Multithreading

A *thread* is a unit of control (stream of instructions) within a process.

When a thread runs, it executes a function in the program.

The process associated with a running program starts with one running thread, called the “main thread”, which executes the “main” function of the program.

In a multithreaded program, the main thread creates other threads, which execute other functions. These other threads can create even more threads, and so on.

Each thread has its own stack of activation records and its own copy of the CPU registers, including the stack pointer and the program counter, which together describe the state of the thread’s execution. T

The threads in a multithreaded process share the data, code, resources, and address space of their process.

Per-process state information is also shared by the threads in the program, which greatly reduces the overhead of creating and managing threads.

Advantages of Multithreading

Multithreading allows a process to overlap IO and computation.

One thread can execute while another thread is waiting for an IO operation to complete. Multithreading makes a GUI (Graphical User Interface) more responsive. The thread that handles GUI events, such as mouse clicks and button presses, can create additional threads to perform long running tasks in response to the events. This allows the event handler thread to respond to more GUI events.

Multithreading can speedup performance through parallelism/multicore.

Multithreading has some advantages over multiple processes. Threads require less overhead to manage than processes, and intra-process thread communication is less expensive than inter-process communication.

Multi-process concurrent programs do have one advantage in that each process can execute on a different machine. (Web browser – Web server.)

The main disadvantage of concurrent programs is that they are extremely difficult to develop. Concurrent programs often contain bugs that are notoriously hard to find and fix.

Threads in Java

Java provides a *Thread* class for defining user threads. One way to define a thread is to define a class that extends (i.e. inherits from) the *Thread* class.

```
class simpleThread extends Thread {
   public simpleThread(int ID) {myID = ID;}
   public void run() {System.out.println("Thread " + myID + " is running.");}
   private int myID;
}
public class javaConcurrentProgram {
   public static void main(String[] args) {
      simpleThread thread1 = new simpleThread(1);
      simpleThread thread2 = new simpleThread(2);
            thread1.start(); thread2.start(); // causes the run() methods to execute
   }
}


class simpleRunnable implements Runnable {
   public simpleRunnable(int ID) {myID = ID;}
   public void run() { System.out.println("Thread " + myID + " is running.");    }
   private int myID;
}
public class javaConcurrentProgram2 {
   public static void main(String[] args) {
         Runnable r = new simpleRunnable(3);
         Thread thread3 = new Thread(r); // thread3 executed r's run() method
         thread3.start();
   }
}
```

Multithreading in Win32

```
#include <iostream>
#include <windows.h>
#include <process.h>

unsigned WINAPI simpleThread (LPVOID myID)
   std::cout << "Thread " << (unsigned) myID << " is running" << std::endl;
   return (unsigned) myID;
}
int main() {
   const int numThreads = 2;
   HANDLE threadArray[numThreads]; // array of thread handles
   unsigned threadID;
   DWORD rc;

   // Create two threads and store their handles in array threadArray
   threadArray[0] = (HANDLE) _beginthreadex(NULL, 0, simpleThread,
               (LPVOID) 1U, 0, &threadID);
   threadArray[1] = (HANDLE) _beginthreadex(NULL, 0, simpleThread,
               (LPVOID) 2U, 0, &threadID);

   rc = WaitForMultipleObjects(numThreads,threadArray,TRUE,INFINITE);
   //wait for the threads to finish

   DWORD result1, result2;// these variables will receive the return values
   rc = GetExitCodeThread(threadArray[0],&result1);
   rc = GetExitCodeThread(threadArray[1],&result2);
   std::cout << "thread1:" << result1 << " thread2:" << result2 << std::endl; // display results
   rc = CloseHandle(threadArray[0]);  // release reference to the thread
   rc = CloseHandle(threadArray[1]);
   return 0;
}
```

Multithreading in PThreads

```
#include <iostream>
#include <pthread.h>
#include <errno.h>

void* simpleThread (void* myID) { // myID is the fourth argument of pthread_create ()
   std::cout << "Thread " << (long) myID << " is running" << std::endl;
   return NULL;      // implicit call to pthread_exit(NULL);
}

int main() {
   pthread_t threadArray[2];              // array of thread IDs
   int status;
   pthread_attr_t threadAttribute;
   status = pthread_attr_init(&threadAttribute);      // initialize the attribute object
   // set the scheduling scope attribute
   status = pthread_attr_setscope(&threadAttribute, PTHREAD_SCOPE_SYSTEM);
   // Create two threads and store their IDs in array threadArray
   status = pthread_create(&threadArray[0], &threadAttribute, simpleThread, (void*) 1L);
   status = pthread_create(&threadArray[1], &threadAttribute, simpleThread, (void*) 2L);
   status = pthread_attr_destroy(&threadAttribute);
   status = pthread_join(threadArray[0],NULL); // wait for threads to finish
   status = pthread_join(threadArray[1],NULL);
}
```

