Threads

Advantages of threads

To make your program do more than one thing at a time, you use *threads*. Possible reasons for using threads:

(i) **CPU Performance**

(ii) **Network Performance**

(iii) **Fairness**

(iv) **Usability**

(v) **More natural programming?** *Very occasionally*.

Disadvantages of threads

(i) **Complexity:** Multi-threaded code is much harder to write and understand than single-threaded code.

(ii) **Testability:** Multi-threaded code is much harder to test than single-threaded code.

(iii) **CPU Performance:** sometimes actually worse.

Some code to run on different threads

```
void primalityTest( int* pointerToInt ) {
    int toTest = *pointerToInt;
    for (int i=2; i<toTest; i++) {
        if ((toTest % i)==0) {
            INFO( toTest << " is not prime" );
            return;
        }
    }
    INFO( toTest << " is prime" );
}
```

Creating threads

```
void testPrimes() {
    int values[3] = {1299817,1299821,1299827};
    thread t1( &primalityTest, &values[0] );
    thread t2( &primalityTest, &values[1] );
    thread t3( &primalityTest, &values[2] );
    t1.join();
    t2.join();
    t3.join();
}
```

A problem

Consider the code:

```
cout << "The answer to calculation "<<i<<" is "<<j<<"\n";
```

When run by several threads.

```
The answer to calculation The answer to calculation 1
2 is 19 is 18
```

A race condition

What if two threads call this at once on the same account?

```
bool debitAccount(Account& account, double amount) {
    if (account.balance >= amount) {
        account.balance -= amount;
        return true;
    }
    return false;
}
```

This is an example of a *race condition*.

Mutual exclusion

```
bool debitAccount(Account& account,
    double amount) {
    account.mtx.lock();
    bool ret = false;
    if (account.balance >= amount) {
        account.balance -= amount;
        ret = true;
    }
    account.mtx.unlock(); // don't do this
    return ret;
}
```

Mutual exclusion with a lock guard

```
bool debitAccount(Account& account,
                  double amount) {
    lock_guard<mutex> lock(account.mtx);
    if (account.balance >= amount) {
        account.balance -= amount;
        return true;
    }
    return false;
}
```

Resource acquisition is initialization (RAII)

- resource acquisition is initialization;
- resource release is deletion.

When we introduced destructors the resource we wished to manage was memory. Here we are viewing holding a lock as another form of resource acquisition. In both cases, there is a limited supply of the resource that needs to be managed: a computer has finite memory; only one thread can hold a lock.

Global variables and race conditions

- Whenever you use non constant global variables and multiple threads there is likely to be a race condition.
- For example, writing to the stream `cout`.
- `INFO` and `DEBUG_PRINT` have been changed in FMLib to deal with this.
- `randuniform` uses a global `mt19937` instance. Must be changed.

Using a mutex for random numbers

```
/*  MersenneTwister random number generator */
static mt19937 mersenneTwister;
/*  Mutex to protect static var */
static mutex rngMutex;

/*  Reset the random number generator.
We ignore the description string */
void rng(const string& description) {
    ASSERT(description == "default");
    lock_guard<mutex> lock(rngMutex);
    mersenneTwister.seed(mt19937::default_seed);
}

/*  Generate random numbers */
Matrix randuniform(int rows, int cols) {
    lock_guard<mutex> lock(rngMutex);
    return randuniform(mersenneTwister, rows, cols);
}
```

Deadlock

1) Thread A locks mutex a.
2) Thread B now locks mutex b.
3) Keeping hold of mutex a, thread A tries to lock mutex b.
4) Keeping hold of mutex b, thread B tries to lock mutex a.
5) Neither thread A or thread B can proceed because they are each waiting or the other to complete.

This situation is called deadlock.

Deadlock example

```
bool transferMoney(Account& from,
    Account& to,
    double quantity) {
    lock_guard<mutex> lock1(from.mtx);
    if (from.balance<quantity) {
        return false;
    }
    lock_guard<mutex> lock2(to.mtx);
    from.balance -= quantity;
    to.balance += quantity;
    return true;
}
```

Deadlock prevention strategies

(i) Write single threaded code instead.

