Week 12 – Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Synchronization design patterns
- Producer-consumer

Questions?

Project 3

Producer-Consumer

Producer-consumer

- The **producer-consumer problem** comes up all the time in concurrent systems
  - One or more threads is producing elements that go into a buffer
  - One or more threads is consuming elements from the buffer
- A producer can't put an item into a full buffer and must block
- A consumer can't remove an item from an empty buffer and must block
- Example:
  - An OS thread is putting data into a buffer that's coming across the network
  - A user thread is trying to read data out of that buffer

Unsafe producer-consumer with a bounded queue

- We can move on to a version with a bounded buffer
- Our implementation uses a circular array (that wraps back around to the beginning)
- The following code is unsafe for two reasons:
  - It doesn't check to see if the buffer is full when enqueuing or empty when dequeuing
  - Changing queue data is **unsafe** for a multi-threaded application

```
void enqueue_unsafe (queue_t *queue, data_t *data)
{
  // Store the data in the array and advance the index
  queue->contents[queue->back++] = data;
  queue->back %= queue->capacity;
}

data_t * dequeue_unsafe (queue_t *queue)
{
  data_t * data = queue->contents[queue->front++];
  queue->front %= queue->capacity;
  return data;
}
```

Safe producer-consumer with a bounded queue and a single producer and consumer

- We could use locks and check a variable giving the total number of elements in the queue
- However, semaphores have this feature built in
- We initialize the **`space`** semaphore to the maximum size of the queue
- We initialize the **`items`** semaphore to **0**

```
void enqueue (queue_t *queue, data_t *data, sem_t *space, sem_t *items)
{
  sem_wait (space);
  enqueue_unsafe (queue, data);
  sem_post (items);
}

data_t * dequeue (queue_t * queue, sem_t *space, sem_t *items)
{
  sem_wait (items);
  data_t * data = dequeue_unsafe (queue);
  sem_post (space);
  return data;
}
```

Safe producer-consumer with a bounded queue and multiple producers and consumers

- Unfortunately, the two semaphores aren't enough when there are multiple producers and consumers
- In that situation, two producers could both be calling **`enqueue_unsafe()`**, potentially causing a race condition with the increment
- The solution is to one lock for producers and one lock for consumers
- We could use a single lock for both, but using two locks allows producers and consumers to modify the queue concurrently yet safely

```
void enqueue (queue_t *queue, data_t *data, sem_t *space, sem_t *items, pthread_mutex_t *producer_lock)
{
  sem_wait (space);
  pthread_mutex_lock (producer_lock);
  enqueue_unsafe (queue, data);
  pthread_mutex_unlock (producer_lock);
  sem_post (items);
}

data_t * dequeue (queue_t * queue, sem_t *space, sem_t *items, pthread_mutex_t *consumer_lock)
{
  sem_wait (items);
  pthread_mutex_lock (consumer_lock);
  data_t * data = dequeue_unsafe (queue);
  pthread_mutex_unlock (consumer_lock);
  sem_post (space);
  return data;
}
```

Readers-Writers

Readers-Writers

- What if we have a situation where we want to allow an unlimited number of reader threads to read data?
- But if a single writer needs to write, no other threads can access the data
- Changing the data can cause race conditions, but merely reading it concurrently is fine
  - And can make reading much faster!

First solution: Lightswitches

- This is exactly the scenario we solved with lightswitches:
  - Initialize a semaphore to **1**
  - Initialize a counter variable to **0**
  - Create a lock
  - Whenever a reader thread wants to read:
    - It acquires the lock
    - Increments the counter
    - If the counter is **1**, call **`sem_wait()`** on the semaphore
    - Unlock the lock
  - Whenever a reader thread is done reading:
    - It acquires the lock
    - It decrements the counter
    - If the counter is **0**, it calls **`sem_post()`** on the semaphore
    - Unlock the lock
  - Writers call **`sem_wait()`** to start writing and **`sem_post()`** when done

