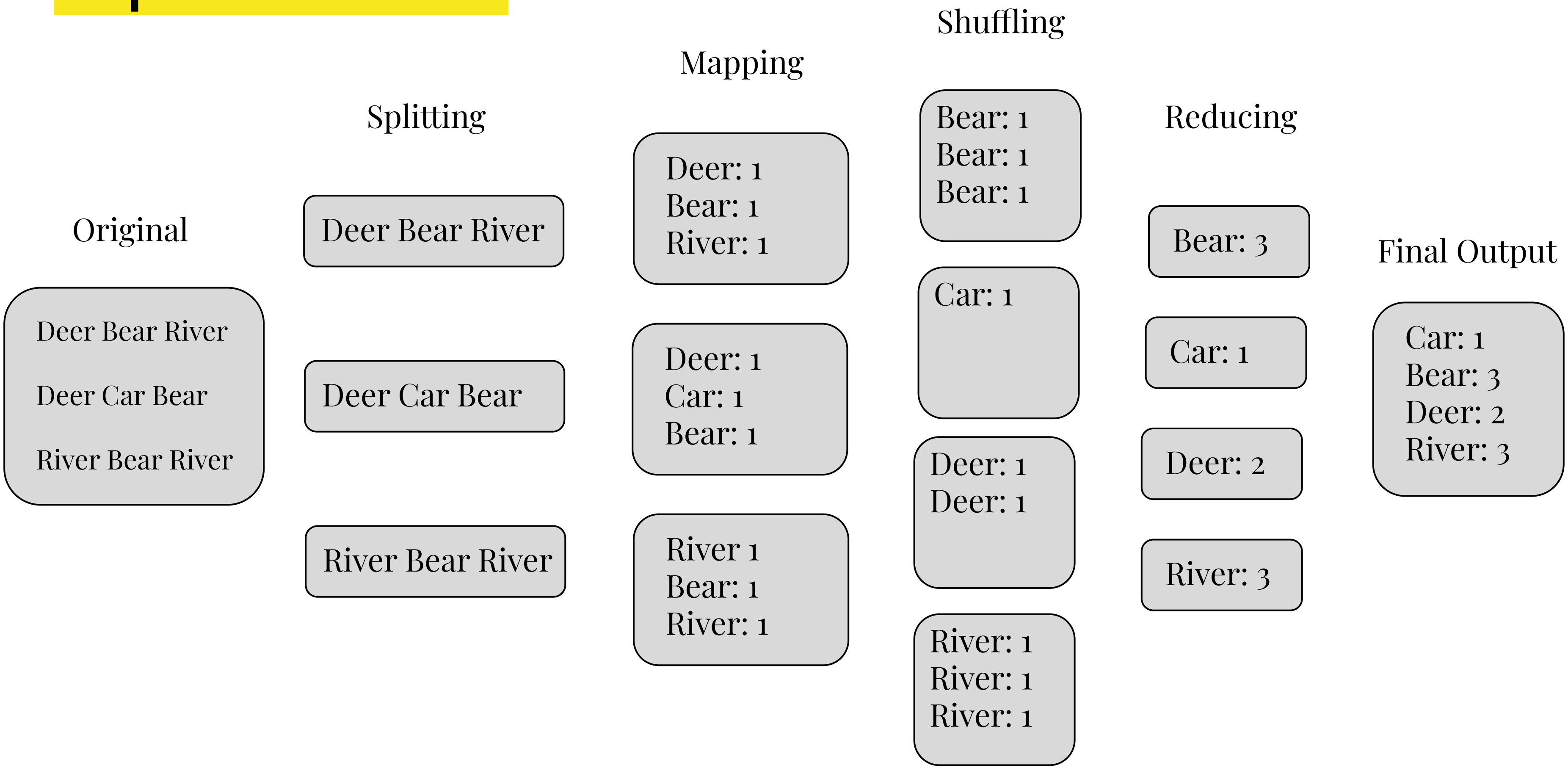
CSC360
Operating Systems

MapReduce / Dining Philosophers

Today!

- A2 part2: how to approach it
  - Ideas from MapReduce (many examples out there, will look at one)
- A2 part3: dining philosophers
  - Race conditions
  - Deadlock
  - Starvation
- Blockchain!
  - What does it have to do with concurrency?

