Tutorial for coursework Part 2

UG3 Computer Communications & Networks (COMN)

Myungjin Lee
myungjin.lee@ed.ac.uk
Overview

• To understand the purpose of multithreading
• To describe Java's multithreading mechanism
• To explain concurrency issues caused by multithreading
• To outline synchronized access to shared resources
What is multithreading?

- Multithreading is similar to multi-processing
- A multi-processing OS can run several processes at the same time
  - Each process has its own address/memory space
  - Separate processes to not have access to each other's memory space
- In a multithreaded application, there are several points of execution **within the same memory space**
  - Each point of execution is called a thread
  - Threads share access to memory
Thread Support in Java

• The Java Virtual machine has its own runtime threads
  – Used for garbage collection

• Threads are represented by a Thread class
  – A thread object maintains the state of the thread
  – It provides control methods such as interrupt, start, sleep, yield, wait

• When an application executes, the main method is executed by a single thread
  – If the application requires more threads, the application must create them
Thread States

- Threads can be in one of four states
  - Created, Running, Blocked, and Dead
- A thread's state changes based on:
  - Control methods such as start, sleep, yield, wait, notify
  - Termination of the run method
How does a thread run?

• The thread class has a run() method
  – run() is executed when the thread's start() method is invoked

• The thread terminates if the run method terminates
  – run() method often has an endless loop to prevent thread termination

• One thread starts another by calling its start method
Creating your own Threads

• A way to create your own threads is to subclass the Thread class and then override the run() method
  – This is the easiest way to do it although not recommended

• The object which provides the run method is usually a subclass of some other class
  – If it inherits from another class, it cannot inherit from Thread

• The solution to this problem is Runnable interface
  – Runnable defines one method - public void run()
  – Thread class constructor can take a reference to a Runnable object
  – When the thread is started, it invokes the run method in the runnable object instead of its own run method
Using Runnable

- When the Thread object is instantiated, it is passed a reference to a "Runnable" object
  - The Runnable object must implement the "run" method
- When the thread object receives a start message, it checks if it has a reference to a Runnable object:
  - If it does, it runs the "run" method of that object
  - If not, it runs its own "run" method
```java
public class thdexpl1 {
    public static int count = 0;
    private static class MyThread implements Runnable {
        public void run() {
            while (count <= 10) {
                System.out.println("MyThread: " + count++);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {}
            }
        }
    }
}
```
public static void main(String[] args) {
    System.out.println("Starting Main Thread...");
    MyThread mythd = new MyThread();
    Thread t = new Thread(mythd);
    t.start();
    while (count <= 10) {
        System.out.println("MainThread: " + count++);
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
    }
}
Creating Multiple Threads

• The previous example illustrates a Runnable class which creates its own thread when the start method is invoked.

• To create multiple threads, one could simply create multiple instances of the Runnable class and send each object a start message:
  – Each instance would create its own thread object.
Synchronization
Critical Sections / Mutual Exclusion

• Sequences of instructions that may get incorrect results if executed simultaneously are called **critical sections**
• (We also use the term **race condition** to refer to a situation in which the results depend on timing)

• **Mutual exclusion** means “not simultaneous”
  – A < B or B < A
  – We don’t care which

• Forcing mutual exclusion between two critical section executions is sufficient to ensure correct execution – guarantees ordering

• One way to guarantee mutually exclusive execution is using **locks**
Critical sections

\[ \rightarrow \text{is the "happens-before" relation} \]

Possibly incorrect

Correct

Correct
When do critical sections arise?

• One common pattern:
  – read-modify-write of
  – a shared value (variable)
  – in code that can be executed concurrently

• Shared variable:
  – Globals and heap-allocated variables
  – NOT local variables (which are on the stack)
Example: shared bank account

• Suppose we have to implement a function to withdraw money from a bank account:

```c
int withdraw(account, amount) {
    int balance = get_balance(account); // read
    balance -= amount; // modify
    put_balance(account, balance); // write
    spit out cash;
}
```

• Now suppose that you and your partner share a bank account with a balance of $100.00
  – what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?
Assume the bank’s application is multi-threaded
A random thread is assigned a transaction when that transaction is submitted

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    spit out cash;
}
```

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    spit out cash;
}
```
Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

```
balance = get_balance(account);
balance -= amount;
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
spit out cash;
put_balance(account, balance);
spit out cash;
```

- What’s the account balance after this sequence?
  - who’s happy, the bank or you?
Locks

• A lock is a memory object with two operations:
  – `acquire()`: obtain the right to enter the critical section
  – `release()`: give up the right to be in the critical section

• `acquire()` prevents progress of the thread until the lock can be acquired

• Note: terminology varies: acquire/release, lock/unlock
Locks: Example

Example execution:

- lock()
- unlock()

Two choices:
- Spin
- Block
- (Spin-then-block)
Java Synchronization Mechanism

• Java has a keyword called synchronized

• In Java, every object has a lock
  – To obtain the lock, you must synchronize with the object

• The simplest way to use synchronization is by declaring one or more methods to be synchronized
public class SavingsAccount
{
    private float balance;

    public synchronized void withdraw(float anAmount)
    {
        if ((anAmount>0.0) && (anAmount<=balance))
            balance = balance - anAmount;
    }

    public synchronized void deposit(float anAmount)
    {
        if (anAmount>0.0)
            balance = balance + anAmount;
    }
}
public class SavingsAccount {
    private float balance;

    public void withdraw(float anAmount) {
        if (anAmount<0.0)
            throw new IllegalArgumentException("Withdraw amount negative");
        synchronized(this) {
            if (anAmount<=balance)
                balance = balance - anAmount;
        }
    }

    public void deposit(float anAmount) {
        if (anAmount<0.0)
            throw new IllegalArgumentException("Deposit amount negative");
        synchronized(this) {
            balance = balance + anAmount;
        }
    }
}
Example Codes

thdexp1.java and thdexp2.java
from

https://drive.google.com/open?id=0B6rUEJFM3QjTWUUxT2E0WHFNOWc
Design choices for Part 2

• Both sender and receiver are implementable without multithreading
  – Definitely no need for multithreading at the receiver side
  – Multithreading may be useful for sender implementation

• Many design choices for the sender are possible
Sketch of one design for Part 2A

Sender 2A

Timer Thread

Timer start

Notify timeout

Data (Re)Send Thread

buffer

Base seqno

Next seqno

ACK Receive Thread

Timer start/stop

Receiver 2A
Sketch of one design for Part 2B

Sender 2B

Timer Queue

Timer restart

Timer + ReTx Thread

Data Send Thread

For ReTx

Timer start

buffer

Base seqno

Next seqno

ACK marking

ACK Receive Thread

Receiver 2B