Outline Part 2

• Objectives:
  – Higher level synchronization mechanisms
  – Classic problems in concurrency
• Administrative details:

Semaphores

• Well-known synchronization abstraction
• Defined as a non-negative integer with two atomic operations
  \[ P(s) \cdot \{ \text{wait until } s > 0; s-- \} \]
  \[ V(s) \cdot \{ s++ \} \]
• The atomicity and the waiting can be implemented by either busywaiting or blocking solutions.

Semaphore Usage

• Binary semaphores can provide mutual exclusion (solution of critical section problem)
• Counting semaphores can represent a resource with multiple instances (e.g. solving producer/consumer problem)
• Signaling events (persistent events that stay relevant even if nobody listening right now)
The Critical Section Problem

while (1) {
  ...other stuff...
  P(mutex)
  critical section
  V(mutex)
}

Semaphore: mutex initially 1

When is a code section a critical section?

Thread 0    Thread 1

a = a + c;   a = a + c;
b = b + c;   b = b + c;

When is a code section a critical section?

Thread 0    Thread 1
P(mutex)    P(mutex)
a = a + c;   a = a + c;
b = b + c;   b = b + c;
V(mutex)    V(mutex)
When is a code section a critical section?

**Thread 0**
- P(mutexa)
- a = a + c;
- V(mutexa)
- P(mutexb)
- b = b + c;
- V(mutexb)

**Thread 1**
- P(mutexa)
- a = a + c;
- V(mutexa)
- P(mutexb)
- b = b + c;
- V(mutexb)

When is a code section a critical section?

**Thread 0**
- a = a + c;
- b = b + c;

**Thread 1**
- a = a + c;
- b = b + c;

**Thread 2**
- c = a + b;
- b = b + c;
### When is a code section a critical section?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(mutexa)</td>
<td>P(mutexa)</td>
<td>P(mutexa)</td>
</tr>
<tr>
<td>a = a + c;</td>
<td>a = a + c;</td>
<td>a = a + c;</td>
</tr>
<tr>
<td>V(mutexa)</td>
<td>V(mutexa)</td>
<td>V(mutexa)</td>
</tr>
<tr>
<td>P(mutexb)</td>
<td>P(mutexb)</td>
<td>P(mutexb)</td>
</tr>
<tr>
<td>b = b + c;</td>
<td>b = b + c;</td>
<td>b = b + c;</td>
</tr>
<tr>
<td>V(mutexb)</td>
<td>V(mutexb)</td>
<td>V(mutexb)</td>
</tr>
</tbody>
</table>

### When is a code section a critical section?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(mutex)</td>
<td>P(mutex)</td>
<td>P(mutex)</td>
</tr>
<tr>
<td>a = a + c;</td>
<td>a = a + c;</td>
<td>a = a + c;</td>
</tr>
<tr>
<td>V(mutex)</td>
<td>V(mutex)</td>
<td>V(mutex)</td>
</tr>
<tr>
<td>P(mutexb)</td>
<td>P(mutexb)</td>
<td>P(mutexb)</td>
</tr>
<tr>
<td>b = b + c;</td>
<td>b = b + c;</td>
<td>b = b + c;</td>
</tr>
<tr>
<td>V(mutexb)</td>
<td>V(mutexb)</td>
<td>V(mutexb)</td>
</tr>
</tbody>
</table>

### When is a code section a critical section?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(mutex);</td>
<td>P(mutex);</td>
<td>P(mutex);</td>
</tr>
<tr>
<td>a = a + c;</td>
<td>a = a + c;</td>
<td>c = a + b;</td>
</tr>
<tr>
<td>b = b + c;</td>
<td>b = b + c;</td>
<td>V(mutex);</td>
</tr>
<tr>
<td>V(mutex);</td>
<td>V(mutex);</td>
<td>V(mutex);</td>
</tr>
</tbody>
</table>
Classic Problems

There are a number of "classic" problems each of which represents a class of synchronization situations

• Critical Section problem
• Producer/Consumer problem
• Reader/Writer problem
• 5 Dining Philosophers

Producer / Consumer

Producer:
while(whatever)
{
    locally generate item
    fill empty buffer with item
}

Consumer:
while(whatever)
{
    get item from full buffer
    use item
}

Producer / Consumer
(with Counting Semaphores)

Producer:
while(whatever)
{
    locally generate item
    P(emptybuf);
    fill empty buffer with item
    V(fullbuf);
}

Consumer:
while(whatever)
{
    P(fullbuf);
    get item from full buffer
    V(emptybuf);
    use item
}

Semaphores: emptybuf initially N; fullbuf initially 0;
What does it mean that Semaphores have persistence?

