Threads and Concurrency

CS2112 Fall 2018 — Recitation 10

What is a Thread?

- A separate process that can perform a computational task independently and concurrently with other threads
  - Most programs have only one thread:
    - *main thread*
  - GUIs have two other threads:
    - *application (or event dispatch) thread*
    - *rendering thread*
  - A program can have many threads
  - You can create new threads in Java

What is a Thread?

- In reality, threads are an illusion
  - The processor shares its time among all the active threads
  - Implemented with support from underlying operating system or virtual machine
  - Gives the illusion of several threads running simultaneously

Concurrency (aka Multitasking)

- Refers to situations in which several threads are running simultaneously

- Special problems arise
  - race conditions
  - deadlock

- The operating system provides support for multitasking
- In reality there is one processor doing all this
- But this is an illusion too – at the hardware level, lots of multitasking
  - memory subsystem
  - video controller
  - buses
  - instruction prefetching

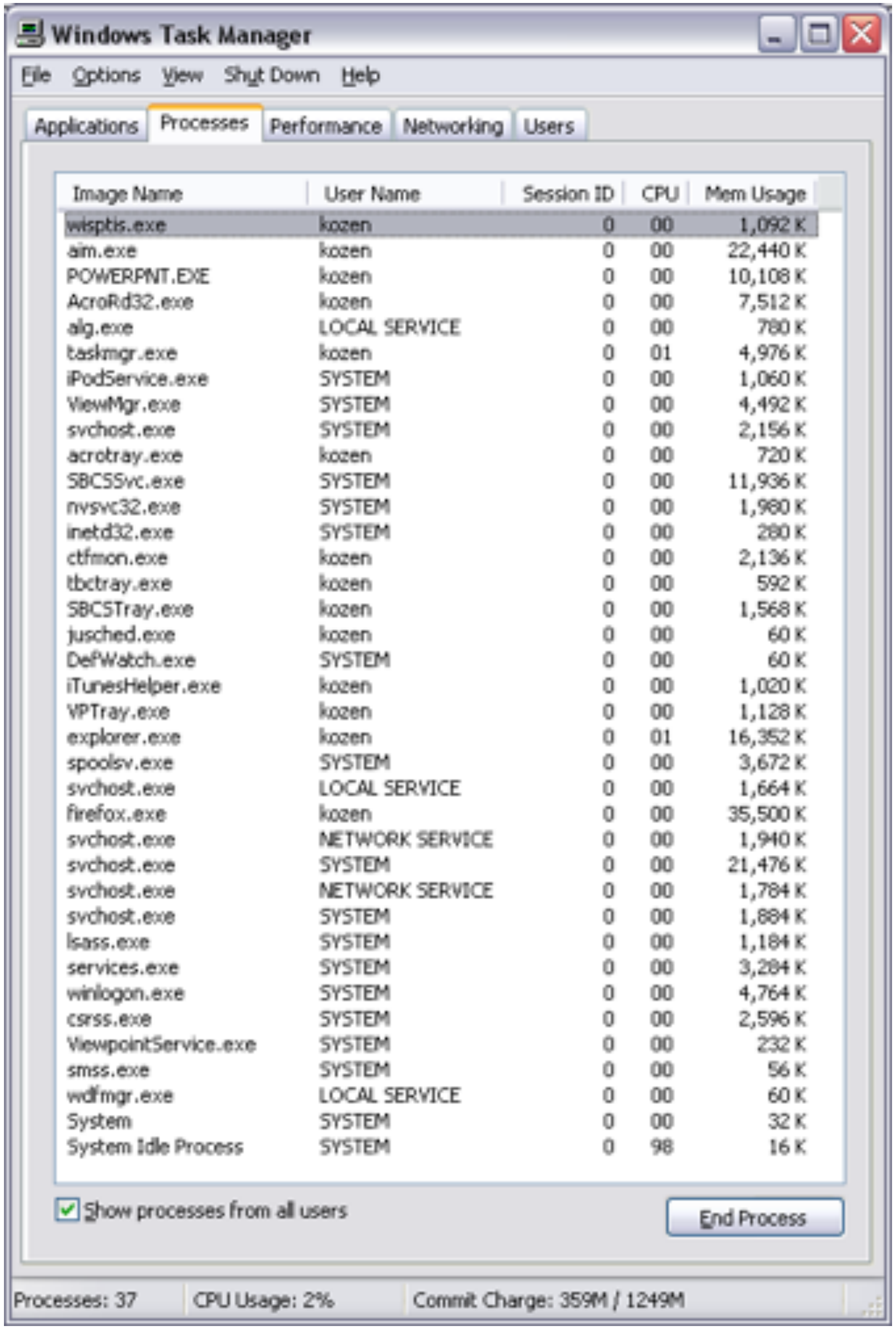

Image Name	User Name	Session ID	CPU	Mem Usage
wisptis.exe	kozen	0	00	1,092 K
aim.exe	kozen	0	00	22,440 K
POWERPNT.EXE	kozen	0	00	10,108 K
AcroRd32.exe	kozen	0	00	7,512 K
alg.exe	LOCAL SERVICE	0	00	780 K
taskmgr.exe	kozen	0	01	4,976 K
iPodService.exe	SYSTEM	0	00	1,060 K
ViewMgr.exe	SYSTEM	0	00	4,492 K
svchost.exe	SYSTEM	0	00	2,156 K
acrotray.exe	kozen	0	00	720 K
SBCSSvc.exe	SYSTEM	0	00	11,936 K
nvsvc32.exe	SYSTEM	0	00	1,980 K
inetd32.exe	SYSTEM	0	00	280 K
ctfmon.exe	kozen	0	00	2,136 K
tbctray.exe	kozen	0	00	592 K
SBCSTray.exe	kozen	0	00	1,568 K
jusched.exe	kozen	0	00	60 K
DefWatch.exe	SYSTEM	0	00	60 K
iTunesHelper.exe	kozen	0	00	1,020 K
VPTray.exe	kozen	0	00	1,128 K
explorer.exe	kozen	0	01	16,352 K
spoolsv.exe	SYSTEM	0	00	3,672 K
svchost.exe	LOCAL SERVICE	0	00	1,664 K
firefox.exe	kozen	0	00	35,500 K
svchost.exe	NETWORK SERVICE	0	00	1,940 K
svchost.exe	SYSTEM	0	00	21,476 K
