Process Synchronization

Chapter 6

Critical resource
- A limited (finite number or amount) commodity that a task needs to complete its task
- Shared data

Critical section
- Code segment that uses a critical resource
- Simultaneous access can result in a race condition
- Must be protected by a mutual exclusion protocol
- Must be small and fast

Multitasking Concepts

A task may be either a thread or a process

Race condition
- Multiple tasks concurrently access shared data
- The outcome depends on the order of access

Mutual exclusion
- Protocol limiting the # of tasks in a critical section
  - when no task is in a critical section, a requesting task is not delayed
  - when two tasks compete for a critical section, the selection cannot be postponed indefinitely
  - no task can prevent another from entering its critical section indefinitely

Starvation
- A task is never allowed access to a critical resource; it is never allowed to run to completion

Deadlock
- 2 or more tasks each have a mutual exclusion lock on a resource needed by another

Atomic test and set
- an “uninterleavable” instruction that tests and sets the value of a variable; used for mutual exclusion

Fairness
- How contention for a critical resource is resolved
  - weak fairness: if a task continuously makes a request, it is eventually satisfied
  - strong fairness: if a task makes a request infinitely often, it is eventually satisfied
  - linear waiting: if a task makes a request, it is granted before any other task is granted the same request more than once
  - FIFO waiting: if a task makes a request, it will be granted before that of a task making a later request

The Critical Section Problem

Race condition example
**Atomic Instructions**

Two views of TestAndSet

boolean locked = false;
while (true)
{
   if (locked)
      locked = true;
   else
      wait();
   // critical section
   locked = false;
}

executed atomically

boolean locked = false;
while (true)
{
   while (TestAndSet(&locked))
      // spin
   // critical section
   lock = false;
}

**Swap**

boolean locked = false;
while (true)
{
   boolean key = true;
   while (key)
      Swap(&lock, &key);
   // critical section
   lock = false;
}

**Competing Tasks**

Mutual exclusion example

mutual exclusion

atomic test and set instruction

if (inUse == false)
   inUse = true
else
   block

inUse = false

POS task

**Semaphores**

Common mutual exclusion device

- Sometimes called PV Semaphores
  - From the Dutch proberen and verhogen
- Have a value called the semaphore value (s)
  - Binary semaphores: 0 or 1
  - Counting semaphores: 0, 1, 2, ..., n
- wait(s)
  - s > 0 decrement s and continue
  - s == 0 wait (i.e., suspend)
- signal(s) increment s and continue

**Semaphore Example**

Inventory / point-of-sale problem

```java
int widgets; // critical resource
void update(int n)
{
   wait(s);
   widgets += n;
   signal(s);
}
void sell()
{
   wait(s);
   widgets -= 1;
   signal(s);
}
```

**Semaphore Example**

Bounded Buffer Problem

```java
Semaphore mutex = 1;
Semaphore spaces = BUF_SIZE;
Semaphore elements = 0;

producer()
{
   wait(spaces);
   wait(mutex);
   buffer[front] = prod;
   front = (front + 1) % BUF_SIZE;
   signal(mutex);
   signal(elements);
}

consumer()
{
   wait(elements);
   wait(mutex);
   consume = buffer[back];
   back = (back + 1) % BUF_SIZE;
   signal(mutex);
   signal(spaces);
}
```
The Problem With Semaphores

All tasks must respect the protocol

- The responsibility for correctness is distributed over all tasks
  - Will fail if even a single task does not follow the protocol
    - Honest programming error
    - Uncooperative programmer
  - Interchanges the order of wait and signal
  - Replaces signal with wait
  - Omits wait, signal, or both

Monitors

High-level synchronization (C.A.R. Hoare)

- A data type that encapsulates private, shared data with public access functions/methods
  - Centralizes responsibility for correctness
  - Tasks request services from the monitor
  - Critical resources are represented by attributes
  - Operations restrict and serialize access

- Correctness does not rely on individual tasks correctly sequencing semaphore operations

Monitor Example

Implementing a monitor in Java

```java
public class Inventory
{
  private int widgets;

  synchronized public void update(int n)
  {
    widgets += n;
  }

  synchronized public void sell()
  {
    widgets -= 1;
  }
}
```

Condition Variables

Extends a monitor’s synchronization ability

- Are attached to a monitor
- Supports two or three operations
  - wait release the monitor lock and block
  - signal resume a task blocked (waiting) on this condition
  - non_empty boolean valued: true if the waiting queue is not empty

- Encapsulates a semaphore and a counter

Monitor with Condition Variables

Multiple queues

```cpp
class Buffer : public Monitor
{
  private:
  int front; // front of Buffer
  int back; // back or end of Buffer
  int counter; // how many ints in Buffer
  int buffer[5]; // circular buffer
  Condition* spaces;
  Condition* elements;

  public:
  Buffer();
  void insert(int data);
  int remove();
};
```

C++ / Bounded Buffer

See posted example for complete syntax
C++ / Bounded Buffer

void Buffer::insert(int data)
{
   Lock( );
   while (counter == BUF_SIZE)
      Wait(spaces);
   buffer[front] = data;
   front = (front + 1) % BUF_SIZE;
   counter++;
   Signal(elements);
}

int Buffer::remove()
{
   Lock( );
   while (counter == 0)
      Wait(elements);
   int  removed = buffer[back];
   back = (back + 1) % BUF_SIZE;
   counter--;
   Signal(spaces);
   return removed;
}

Java Monitor / Bounded Buffer

public class Buffer
{
   int front = 0;
   int back = 0;
   int counter = 0;
   int[] buffer = new int[BUF_SIZE];
   synchronized public void insert(int stuff)
   {
      while (counter == BUF_SIZE)
         wait( );
      buffer[front] = stuff;
      front = (front + 1) % BUF_SIZE;
      counter++;
      notifyAll( );
   }

   synchronized public int remove()
   {
      while (counter == 0)
         wait( );
      int temp = buffer[back];
      back = (back + 1) % BUF_SIZE;
      counter--;
      notifyAll( );
      return temp;
   }
}

Monitors

An alternate view of Java monitors

Emulating Monitors w/ Semaphores

Condition variables

void Condition::wait()
{
   count++;
   ReleaseSemaphore(*monitor_sem, 1, 0);
   WaitForSingleObject(semaphore, INFINITE);
   count--;
}

void Condition::signal()
{
   if (count > 0)
      ReleaseSemaphore(semaphore, 1, 0);
   else
      ReleaseSemaphore(*monitor_sem, 1, 0);
}

class Condition
{
   private:
      int count;
      HANDLE semaphore;
      HANDLE* monitor_sem;
   public:
      Condition(HANDLE* sem);
      void wait();
      void signal();
}

Emulating Monitors w/ Semaphores

Continued

class Monitor
{
   private:
      HANDLE mutex;
   public:
      void Lock();
      void Wait(Condition* c);
      void Signal(Condition* c);
      Condition* getCondition();
};

void Monitor::Lock()
{
   WaitForSingleObject(mutex, INFINITE);
}

void Monitor::Wait(Condition* c)
{
   c->wait();
}

void Monitor::Signal(Condition* c)
{
   c->signal();
}

Condition* Monitor::getCondition()
{
   return new Condition(&mutex);
}

Dining Philosophers

Another classic problem

- Philosophers
  - Think
  - Eat
- Eating requires 2 chop sticks
  - get left chop stick
  - get right chop stick
  - eat
  - release right chop stick
  - release left chop stick

Shared bowl of rice (data). Overlapping set of chop sticks (binary semaphores).