Outline for Today

- Objective:
  - To continue talking about the critical section problem and get more practice thinking about possible interleavings.
  - Start talking about synchronization primitives.
  - Introduce other "classic" concurrency problems
- Administrative details:
  - Look on the web for TAs' office hours or check newsgroup for UTAs' office hours for assignment 1.
  - On collecting problem sets…

Semaphores

- Well-known synchronization abstraction
- Defined as a non-negative integer with two atomic operations

  P(s) - [wait until s > 0; s--]   Reminder: notation [ ] = atomic

  V(s) - [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

Fill in the boxes

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

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;

Tweedledum and Tweedledee

- Separate threads executing their respective procedures. The idea 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

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

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

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.

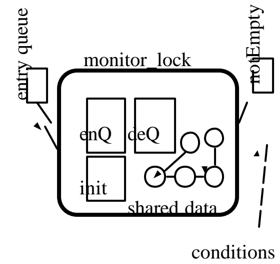
```
void Tweedledum()
{
  while(1) {
    Sleep();            P(dum);
    x = Quarrel(x);
    Wakeup(Tweedledee); V(dee);
  }
}

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

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


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.

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)**

Monitor Abstraction

```
EnQ:{Lock->Acquire ( );
     if (head == null)
          {head = item;
          notEmpty->Signal (Lock);}
     else tail->next = item;
     tail = item;
     Lock->Release( );}
deQ:{Lock->Acquire (lock);
     if (head == null)
          notEmpty->Wait (lock);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     Lock->Release( );}
```

entry queue
monitor_lock
notEmpty
enQ
deQ
init
shared data
conditions

Monitor Abstraction

```
EnQ:{Lock->Acquire ( );
     if (head == null)
          {head = item;
          notEmpty->Signal (Lock);}
     else tail->next = item;
     tail = item;
     Lock->Release( );}
deQ:{Lock->Acquire (lock);
     if (head == null)
          notEmpty->Wait (lock);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     Lock->Release( );}
```

entry queue
monitor_lock
notEmpty
enQ
deQ
init
shared data
conditions

Monitor Abstraction

```
EnQ:{Lock->Acquire ( );
     if (head == null)
          {head = item;
          notEmpty->Signal (Lock);}
     else tail->next = item;
     tail = item;
     Lock->Release( );}
deQ:{Lock->Acquire (lock);
     if (head == null)
          notEmpty->Wait (lock);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     Lock->Release( );}
```

entry queue
monitor_lock
notEmpty
enQ
deQ
init
shared data
conditions

Monitor Abstraction

```
EnQ:{Lock->Acquire ( );
     if (head == null)
          {head = item;
          notEmpty->Signal (Lock);}
     else tail->next = item;
     tail = item;
     Lock->Release( );}
deQ:{Lock->Acquire (lock);
     if (head == null)
          notEmpty->Wait (lock);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     Lock->Release( );}
```

entry queue
monitor_lock
notEmpty
enQ
deQ
init
shared data
conditions

Monitor Abstraction

```
EnQ:{Lock->Acquire ( );
    if (head == null)
        {head = item;
        notEmpty->Signal (Lock);}
    else tail->next = item;
    tail = item;
    Lock->Release( );}
deQ:{Lock->Acquire (lock);
    if (head == null)
        notEmpty->Wait (lock);
     item = head;
     if (tail == head) tail = null;
     head=item->next;
    Lock->Release( );}
```


Monitor Abstraction

```
EnQ:{Lock->Acquire ( );
    if (head == null)
        {head = item;
        notEmpty->Signal (Lock);}
    else tail->next = item;
    tail = item;
    Lock->Release( );}
deQ:{Lock->Acquire (lock);
    if (head == null)
        notEmpty->Wait (lock);
     item = head;
     if (tail == head) tail = null;
     head=item->next;
    Lock->Release( );}
```


Classic Problems

There are a number of “classic” problems that represent a class of synchronization situations

- ✓Critical Section problem
- ✓Producer/Consumer problem
- Reader/Writer problem
- 5 Dining Philosophers

5 Dining Philosophers


```
while(food available)
{pick up 2 adj. forks;
 eat;
 put down forks;
 think awhile;
}
```

Template for Philosopher

```
while (food available)
{  [          ]  /*pick up forks*/
   eat;
   [          ]  /*put down forks*/
   think awhile;
}
```


Naive Solution

```
while (food available)
{  P(fork[left(me)]);     /*pick up forks*/
   P(fork[right(me)]);
   eat;
   V(fork[left(me)]);     /*put down forks*/
   V(fork[right(me)]);
   think awhile;
}
```

Does this work?

Simplest Example of Deadlock

Thread 0	Interleaving	Thread 1
P(R1)	P(R1)	P(R2)
P(R2)	P(R2)	P(R1)
V(R1)	P(R1) waits	V(R2)
V(R2)	P(R2) waits	V(R1)

R1 and R2 initially 1 (binary semaphore)

Conditions for Deadlock

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and Thread 1 waits for Thread 0 to V(R1)
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are removed from their respective holding Threads.

Philosophy 101
(or why 5DP is interesting)

- How to eat with your Fellows without causing ***Deadlock.***
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why ***Starvation*** exists and what we can do about it.

Dealing with Deadlock

It can be ***prevented*** by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- circular waiting
  - Example: Define an ***ordering*** on resources and acquire them in order
- hold and wait
- no pre-emption

Circular Wait Condition

```
while (food available)
{   if (me == 0) {P(fork[left(me)]); P(fork[right(me)]);}
    else {(P(fork[right(me)]); P(fork[left(me)]); }
    eat;
    V(fork[left(me)]); V(fork[right(me)]);

    think awhile;
}
```


Hold and Wait Condition

```
while (food available)
{   P(mutex);
    while (forks [me] != 2)
        {blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);}
    forks [leftneighbor(me)] --;  forks [rightneighbor(me)]--;
    V(mutex):
    eat;
    P(mutex); forks [leftneighbor(me)] ++;  forks [rightneighbor(me)]++;
    if (blocking[leftneighbor(me)]) {blocking [leftneighbor(me)] = false;
    V(sleepy[leftneighbor(me)]); }
    if (blocking[rightneighbor(me)]) {blocking[rightneighbor(me)] = false;
    V(sleepy[rightneighbor(me)]); }    V(mutex);
    think awhile;
}
```