CS 537
Lecture 19
Condition Variables/Monitors

Michael Swift

Barber code

Variables:
```
sem_t barber;
sem_t customer;
sem_t mutex;
int open_chairs = 3;
```
Initialization:
```
sem_init(&barber, 0);
sem_init(&customer, 0);
sem_init(&mutex, 1);
```

```
boolean shop_full = false;
sem_wait(&mutex);
if (open_chairs == 0)
  shop_full = true;
else open_chairs--;
sem_signal(&mutex);

if (!shop_full) {
  sem_signal(&barber);
  sem_wait(&customer);
}
```

Barber Code

```
Barber:
 while (1) {
  sem_wait(&barber);

  sem_wait(&mutex);
  open_chairs++;
  move_customer_to_barber_chair();
  sem_signal(&mutex);

  cut_hair();
  sem_signal(&customer);
 }
```

Dining Philosophers

- Problem Statement:
  - N Philosophers sitting at a round table
  - Each philosopher shares a fork with neighbor
  - Each philosopher must have both forks to eat
  - Neighbors can't eat simultaneously
  - Philosophers alternate between thinking and eating
- Each philosopher/thread i runs following code:

```
while (1) {
  think();
  take_forks(i);
  eat();
  put_forks(i);
}
```

Dining Philosophers: Attempt #1

- Two neighbors can't use fork at same time
- Must test if fork is there and grab it atomically
  - Represent each fork with a semaphore
  - Grab right fork then left fork
- Code for 5 philosophers:

```
sem_t fork[5]; // Initialize each to 1
take_forks(int i) {
  wait(&fork[i]);
  wait(&fork[(i+1)%5]);
}
put_forks(int i) {
  signal(&fork[i]);
  signal(&fork[(i+1)%5]);
}
```

- What is wrong with this solution???

Dining Philosophers: Attempt #2

- Approach
  - Grab lower-numbered fork first, then higher-numbered
- Code for 5 philosophers:
-

```
sem_t fork[5]; // Initialize to 1
take_forks(int i) {
  if (i < 4) {
    wait(&fork[i]);
    wait(&fork[i+1]);
  } else {
    wait(&fork[0]);
    wait(&fork[4]);
  }
```

- What is wrong with this solution???

Dining Philosophers: How to Approach

- Guarantee two goals
  - Safety: Ensure nothing bad happens (don't violate constraints of problem)
  - Liveness: Ensure something good happens when it can (make as much progress as possible)
- Introduce state variable for each philosopher i
  - `state[i] = THINKING, HUNGRY, or EATING`
- Safety: No two adjacent philosophers eat simultaneously
  - `for all i: !(state[i]==EATING && state[i+1%5]==EATING)`
- Liveness: Not the case that a philosopher is hungry and his neighbors are not eating
  - `for all i: !(state[i]==HUNGRY && (state[i+4%5]!=EATING && state[i+1%5]!=EATING))`

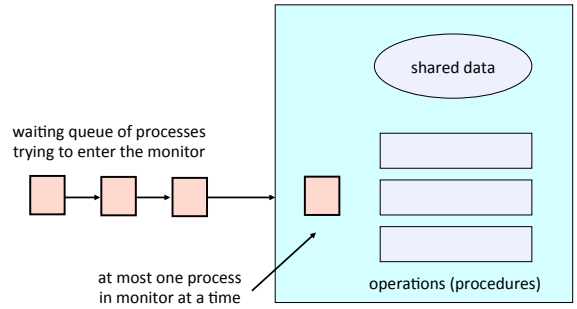
Dining Philosophers: Solution

```
sem_t mayEat[5]; // how to initialize?
sem_t mutex; // how to init?
int state[5] = {THINKING};
take_forks(int i) {
    wait(&mutex); // enter critical section
    state[i] = HUNGRY;
    testSafetyAndLiveness(i); // check if I can run
    signal(&mutex); // exit critical section
    wait(&mayEat[i]);
}
put_forks(int i) {
    wait(&mutex); // enter critical section
    state[i] = THINKING;
    test(i+1 %5); // check if neighbor can run now
    test(i+4 %5);
    signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
    if (state[i]==HUNGRY && state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
     state[i] = EATING;
     signal(&mayEat[i]);
    }
}
```