MapReduce A2/P2



What does this code do?

```c
void** map(void** things, void* (*f)(void*), int length){

    void** results = malloc(sizeof(void*)*length);

    for(int i = 0; i < length; i++){
        void* thing = things[i];
        void* result = (*f)(thing);
        results[i] = result;
    }

    return results;
}
```

http://www.datastuff.tech/programming/how-to-code-mapreduce-in-c-from-scratch-using-threads-pt-1-map/

What about this code?

```c
// returns 1 if the number is a prime, 0 otherwise
void* naivePrime(void* number){
    int n = *((int*) number);
    int* res = malloc(sizeof(int));
    *res = 1;
    for(int i = 2; i < n; i++){
        if(n%i==0){
            *res=0;
            return res;
            }
    }
    return res;
}

int main(int argc, char** argv){
  int N = 1000;

  int** numbers = malloc(sizeof(int*)*N);
  for(int i = 0 ; i < N ; i++){
      int* n = malloc(sizeof(int));
      *n = i;
      numbers[i] = n;
  }

  int** resulting_numbers = NULL;
  void** is_prime = map((void**) numbers, naivePrime, N);
  resulting_numbers = (int**) is_prime;
}
```

How can we use threads to speed this up?

- Partition the data to be processed into separate threads
  - How many threads?
- Experiment! For example, the naïve prime single versus multi-threaded

Time (s):	150000 elements	300000 elements
single-threaded	5.02	18.73
2-threads	3.76	13.78
4-threads	2.73	10.14
8-threads	2.43	8.70

“Embarrassingly Parallelizable!”

```c
struct map_argument {
    void** things;
    void** results;
    void* (*f)(void*);
    int from;
    int to;
    };
```

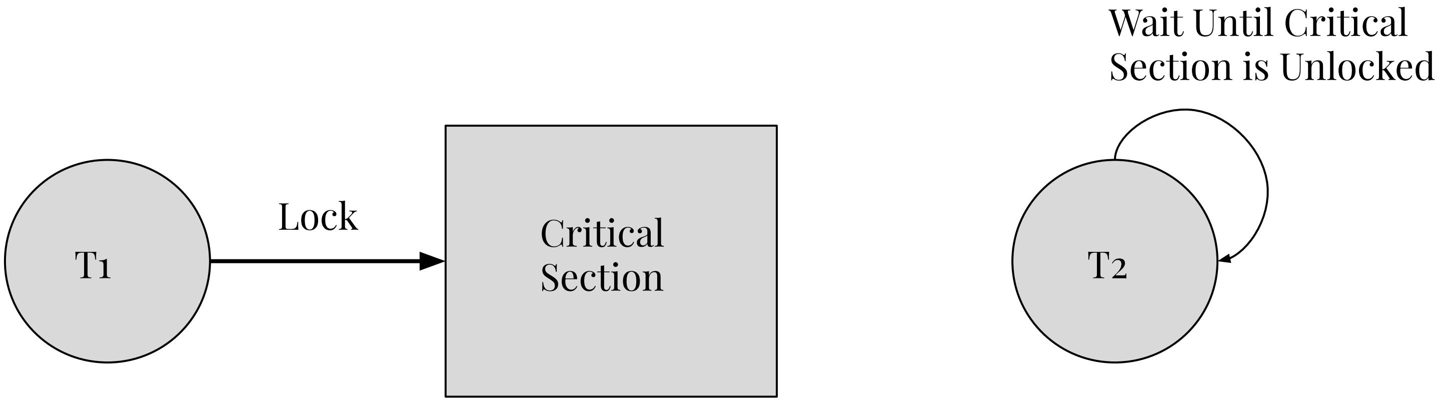
```c
void* chunk_map(void* argument){
    struct map_argument* arg = (struct map_argument*) argument;
    for(int i = arg->from; i < arg->to; i++){
        arg->results[i] = (*(arg->f))(arg->things[i]);
    }
    return NULL;
}
```

```c
concurrent_map((void**) numbers, twice, N, NTHREADS)
```

```c
void** concurrent_map(void** things, void* (*f)(void*), int length,
                      int nthreads){

    void** results = malloc(sizeof(void*)*length);
    struct map_argument arguments[nthreads];
    pthread_t threads[nthreads];
    int chunk_size = length/nthreads;

    for(int j = 0 ; j < nthreads; j++){
        int from = j*chunk_size;
        struct map_argument* argument = &arguments[j];
        init_map_argument(argument, things, results, f, from, from+chunk_size);
        pthread_create(&threads[j], NULL, chunk_map, (void*) argument);
    }


    for(int i = 0; i < length % nthreads; i++){
        int idx = chunk_size*nthreads + i;
        results[idx] = (*f)(things[idx]);
    }

    for(int k = 0; k < nthreads; k++){
        pthread_join(threads[k], NULL);
    }

    return results;
}
```

