ECE 462 Exam 3

11:30AM-12:20PM, November 17, 2010

I will not receive nor provide aid to any other student for this exam.

Signature:

You must sign here. Otherwise, the exam is not graded.

This exam is printed **double sides**.

Write your answers next to the questions. If you need more space, you can use the two blank pages.

This is an *open-book, open-note* exam. You can use any book or note or program printouts.

Please turn off your cellular phone **now** .

Two outcomes are tested in this exam. To pass each outcome, you must receive 50% of the points.

- Outcome 7: exception handlin
- Outcome 8: multiple threads and synchronization among threads

You need to obtain 50% of the points in each outcome to pass the outcome. If a program has a syntax error, **point out** which line causes the error.

Contents

Passed Outcomes: 7 8

Total Score:

1 Java Exception (outcome 7, 3 points)

Fill in the code.

Answer:

```
class Exception1 extends Exception
{
    public Exception1 (String m, int v)
    {
        message = m;
        value = v;
    }
    public String toString()
    {
        String str = "message: " + message +
            ", value: " + value;
        return str;
    }
    private String message;
    private int value;
}

class Exception2 extends Exception1
{
    public Exception2 (String m, int v, String n)
    {
        super(m, v);
        name = n;
    }
    public String toString()
    {
        String str1 = super.toString();
        String str2 = str1 + ", name: " + name;
        return str2;
    }
    private String name;
}

class TroubleMaker
{
    public void f1() throws Exception1
    {
        throw new Exception1("ece", 462);
    }
    public void f2() throws Exception2
```

```
    {
        throw new Exception2("ece", 462, "java");
    }
}

class outcome71
{
    public static void main(String[] args)
    {
        TroubleMaker tm = new TroubleMaker();
        try {
            tm.f1();
            tm.f2();
        } catch (Exception2 e2) {
            System.out.println("caught Exception2");
            System.out.println(e2);
        } catch (Exception1 e1) {
            System.out.println("caught Exception1");
            System.out.println(e1);
        } finally {
            System.out.println("reach finally");
        }
    }
}

// <--- FIX ME
// create a class called Exception1 as a derived class of Exception



{
    public Exception1 (String m, int v)
    {
        message = m;
        value = v;
    }
    public String toString()
    {
        String str = "message: " + message + ", value: " + value;
        return str;
    }
    private String message;
    private int value;
}
// <--- FIX ME
// create a class called Exception2 as a derived class of Exception1
```

```
{
    public Exception2 (String m, int v, String n)
    {
        super(m, v);
        name = n;
    }
    public String toString()
    {
        String str1 = super.toString();
        String str2 = str1 + ", name: " + name;
        return str2;
    }
    private String name;
}
class TroubleMaker
{
    // <--- FIX ME
    // function f1, throws Exception1("ece", 462);









    // <--- FIX ME
    // function f2, throws Exception2("ece", 462, "java");




}
class outcome71
{
    public static void main(String[] args)
    {
        TroubleMaker tm = new TroubleMaker();
        try {
            tm.f1();
            tm.f2();
            // <--- FIX ME
            // write two catch blocks to catch Exception1 and
```

```
            // Exception2. One and only one catch block is executed.
            // Inside each block, print the caught exception object
            // using System.out.println
            // <--- WARNING: the order of the two catch blocks is
            // important









        } finally {
            System.out.println("reach finally");
        }
    }
}
```

2 C++ Exception (outcome 7, 2 points)

What is the output of this program? Please be careful about the **order** of the output.

Answer:

E C B D caught Type 2

```
#include <iostream>
#include <string>
using namespace std;
class ExceptionType1
{
};

class ExceptionType2: public ExceptionType1
{
};

void f (int i) throw (ExceptionType1, ExceptionType2)
{
  switch (i)
    {
    case 1:
      cout << "A" << endl;
      throw ExceptionType1();
      break;
    case 2:
      cout << "B" << endl;
      throw ExceptionType2();
      break;
    default:
      cout << "no exception" << endl;
    }
}

void g (int i) throw (ExceptionType1, ExceptionType2)
{
  try
    {
      cout << "C" << endl;
      f(i);
    }
  catch (ExceptionType1 et1)
    {
      cout << "D" << endl;
```

```
      throw ExceptionType2();
    }
}

int main()
{
  try
    {
      cout << "E" << endl;
      g(2);
      cout << "F" << endl;
      g(1);
      cout << "G" << endl;
      g(0);
    }
  catch (ExceptionType2 et2 )
    {
      cout << "caught Type 2 " << endl;
    }
  catch (ExceptionType1 et1 )
    {
      cout << "caught Type 1 " << endl;
    }
  return 0;
}
```

3 Exception Handling in C++ and Java (outcome 7, 1 point)

Choose the correct statement (or statements).

