Threads

Processes

- Many programs execute on a computer simultaneously
  - on this machine we're running: PowerPoint, a demo, some networking software and so on
  - generally there are about 60 processes running
  - a process is just a program running on the machine (i.e., your Java program)
- But CPUs/cores can each only execute 1 instruction at a time
- So how do all of these programs run simultaneously?
- They don't

- Time Slicing
  - process 1 runs for a little while, and then process 2 runs for a while, and then process 3, etc.
  - like a master chess player making moves for several players across multiple games “simultaneously”
  - based on some scheduler in the operating system
    - (take cs167/9…)
  - basis of time-sharing on mainframes

Threads

- Individual programs can also take advantage of this “simultaneous” computing
  - Inside an individual process there can be multiple **threads**
- This type of program is referred to as “multithreaded” or “concurrent”
- Each thread represents a “context of execution”
  – at any point, the code that is being executed is running inside a thread

- All threads are running concurrently and share the machines resources (memory, CPU, hard disk etc.)
  - This means they can share memory, whereas processes cannot

- Thread 1 runs for a while, Thread 2 runs for a while, Thread 3 runs for a while, …, Thread 1 runs again
  - if the delay is small enough, it will appear as if everything is running at the same time

- Every program has at least one main thread

Why do we need them?

- Most programs are not simply outright computation, during which you start up the program and run continuously until completion
- In fact, many programs spend a significant amount of time waiting
  - For the user to type a key or click on the GUI
  - For data to come through the network or to read from a file
  - we commonly refer to the thread as being “blocked on <file, network, etc.>”

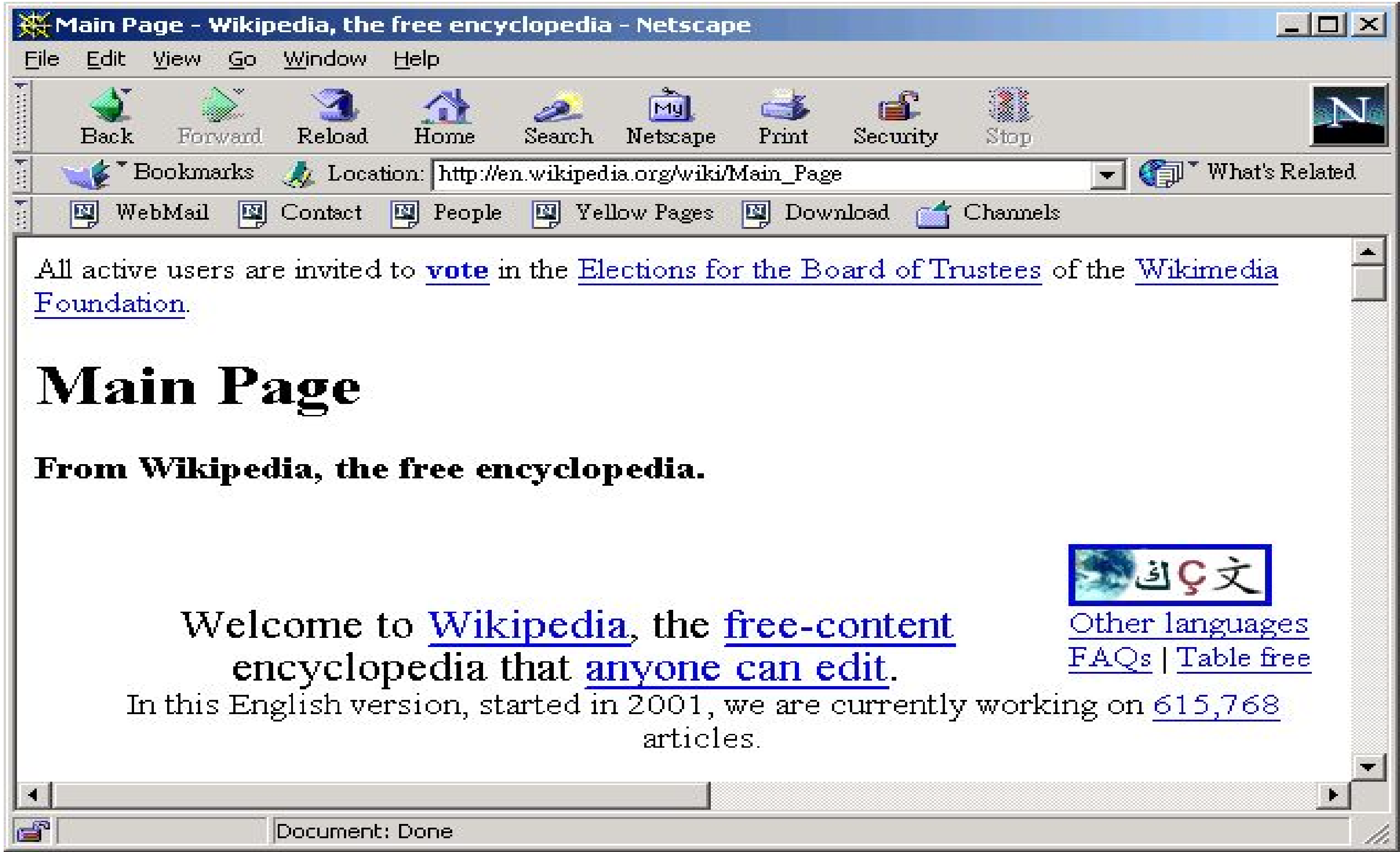
- While the program is waiting, we would like it if we could be doing useful work in another thread