Two Classes of Synchronization Problems

- Uniform resource usage with simple scheduling constraints
  - No other variables needed to express relationships
  - Use one semaphore for every constraint
  - Examples: thread join and producer/consumer
- Complex patterns of resource usage
  - Cannot capture relationships with only semaphores
  - Need extra state variables to record information
  - Use semaphores such that
    - One is for mutual exclusion around state variables
    - One for each class of waiting
- Always try to cast problems into first, easier type

A monitor



Readers and Writers Monitor Example

```
Monitor ReadersNWriters {
  int WaitingWriters,
      WaitingReaders,
      NReaders, NWriters;
  Condition CanRead, CanWrite;
  mutex_t lock;

Void BeginWrite()  {
   acquire(lock);
   while(NWriters == 1 ||
      NReaders > 0) {
     ++WaitingWriters;
     wait(CanWrite);
     --WaitingWriters;
   }
   NWriters = 1;
   release(lock);
}
```

```
Void EndWrite() {
  acquire(lock);
  NWriters = 0;
  if(WaitingReaders)
    Signal(CanRead);
  else Signal(CanWrite);
  Release(lock);
}
```


Readers and Writers Monitor Example

```
Void BeginRead() {
  acquire(lock);
  bool didWait = F;
  while(NWriters == 1 ||
     (WaitingWriters > 0 &&
      NReaders > 0 &&
      !didWait)) {
    ++WaitingReaders;
    Wait(CanRead);
    --WaitingReaders;
    didWait = T;
  }
  ++NReaders;
  Signal(CanRead);
  release(lock);
}
```

```
Monitor
ReadersNWriters {
 int WaitingWriters,
     WaitingReaders,
      NReaders,
     NWriters;
 Condition CanRead,
           CanWrite;
 mutex_t lock;

Void EndRead() {
  acquire(lock);
  if(--NReaders ==
       0)
  Signal(CanWrite);
  release(lock);
}
```

Examples

- Traffic light
  - Only one direction of traffic can flow at a time

Traffic light

```
struct traffic_light{
   enum direction = {left, right};
   enum color = {green, yellow, red};
   color current_color[direction] = {green, red};
   cond_t changed[direction];
   direction current_dir = left;
   direction light_dir = left;
   int in_intersection = 0;
   mutex_t *lock;

enter_left(dir)
   mutex_lock(lock)
   while ((current_dir != dir) && (current_color != green))
    cond_wait(changed[dir], loock);
   in_intersection++;
   mutex_unlock(lock);
   return;

exit(dir)
   mutex_lock(lock)
   in_intersection--;
   if (in_intersection == 0) && (current_color[dir] == red) {
         current_dir = light_dir;
    broadcast(changed[other_dir(dir)]);
   }
   mutex_unlock(lock);
```


Traffic light

```
struct traffic_light{
   enum direction = {left, right};
   enum color = {green, yellow, red};
   color current_color[direction] = {green, red};
   cond_t changed[direction];
   direction current_dir = left;
   int in_intersection = 0;
   mutex_t *lock;


timer()
   mutex_lock(lock);
   switch(current_color[light_dir]) {
    case green:
        current_color[light_dir] = yellow;
    case yellow:
        current_color[light_dir] = red;
        current_dir = other_dir(light_dir);
        current_color[light_dir] = green;
        if (in_intersection == 0) {
                        current_dir = light_dir;
            broadcast(changed[current_dir]);
                }
   }
   mutex_unock(lock);
```


In-class Problem

- A file is to be shared among different threads, each of which has a unique number.
- The file can be accessed simultaneously by several threads, subject to a single constraint: the sum of the numbers of the threads cannot exceed n, where n is a constant.
- Write code using locks and condition variables to coordinate access to the file. The interface to the file should be:
  - `void access_file(void)`
  - `void release_file(void)`
- The access_file() function should block until the file is available, and the release_file() function should wake up any necessary waiting threads.

Problem Solution

```
pthread_mutex_t lock;
pthread_cond_t cond;
int max_sum = n;
int current_sum = 0;

void access_file(void) {
  pthread_mutex_lock(&lock);
  while (current_sum + thread_number > max_sum) {
    pthread_cond_wait(&cond, &lock);
  }
  current_sum += thread_number;
  pthread_mutex_unlock(&lock);
}

void release_file(void) {
  pthread_mutex_lock(&lock);
  current_sum -= thread_number;
  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&lock);
}
```