(ii) Only use one lock for all accounts.

(iii) Give the locks an ordering and insist that (say) lock A is always acquired before lock B.

(iv) Include a time-out. So, if you don't acquire a lock in a reasonable time frame, you should release all the locks that you hold.

(v) Incorporate some deadlock detection and resolution algorithm in your locking classes. Databases do this.

(vi) Use a database for your data.

Guidelines for multi-threaded code

- Don't use global variables. If you must have global variables other than constants, you will need to use locks to protect them.
- Minimize the data shared between threads. The less that is shared, the less locking required.
- Where possible use `const` data between threads as this won't require locking.
- Divide your code into simple sequential algorithms and small separate sections where threads communicate.
- Use standard, established design patterns and classes for multi-threaded code.
- Don't write multi-threaded code unless there is a clear benefit. Even then only a tiny part of your code should involve threading.

The command pattern

```
class Task {
public:
    virtual ~Task() {}
    virtual void execute() = 0;
};
```

The executor interface

```
class Executor {
public:
    /*  Destructor */
    virtual ~Executor() {}
    /*  Add a task to the executor */
    virtual void addTask(
        std::shared_ptr<Task> task ) = 0;
    /*  Wait until all tasks are complete */
    virtual void join() = 0;
    /*  Factory method */
    static std::shared_ptr<Executor> newInstance();
    /*  Factory method */
    static std::shared_ptr<Executor> newInstance(
        int maxThreads );
};
```

Advantages of the command pattern

- You can easily use a more sophisticated scheduling algorithm (e.g., run all tasks at midnight).
- It divides the responsibility of writing clever threading code and writing business logic.

Monte Carlo pricing

- Launch several threads and take the average result.
- Each thread has its own random number generator.
- We need to ensure these are independent.

Changes to FMLib

```
/*  Create uniformly distributed random numbers */
Matrix randuniform( int rows, int cols );
/*  Create normally distributed random numbers */
Matrix randn( int rows, int cols );
/*  Create uniformly distributed random numbers */
Matrix randuniform(std::mt19937& random,
                   int rows, int cols);
/*  Create normally distributed random numbers */
Matrix randn(std::mt19937& random,
             int rows, int cols);
```

Pass round random number generator everywhere.

```
MarketSimulation generatePricePaths(
    std::mt19937& rng,
    double toDate,
    int nPaths,
    int nSteps) const;
```

Ensure we know how many random numbers are needed:

```
/* How many random numbers are needed
   to generate the given paths? */
long long randSize(long long nPaths,
                   long long nSteps) {
    return stockNames.size()*nPaths*nSteps;
}
```

Writing a multi-threaded pricer

```
double singleThreadedPrice(
        int taskNumber,
        int nScenarios,
        int nSteps,
        const ContinuousTimeOption& option,
        const MultiStockModel& model ) {
```

This essentially repeats the old code except we initialize a random number generator as shown.

```
    MultiStockModel subModel = model.getSubmodel(
        option.getStocks());

    long long randSize = subModel.randSize(nScenarios,
                                           nSteps);
    mt19937 rng;
    rng.discard(randSize*taskNumber);
```

```
class PriceTask : public Task {
public:
    /*  Amount of random numbers to skip */
    int taskNumber;
    int nScenarios, nSteps;
    const ContinuousTimeOption& option;
    const MultiStockModel& model;
    /*  Output data */
    double result;

    PriceTask(
            int taskNumber,
            int nScenarios,
            int nSteps,
            const ContinuousTimeOption& option,
            const MultiStockModel& model)
        :
        taskNumber(taskNumber),
        nScenarios(nScenarios),
        nSteps(nSteps),
        option(option),
        model(model) {
    }

    void execute() {
        result = singleThreadedPrice( taskNumber,
            nScenarios, nSteps, option, model);
    }
};
```