First solution code

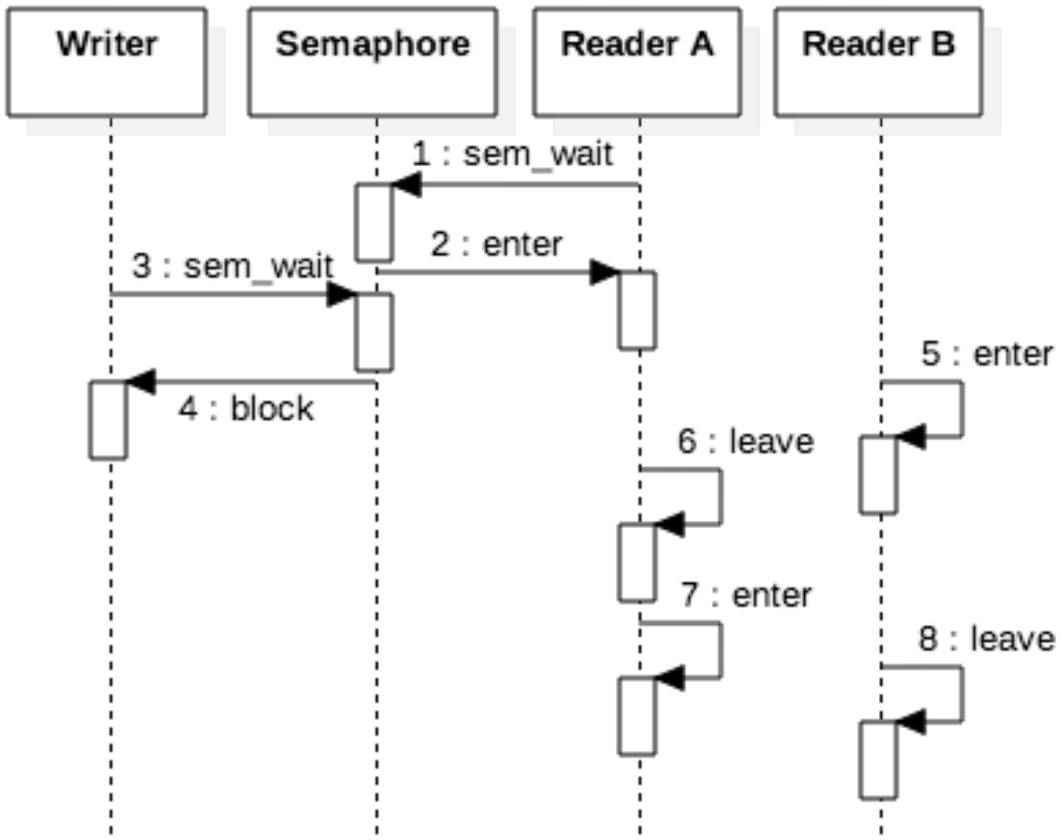
- The lightswitch has the behavior described: waiting on the semaphore for the first reader in the room and posting on it for the last reader to leave

```
void *reader (void * args)
{
  ls_t *lightswitch = (ls_t  *) args;
  enter (lightswitch);
  // Read the shared data
  leave (lightswitch);
  // Do other work and exit thread
}

void * writer (void * _args)
{
  ls_t *lightswitch = (ls_t  *) args;
  sem_wait (lightswitch->semaphore);
  // Write to the shared data
  sem_post (lightswitch->semaphore);
  // Do other work and exit thread
}
```

What's the problem with this solution?

- When a reader comes into the room, it becomes blocked for writers
- If more readers come in before others leave, writers might *never* get to enter
- What do we do?



