CSE 3100 Exam 3 Review

POSIX Threads

- Thread creation
- Thread joining
- Synchronization Methods
  - Mutex
  - Condition variables
  - Barriers

Creating/Joining With Threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- Threads always receive a void* and return a void*
  - Why is this?
- We usually package thread arguments in a struct

```
int pthread_join(pthread_t thread, void **retval);
```

- We must know which thread we want to join with
- Notice the slight difference in types

Creating/Joining With Threads Example

```c
typedef struct thread_arg_tag {
    int num;
    double data[16];
    // we can put whatever we want here
} thread_arg_t;

void* thread_func(void* arg) {
    // Do things!!
    return NULL; // or some value in a void*
}

int main() {

    pthread_t my_thread;
    thread_arg_t my_arg;
    // set values in the my_arg struct
    pthread_create(&my_thread, NULL, thread_func, &my_arg);
    // ...
    pthread_join(my_thread, NULL); // if we don't care about the return value, use NULL
}
```

Mutexes

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
const pthread_mutexattr_t *restrict attr);
```

- We usually initialize mutexes with default attributes (NULL)
- You can only destroy an *unlocked* mutex!
- Use mutexes to protect shared data/resources
- We need to avoid **data races**

Mutex Rules of Thumb

We should use a mutex when:

- We need to “protect” a certain value
- Multiple threads are writing and/or reading this same value
- When only one thread should be executing a section of code at a certain time
  - “Critical Section”

We should NOT use a mutex when:

- Multiple threads are only reading a value (no data races)
- Threads are guaranteed not to interfere with one another

Mutex Design Example

Let's say we want to count the number of prime numbers between 2 and 1000. We split the work between two threads.

Thread 1 takes numbers 2-500, and thread 2 takes 501-1000

Let's come up with some pseudocode for these threads

Mutex Design Example (Counting Primes)

Design 1:

```
for i between [min, max]:
    if i is prime:
        lock mutex
        shared_prime_count++
        unlock mutex
```

Design 2:

```
for i between [min, max]:
    if i is prime:
        local_count++

lock mutex
shared_prime_count += local_count
unlock mutex
```

Which design is better?

Condition Variables

```c
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
const pthread_condattr_t *restrict attr);
```

- Again, initialize with default attributes (NULL)
- Each cond is associated with only one mutex

```c
int pthread_cond_wait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex);
```

```c
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Condition Variables

```c
typedef struct thread_arg_tag {
    int num;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_arg_t;

void* thread_func(void* arg) {
    thread_arg_t* thread_arg = arg;
    pthread_mutex_lock(&thread_arg->mutex);
    while(thread_arg->num < 100) {
        pthread_cond_wait(&thread_arg->cond, &thread_arg->mutex);
    }
    // Do things
    pthread_mutex_unlock(&thread_arg->mutex);

    // ...

    return NULL;
}
```

- Always in a while loop
- Should be predicated on some condition
    - Perhaps a status variable
- Unlocks the mutex while waiting
- Has the mutex when we wake up

Barriers

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

- Again, initialize with default attributes (NULL)
- Initialize with a *count*, the number of threads to hit a barrier before they are unblocked

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

- Note that barrier_wait calls do not take a mutex
- Typically barriers are used independent of mutexes

Common Mistakes

```c
enum {S_INIT, S_READY};

typedef struct thread_arg_tag {
    int num;
    int status;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_arg_t;

void* some_func(void* arg) {
    thread_arg_t* thread_arg = arg;

    if(thread_arg->status == S_READY) {
        pthread_mutex_lock(&thread_arg->mutex);
        thread_arg->num += 100;
        pthread_mutex_unlock(&thread_arg->mutex);
    }

    // ...

    return NULL;
}
```

Common Mistakes

```c
enum {S_INIT, S_READY};

typedef struct thread_arg_tag {
    int num;
    int status;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_arg_t;

void* some_func(void* arg) {
    thread_arg_t* thread_arg = arg;

    if(thread_arg->status == S_READY) {
        pthread_mutex_lock(&thread_arg->mutex);
        thread_arg->num += 100;
        pthread_mutex_unlock(&thread_arg->mutex);
    }

    // ...

    return NULL;
}
```