Tweedledum and Tweedledee Problem

- Separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation.
- The `Sleep()` and `Wakeup()` routines operate as follows:
  - `Sleep` blocks the calling thread.
  - `Wakeup` unblocks a specific thread if that thread is blocked, otherwise its behavior is unpredictable.

The code shown above exhibits a well-known synchronization flaw. Outline a scenario in which this code would fail, and the outcome of that scenario.

```c
void Tweedledum()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee);
    }
}

void Tweedledee()
{
    while(1) {
        x = Quarrel(x);
        Wakeup(Tweedledum);
        Sleep();
    }
}
```

Lost Wakeup:
If dee goes first to sleep, the wakeup is lost (since dum isn’t sleeping yet). Both sleep forever.

Show how to fix the problem by replacing the `Sleep` and `Wakeup` calls with semaphore `P` (down) and `V` (up) operations.

```c
semaphore dum = 0;
semaphore dee = 0;

void Tweedledum()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledum);
        V(dee);
    }
}

void Tweedledee()
{
    while(1) {
        x = Quarrel(x);
        Wakeup(Tweedledum);
        Sleep();
        P(dum);
    }
}
```
Monitor Abstraction

- Encapsulates shared data and operations with mutual exclusive use of the object (an associated lock).
- Associated Condition Variables with operations of Wait and Signal.

Condition Variables

- We build the monitor abstraction out of a lock (for the mutual exclusion) and a set of associated condition variables.
- Wait on condition: releases lock held by caller, caller goes to sleep on condition’s queue. When awakened, it must reacquire lock.
- Signal condition: wakes up one waiting thread.
- Broadcast: wakes up all threads waiting on this condition.

Monitor Abstraction

EnQ(acquire (lock);
  if (head == null)
    head = item;
  else tail->next = item;
  tail = item;
  release(lock);)

deQ(acquire (lock);
  if (head == null)
    wait (lock, notEmpty);
  item = head;
  if (tail == head) tail = null;
  head = item->next;
  release(lock);)
Monitor Abstraction

EnQ: {acquire (lock);
if (head == null)
    {head = item;
    signal (lock, notEmpty);}
else tail->next = item;
tail = item;
release(lock);}

deQ: {acquire (lock);
if (head == null)
    wait (lock, notEmpty);
item = head;
if (tail == head) tail = null;
head=item->next;
release(lock);}
Monitor Abstraction

**EnQ**: (acquire (lock);
  if (head == null)
    (head = item;
    signal (lock, notEmpty);
  else tail->next = item;
  tail = item;
  release(lock);
)

**deQ**: (acquire (lock);
  while (head == null)
    wait (lock, notEmpty);
  item = head;
  if (tail == head) tail = null;
  head=item->next;
  release(lock);
)

The Critical Section Problem

while (1)
{
  ...other stuff...

  acquire (mutex);

  critical section  // conceptually "inside" monitor

  release (mutex);

  }
P&V using Locks & CV (Monitor)

P: {acquire (lock);
   while (Sval == 0)
      wait (lock, nonZero);
   Sval = Sval - 1;
   release(lock);}

V: {acquire (lock);
   Sval = Sval + 1;
   signal (lock, nonZero);
   release(lock);}

Nachos-style Synchronization

synch.h, cc
• Semaphores
  Semaphore::P
  Semaphore::V
• Locks and condition variables
  Lock::Acquire
  Lock::Release
  Condition::Wait (conditionLock)
  Condition::Signal (conditionLock)
  Condition::Broadcast (conditionLock)

Design Decisions / Issues
• Locking overhead (granularity)
• Broadcast vs. Signal
• Nested lock/condition variable problem
  LOCK a DO
  LOCK b DO
  while (not_ready) wait (b, c) //releases b not a
  END
  END
• My advice – correctness first!
Lock Granularity – how much should one lock protect?

Concurrency vs. overhead
Complexity threatens correctness

Using Condition Variables

while (!required_conditions) wait (m, c);

- Why we use “while” not “if” – invariant not guaranteed
- Why use broadcast vs. signal – can arise if we are using one condition queue for many reasons. Waking threads have to sort it out (spurious wakeups). Possibly better to separate into multiple conditions (but more complexity to code).