Use a Thread/Runnable class – write your own!

```
class simpleRunnable: public Runnable {
public:
   simpleRunnable(int ID) : myID(ID) {}
   virtual void* run() {
      std::cout << "Thread " << myID << " is running" << std::endl;
            return reinterpret_cast<void*>(myID);
   }
private:
   int myID;
};
   class simpleThread: public Thread {
public:
   simpleThread (int ID) : myID(ID) {}
   virtual void* run() {
      std::cout << "Thread " << myID << " is running" << std::endl;
            return reinterpret_cast<void*>(myID);
   }
private:
   int myID;
};
int main() {
   std::auto_ptr<Runnable> r(new simpleRunnable(1));
   std::auto_ptr<Thread> thread1(new Thread(r));
   thread1->start();
   std::auto_ptr<simpleThread> thread2(new simpleThread(2));
   thread2->start();
   simpleThread thread3(3);
   thread3.start();
   // thread1 and thread2 are created on the heap; thread3 is created on the stack
   int result1 = reinterpret_cast<int>(thread1->join());    // wait for the threads to finish
   int result2 = reinterpret_cast<int>(thread2->join());
   int result3 = reinterpret_cast<int>(thread3.join());
   std::cout << result1 << ' ' << result2 << ' ' << result3 << std::endl;
   return 0;
   // destructors for thread1 and thread2 will automatically delete pointed-to thread objects
}
```

Thread Communication

```
int s=0; // shared variable s

class communicatingThread: public Thread {
public:
   communicatingThread (int ID) : myID(ID) {}
   virtual void* run();
private:
   int myID;
};
void* communicatingThread::run() {
   std::cout << "Thread " << myID << " is running" << std::endl;
   for (int i=0; i<10000000; i++)         // increment s ten million times
      s = s + 1;
   return 0;
}

int main() {
   std::auto_ptr<communicatingThread> thread1(new communicatingThread (1));
   std::auto_ptr<communicatingThread> thread2(new communicatingThread (2));
   thread1->start();   thread2->start();
   thread1->join();    thread2->join();
   std::cout << "s: " << s << std::endl;
   return 0;
}
```

The expected final value of *s* is 20000000.

What will you see when you run this program?

The execution of a concurrent program is non-deterministic – two executions of the *same* program with the *same* input can produce *different* results.

Uh-Oh!

Example 1. Assume that integer *x* is initially 0.

Thread1	Thread2	Thread3
(1) x = 1;	(2) x = 2;	(3) y = x;

The final value of *y* is unpredictable, but it is expected to be either 0, 1 or 2. Below are some of the possible interleavings of these three statements.

(3), (1), (2) ⇒ final value of *y* is 0
(2), (1), (3) ⇒ final value of *y* is 1
(1), (2), (3) ⇒ final value of *y* is 2

Example 2. Assume that *y* and *z* are initially 0.

Thread1	Thread2
x = y + z;	y = 1;
z = 2;	

If we (incorrectly) assume that that the execution of each assignment statement is an atomic action, the expected final value of *x* computed by *Thread1* is 0, 1, or 3, representing the sums 0+0, 1+0, and 1+2, respectively.

However, the machine instructions for *Thread1* and *Thread2* will look something like the following:

Thread1	Thread2
(1) load r1, y	(4) assign y, 1
(2) add r1, z	(5) assign z, 2
(3) store r1, x	

Below are some of the possible interleavings of these machine instructions. The character "*" indicates an unexpected result:

(1), (2), (3), (4), (5) ⇒ x is 0
(4), (1), (2), (3), (5) ⇒ x is 1
(1), (4), (5), (2), (3) ⇒ x is 2 *
(4), (5), (1), (2), (3) ⇒ x is 3

Example 3. Assume that the initial value of *x* is 0.

Thread1	Thread2
x = x + 1;	x = 2;

Again we are incorrectly assuming that the execution of each assignment statement is an atomic action, so the expected final value of *x* is 2 or 3. The machine instructions for *Thread1* and *Thread2* are:

Thread1	Thread2
(1) load r1, x	(4) assign x, 2
(2) add r1, 1	
(3) store r1, x	

Below are some of the possible interleavings of these machine instructions. Once again, the character "*" indicates an unexpected result.

(1), (2), (3), (4) => x is 2
(4), (1), (2), (3) => x is 3
(1), (2), (4), (3) => x is 1 *

Example 5**.** Variable *first* points to the first *Node* in the list. Assume that the list is not empty

```
class Node {
public:
   valueType value;
   Node* next;
}
Node* first;                    // first points to the first Node in the list;
void deposit(valueType value) {
   Node* p = new Node;          // (1)
   p->value = value;            // (2)
   p->next = first;             // (3)
   first = p;                   // (4) insert the new Node at the front of the list
}
valueType withdraw() {
   valueType value = first->value;  // (5) withdraw the first value in the list
   first = first->next;             // (6) remove the first Node from the list
   return value;                    // (7) return the withdrawn value
}
```

If two threads try to *deposit* and *withdraw* a value at the same time, the following interleaving of statements is possible:

```
valueType value = first->value;     // (5) in withdraw
Node* p = new Node();               // (1) in deposit
p->value = value                    // (2) in deposit
p->next = first;                    // (3) in deposit
first = p;                          // (4) in deposit
first = first->next;                // (6) in withdraw
return value;                       // (7) in withdraw
```

At the end of this sequence, the withdrawn item is still pointed to by *first* and the deposited item has been lost. To fix this problem, each of methods *deposit* and *withdraw* must be implemented as an atomic action.

If there are *n* threads (*Thread*$_1$, *Thread*$_2$, …, *Thread*$_n$) such that *Thread*$_i$ executes *mi* atomic actions, then the number of possible interleavings of the atomic actions is:

$$\frac{(m1 + m2 + \ldots + mn)!}{(m1! * m2! * \ldots * mn!)}$$

Uh-Oh!