Correct Approach

```c
enum {S_INIT, S_READY};

typedef struct thread_arg_tag {
    int num;
    int status;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_arg_t;

void* better_func(void* arg) {
    thread_arg_t* thread_arg = arg;

    pthread_mutex_lock(&thread_arg->mutex);
    while(thread_arg->status != S_READY) {
        pthread_cond_wait(&thread_arg->cond, &thread_arg->mutex);
    }
    thread_arg->num += 100;
    pthread_mutex_unlock(&thread_arg->mutex);

    // ...

    return NULL;
}
```

Common Mistakes

```
enum {S_INIT, S_READY};

typedef struct thread_arg_tag {
    int num;
    int status;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_arg_t;


void* slow_func(void* arg) {
    thread_arg_t* thread_arg = arg;
    pthread_mutex_lock(&thread_arg->mutex);
    do_one_million_calculations(thread_arg->num);
    pthread_mutex_unlock(&thread_arg->mutex);
}
```

Common Mistakes

```c
enum {S_INIT, S_READY};

typedef struct thread_arg_tag {
    int num;
    int status;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_arg_t;


void* slow_func(void* arg) {
    thread_arg_t* thread_arg = arg;
    pthread_mutex_lock(&thread_arg->mutex);
    do_one_million_calculations(thread_arg->num);
    pthread_mutex_unlock(&thread_arg->mutex);
}
```

Correct Approach

```c
enum {S_INIT, S_READY};

typedef struct thread_arg_tag {
    int num;
    int status;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_arg_t;


void* faster_func(void* arg) {
    thread_arg_t* thread_arg = arg;
    int temp;
    pthread_mutex_lock(&thread_arg->mutex);
    temp = thread_arg->num;
    pthread_mutex_unlock(&thread_arg->mutex);
    do_one_million_calculations(temp);
    return NULL;
}
```

