Today's class

- Will cover sections 6.5.3—6.7
- Terms and keywords
  - Deadlock, priority inversion, priority inheritance, bounded-buffer problem, readers-writers problem, dining philosophers problem, monitors
- Learning objectives
  - To understand the concept of deadlock and the programming techniques to avoid having one
  - To understand what monitors are and how they are used
But, first, term projects

- Class Term Projects
  - Class term projects can be either individual or group projects (groups of up to 4 people).
  - Effort of the project should be proportional to the group size.
  - Dates:
    - Proposals due next week's Thursday, October 28
      - A 2-page proposal with the group constitution, title and an extended abstract of what the project
    - Progress reports are due by Tuesday, November 18
    - Presentations should be held the after thanksgiving
      - A preliminary draft of the final report due with the presentation
    - Final reports due by last day of class, Tuesday, December 7
      - Term project's grade can replace or compete with final grade.
    - Can do project and not final (replace) or can do both (compete).
Term project topics

- Almost anything, from survey to research to programming
  - Survey projects: find what is out there about a technology that relates to OS, various implementations, pros and cons of each
  - Study projects: study a particular OS feature and perform some kind of evaluation
  - Research projects: hypothesis testing on some aspect of OS performance
  - Programming projects: develop a feature on an OS or a utility program
- Note: effort should be proportional to group size
Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let $s$ and $q$ be two semaphores initialized to 1

  \[
  P_0 \quad \quad \quad P_1
  \]

  \[
  \begin{align*}
  &\text{wait (S);} & \text{wait (Q);} \\
  &\text{wait (Q);} & \text{wait (S);} \\
  &\vdots & \vdots \\
  &\text{signal (S);} & \text{signal (Q);} \\
  &\text{signal (Q);} & \text{signal (S);}
  \end{align*}
  \]

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

- **Priority Inversion** - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
Priority Inversion

- **Priority inversion** may happen when processes of various priorities race over the same resource

- Example:
  - Process A: high priority
  - Process B: low priority
  - Process C: medium priority
  - “B” acquires lock over a resource
  - “A” tries to lock over the resource and waits
  - “C” dominates CPU because it is of higher priority
    - “A” waits as if it is of low priority.

- Solution: **priority inheritance**
  - A locking process inherits the priority of the highest-blocked process over the same resource, for the duration of the critical section.
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded-Buffer Problem

- $N$ buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value $N$. 
Bounded Buffer Problem (Cont.)

- The structure of the producer process

  ```
  do {
    // produce an item in nextp
    wait (empty);
    wait (mutex);

    // add the item to the buffer
    signal (mutex);
    signal (full);

  } while (TRUE);
  ```
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```c
    do {
        wait (full);
        wait (mutex);
        // remove an item from buffer to nextc
        signal (mutex);
        signal (empty);
        // consume the item in nextc
    } while (TRUE);
```
Readers-Writers Problem

• A data set is shared among a number of concurrent processes
  – Readers – only read the data set; they do not perform any updates
  – Writers – can both read and write

• Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time

• Shared Data
  – Data set
  – Semaphore mutex initialized to 1
  – Semaphore wrt initialized to 1
  – Integer readcount initialized to 0
Readers-Writers Problem (Cont.)

- The structure of a writer process

```c
do {
    wait (wrt) ;

    // writing is performed

    signal (wrt) ;
} while (TRUE);
```
Readers-Writers Problem (Cont.)

- The structure of a reader process

```c
do {
    wait (mutex);
    readcount ++;
    if (readcount == 1)
        wait (wrt);
    signal (mutex);

    // reading is performed

    wait (mutex);
    readcount --;
    if (readcount == 0)
        signal (wrt);
    signal (mutex);
} while (TRUE);
```
Dining-Philosophers Problem

- **Shared data**
  - Bowl of rice (data set)
  - Semaphore `chopstick [5]` initialized to 1
The structure of Philosopher $i$:

```c
    do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
        // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        // think
    } while (TRUE);
```
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }
    ...
    procedure Pn (...) {......}
    Initialization code ( ....) { ... }
    ...
}
```
Schematic view of a Monitor
Condition Variables

• condition x, y;

• Two operations on a condition variable:
  – x.wait() – a process that invokes the operation is suspended.
  – x.signal() – resumes one of processes (if any) that invoked x.wait()
Monitor with Condition Variables

- queues associated with x, y conditions
- shared data
- entry queue
- operations
- initialization code
Solution to Dining Philosophers

monitor DP
{
    enum { THINKING; HUNGRY, EATING } state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
void test (int i) {
    if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}

Solution to Dining Philosophers (cont)

- Each philosopher $I$ invokes the operations `pickup()` and `putdown()` in the following sequence:

  ```
  DiningPhilosophers.pickup (i);
  EAT
  DiningPhilosophers.putdown (i);
  ```
Monitor Implementation Using Semaphores

• Variables
  
  semaphore mutex;  // (initially = 1)
  semaphore next;   // (initially = 0)
  int next-count = 0;

• Each procedure $F$ will be replaced by

  wait(mutex);
  ...

  body of $F$;

  ...

  if (next_count > 0)
  signal(next)
  else
  signal(mutex);

• Mutual exclusion within a monitor is ensured.
Monitor Implementation

- For each condition variable \( x \), we have:

  ```
  semaphore x_sem; // (initially = 0)
  int x-count = 0;
  ```

- The operation \( x.wait \) can be implemented as:

  ```
  x-count++;
  if (next_count > 0)
      signal(next);
  else
      signal(mutex);
  wait(x_sem);
  x-count--;
  ```
Monitor Implementation

• The operation `x.signal` can be implemented as:

```java
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```
A Monitor to Allocate Single Resource

monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}