Answer:

C, E

A. In C++, each `try` has one and only one correspoding `catch`.

B. In Java, each `try` may have several `catch` blocks. After the last `catch`, an `always` block can be added and the code inside is always executed.

C. In C++, a function may be called inside a `try` block even if the function does not throw any exceptions.

D. In Java, an exception is thrown if a function returns -1.

E. In Java, a `catch` block can throw an exception.

4 Amdahl's Law (outcome 8, 1 point)

Briefly explain Amdahl's Law and give an example.

Answer:

Amdahl's Law says that improving performance by parallelization will receive diminishing return, limited by the part that cannot be parallelized.

If 10% of a program cannot be parallelized, the program's performance can be improved by at most 10 times, even if many processors are used.

5 C++ Mutex Locks (outcome 8, 3 points)

Add appropriate code so that `balance` in `Account` can be written or read by only one thread at any moment. Moreover, the balance is never negative. In other words, a `Withdrawer` thread cannot proceed if the balance is too low.

Answer:

```
class Account
{
private:
  QMutex mutex;
  QWaitCondition cond;
  int balance;
public:
  Account() { balance = 0; }
  void deposit( int dep )
  {
    mutex.lock();
    balance += dep;
    cond.wakeAll();
    mutex.unlock();
  }
  void withdraw( int draw )
  {
    mutex.lock();
    while ( balance < draw )
      {
        cond.wait( & mutex );
      }
    balance -= draw;
    mutex.unlock();
  }
  void getBalance()
  {
    mutex.lock();
    cout << "balance after deposits: " << balance << endl;
    mutex.unlock();
  }
};
```

```
#include <QtCore>
#include <cstdlib>
#include <iostream>
using namespace std;
```

```
class Account
{
private:
  // <--- FIX ME
  // add required attributes




  int balance;
public:
  Account() { balance = 0; }
  void deposit( int dep )
  {
    // <--- FIX ME
    // protect the balance so that only one thread can modify the balance



    balance += dep;




  }
  void withdraw( int draw )
  {
    // <--- FIX ME
    // protect the balance so that only one thread can modify the
    // balance. the balance should never become negative.  If the
    // balance is smaller than draw, wait until someone deposits
    // enough money



    balance -= draw;
```

```
  }

  void getBalance()
  {
    // <--- FIX ME
    // protect the balance so that it cannot be read if another thread
    // is modifying the balance



    cout << "balance after deposits: " << balance << endl;




  }
};

// You do not need to modify anything below.

class Depositor : public QThread
{
private:
  Account * act;
public:
  Depositor (Account * a) { act = a; }
  void run()
  {
    int i = 0;
    while ( true )
      {
        int x = (int) ( rand() % 10 );
        act -> deposit( x );
        if ( i++ % 100 == 0 )
        {
          act -> getBalance();
        }
      }
  }
};

class Withdrawer : public QThread
{
private:
  Account * act;
```

```
public:
  Withdrawer(Account * a) { act = a; }
  void run()
  {
    int i = 0;
    while ( true )
      {
        int x = (int) ( rand() % 100 );
        act -> withdraw( x );
        if ( i++ % 100 == 0 )
          {
            act -> getBalance();
          }
      }
  }
};

int main()
{
  Account act;
  Depositor* depositors[5];
  Withdrawer* withdrawers[5];

  for ( int i=0; i < 5; i++ )
    {
      depositors[ i ] = new Depositor(& act);
      withdrawers[ i ] = new Withdrawer(& act);
      depositors[ i ]->start();
      withdrawers[ i ]->start();
    }
  for ( int i=0; i < 5; i++ )
    {
      depositors[ i ]->wait();
      withdrawers[ i ]->wait();
    }
}
```

6 Java Producer - Consumer (outcome 8, 2 points)

In the following producer - consumer program, which functions **must** be `synchronized` so that (1) every produced item is consumed exactly once, (2) the elements are removed in the order as they are added (first-in-first-out), and (3) only produced items are consumed. The shared buffer does neither overflow nor underflow.

