Lecture 11: Synchronization (Chapter 6, cont)
Semaphores (by Dijkstra 1930 – 2002)

- Born in Rotterdam, The Netherlands
- 1972 recipient of the ACM Turing Award
- Responsible for
  - The idea of building operating systems as explicitly synchronized sequential processes
  - The formal development of computer programs
- Best known for
  - His efficient shortest path algorithm
  - Having designed and coded the first Algol 60 compiler.
  - Famous campaign for the abolition of the \texttt{GOTO} statement
- Also known for his hand-written communications with friends and colleagues. For example: http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1205.PDF
Semaphores

- Synchronization tool that does not require busy waiting
- Semaphore $S$ – integer variable
- Two standard operations modify $S$: \texttt{wait()} and \texttt{signal()}
  - Originally called \texttt{P()} and \texttt{V()}
  - Also called \texttt{down()} and \texttt{up()}
- The value of $S$ can only be accessed through \texttt{wait()} and \texttt{signal()}

```c
wait (S) { while S <= 0 S++; // no-op S--; }
signal (S) { S++; }
```
Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue.

```c
typedef struct{
    int value;
    struct process *list;
} semaphore;
```

- Two operations on processes:
  - **block** – place the process invoking the operation on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.
Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of **wait**:
  ```c
  wait(semaphore *S) {
   S->value--; if (S->value < 0) {
      add this process to S->list;
      block();
   }
  }
  ```

- Implementation of **signal**:
  ```c
  signal(semaphore *S) {
   S->value++; if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
  }
  ```
Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Can implement a counting semaphore $S$ as a binary semaphore
- Provides mutual exclusion

```c
Semaphore mutex; // initialized to 1
do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
} while (TRUE);
```
One thread sends a signal to another thread to indicate that something has happened.
Group Work (2): rendezvous

Generalize the signal pattern so that it works both ways: Thread A has to wait for Thread B and vice versa.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
</tr>
</tbody>
</table>

we want to guarantee that a1 happens before b2 and b1 happens before a2
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

  \[ P_0 \]
  
  ```
  wait (S);
  wait (Q);
  wait (Q);
  wait (S);
  ... ... 
  signal (S);
  signal (Q);
  signal (Q);
  signal (S);
  ```

  \[ P_1 \]
  
  ```
  wait (Q);
  wait (S);
  ... ... 
  signal (Q);
  signal (S);
  ```

- **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
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 1
- Semaphore `full` initialized to 0
- Semaphore `empty` initialized to $N$. 
Bounded Buffer Solution

- **Producer:**

  ```
  do {
    // produce an item in nextp
    wait (empty);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (full);
  } while (TRUE);
  ```

- **Consumer:**

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

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

- Problem:
  - Allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time

- Shared Data
  - Data set
  - Semaphore $\text{mutex}$ initialized to 1
  - Semaphore $\text{wrt}$ initialized to 1
  - Integer $\text{readcount}$ initialized to 0
Readers-Writers Solution

- **Writer:**
  
  ```c
  do {
    wait (wrt) ;
    // writing is performed
    signal (wrt) ;
  } while (TRUE);
  ```

- **Reader:**
  
  ```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);
  ```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true) {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}
void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
```c
void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}

void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```
Dining Philosophers Problem

Figure 6.11  Dining Arrangement for Philosophers
The Problem

- Devise a ritual (algorithm) that will allow the philosophers to eat.
  - No two philosophers can use the same fork at the same time (mutual exclusion)
  - No philosopher must starve to death (avoid deadlock and starvation ... literally!)
/ * program diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
Avoiding deadlock (only 4 philosophers)

```c
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [ (i+1) mod 5 ]); 
        eat();
        signal (fork [ (i+1) mod 5 ]); 
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
                philosopher (3), philosopher (4));
}
```
Dining Philosophers: Solution

```
#define N 5        /* number of philosophers */
#define LEFT (i+N-1)%N /* number of i’s left neighbor */
#define RIGHT (i+1)%N /* number of i’s right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

void take_forks(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}