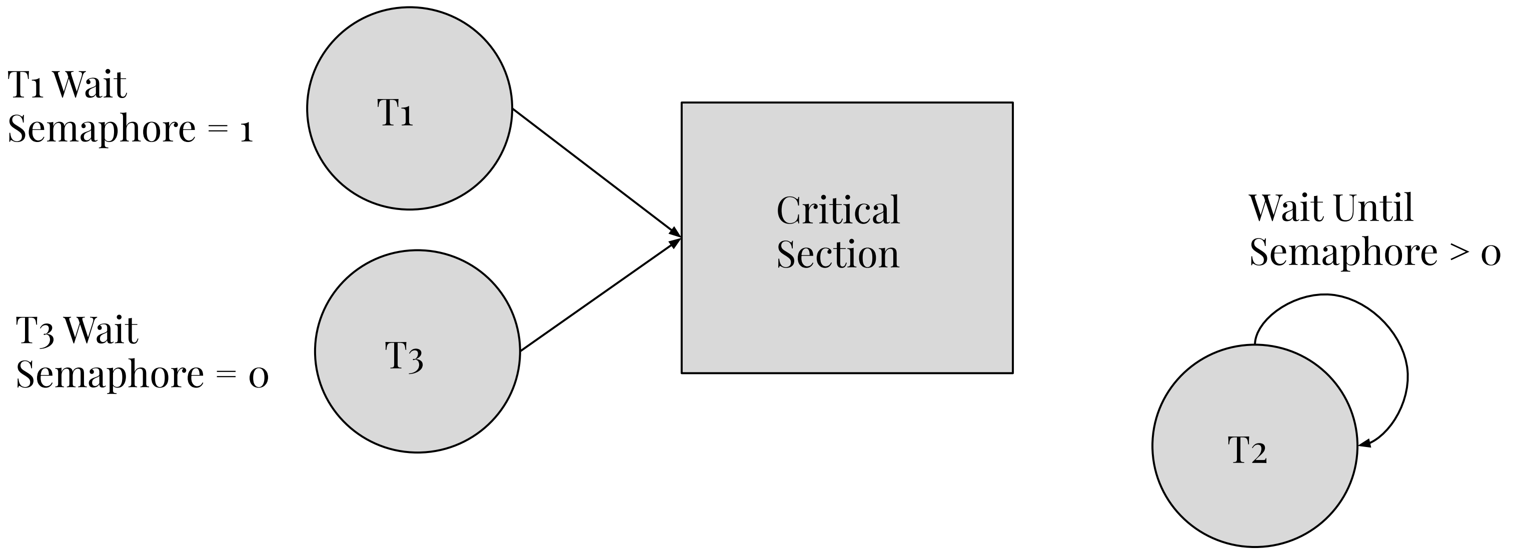
Mutex

- Either lock or unlock



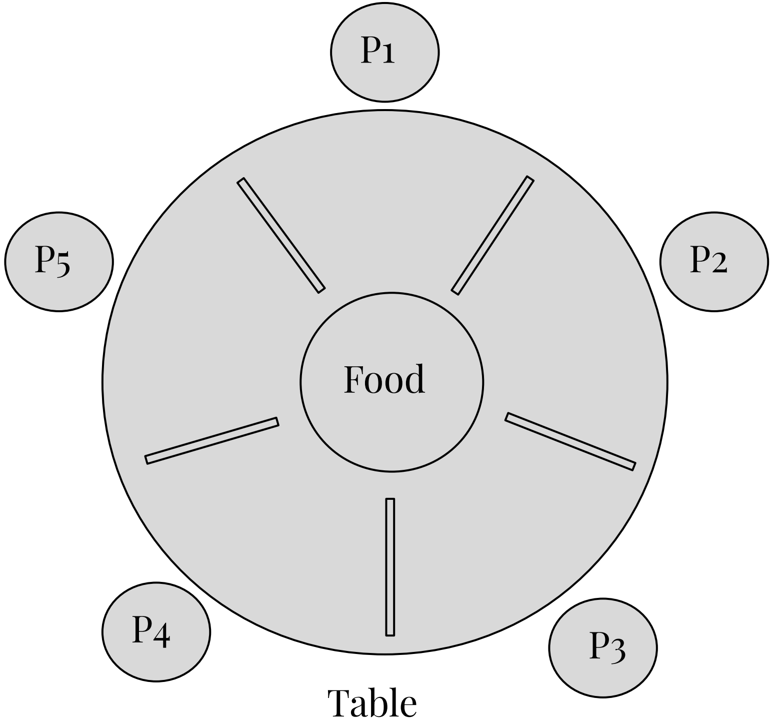
Semaphores

- Wait (-1) or Signal (+1)
- ex) Semaphore = 2
  - This means 2 Threads can be in Critical Section
  - T1 enters after Wait, Semaphore now equals 1
  - T3 enters, Semaphore = 0, no room left
  - T2 waits until Semaphore > 0



Dining Philosophers A2/P3

5 Philosophers
5 Chopsticks



- A Philosophers needs 2 Chopsticks to eat.
- We need to implement this solution using Semaphores or Mutexes.
- A Philosopher must Lock a Left and a Right Chopstick down until done eating.
- Start: Put a lock on each chopstick

Dining Philosophers - Solution 1

1. Think until the left fork is available; when it is, pick it up
2. Think until the right fork is available; when it is, pick it up
3. With both forks, eat for a fixed amount of time
4. Put the right fork down
5. Put the left fork down
6. Repeat

Issues

Deadlock

- If each Philosopher takes the left chopstick, no one can take a right chopstick

Starvation

- 2 Philosophers are fast thinkers and fast eaters
- The other 3 Philosophers have no chance to eat

Solution 1 (Bad Solution) for a philosopher

```
Think();
Pick up left chopstick;
Pick up right chopstick;
Eat();
Put down right chopstick;
Put down left chopstick;
```

- Problem: POSSIBLE Deadlock
- How?
  - Each philosopher can pick up left fork before anyone picks up their right fork