Second solution: Add a turnstile

- We add a turnstile for the readers
  - They pass through without any problem at first
- When a writer wants to write, it waits on the reader semaphore
- This blocks any new readers from entering

Second solution code

- The lightswitch has the same behavior as before

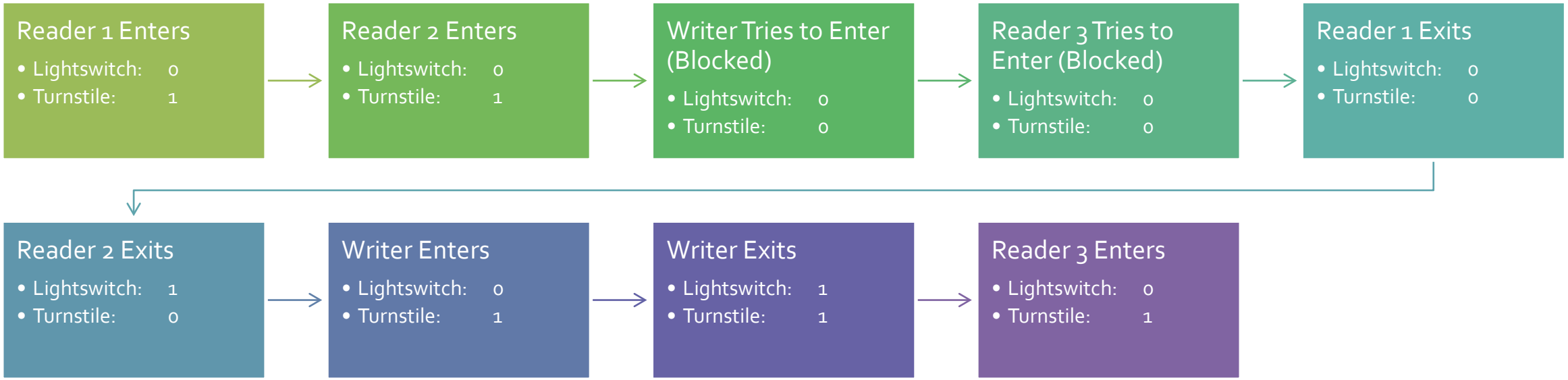
```
struct args {
  ls_t *lightswitch;
  sem_t *turnstile;
};

void * reader (void * _args)
{
  struct args *args = (struct args *) _args;
  sem_wait (args->turnstile);
  sem_post (args->turnstile);
  enter (args->lightswitch);
  // Read the shared data
  leave (args->lightswitch);
}

void * writer (void * _args)
{
  struct args *args = (struct args *) _args;
  sem_wait (args->turnstile);
  sem_wait (lightswitch->semaphore);
  sem_post (args->turnstile);
  // Write to the shared data structure
  sem_post (lightswitch->semaphore);
}
```

Illustration of second solution

- The system starts off with its two semaphores having the following values:
  - Lightswitch: 1
  - Turnstile: 1



Search-insert-delete problem

- The readers-writers problem can be extended to a problem with the following characteristics:
  - Searchers are searching for data (similar to regular readers)
  - Inserters are a kind of writer that only adds data
  - Deleters are a kind of writer that only removes data
- Rules:
  - Searchers can be concurrent with each other and an inserter
  - Inserters can be concurrent with searchers, but there can only be one inserter at a time
  - Deleters must be mutually exclusive with everyone
- You can imagine a version of this problem for concurrent accesses to databases

Search-insert-delete solution

- Searchers use a lightswitch as before
- Inserters use their own lightswitch but also have a lock to prevent concurrent insertions with each other
- Deleters must wait on both lightswitches
- This solution works because a deleter can enter only when there are no searchers or inserters

Search code

- The searcher code is essentially the same as the reader code from our first reader-writer solution

```
void *searcher (void * args)
{
  ls_t *search_switch = (ls_t  *) args;
  enter (search_switch);
  // Search through data
  leave (search_switch);
  // Do other work and exit thread
}
```

Inserter code

- The insert code is similar except that it has a lock as well

```
struct ins_args {
  ls_t *insert_switch;
  pthread_mutex_t *insert_lock;
};

void *inserter (void * args)
{
  struct ins_args *args = (struct ins_args *) args;
  enter (args->insert_switch);
  pthread_mutex_lock (args->insert_lock);
  // Do insertion
  pthread_mutex_unlock (args->insert_lock);
  leave (args->insert_switch);
  // Do other work and exit thread
}
```

