Assignment 6 Recitation

Concurrency

Concurrency Primitives

- Focus on three types:
  - Threads
  - Mutexes
  - Semaphores
- Already covered high level in lecture
- API needed for assignment + concrete examples
- Homework provides a library in Thread.h (wrapper around pthreads or POSIX threads)

Threads

- Before we can create a new thread, we need a function the thread will run
  - `void* f(void* arg);`
  - Takes a returns a void pointer.
- Void*?
  - A pointer to anything. (Need to cast to use)

Threads

```
void* myprint(void* arg) {
        int* intarg = (int*) arg;
        int intcopy = *intarg;
        printf("%d\n", intcopy);
        return NULL;
}

int i = 10;
myprint((void*) &i); // Prints 10
```

Threads

- What if I need multiple arguments?
  - Structure!

```
struct threadarg {
        int a;
        int b;
};

void* threadadder(void* arg) {
        threadarg* t = (threadarg*) t;
        printf("%d+ %d=%d", t->a, t->b, t->a+ t->b);
        return NULL;
}
```

Threads

```
threadarg t;
t.a = 1;
t.b = 2;
threadadder((void*) &t); // 1+2 = 3
```

Threads

- On to making threads now that we have a function the thread can run!

```
threadarg a;
a.a = 1;
a.b = 2;
Thread* t = new Thread();
t->run(threadadder, (void*) &a);
// 1+2 = 3;
t->join();
```

Mutexes

- Used only when one thread can use a resource at a time

```
Mutex* m = new Mutex();      // Create a mutex.
m->lock();                   // Lock mutex.
m->unlock();                 // Unlock mutex.
```

Mutex Example

```
void a(Mutex* m) {
        m->lock();
        printf("In a!\n");
}
void b(Mutex* m) {
        m->lock();
        printf("In b!\n");
        m->unlock();
}
```

```
//Lock shared by a, b
Mutex m;
a(&m);
b(&m);
```

Deadlock!

```
void a(Mutex* m) {
        m->lock();
        printf("In a!\n");
}
void b(Mutex* m) {
        m->lock();
        printf("In b!\n");
        m->unlock();
}
```

```
//Lock shared by a, b
Mutex m;
a(&m);
b(&m);
```

Deadlock fixed!

```
void a(Mutex* m) {
        m->lock();
        printf("In a!\n");
        m->unlock();
}
void b(Mutex* m) {
        m->lock();
        printf("In b!\n");
        m->unlock();
}
```

```
//Lock shared by a, b
Mutex m;
a(&m);
b(&m);
```

Another Mutex Example

```
void* tf(void* arg) {
        vector<int>* ns = (vector<int>*) arg;
        for (int i = 0; i < 2000; ++i) {
                ns.push_back(i);
        }
        return NULL;
}
```

Another Mutex Example

```
vector<int> nums;
Thread a, b;
a.run(tf, (void*) &nums);
b.run(tf, (void*) &nums);
a.join();
b.join();
printf("size of nums: %d\n", nums.size());
```

Race conditions!

```
vector<int> nums;
Thread a, b;
a.run(tf, (void*) &nums);
b.run(tf, (void*) &nums);
a.join();
b.join();
printf("size of nums: %d\n", nums.size());
```

A quick fix

```
Mutex lock;
void* tf(void* arg) {
        vector<int>* ns = (vector<int>*) arg;
        for (int i = 0; i < 2000; ++i) {
                lock.lock();
                ns.push_back(i);
                lock.unlock();
        }
        return NULL;
}
```

Semaphores

- Can be thought of as generalized mutexes
  - Instead of only 0 or 1 threads holding the lock, 0..n threads can hold the lock

```
Semaphore*s = new Semaphore(2);
s->inc();
s->inc();  // Semaphore full, more incs will block
s->dec(); // Semaphore now has space.
s->dec();
s->value(); // Returns 0
```

Consumer Producer Queues

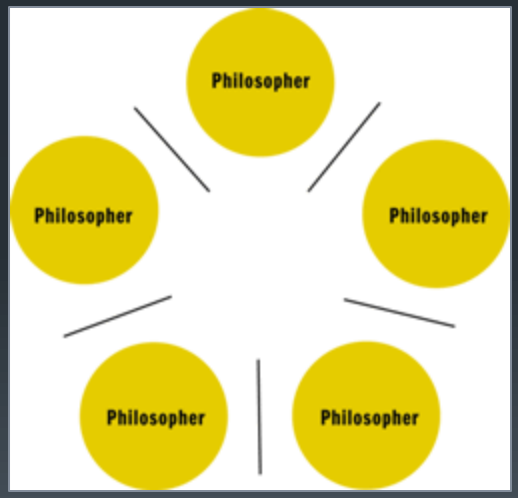
- Some threads produce date
- Some threads read/consume date
- These consumers and producers need to share a fixed size buffer
- Two potential problems
  - Empty buffer, consumer must wait
  - Full buffer, producers must wait
- Apply semaphores?

Consumer Producer Queues

- What do we need to track?
  - Number of slots filled.
  - Number of slots empty
  - Use a semaphore for each value
- Producers
  - Decrement empty slots semaphore
  - Add value
  - Increment filled slots semaphore
- Consumers
  - Decrement filled slots semaphore
  - Use value
  - Increment empty slots semaphore

Dining philosophers

- 5 silent hungry philosophers
- One bowl of spaghetti
- 5 forks placed as shown
- Philosophers alternate eating and thinking
- Must have both forks to left+right to eat
- Cannot grab forks for each other
- Replace forks after eating
- How to design behavior such that each philosopher won't starve?



Proposal

- Think until the right fork is free; when it is, pick it up
- Think until the left fork is free; when it is, pick it up
- When both forks are held, eat for a fixed amount of time
- Then, put the left fork down
- Then, put the right fork down
- Repeat

Deadlock

- All start picking up the right fork at the same time
- Wait forever for the left fork to be ready!

Round Robin

- Since they are silent, imagine a third party helper
- Tells the philosophers when to eat
- Only allows **one** philosopher to eat
- Deadlock?
- Implement using primitives mentioned above

Homework notes:

- Each philosopher runs in its own thread!
- Every philosopher must eventually eat!

Questions?