The multi-threaded price function

```
double MonteCarloPricer::price(
    const ContinuousTimeOption& option,
    const MultiStockModel& model) const {
    ASSERT(nTasks >= 1);
    vector< shared_ptr<PriceTask> > tasks;
    shared_ptr<Executor> executor =
        Executor::newInstance(nTasks);
    for (int i = 0; i<nTasks; i++) {
        shared_ptr<PriceTask> task(new PriceTask(
            i, nScenarios/nTasks,
            nSteps, option, model));
        tasks.push_back(task);
        executor->addTask(task);
    }
    executor->join();
    double total = 0.0;
    for (int i = 0; i<nTasks; i++) {
        total += tasks[i]->result;
    }
```

The Pipeline pattern

- Always use standard patterns when writing multi-threaded code.
- The pipeline pattern is a common one. One thread performs a work and then sends a message to another thread along a pipeline.
- Versions of this pattern are used heavily in messaging architectures and in the Unix operating system.
- We will use it as a simple example of how threads can talk to each other.

The Pipeline pattern

```
class Pipeline {
public:
    Pipeline();
    void write( double value );
    double read();
private:
    bool empty;
    double value;
    /*  Mutex to coordinate threads */
    std::mutex mtx;
    /* Condition variable to signal between threads */
    std::condition_variable cv;
};
```

Writing to the pipeline

```
class WriteTask : public Task {
public:
    Pipeline& pipeline;

    void execute() {
        for (int i=0; i<100; i++) {
            pipeline.write(i);
        }
    }

    WriteTask( Pipeline& pipeline ) :
        pipeline( pipeline ) {
    }
};
```

Reading from the pipeline

```
class ReadTask : public Task {
public:
    Pipeline& pipeline;
    double total;

    void execute() {
        for (int i=0; i<100; i++) {
            total+=pipeline.read();
        }
    }

    ReadTask( Pipeline& pipeline ) :
        pipeline( pipeline ),
        total(0.0 ) {
    }
};
```

Putting it all together

```
static void testTwoThreads() {
    Pipeline pipeline;
    auto w = make_shared<WriteTask>( pipeline );
    auto r = make_shared<ReadTask>( pipeline );
    SPExecutor executor = Executor::newInstance(2);
    executor->addTask( r );
    executor->addTask( w );
    executor->join();

    ASSERT_APPROX_EQUAL( r->total, 99.0*50.0, 0.1);
}
```