An example program

Check the Exam 3 Review Session Folder in HuskyCT

Download *workshop.c* and setup your environment

We will code along together

Solutions will be posted after the review session

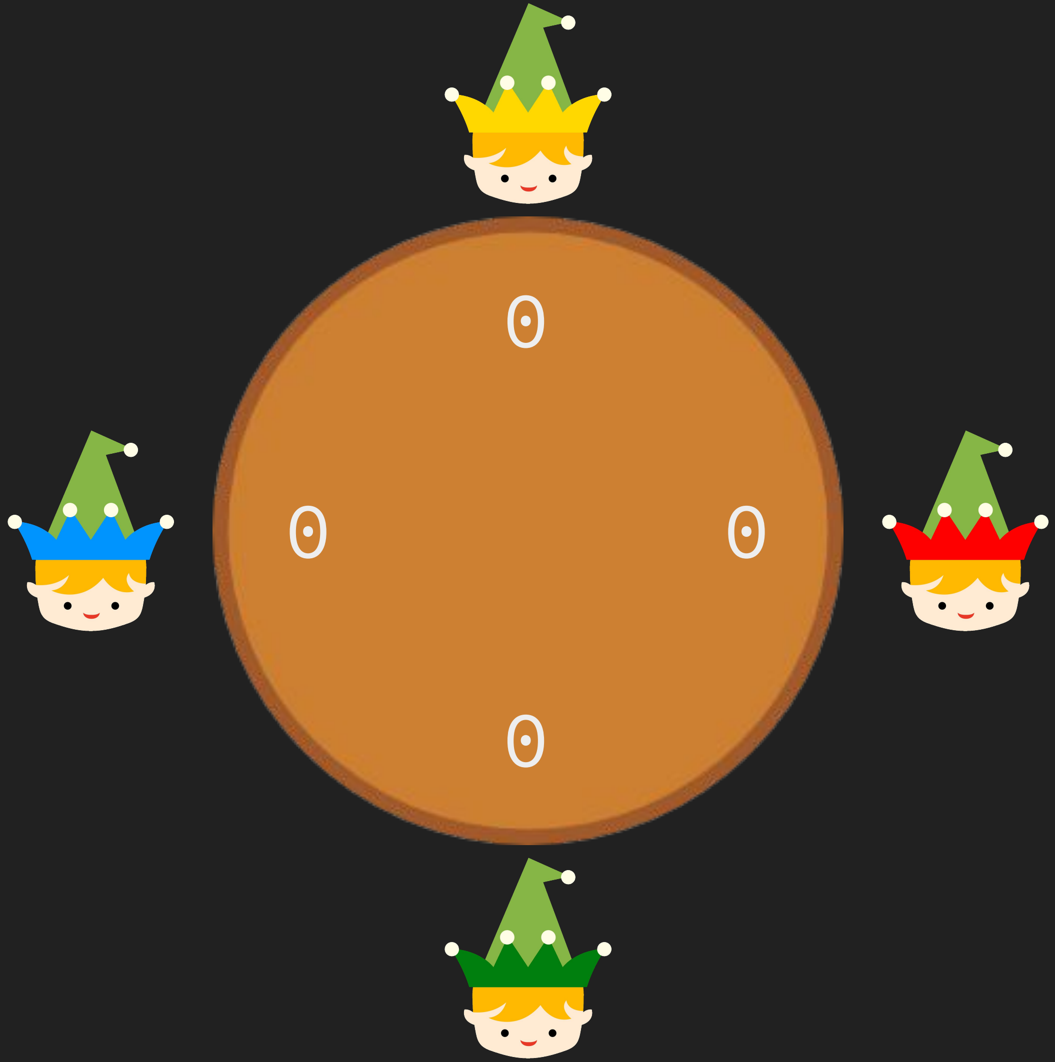
Problem Description

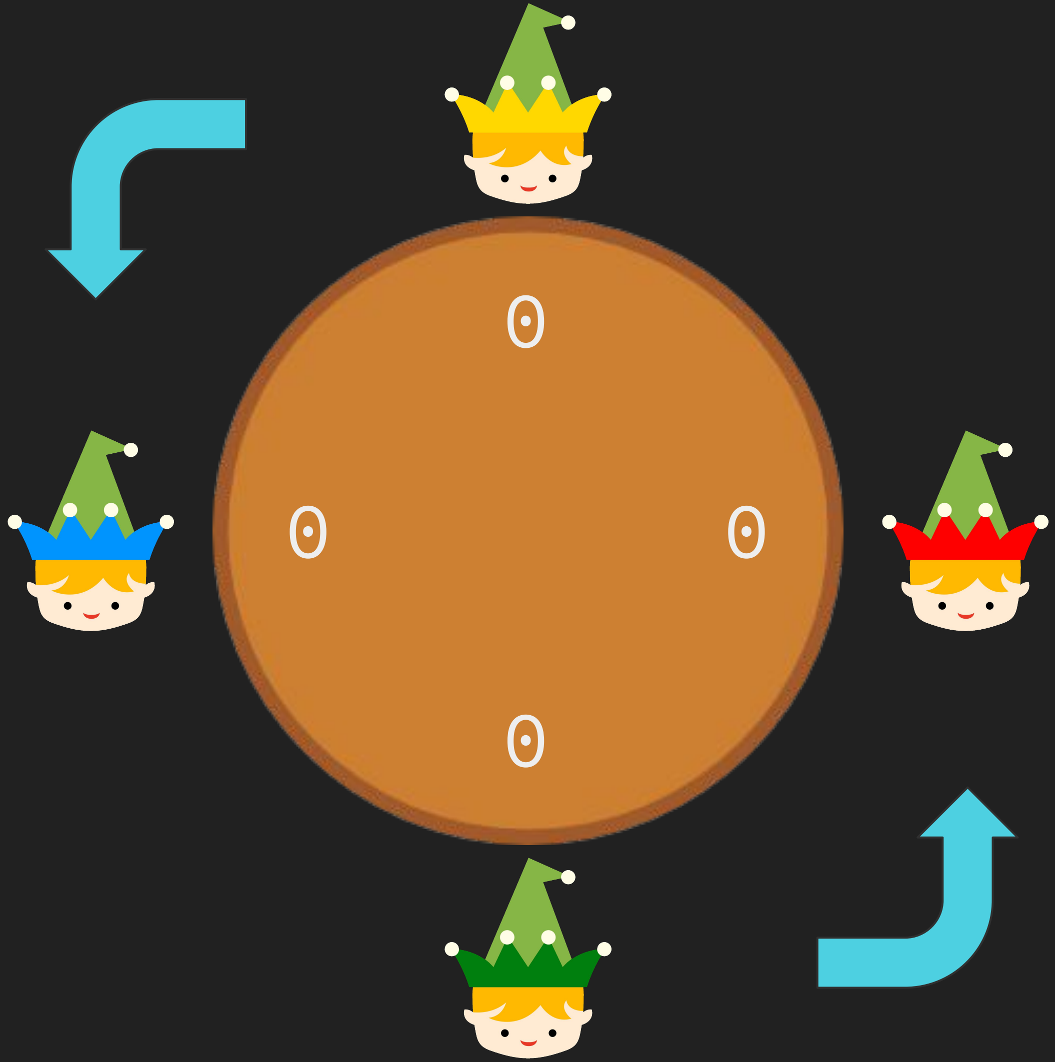
- We will simulate elves in a workshop making toy cars.
- Each car needs a certain number of parts to function
- Let’s say we have four elves and four parts in a car