    - Now everyone is waiting for right fork!

using pthreads...

```
#include <stdio.h>
#include <pthread.h>
#include "dphil.h"


typedef struct sticks {
  pthread_mutex_t *lock[MAXTHREADS];
  int phil_count;
} Sticks;


void pickup(Phil_struct *ps)
{
  Sticks *pp;
  int i;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  pthread_mutex_lock(pp->lock[ps->id]);       /* lock up left stick */
  pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]); /* lock up right stick */
}
```

```
void putdown(Phil_struct *ps)
{
  Sticks *pp;
  int i;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]); /* unlock right stick */
  pthread_mutex_unlock(pp->lock[ps->id]);  /* unlock left stick */
}

void *initialize_state(int phil_count)
{
  Sticks *pp;
  int i;

  pp = (Sticks *) malloc(sizeof(Sticks));

  pp->phil_count = phil_count;
  for (i = 0; i < phil_count; i++) {
    pp->lock[i] = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
  }
  for (i = 0; i < phil_count; i++) {
    pthread_mutex_init(pp->lock[i], NULL);
  }

  return (void *) pp;
}
```

https://sites.cs.ucsb.edu/~rich/class/cs170/notes/DiningPhil/index.html

Solution 2: Global lock... what are the tradeoffs?

```
Think();
table.lock();
while(!both chopstick available)
  chopstickPutDown.wait();
Pick up left chopstick;
Pick up right chopstick;
table.unlock();
Eat();
Put down right chopstick;
Put down left chopstick;
chopstickPutDown.signal();
```

