# CS 315 LAB #2: Resource contention

## NAME: Elisha Parslow

**GRADE:**

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| EX02\_01: hardware concurrency |  | 25 |
| EX02\_02: Mutex and condition variables |  | 50 |
| **TOTAL** |  | 75 |

## Learning Goals for this Exercise

* Become familiar with the reference website http://en.cppreference.com
* Practice using C++11 mutex and condition variables

**Save the answers to questions in this document in** [**\\CS1\home \<yourname>\CS315-1\Lab02\**](file:///\\CS1\home%20\%3cyourname%3e\CS315-1\Lab02\)

### EX02\_01

Sometimes it is useful for a program to know how many actual CPU cores a computer has. This information is useful for guiding us on how many threads can truly run concurrently at the same time on a given computer. Visit <http://en.cppreference.com/w/cpp/thread/thread> and look for the static function that allows you to query the number of concurrent threads supported by the computer that is running your C++11 program. Read the description of this function.

Create a visual studio Win32 console project in [\\CS1\home\<yourname>\CS315-1\EX02\_01\](file:///\\CS1\home\%3cyourname%3e\CS315-1\EX02_01\) and demonstrate the use of this static function in a simple program of your choice.

### EX02\_02 (please answer the 5 questions in this exercise)

Create a visual studio **Win32 console** project in [\\CS1\home\<yourname>\CS315-1\EX02\_02\](file:///\\CS1\home\%3cyourname%3e\CS315-1\EX02_02\) Copy the program in **Table 1** into a C++ file in your new project.

The program is simple. It defines a class called **BankAccount**, which you can use to *deposit* to and *withdraw* money from. The **BankAccount** *constructor* allows you to initialize the amount of money you have in the account when you first create it.

In the main function, we create one **BankAccount** object, with the initial amount of $0. We then spawn 1000 threads (using PPL). Half of the threads will withdraw $20, and the other half will deposit $20.

The last **cout** statement in the main function prints out the final balance in the account.

The **correct final balance for this program should be “0”,** since 500 threads will be depositing $20, and 500 threads will be withdrawing $20.

Now, compile and run the program. Run it at least 5 times.

**Question 1**: What are the values printed out by the last **cout** statement from 5 runs of the program?

**Answer:** $0, $0, $80, $0, $80

**Question 2**: Why are we not getting the value “0” as the final balance?

**Answer:** The dollar variable is being shared with all of the threads so some of results are being affected from the treads that complete their required actions sooner than others.

Table 1

|  |
| --- |
| #include <iostream>  #include <thread>  #include <ppl.h>  using namespace std;  using namespace concurrency;  #define AMOUNT 20  class BankAccount {  private:  int amount;  public:  BankAccount(int amount) : amount(amount) {}  /\*\*  \* Deposits money into account  \*/  void Deposit(int dollar) {  amount += dollar;  }  /\*\*  \* Withdraws money from account  \* If account has less money than the amount requested,  \* it does not perform the withdrawal.  \*/  bool Withdraw(int dollar) {  if (amount >= dollar) {  amount -= dollar;  return true;  }  else {  return false;  }  }  int Amount() {  return amount;  }  };  int main()  {  BankAccount myAccount(0); // Open an account with an initial balance of $0  parallel\_for(0, 1000,[&](int tid) {  if (tid % 2 == 0) { // every other thread will withdraw  myAccount.Withdraw(AMOUNT); // Withdraw some amount  }  else {  myAccount.Deposit(AMOUNT); // Deposit some amount  }  });  cout << "My account has $" << myAccount.Amount() << endl;  } |

Now let’s define a **critical section** in the **lambda function** to allow the calls to **Withdraw()** and **Deposit()** be mutually exclusive, i.e. only ONE thread at a time gets to invoke either **Withdraw()** or **Deposit().**

**Step 1:** include <mutex>

**Step 2:** define a **member object in BankAccount of type** mutex, e.g.

mutex m;

**Step 3**: **in each of the member functions, Withdraw and Deposit**, define a **scoped lock** using the **unique\_lock** type, and pass the **mutex** object m, that you just defined as a member variable to the scoped lock’s constructor, e.g.

unique\_lock<mutex> lk(m);

**Question 3**: What methods of the **mutex m** are automatically called in the **constructor** and **destructor** for **unique\_lock** object **lk**?