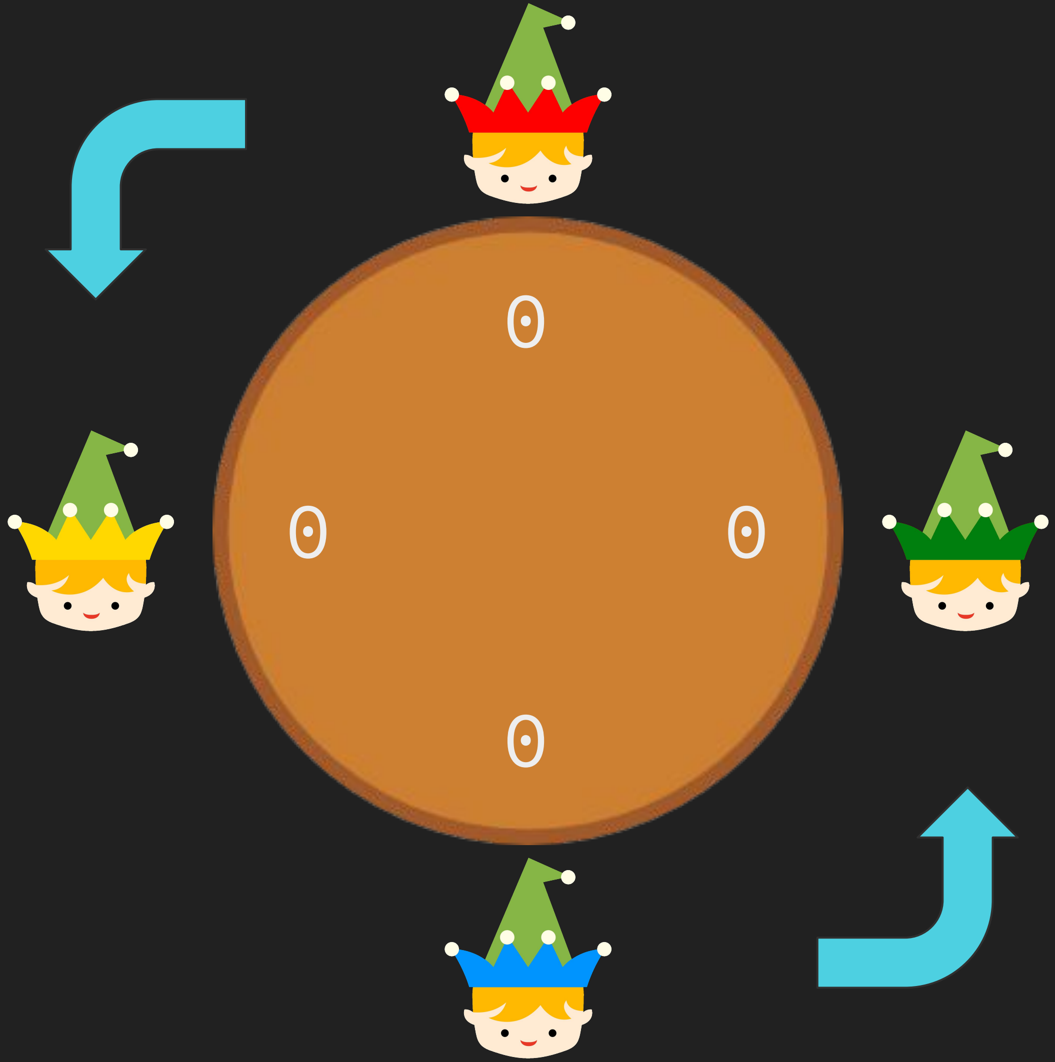
Restriction:

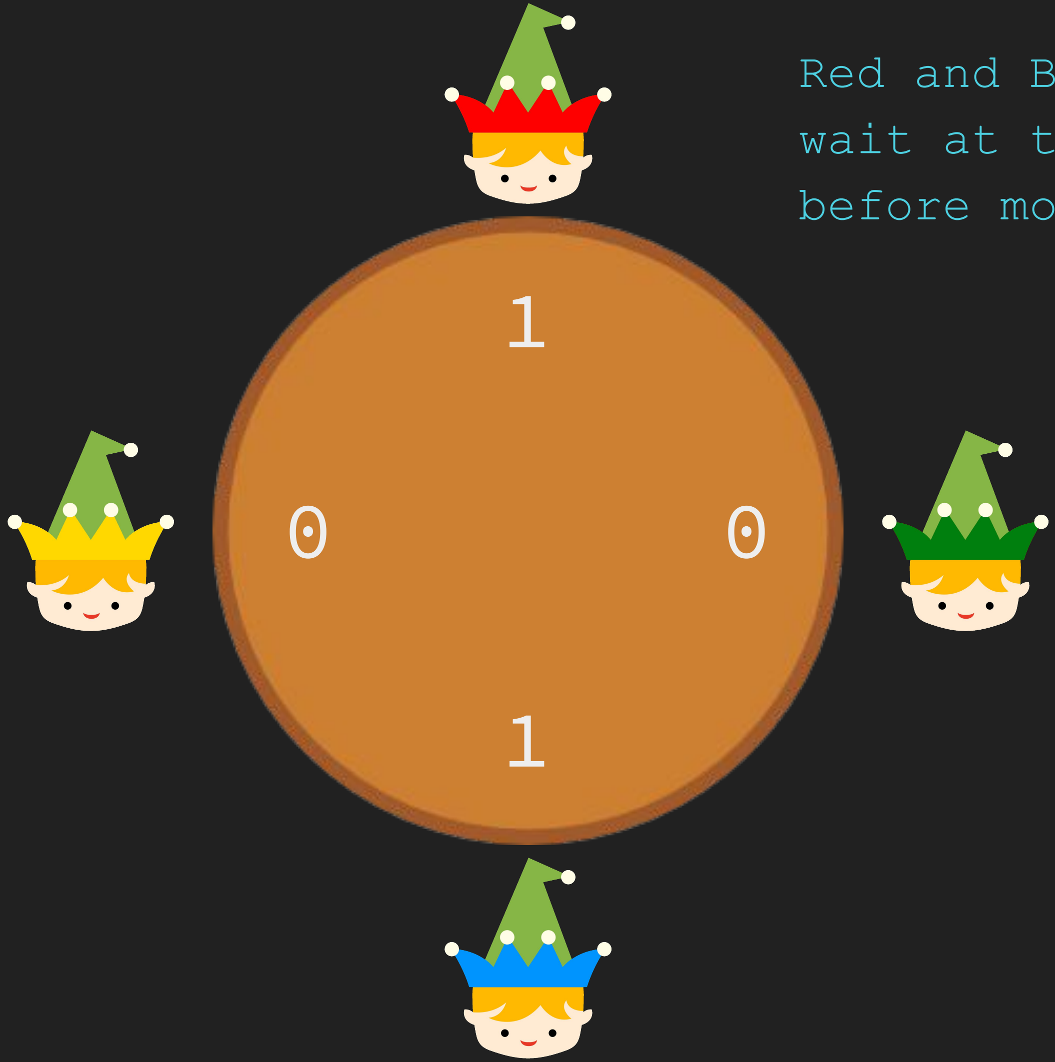
Elves love teamwork, so each elf must add exactly one part to each car.

