Two Classes of Semaphore Problems

Uniform resource usage w/ simple scheduling constraints
- No other variables to express relationships
- Use one semaphore for every constraint
- Examples: Thread join and producer/consumer (Previous Lecture)

Complex patterns of resource usage
- Can’t capture relationships with only semaphores
- Need extra state variables to record information
- Use of semaphores
  - One for mutual exclusion for state variables
  - One for each class of waiting

Try to cast problems into first type
Today: Two examples that use second approach

Classic Examples and Monitors

Questions answered in these notes:
- Semaphore Examples: Dining Philosophers and Reader/Writers
- Monitors and condition variables: Three styles
  - Hoare
  - Mesa
  - Java

Reading
- Chapter 6 (through 6.7)

Dining Philosophers

Each philosopher must have both chopsticks to eat

Philosophers alternate between thinking and eating

class Philosopher implements Runnable {
    int i; // which Philosopher am I?
    public void run() {
        while (true) {
            think();
            take_chopsticks(i);
            eat();
            put_chopsticks(i);
        }
    }
}

Dining Philosophers: Attempt 1

Represent each chopstick with a semaphore; Grab right then left

Semaphore chopstick[5] = Initialize each to 1;
void take_chopsticks(int i) {
    chopstick[i].P();
    chopstick[(i+1) % 5].P();
}
void put_chopsticks(int i) {
    chopstick[i].V();
    chopstick[(i+1) % 5].V();
}

What is wrong with this solution?
Dining Philosophers: Attempt 2

Grab lower-numbered chopstick first, then higher-numbered

Semaphore chopstick[5] = 1;
void take_chopsticks(int i) {
  if (i < 4) {
    chopstick[i].P();
    chopstick[i+1].P();
  } else {
    chopstick[0].P();
    chopstick[4].P();
  }
}

What is wrong with this solution?

Dining Philosophers: How to Approach

Guarantee two goals
- Safety: Make sure nothing bad happens
- Liveness: Make sure something good happens when it can

Introduce state variable for each philosopher i
- state[i]: THINKING, HUNGRY, or EATING

Safety: No two adjacent philosophers are eating simultaneously
- for all i: !(state[i] == EATING && state[i+1 % 5] == EATING)

Liveness: No philosopher is hungry unless a neighbor is eating
- for all i: !(state[i] == HUNGRY && (state[i+4 % 5] != EATING && state[i+1 % 5] != EATING))

Dining Philosophers: Solutions

Semaphore mayEat[5] = { 0, 0, 0, 0, 0};
Semaphore mutex = 1;
final static public int THINKING = 0; final static public int HUNGRY = 1;
final static public int EATING = 2;
int state[5] = { THINKING, THINKING, THINKING, THINKING, THINKING};
void take_chopsticks(int i) {
  mutex.P();
  state[i] = HUNGRY;
  testSafetyAndLiveness(i);
  mutex.V();
  mayEat[i].P();
}
void put_chopsticks(int i) {
  mutex.P();
  state[i] = THINKING;
  test(i+1 % 5);
  test(i+4 % 5);
  mutex.V();
}
void testSafetyAndLiveness(int i) {
  if (state[i]==HUNGRY&&state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
    state[i] = EATING;
    mayEat[i].V();
  }
}

Readers and Writers Problem Statement

Two different classes of users
- Any number of readers can access data
- Each writer must have exclusive access

Two possibilities for priorities
- 1 – No reader waits unless writer in critical section
  Writers can starve
  Solution presented in text book
- 2 – No writer waits longer than absolute minimum
  Readers can starve
  Much more difficult, solution here

Four state variables:
- ActiveReaders, WaitingReaders, ActiveWriters, WaitingWriters

Three semaphores:
- mutex = ??, OKToRead = ??, OKToWrite = ??
Readers and Writers Implementation

**Reader Process:**

mutex.P();
if (ActiveWriters + WaitingWriters == 0) {
  OKToRead.V();
  ActiveReaders++;
} else WaitingReaders++;
mutex.V();
OKToRead.P();

// Do read
mutex.P();
ActiveReaders--;
if (ActiveReaders == 0 && WaitingWriters > 0) {
  OKToWrite.V();
  ActiveWriters++; WaitingWriters--;
} else WaitingWriters--;
mutex.V();
OKToRead.P();

**Writer Process:**

mutex.P();
if (ActiveWriters + ActiveReaders + WaitingWriters == 0) {
  OKToWrite.V();
  ActiveWriters++;
} else WaitingWriters++;
mutex.V();
OKToWrite.P();

