CS 537
Lecture 8
Monitors

Michael Swift

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
- Today: Two examples using second approach

Thread Join with Semaphores

- General case: One thread waits for another to reach some point
- Example: Implement thread_join()
  - Parent thread calls thread_join(), which must wait for child thread to call exit();
  - Shared sem between parent and child (created when child thread is created)
    To what value is sem initialized???

Parent thread

```
Thread_join() {
        sem_wait(&sem);
}
```

Child thread

```
exit() {
        sem_signal(&sem);
}
```


Dining Philosophers

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

```
while (1) {
  think();
  take_chopsticks(i);
  eat();
  put_chopsticks(i);
}
```

Dining Philosophers: Attempt #1

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

```
sem_t chopstick[5]; // Initialize each to 1
take_chopsticks(int i) {
  wait(&chopstick[i]);
  wait(&chopstick[(i+1)%5]);
}
put_chopsticks(int i) {
  signal(&chopstick[i]);
  signal(&chopstick[(i+1)%5]);
}
```

- What is wrong with this solution???

Dining Philosophers: Attempt #2

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

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

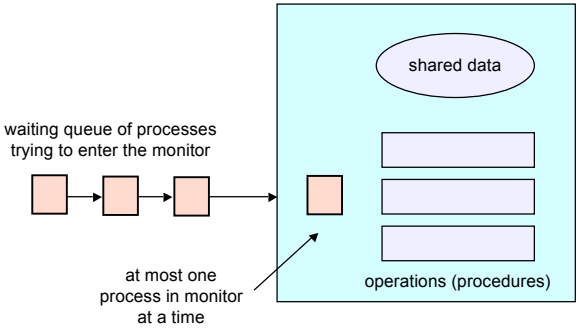
```
sem_t mayEat[5]; // how to initialize?
sem_t mutex; // how to init?
int state[5] = {THINKING};
take_chopsticks(int i) {
   wait(&mutex); // enter critical section
   state[i] = HUNGRY;
   testSafetyAndLiveness(i); // check if I can run
   signal(&mutex); // exit critical section
   wait(&mayEat[i]);
}
put_chopsticks(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]);
   }
}
```

Dining Philosophers: Example Execution

Monitors

- A programming language construct that supports controlled access to shared data
  - synchronization code added by compiler, enforced at runtime
  - why does this help?
- Monitor is a software module that encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent processes that invoke those procedures
- Monitor protects the data from unstructured access
  - guarantees only access data through procedures, hence in legitimate ways

A monitor



Monitor facilities

- Mutual exclusion
  - only one process can be executing inside at any time
    - thus, synchronization implicitly associated with monitor
  - if a second process tries to enter a monitor procedure, it blocks until the first has left the monitor
    - more restrictive than semaphores!
    - but easier to use most of the time
- Once inside, a process may discover it can't continue, and may wish to sleep
  - or, allow some other waiting process to continue
  - condition variables provided within monitor
    - processes can wait or signal others to continue
    - condition variable can only be accessed from inside monitor

Implementation

- As a library (pthreads)

```
pthread_mutex_t mu;
pthread_cond_t co;
boolean ready;
void foo( ) {
 pthread_mutex_lock(&mu);
 if (!ready)
   pthread_cond_wait(&co, &mu);
 …
 ready = TRUE;
 pthread_cond_signal(&co);  // unlock and signal
 pthread_mutex_unlock(&mu);
```

- As a language (Java)

```
synchronized withdraw(int amount) {
  while (balance < amount) {
   wait();
  balance -= amount;
  if (balance == 0) {
   notify();
}
```


Condition Variables

- A place to wait; sometimes called a rendezvous point
  - Always used with a monitor lock
  - No value (history) associated with condition variable
- Three operations on condition variables
  - wait(c)
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have wait queues
  - signal(c)
    - wake up at most one waiting process/thread
    - if no waiting processes, signal is lost
    - this is different than semaphores: no history!
  - broadcast(c)
    - wake up all waiting processes/threads

Signaling

- Mesa monitors: signal(c) means
  - Wake one thread waiting on this condition variable (if any)
    - Signaller can keep lock and CPU
  - waiter is made ready, but the signaller continues
    - waiter runs when signaller leaves monitor (or waits)
    - condition is not necessarily true when waiter runs again
  - signaller need not restore invariant until it leaves the monitor
  - being woken up is only a hint that something has changed
    - must recheck conditional case
- Broadcast (or NotifyAll)
  - Wake all threads waiting on condition variable
  - Avoids need for multiple condition variables

Producer/Consumer: pthread monitors

- Another thread may be scheduled and acquire lock before signalled thread runs
- Implication: Must recheck condition with while() loop instead of if()

Shared variables

```
cond_t empty, full;
int slots = 0;
```

Producer

```
While (1) {
   mutex_lock(&lock);
   while (slots==N)
       cond_wait(&empty,&lock);
   myi = findempty(&buffer);
   Fill(&buffer[myi]);
   slots++;
   cond_signal(&full);
   mutex_unlock(&lock);
}
```

Consumer

```
While (1) {
   mutex_lock(&lock);
   while(slots==0)
       cond_wait(&full,&lock);
   myj = findfull(&buffer);
   Use(&buffer[myj]);
   slots--;
   cond_signal(&empty);
   mutex_unlock(&lock);
```

Traffic light

```
monitor 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;

enter_left(dir)
   while ((current_dir != dir) && (current_color != green))
        cond_wait(changed[dir]);
   in_intersection++;
   return;

exit(dir)
   in_intersection--;
   if (in_intersection == 0) && (current_color[dir] == red)
        broadcast(changed[other_dir(dir)]);

timer()
   switch(current_color[direction]) {
        case green:
                current_color[current_dir] = yellow;
        case yellow:
                current_color[current_dir] = red;
                current_dir = other_dir(current_dir);
                current_color[current_dir] = green;
                if (in_intersection == 0) {
                        broadcast(changed[current_dir]);
```


Examples

- Traffic light
  - Only one direction of traffic can flow at a time
- Try more at home from the book!
  - I will correct them if you would like