- Multi-threaded programs can take advantage of multiple CPU cores
  - application performance can increase substantially

Example

- Web browsers can spend a lot of their execution time waiting
- Web browser has to receive user input (URL, clicking on a link, etc.), get the data from the address and display the contents
  - if the user does not interact with the browser, the browser should wait
  - while data is being transmitted across the network, the browser is waiting
- With all that waiting, a multithreaded implementation seems like a better solution…

Blast from the past



- The earliest versions of Netscape were not multithreaded.
- After the user typed in a URL, the browser would issue a request to get the data from the computer at the URL and wait until the data arrived.
- Since it was single threaded, nothing else could go on while the browser was waiting.
- What if you typed the wrong URL?

Relevant to you

- Whenever code is executing, it’s always run in the context of a thread
  - all of the programs you’ve written have been running in threads!
  - if your program is single-threaded, you can ignore the fact that threads exist entirely
- Swing Timer is running in its own thread
  - the thread waits for a fixed time and then wakes up to tell your program to do something (`actionPerformed`) and then waits again

Non-Threaded GUI

- Some Swing components do start their own thread, but it's far from being a fully multi-threaded API

- Example:
  - A program has two buttons. Clicking on one prints to console that button was clicked. Clicking on other button starts a very time-consuming computation

- What happens if click on time-consuming computation button and then click on other button?
- Program tries to deal with the time-consuming computation, but since it’s single-threaded, can’t do anything else until computation is done
- Only after it’s complete can it respond to the other button

Threaded GUI

- In the non-threaded version we could have a class that performs the calculations:

```
public class SlowComputation {
  public int performLotsOfCalculations() {
    int j = 0;
    for (int i = 0; i < 100000000; i++)  {
        /*
         Do something really time consuming
        */
    }
    return j;
  }
}
```

- Rewrite this program so that time-consuming computation can be running in its own thread

Threaded GUI - Code

- Making threads in Java is easy! Have the class extend java.lang.Thread or have it implement the java.lang.Runnable interface

- Let’s write a new class that performs the same computation in its own thread

```
public class ThreadedComputation extends java.lang.Thread {
      private boolean _stop;

      public ThreadedComputation() {
        _stop = true;
    }
/*  run() is key thread's method to override */
    public void run() {
        _stop = false;
        this.performLotsOfCalculations();
    }
    public void stop() {
        _stop = true;
    }

      public int performLotsOfCalculations() {
            int j = 0;
        int i = 0;
            while (i < 100000000 && !_stop)  {
            /*
             Do something really time consuming
            */
            i++;
        }
        return j;
      }
    }
```

Note: **run()** will be called automatically when **start()** is called on the thread object

- Using multiple threads seems like a really good idea so why don’t we use it all the time?
- Writing multithreaded code can be tricky when two threads are modifying the same stuff
- If things are happening “at the same time,” we can have weird errors because of dependencies

- In the single-threaded version:

```
_michelle.eatCookie(cookieJar);
_wendy.eatCookie(cookieJar);
```

Michelle eats the cookie first, emptying the jar. When Wendy asks for a cookie, there is nothing left

- In the multi-threaded version, both of these things can be happening “at the same time”

