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

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);
}

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: 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

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})
Monitors

Motivation
• Users can inadvertently misuse locks and semaphores (e.g., never unlock a mutex)

Idea
• Provide language support to automatically lock and unlock monitor lock when in critical section
  - Lock is added implicitly; never seen by user
• Provide condition variables for scheduling constraints

Examples
• Mesa language from Xerox
• Java from Sun
  - Use synchronized keyword when defining method
  synchronized deposit(int amount) {
    balance += amount;
  }

Condition Variables

Idea
• Used to specify scheduling constraints
• Always used with a monitor lock
• No value (history) associated with condition variable

Allocate: Cannot initialize value!
• Must allocate a monitor lock too (implicit with language support, explicit in POSIX and C)
  pthread_mutex_t monitor_lock = PTHREAD_MUTEX_INITIALIZER;

Wait
• Call with monitor lock held; Releases monitor lock, sleeps until signalled, reacquires lock when woken
• NOTE: No test inside of wait(); will always sleep!
  pthread_mutex_lock(&monitor_lock);
  if (expression) pthread_cond_wait(&cond, &monitor_lock);
  pthread_mutex_unlock(&monitor_lock);

Broadcast (or NotifyAll)
• Wake all threads waiting on condition variable

Producer/Consumer: Hoare

Attempt #1

Final case:
• Multiple producer threads, multiple consumer threads
• Shared buffer with N elements between producer and consumer

Shared variables
lock_t monitor;
cond_t empty, full;
Producer

While (1) {
  mutex_lock(&monitor);
  cond_wait(&empty, &monitor);
  myi = findempty(&buffer);
  Fill(&buffer[myi]);
  mutex_unlock(&monitor);
}

Consumer

While (1) {
  mutex_lock(&monitor);
  cond_wait(&full, &monitor);
  myj = findfull(&buffer);
  Use(&buffer[myj]);
  cond_signal(&empty);
  mutex_unlock(&monitor);
}

Why won’t this work?
Producer/Consumer: Hoare
Attempt #2

Shared variables
lock_t monitor;
cond_t empty, full;
int slots = 0;

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

Consumer
While (1) {
    mutex_lock(&monitor);
    if (slots==0)
        cond_wait(&full,&monitor);
    myj = findfull(&buffer);
    Use(&buffer[myj]);
    slots--;
    cond_signal(&empty);
    mutex_unlock(&monitor);
}

Producer/Consumer: Mesa
Mesa: Another thread may be scheduled and acquire lock before
signalled thread runs
Repeat Example: Two producers, two consumers...

What can go wrong?

Producer/Consumer: Mesa
Mesa: 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);
}