Mark only the methods that **must** be `synchronized`. You will lose 1 point for each method that does not have to be `synchronized`.

Answer:

SharedBuffer.put and SharedBuffer.get

```
class SharedBuffer
{
    private int sb_head;
    private int sb_tail;
    private int sb_numElem; // number of valid elements
    private int [] sb_element; // circular buffer
    public SharedBuffer(int size)
    {
        sb_element = new int [size];
        sb_head = 0;
        sb_tail = 0;
        sb_numElem = 0;
    }
    public void put(int v)
    {
        while (sb_numElem >= sb_element.length)
            {
                try
                    { wait(); }
                catch (InterruptedException e )
                    { System.out.println("caught exception in put"); }
            }
        sb_numElem ++;
        sb_element [sb_tail] = v;
        sb_tail = (sb_tail + 1) % sb_element.length;
        notifyAll();
    }
    public int get()
    {
        while (sb_numElem < 0)
            {
                try
```

```
                    { wait(); }
                catch (InterruptedException e )
                    { System.out.println("caught exception in get"); }
            }
        int v = sb_element [sb_head];
        sb_numElem --;
        sb_head = (sb_head + 1) % sb_element.length;
        notifyAll();
        return v;
    }
}

class Producer extends Thread
{
    private SharedBuffer p_sbuf;
    private final int p_numItem = 300;
    public Producer(SharedBuffer buf)
    { p_sbuf = buf; }
    public void run()
    {
        for (int iter = 0; iter < p_numItem; iter ++)
            { p_sbuf.put(iter); }
    }
}

class Consumer extends Thread
{
    private SharedBuffer c_sbuf;
    private final int c_numItem = 300;
    public Consumer(SharedBuffer buf)
    { c_sbuf = buf; }
    public void run()
    {
        int val;
        for (int iter = 0; iter < c_numItem; iter ++)
            { val = c_sbuf.get(); System.out.println(val); }
    }
}

public class outcome82 {
    public static void main (String[] args ) {
        // create a shared buffer
        int bufSize = 20;
        SharedBuffer sbuf = new SharedBuffer(bufSize);
```

```
        // create threads
        int numThread = 4;
        Producer [] prod = new Producer [numThread];
        Consumer [] cons = new Consumer [numThread];
        for (int index = 0; index < numThread; index ++)
            {
                prod[index] = new Producer (sbuf);
                cons[index] = new Consumer (sbuf);
            }

        // start producing and consuming
        for (int index = 0; index < numThread; index ++)
            {
                prod[index].start();
                cons[index].start();
            }

        // wait until all threads complete
        try
            {
                for (int index = 0; index < numThread; index ++)
                    {
                        prod[index].join();
                        cons[index].join();
                    }
            }
        catch (Exception je)
            { System.out.println("join exception " + je); }

    }
}
```

7 Deadlock (3 points)

List the necessary conditions for a deadlock (1 point)

Is it **possible** that the following program has a deadlock? (1 point)

Explain the reason. (1 point)

Answer:

This program may have deadlock because 1. There are two threads and two objects. Each thread owns the key of one object (mutual exclusion). 2. Each thread wants to acquire the key of the other object, while inside synchronized methods (hold and wait). 3. The threads cannot release the keys since they are inside synchronized methods (hold and wait). 4. Each thread is waiting for the other thread to release the keys (circular wait).

```
class Account {
    public Account() {
    }
    synchronized void transfer (Account a2) {
        // waste some time to increase the chance of interleaving
        for (int i = 0; i < 10000; i ++) { }
        a2.balance();
    }
    synchronized void balance() {
        for (int i = 0; i < 10000; i ++) { }
    }
}

class TransferThread extends Thread {
    Account tt_act1;
    Account tt_act2;
    public TransferThread(Account act1, Account act2) {
        tt_act1 = act1;
        tt_act2 = act2;
    }
    public void run() {
        tt_act1.transfer(tt_act2);
    }
}

public class deadlock {
    public static void main( String[] args ) {
        Account act1 = new Account();
        Account act2 = new Account();
        TransferThread tt1 = new TransferThread(act1, act2);
```

```
        TransferThread tt2 = new TransferThread(act2, act1);
        tt1.start();
        tt2.start();
        try {
            tt1.join();
            tt2.join();
        } catch (Exception je) {
            System.out.println("join exception " + je);
        }
        System.out.println("all transactions completed");
    }
}
```