An example of why it could be useful

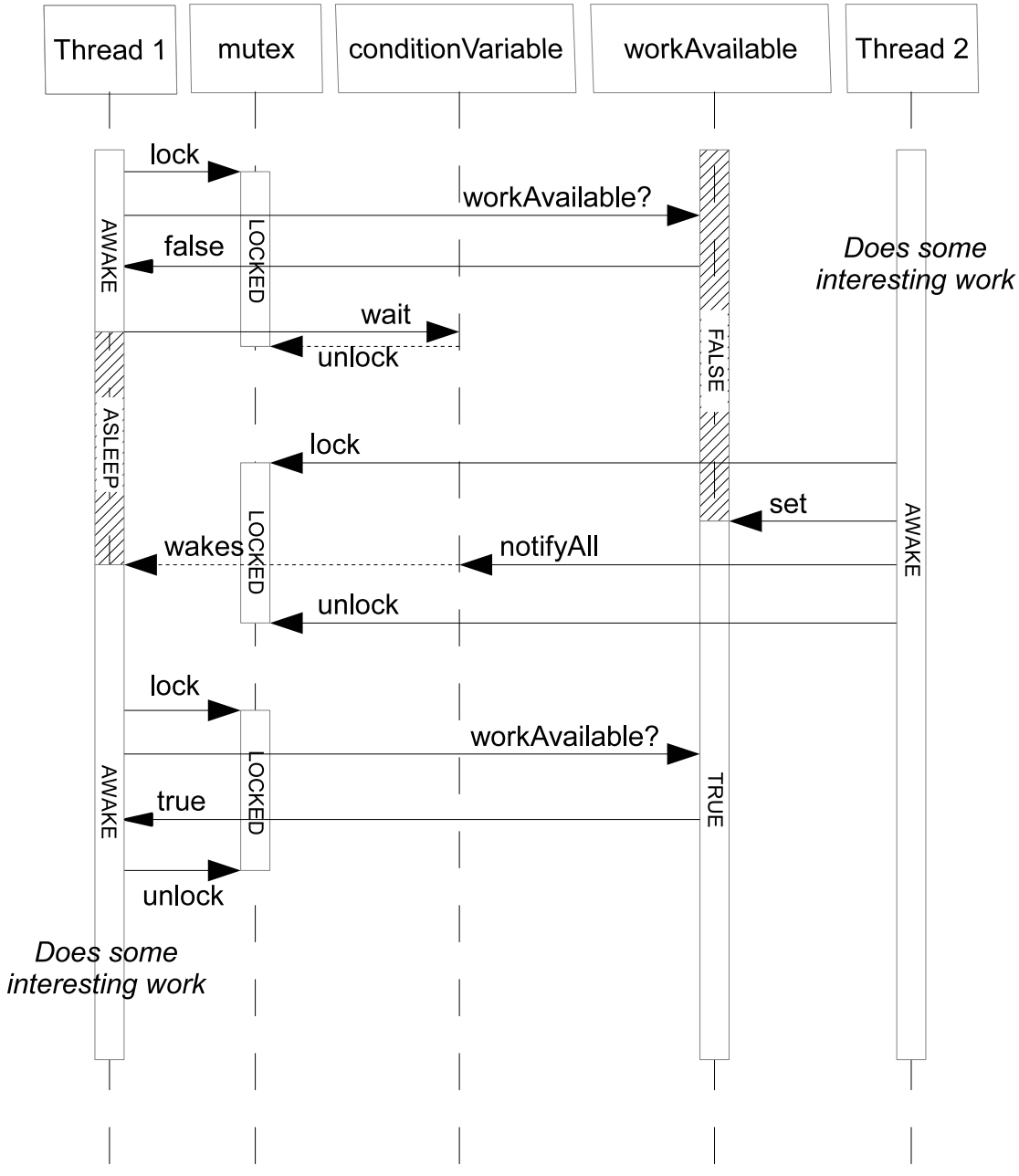
```
void priceByMonteCarlo() {
    double total = 0.0;
    or (int i=0; i<nScenarios; i++) {
        vector<double> path = generatePricePath();
        double payoff = computePayoff( path );
        total += payoff;
    }
    return total/nScenarios;
}
```

We can't parallelise using command as the random-number generator isn't being passed round. We can generatePricePaths on one thread and compute payoffs on another however.

Communicating between threads

- How can threads communicate? E.g. how can one thread tell another that there is some work for it to do?
- Answer is to use condition variables.
- Condition variables implement a fairly complicated protocol to communicate between threads without any possibility of messages being missed.

Condition variables



Using condition variables

(a) Whenever you use a `condition_variable` you should also create a `mutex` to guard the data.

(b) Test if the condition is met in a while loop. You will want to lock the mutex while testing your condition.

(c) Change the data that determines whether your condition passes before calling `notifyAll`. Hold the lock while modifying the data: you should keep holding it until you have called `notify_all`.

(d) If you were to release the lock before calling `wait` it is possible that the condition may change just before you start waiting. As a result you must hold the lock using a `unique_lock<mutex>`, and you must pass the lock as a parameter when calling `wait`.

(e) The `condition_variable` will release your lock and start waiting as one atomic operation.

```
void Pipeline::write( double value ) {
    unique_lock<mutex> lock(mtx);
    while (!empty) {
        cv.wait(lock);
    }
    empty = false;
    this->value = value;
    cv.notify_all();
}

double Pipeline::read() {
    unique_lock<mutex> lock(mtx);
    while (empty) {
        cv.wait(lock);
    }
    empty = true;
    cv.notify_all();
    return value;
}
```

Moral

- C++ threading primitives are tricky to use.
- Use a threading library that makes life easier.
- e.g., `Pipeline` is easy to use but hard to write.

Summary

- We have seen how to take advantage of multiple CPUs to write a multi-threaded Monte Carlo pricer.
- We have seen how threads can interact and seen the Pipeline design pattern.
- We have learned about race conditions, deadlocks and mutexes.