svchost.exe	NETWORK SERVICE	0	00	1,784 K
svchost.exe	SYSTEM	0	00	1,884 K
lsass.exe	SYSTEM	0	00	1,184 K
services.exe	SYSTEM	0	00	3,284 K
winlogon.exe	SYSTEM	0	00	4,764 K
csrss.exe	SYSTEM	0	00	2,596 K
ViewpointService.exe	SYSTEM	0	00	232 K
smss.exe	SYSTEM	0	00	56 K
wdfmgr.exe	LOCAL SERVICE	0	00	60 K
System	SYSTEM	0	00	32 K
System Idle Process	SYSTEM	0	98	16 K

☑ Show processes from all users

Processes: 37 | CPU Usage: 2% | Commit Charge: 359M / 1249M

Threads in Java

- Threads are instances of the class **`Thread`**
  - you can create as many as you like

- The Java Virtual Machine permits multiple concurrent threads
  - initially only one thread (executes **`main`**)

- Threads have a priority
  - higher priority threads are executed preferentially
  - a newly created **`Thread`** has initial priority equal to the thread that created it (but can change)

Creating a new Thread (Method 1)

```
class PrimeThread extends Thread {
   long a, b;

   PrimeThread(long a, long b) {
      this.a = a; this.b = b;
   }

   public void run() {
      //compute primes between a and b
      ...
   }
}
```

overrides **`Thread.run()`**

can call **`run()`** directly – the calling thread will run it

```
PrimeThread p = new PrimeThread(143, 195);
p.start();
```

or, can call **`start()`** – will run **`run()`** in new thread

Creating a new Thread (Method 2)

```
class PrimeRun implements Runnable {
   long a, b;

   PrimeRun(long a, long b) {
      this.a = a; this.b = b;
   }

   public void run() {
      //compute primes between a and b
      ...
   }
}
```

```
PrimeRun p = new PrimeRun(143, 195);
new Thread(p).start();
```

Example

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[Thread-0,5,main] 0
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
Thread[Thread-0,5,main] 1
Thread[Thread-0,5,main] 2
Thread[Thread-0,5,main] 3
Thread[Thread-0,5,main] 4
Thread[Thread-0,5,main] 5
Thread[Thread-0,5,main] 6
Thread[Thread-0,5,main] 7
Thread[Thread-0,5,main] 8
Thread[Thread-0,5,main] 9
```

Example

