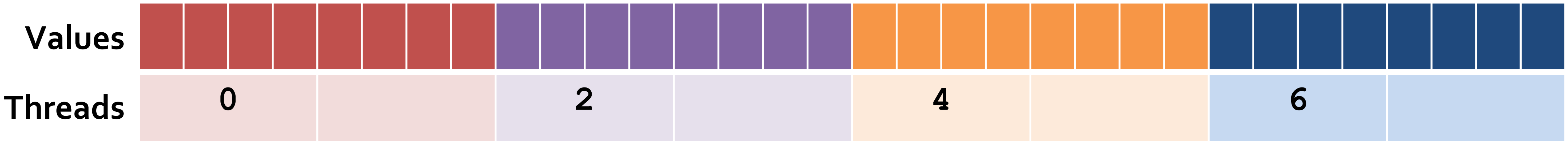
- Each thread is assigned a section of an array and sorts it



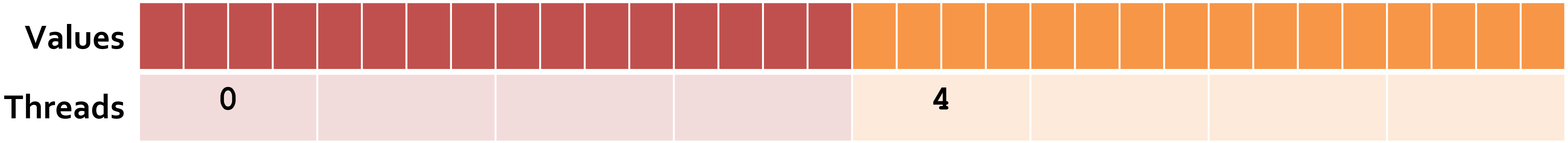
- Since there's no overlap, each thread can work independently
- After sorting, all threads wait on a barrier to be sure that every thread has finished sorting

Final merging visualized

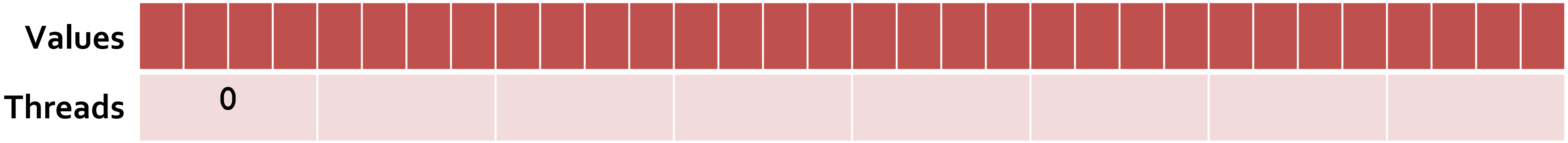
- Threads can't merge the same parts of the array without causing race conditions
- Half the threads merge with their neighbors



- Then, half of those merge



- And so on, until it's all merged



Threaded merge sort in code

- Here are some defined constants and the input structure the threads will use to do the merge sort
- Note that the number of threads evenly divides the array length and is a power of 2, to keep everything simple

```
#define THREADS 8
#define SIZE (1024*1024)

struct args {
  pthread_barrier_t *barrier;
  int id;
  int *array;
  int *scratch;
  int length;
  int threads;
};
```

Threaded merge sort in code

- Here's the **`main()`** for the threaded merge sort

```
int main() {
  pthread_t threads[THREADS];
  struct args args[THREADS];
  int* array = malloc(sizeof(int) * SIZE);         // Create array
  int* scratch = malloc(sizeof(int) * SIZE);       // Merge sort needs a scratch array
  srand(time(NULL));
  for (int i = 0;  i < SIZE; ++i)
    array[i] = rand();

  pthread_barrier_t barrier;
  pthread_barrier_init (&barrier, NULL, THREADS);  // Create barrier

  for (int i = 0; i < THREADS; ++i) {
    args[i].id = i;
    args[i].barrier = &barrier;
    args[i].array = array;
    args[i].scratch = scratch;
    args[i].length = SIZE;
    args[i].threads = THREADS;
    pthread_create (&threads[i], NULL, sorting, &args[i]);
  }

  for (int i = 0; i < THREADS; ++i)
    pthread_join (threads[i], NULL);

  pthread_barrier_destroy (&barrier);
  free (array);
  free (scratch);
  pthread_exit (NULL);
}
```

Threaded merge sort in code

- The thread itself is more complex than what we've done before

```
void * sorting(void *args) {
  struct args* input = (struct args*)args;
  int stride = input->length / input->threads;
  int start = stride * input->id;
  int end =   start + stride;
  merge_sort (start, end, input->array, input->scratch);
  pthread_barrier_wait (input->barrier); // Wait for threads to finish sorting

  int multiple = 2;
  while (multiple <= input->threads) {      // Threaded merge
    if (input->id % multiple == 0) {
      int aStart = start;
      int aEnd = aStart + (stride * multiple / 2);
      int bStart = aEnd;
      int bEnd = bStart + (stride * multiple / 2);
      merge (aStart, aEnd, bStart, bEnd, input->array, input->scratch);
    }

    pthread_barrier_wait (input->barrier);
    multiple *= 2;
  }

   pthread_exit (NULL);
}
```

Programming practice

- Although the threading part is done, we can still do the other two methods

```
void merge_sort (int start, int end, int array[],
                 int scratch[])
```

  - Recursively merge sorts the contents of **`array`** from **`start`** up to (but not including) **`end`**, using **`scratch`** as extra space

```
void merge (int aStart, int aEnd, int bStart, int bEnd,
            int array[], int scratch[])
```

  - Merges sorted values from **`aStart`** up to (but not including) **`aEnd`** with sorted values from **`bStart`** up to **`bEnd`**, using scratch as extra space
  - Note that the range from **`aStart`** up to **`bEnd`** is expected to be contiguous

Ticket Out the Door

Upcoming

Next time…

- Condition variables
- Deadlock
- Synchronization design patterns

Reminders

- Work on Project 3
- Read sections 7.6, 7.7, 8.1, and 8.2