Deleter code

- The delete code has to wait on *both* lightswitches

```
void * deleter ()
{
  sem_wait (search_switch->semaphore);
  sem_wait (insert_switch->semaphore);
  // Do deletion
  sem_post (insert_switch->semaphore);
  sem_post (search_switch->semaphore);
}
```

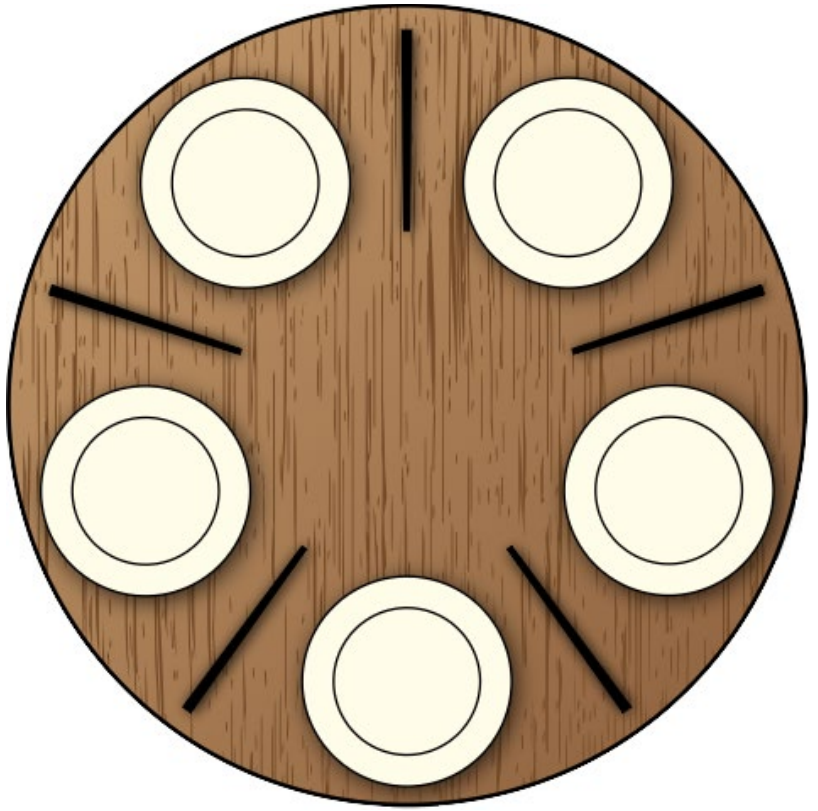
Issues with this solution

- Like our first solution for readers-writers, deleters can be starved if searchers or inserters continue to arrive
  - Never getting to run is called **starvation**
- We could increase fairness for this solution by adding turnstiles as well
  - One turnstile semaphore could be shared by all searchers and inserters
  - When a deleter comes along, it waits on the turnstile, blocking all new searchers and inserters from entering
  - When a deleter gets access to the critical section, it posts on the turnstile, allowing all waiting threads to get to their respective lightswitches