```java
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      currentThread().setPriority(4);
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
Thread[Thread-0,4,main] 0
Thread[Thread-0,4,main] 1
Thread[Thread-0,4,main] 2
Thread[Thread-0,4,main] 3
Thread[Thread-0,4,main] 4
Thread[Thread-0,4,main] 5
Thread[Thread-0,4,main] 6
Thread[Thread-0,4,main] 7
Thread[Thread-0,4,main] 8
Thread[Thread-0,4,main] 9
```

Example

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      currentThread().setPriority(6);
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[Thread-0,6,main] 0
Thread[Thread-0,6,main] 1
Thread[Thread-0,6,main] 2
Thread[Thread-0,6,main] 3
Thread[Thread-0,6,main] 4
Thread[Thread-0,6,main] 5
Thread[Thread-0,6,main] 6
Thread[Thread-0,6,main] 7
Thread[Thread-0,6,main] 8
Thread[Thread-0,6,main] 9
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
```

Example

```java
public class ThreadTest extends Thread {
    static boolean ok = true;

    public static void main(String[] args) {
        new ThreadTest().start();
        for (int i = 0; i < 10; i++) {
            System.out.println("waiting...");
            yield();
        }
        ok = false;
    }

    public void run() {
        while (ok) {
            System.out.println("running...");
            yield();
        }
        System.out.println("done");
    }
}
```

allows other waiting threads to run

```
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
done
```

Stopping Threads

- Threads normally terminate by returning from their run method
- **`stop(), interrupt(), suspend(), destroy()`**, etc. are all deprecated
  - can leave application in an inconsistent state
  - inherently unsafe
  - don't use them
  - instead, set a variable telling the thread to stop itself

Daemon and Normal Threads

- A thread can be *daemon* or *normal*
  - the initial thread (the one that runs **`main`**) is normal

- Daemon threads are used for minor or ephemeral tasks (e.g. timers, sounds)

- A thread is initially a daemon iff its creating thread is
  - but this can be changed

- The application halts when either
  - **`System.exit(int)`** is called, or
  - all normal (non-daemon) threads have terminated

Race Conditions

- A *race condition* can arise when two or more threads try to access data simultaneously

- Thread B may try to read some data while thread A is updating it
  - updating may not be an atomic operation
  - thread B may sneak in at the wrong time and read the data in an inconsistent state

- Results can be unpredictable!

Example – A Lucky Scenario

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s = stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call **`doSomething()`**, and there is one element on the stack

1. thread A tests **`stack.isEmpty()`** ⇒ false
2. thread A pops ⇒ stack is now empty
3. thread B tests **`stack.isEmpty()`** ⇒ true
4. thread B just returns – nothing to do

Example – An Unlucky Scenario

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s = stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call **`doSomething()`**, and there is one element on the stack

1. thread A tests **`stack.isEmpty()`** ⇒ false
2. thread B tests **`stack.isEmpty()`** ⇒ false
3. thread A pops ⇒ stack is now empty
4. thread B pops ⇒ Exception!

Solution – Locking

```java
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   synchronized (stack) {
      if (stack.isEmpty()) return;
      String s = stack.pop();
   }
   //do something with s...
}
```

`synchronized` block

- Put critical operations in a **`synchronized`** block
- The **`Stack`** object acts as a lock
- Only one thread can own the lock at a time

Solution – Locking

- You can lock on any object, including **this**

```
public synchronized void doSomething() {
    ...
}
```

is equivalent to

```
public void doSomething() {
    synchronized (this) {
        ...
    }
}
```

File Locking

- In file systems, if two or more processes could access a file simultaneously, this could result in data corruption
- A process must *open* a file to use it – gives exclusive access until it is *closed*
- This is called *file locking* – enforced by the operating system
- Same concept as **`synchronized(obj)`** in Java

Deadlock

- The downside of locking – *deadlock*

- A *deadlock* occurs when two or more competing threads are waiting for the other to relinquish a lock, so neither ever does

- Example:
  - thread A tries to open file X, then file Y
  - thread B tries to open file Y, then file X
  - A gets X, B gets Y
  - Each is waiting for the other forever

wait/notify

- A mechanism for event-driven activation of threads

- Animation threads and the GUI event-dispatching thread in can interact via **wait/ notify**

wait/notify

animator:

```
boolean isRunning = true;

public synchronized void run() {
    while (true) {
        while (isRunning) {
            //do one step of simulation
        }
        try {
            wait();
        } catch (InterruptedException ie) {}
        isRunning = true;
    }
}
```

relinquishes lock on **animator** –
awaits notification

```
public void stopAnimation() {
    animator.isRunning = false;
}

public void restartAnimation() {
    synchronized(animator) {
        animator.notify();
    }
}
```

notifies processes waiting
for **animator** lock