**Answer**: When the unique lock is reached in the process, the critical code is accessed one at a time by each thread, and the lock deconstructs after each thread finishes the code to allow another thread in.

Look at the constructor and destructor descriptions in the reference page for **std::unique\_lock**: <http://en.cppreference.com/w/cpp/thread/unique_lock>

Now. compile and run the program. Run it at least 5 times.

**Question 4:** What are the values printed out by the last **cout** statement from 5 runs of the program?

**Answer**: $60, $60, $80, $80, $80

We should still not be getting the correct answer. Think about why this is so before continuing to read further?

**Answer**: We are not getting the correct answer because the withdraw function is trying to take more money out than the account has and so those threads are not actually effecting the dollar amount and thus money is left over.

The reason why we are still not getting the correct answer is because some calls to **Withdraw()** will fail because it may be called when the account has less than $20 in its balance.

Other threads that were previously spawned (but not executed yet) were meant to “add $20”, thus the total number of adds does not match the total number of withdrawals! Thus we may possibly end up with money left in the account! Remember, all the threads are running in parallel!

There are two ways we can fix this.

1. (Bad Approach) Since we added a mutual exclusion to each of our functions, we could add another variable to our bank class which is initialized to 0. Then on every successful deposit, we increment this number to 1 and on every successful withdrawal we decrement this number. Then, before we make a withdrawal, we check to see if this signal is a 1. This will force the number of deposits to match the number of withdrawals. Why is this a bad approach?
2. (Good Approach) To fix this, we need to modify our withdrawal function to **first check** if the amount in the account **has sufficient funds** before we actually withdraw money! If there is not sufficient funds, the function should wait on a condition variable

**Step 4**: Add #include <condition\_variable>

**Step 5**: define another **member** condition variable to our Bank account class, e.g.

condition\_variable cv;

**Step 6:** in the **BankAccount::Deposit()** method, add code to use your **condition variable** to **notify** a waiting thread when the amount in the account, **after depositing,** has at least the AMOUNT that will need to be withdrawn, i.e.

if (amount >= AMOUNT)

cv.notify\_one();

**Step 7**: in the **BankAccount::whithdraw()** method add the following code segment directly after the definition of the unique\_lock object lk:

unique\_lock<mutex> lk(m);

if ( amount < AMOUNT) {

cv.wait(lk, [this]() {

return this->amount >= AMOUNT;

});

}

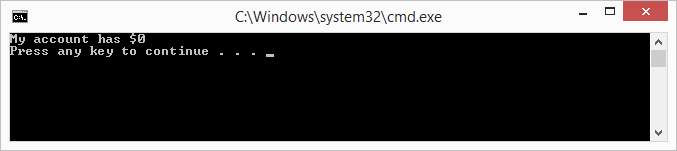
This code will force the thread to **block** (**wait**) until either the account has at least AMOUNT in it (or a spurious wake-up occurs... ask your instructor about spurious\_wakeups…) The **condition\_variable::wait** we are using here is a slightly different from the **wait** method we used in class.

Visit <http://en.cppreference.com/w/cpp/thread/condition_variable/wait> and carefully read the description of **std::condition\_variable::wait**. You should see TWO overloaded versions of the **wait** method.

**Question 5:** What is the **Predicate pred** parameter used for, and how are we using it in the code segment you just added to your program? Now compile, and run the program.

**Answer**: A predicate pred is used as a Boolean that will return false and hold a lock on a piece of code until it returns true to break the lock and allow another thread to come through.

In the final implementation, your console output should print out the correct balance as shown below:



Save your visual studio project and exit.

**Final thought**: Implementing a “correct” parallel program can be tricky. It requires careful thought, and plenty of practice. But once you get the hang of it, it will become fun.