Dining Philosophers

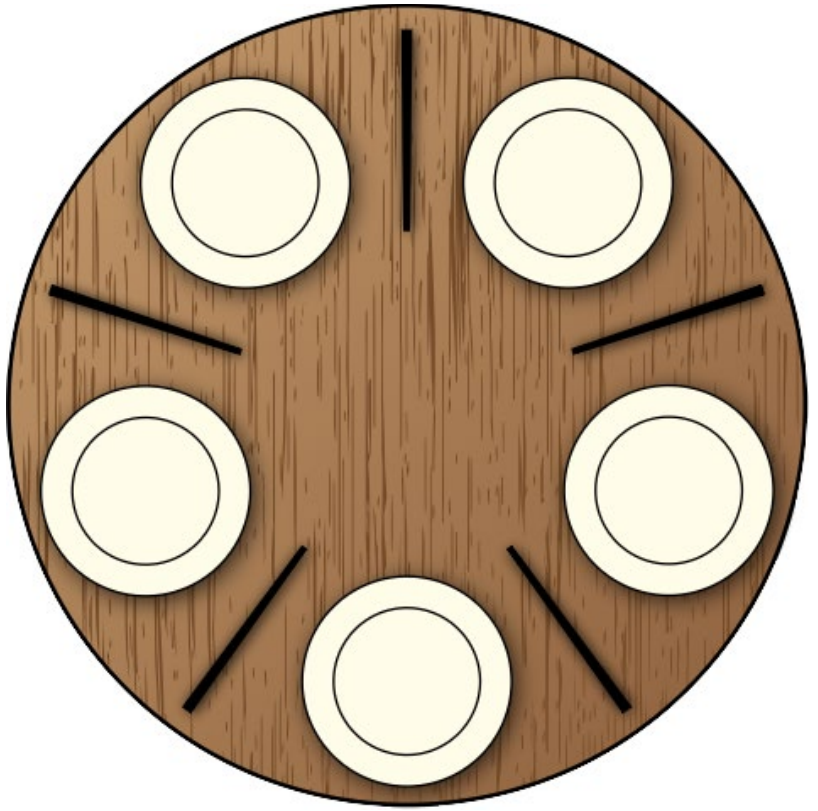
Dining philosophers

- A classic problem illustrating the difficulties of concurrency is the **dining philosophers problem**
- Some number of philosophers sit at a round table and only do two things:
  - Think
  - Eat rice
- In order to eat rice, they have to pick up two chopsticks, one on the left and one on the right
  - The book has them eat with forks, but chopsticks make more sense for the problem
  - You can eat rice with one fork, but you can't eat rice with one chopstick
  - **Critically important:** The numbers of chopsticks and philosophers are equal



The problem

- We have to enforce mutual exclusion for the chopsticks
- Two philosophers can't hold onto the same chopstick at the same time
- It's unpredictable when each philosopher is going to finish thinking and start eating
- We need a solution that works no matter what



A solution with deadlock

- Let's say there are **`SIZE`** philosophers (and **`SIZE`** chopsticks)
- We can create **`SIZE`** locks, one for each chopstick
- Then, each philosopher will acquire the lock for her left chopstick followed by the lock for her right chopstick
- In the following code, **`self`** is the index of the philosopher

```
void * philosopher (void * _args)
{
  struct args *args = (struct args *) _args;
  int self = args->self;                          // Philosopher index
  int next = (self + 1) % SIZE;
  pthread_mutex_lock (args->locks[self]);    // Pick up left chopstick
  pthread_mutex_lock (args->locks[next]);    // Pick up right chopstick
  // Eat rice
  pthread_mutex_unlock (args->locks[next]); // Put down right chopstick
  pthread_mutex_unlock (args->locks[self]); // Put down left chopstick
  // Do other work and exit thread
}
```

Why it has deadlock

- Imagine that every philosopher picks up her left chopstick at the same moment
- Now, each will wait for another one to give up what would be their right chopstick … forever
- We have the four conditions for deadlock:
  - **Mutual exclusion:** Only one philosopher can hold the lock for a chopstick
  - **Hold-and-wait:** Each philosopher acquires chopstick and tries to get another
  - **No preemption:** No philosopher can force another to give up her chopstick
  - **Circular wait:** Under the right circumstances, every philosopher can be waiting for every other in a circle

Code that's equivalent to dining philosophers

- The dining philosophers problem is intentionally absurd
- But it's not too hard to write code that's almost identical, like this code that tries to find an open port for incoming traffic and an open port for outgoing traffic

```
while (true)
  {
    in++;
    in %= NUMBER_OF_PORTS;
    if (!pthread_mutex_trylock (nic_locks[in]))
      break;
  }
// Locked network card in port for incoming data

while (true)
  {
    out++;
    out %= NUMBER_OF_PORTS;
    if (!pthread_mutex_trylock (nic_locks[out]))
      break;
  }
// Locked network card out port for outgoing data
```