We can have the elves move in a circle!

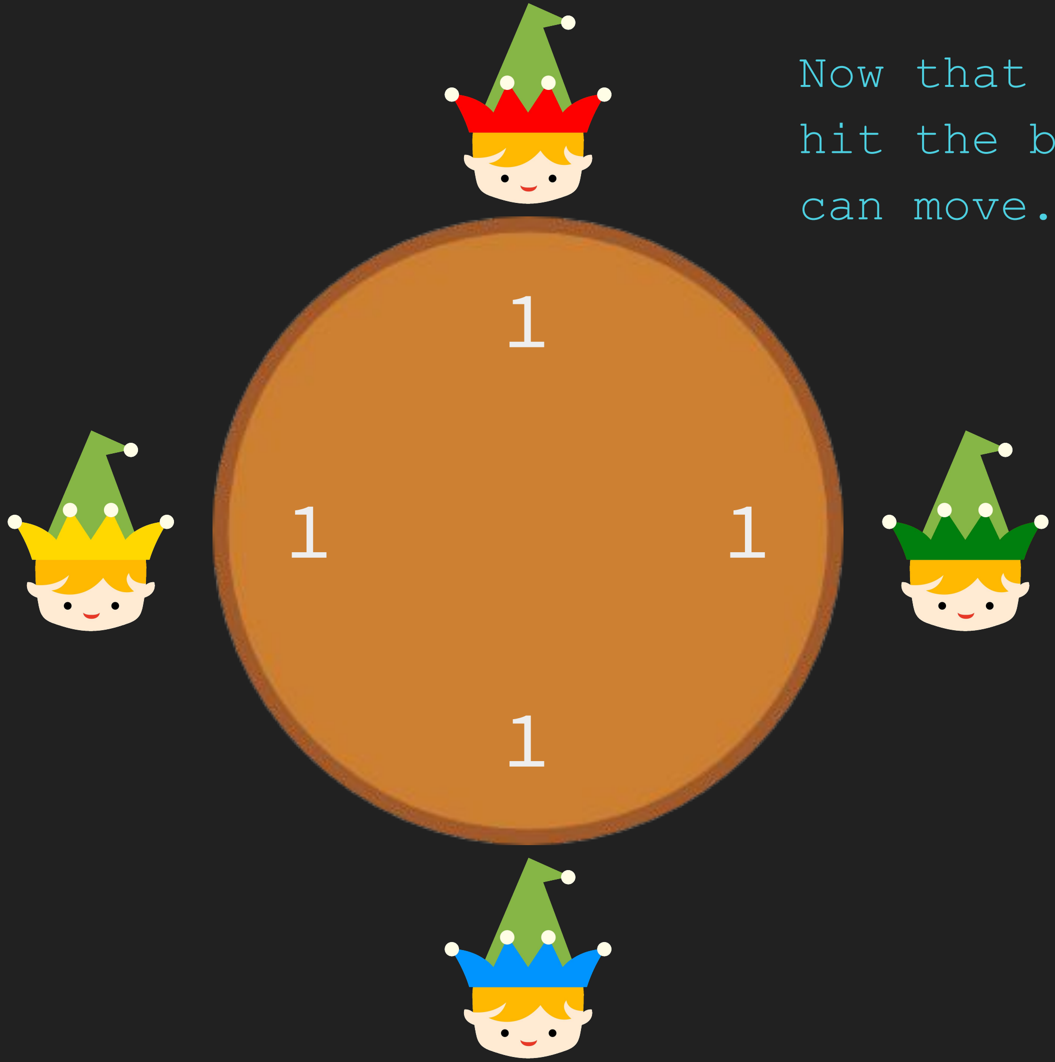

0
0
0
0


0
0
0
0

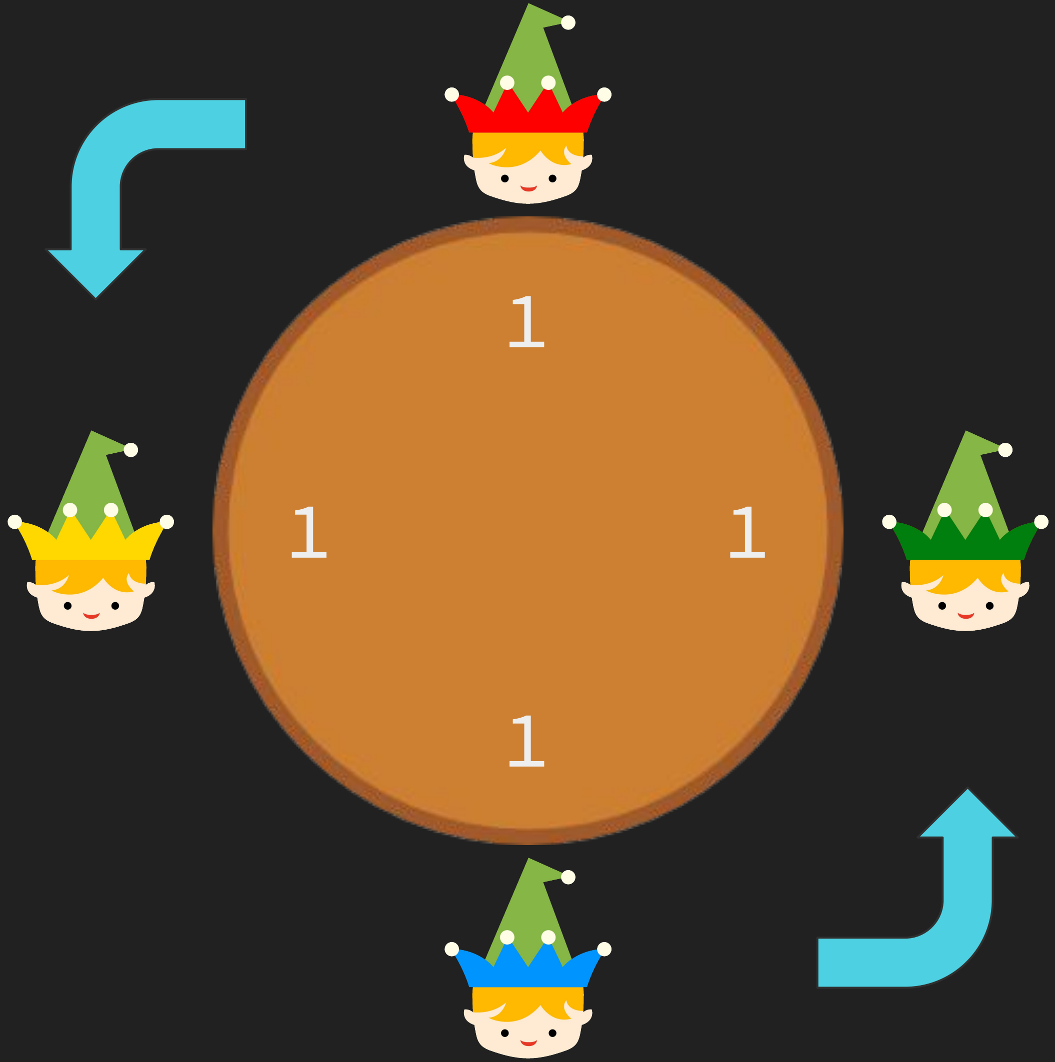

0
0
0
0

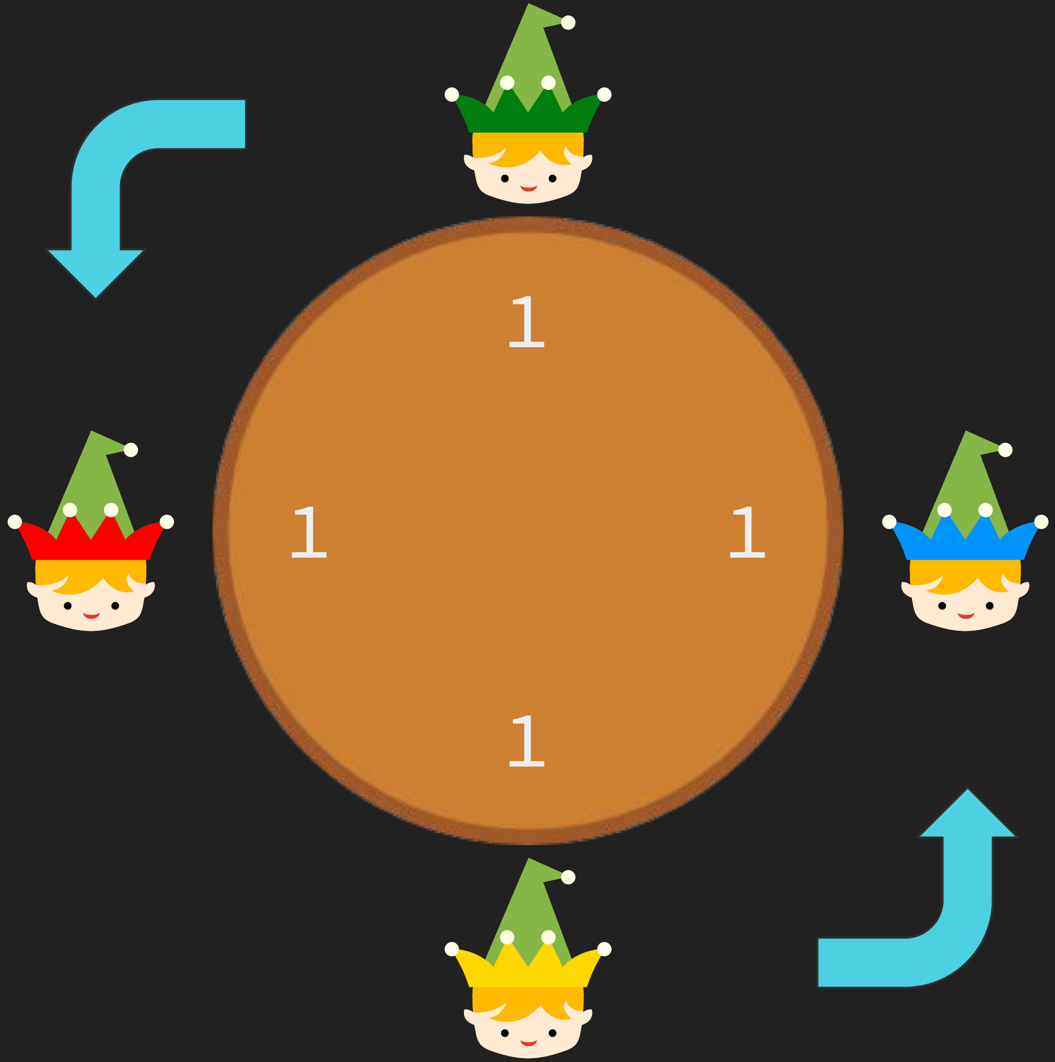


Red and Blue must wait at the barrier before moving!



Now that each elf has hit the barrier, they can move...


1
1
1
1


1
1
1
1

The elves work and wait at the barrier again...

If an elf sees that the car has 4 parts, free it! Then stop.

Summary:

1. Grab a car
2. Wait at the barrier
3. Move one step
4. If the car has less than 4 parts, add one and repeat from step 2
5. Else, free the car and stop working

```
/*
 * The elves sit in a circle and take the following steps:
 * First, each elf grabs a car and places it in the spot corresponding to its id
 * Next, each elf moves to the next spot around the table. 0->1->2->3->0->...
 * Once the elves have moved, each elf adds a part to the car
 * After all the elves complete this task, they move once again (this must be synchronized!)
 * Eventually, every car will have the necessary number of parts and the elves should send them to santa! (Free them)
 * Afterwords, the elves will stop working
 */
```

Extra Notes:

- It is highly recommended that you run your code locally rather than submit to gradescope to test your code
  - It takes a long time for gradescope to run
- PLEASE make sure your code **compiles**
  - Recommendation: each time you have compiled code, copy it into a notepad so that if your changes are unrecoverable, you always have compiled code to submit
- Helpful tool to find thread errors:

  valgrind --tool=helgrind *executable*

Questions?