// Do write
mutex.P();
ActiveWriters--;
if (WaitingWriters > 0) {
  OKToWrite.V();
  ActiveWriters++;
  WaitingWriters--;
} else
  while (WaitingReaders > 0) {
    OKToRead.V();
    ActiveReaders++;
    WaitingReaders--;
  }
mutex.V();

Monitors

Even higher-level data abstraction for concurrent accesses

- Implicit locks for mutex
- Zero or more **condition variables** for scheduling constraints

Programming language construct

- Examples: Mesa language from Xerox and Java from Sun
- Acquire monitor lock when call **synchronized** methods in class

```java
class Queue {
  int head, tail; // shared data
  public synchronized Add(val) {
    // adds lock.Acquire();
    // code to add val to queue
    // adds lock.Release();
  }
  public synchronized int Remove() {
    // adds lock.Acquire();
    // code to remove from queue
    // adds lock.Release();
  }
}
```

Java example

```java
// compiler add Lock lock = new Lock();
public synchronized Add(val) {
  // adds lock.Acquire();
  // code to add val to queue
  // adds lock.Release();
}
```

Condition Variables

- Three basic atomic operations
  - Slightly different semantics depending on Hoare-style or Mesa-style

- **wait()**
  - Release monitor lock, sleep, reacquire lock when awoken
  - **Usage:** if (!expression) condition.wait();

- **signal() (notify() in Java)**
  - Wake one process waiting on condition (if there is one)
  - **Hoare:** Signaller relinquishes lock and processor to waiter (Theory)
  - **Mesa and Java:** Signaller keeps lock and processor (Practice)
  - No history in condition variable

- **broadcast() (notifyAll() in Java)**
  - Wake all processes waiting on condition
  - Useful when condition processes are waiting for varies

Hoare-style Semantics

- **Producer/Consumer with bounded buffers**
  - Guaranteed to run signalled process immediately

```java
Producer Monitor
if (fullEntries == MAX) {
  empty.wait();
  // adds lock.acquire()
  FillBuffer();
  fullEntries++;
  full.signal();
}
Consumer Monitor
if (fullEntries == 0) {
  empty.wait();
  // lock.release()
  UseBuffer();
  fullEntries--;
  full.signal();
}
```
Mesa-style Semantics

- Producer/Consumer with bounded buffers
  Condition empty, full;
  int fullEntries = 0;

Producer Monitor

\[
\text{while (fullEntries == MAX) \{ }
\text{  // lock.release();}
\text{  empty.wait();}
\text{  // lock.acquire();}
\text{\}}
\text{FillBuffer();}
\text{fullEntries++;}
\text{full.signal();}
\]

Consumer Monitor

\[
\text{while (fullEntries == 0) \{ }
\text{  // lock.release();}
\text{  full.wait();}
\text{  // lock.acquire();}
\text{\}}
\text{UseBuffer();}
\text{fullEntries--;}
\text{empty.signal();}
\]

Another process may be scheduled before signalled process runs
- Implication: Must recheck condition with while() loop

Java-style Semantics

- Producer/Consumer with bounded buffers
  // compiler adds one Condition implicitly for each object
  int fullEntries = 0;

Producer Monitor

\[
\text{synchronized AddToBuffer () \{ }
\text{  while (fullEntries == MAX) \{ }
\text{    // lock.release();}
\text{    wait();}
\text{    // lock.acquire();}
\text{\}}}
\text{FillBuffer();}
\text{fullEntries++;}
\text{notify();}
\text{\}}
\]

Consumer Monitor

\[
\text{synchronized RemoveFromBuffer()\{ }
\text{  while (fullEntries == 0) \{ }
\text{    // lock.release();}
\text{    wait();}
\text{    // lock.acquire();}
\text{\}}}
\text{UseBuffer();}
\text{fullEntries--;}
\text{notify();}
\text{\}}
\]

Share one condition variable between producer and consumer
Will this work?

Java with notifyAll()

- Producer/Consumer with bounded buffers
  // compiler adds one Condition implicitly for each object
  int fullEntries = 0;

Producer Monitor

\[
\text{synchronized AddToBuffer () \{ }
\text{  while (fullEntries == MAX) \{ }
\text{    // lock.release();}
\text{    wait();}
\text{    // lock.acquire();}
\text{\}}}
\text{FillBuffer();}
\text{fullEntries++;}
\text{notifyAll();}
\text{\}}
\]

Consumer Monitor

\[
\text{synchronized RemoveFromBuffer()\{ }
\text{  while (fullEntries == 0) \{ }
\text{    // lock.release();}
\text{    wait();}
\text{    // lock.acquire();}
\text{\}}}
\text{UseBuffer();}
\text{fullEntries--;}
\text{notifyAll();}
\text{\}}
\]

Wake up all waiting processes with notifyAll()
Will this work? Is it efficient?