Solution by limiting access

- One solution is to add a semaphore initialized to **`SIZE - 1`**
- Then, only **`SIZE - 1`** philosophers could try to grab a chopstick

```
void * philosopher (void * _args)
{
  struct args *args = (struct args *) _args;
  int self = args->self;                          // Philosopher index
  int next = (self + 1) % SIZE;
  sem_wait (args->can_eat);                       // Multiplexing semaphore
  pthread_mutex_lock (args->locks[self]);         // Pick up left chopstick
  pthread_mutex_lock (args->locks[next]);         // Pick up right chopstick
  sem_post (args->can_eat);                       // Multiplexing semaphore
  // Eat rice
  pthread_mutex_unlock (args->locks[next]); // Put down right chopstick
  pthread_mutex_unlock (args->locks[self]); // Put down left chopstick
  // Do other work and exit thread
}
```

Solution by breaking hold and wait

- In our example, the philosopher gets the first chopstick and immediately tries to get the second
- In real situations, some work might need to get done between acquiring resources
- To avoid delays, it might be desirable to instead get a chopstick and then *try* to get the second, releasing the first if that fails

```
while (! success)
  {
    pthread_mutex_lock (args->locks[self]);      // Pick up left chopstick
    // Perform some work
    // Then, try to get the right chopstick
    if (pthread_mutex_trylock (args->locks[next]) != 0)
      {
        // Undo current progress
        pthread_mutex_unlock (args->locks[self]); // Put down left chopstick
      }
    else
      success = true;
  }
```

Solution by imposing order

- We can break the circular wait condition with a clever ordering
- If every philosopher picks up her left chopstick at the same time, we're stuck
- But what if exactly one picked up her right chopstick first?
  - Deadlock would become impossible!

```
void * philosopher (void * _args)
{
  struct args *args = (struct args *) _args;
  int self = args->self;                          // Philosopher index
  int next = (self + 1) % SIZE;
  if (self > next) swap (&self, &next);           // Last philosopher swaps order
  pthread_mutex_lock (args->locks[self]);         // Pick up left chopstick
  pthread_mutex_lock (args->locks[next]);         // Pick up right chopstick
  // Eat rice
  pthread_mutex_unlock (args->locks[next]); // Put down right chopstick
  pthread_mutex_unlock (args->locks[self]); // Put down left chopstick
  // Do other work and exit thread
}
```

Solution with atomic chopsticks

- I just like saying "atomic chopsticks"
- But it's also possible to use condition variables to acquire the chopsticks as a set
  - Acquire a chopstick-getting lock
  - As long as your chopsticks aren't available
    - Wait on a condition variable
  - When they're available, mark them in-use
  - Release the lock
  - Eat
  - Acquire the lock
  - Mark your chopsticks available
  - Broadcast a wake-up to everyone waiting on the condition variable

Takeaways

- Getting concurrent code to work is challenging:
  - Just working at all
  - Avoiding deadlock
  - Providing fairness for different threads
  - Tuning performance
- If you have a problem that involves coordinating multiple threads, try to see if there's a similar problem in the literature
- **It's hard but not impossible!**
- Some computer scientists see concurrency as a challenge worth pursuing

Patterson's 13 dwarfs

- David Patterson is a Turing Award winner and former president of the ACM
- He's a deep hardware guy who was watching the increase in processor core count in the early 2000s
- In 2007, he proposed the following "13 dwarfs"
  1. Dense linear algebra
  2. Sparse linear algebra
  3. Spectral methods
  4. N-body methods
  5. Structured grids
  6. Unstructured grids
  7. Map-reduce
  8. Combinatorial logic
  9. Graph traversal
  10. Dynamic programming
  11. Back-track/branch and bound
  12. Graphical model inference
  13. Finite state machine

- A dwarf is a kind of problem we'd really like to have good hardware and software approach for running in parallel
- After more than a decade, we still don't have great approaches for most of these
- What was unexpected at the time is that we'd have such good neural networks and ways of training them in parallel

Ticket Out the Door

Upcoming

Next time…

- Parallel and distributed systems
- Parallelism vs. concurrency
- Parallel design patterns

Reminders

- Finish Project 3
  - **Due Friday by midnight!**
- Read sections 9.1, 9.2, and 9.3