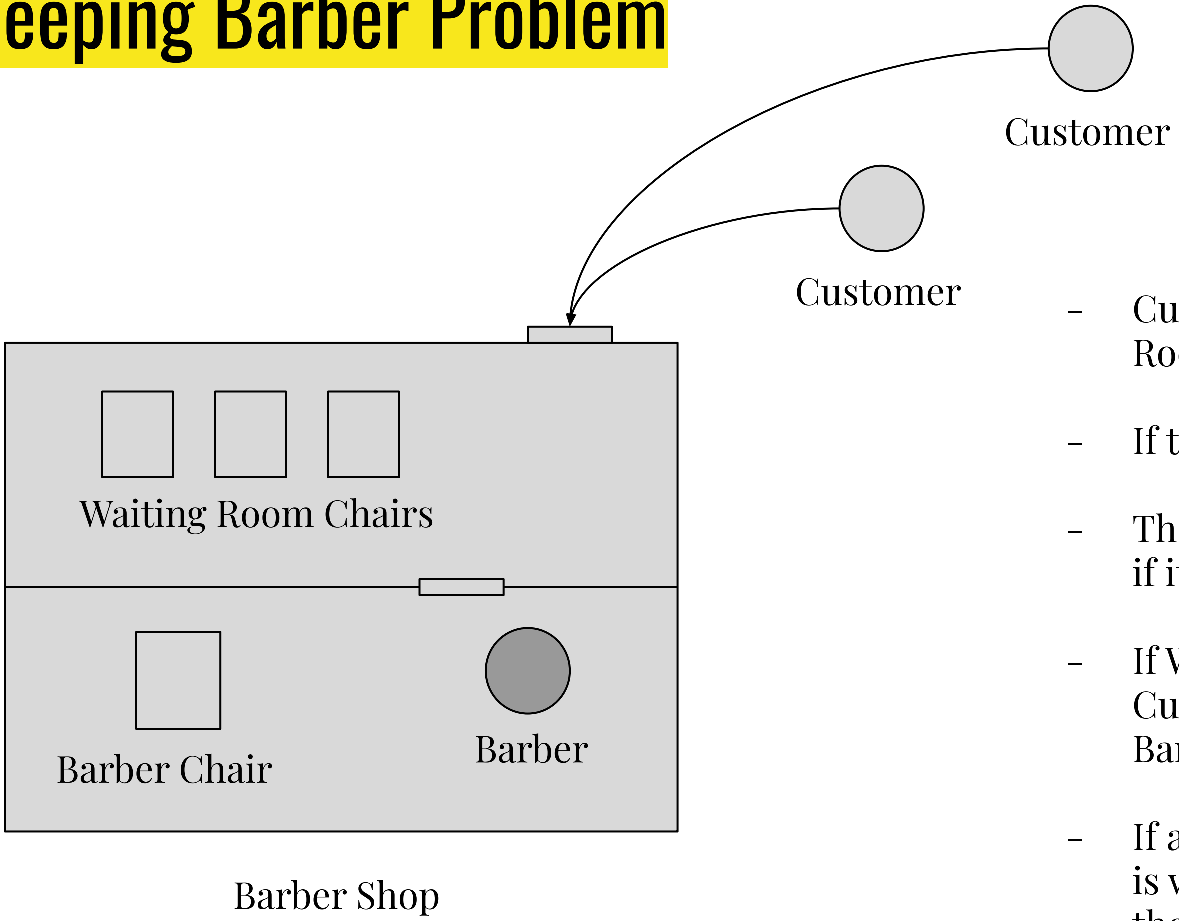
Solution 3: Reactive

```
Think();
Pick up left chopstick;
if(right chopstick available) {
    Pick up right chopstick;
} else {
    Put down left chopstick;
    continue; //Go back to Thinking
}
Eat();
```

Solution 4: Global ordering

```
Think();
Pick up “smaller” chopstick from left and right;
Pick up “bigger” chopstick from left and right;
Eat();
Put down “bigger” chopstick from left and right;
Put down “smaller” chopstick from left and right;
```

Sleeping Barber Problem



- Customers must wait in Waiting Room.
- If the waiting room is full - leave.
- The Barber checks Waiting Room - if it's empty goes to sleep.
- If Waiting Room not empty Customer sits in Barber Chair, Barber cuts hair.
- If a Customer comes in and no one is waiting, the Customer wakes up the Barber.