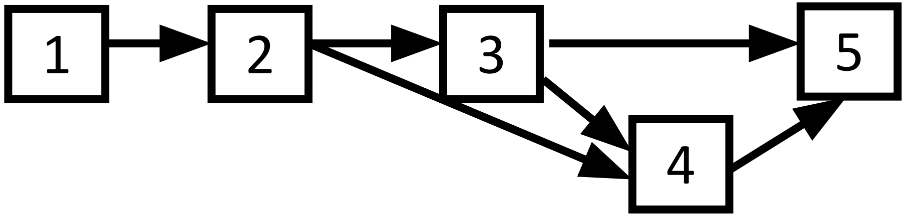
```
_michelle.eatCookie(cookieJar) _wendy.eatCookie(cookieJar)
```

- They’ll both be eating the same cookie which is not what we want

Complications with Threads

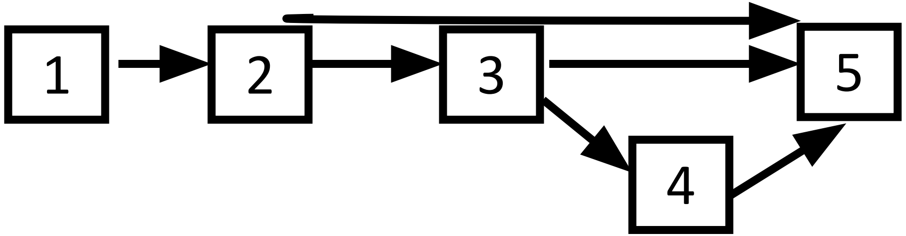
- Anytime you use a shared object (the `cookieJar`) you'll have to worry about concurrently modifying the object
- Because of time slicing, a thread's execution can be taken away from it at anytime
  - the first two lines of the method run but it's not until a while later (you just got time sliced) that the rest of the method gets to run.
  - the exact time you get time-sliced is non-deterministic (it depends on how busy the machine is) making these kind of bugs very hard to track down

- Linked list example:



- Let’s insert 4 and then remove 3.
  - the order pointers change is very important!
  - everything works as expected

- Linked list example:



- Let’s insert 4 and then remove 3.
  - the order pointers change is very important!
  - everything works as expected
- Let’s have one thread insert 4 at the same time another thread removes 3.
  - since we can be time sliced at any time, we have to make sure that the program works in the worst case: we are time sliced between pointer changes
  - we get time sliced and the remove code runs.
  - the insertion code continues to run.
  - when the methods end, local references and objects are GC’ed
  - the end result is that 4 is not inserted, even though the method executed

Locks

- What we would like is to prevent another linked list operation from executing if we are currently modifying the list

- If you attempt to lock something that is already locked, you are put on a queue and must wait until the lock is freed up

- If you attempt to lock something that is not locked, you grab the lock and continue executing

- Only one thread can have a lock at any time

Synchronized

- In Java, locks are done using the **`synchronized`** keyword.
  - there are many ways to use it but they all basically do the same thing.

- The "lock" is the object ownership which java assigns to whichever thread (if any) is currently executing synchronized methods for this object
  - Any object can be a "lock" for its instance variables. In our example, lock is **`this`**

- You can make a method synchronized:

```
public synchronized void someMethod() {
    /* only one thread will be able to
run the code in here at a time */
}
```

- The **synchronized** keyword ensures that one thread cannot start executing that block of code until the previous thread has finished
- You can synchronize an object for a specific portion of a method

```
public void someMethod() {
    //some java code - multiple threads can
  //be executing these lines
    synchronized(this) {
        //some java code - only one thread
    //can get in here
    }
    //more java code that multiple threads
  //can run
}
```

Joining Threads

- Suppose that the main thread needs to do a very time-consuming computation. It "spawns" another thread to perform the task while it continues to work on other things
- Later on, the main thread realizes it needs a result from the computation, so it needs to wait for the computation thread to finish (i.e., closing an application during an auto-save)

- You can do this in Java by calling the thread's `join()` method

```
public void someMethod() {
        /* do some work */
        ThreadedComputation computeThread = new
              ThreadedComputation();
        computeThread.start();

      /* do some work */

      computeThread.join();

      /* we won't get here until the computeThread has
   completed */
}
```

- Have you ever wondered why your Swing Applications don’t just exit? You are not sitting in a loop of any kind, and it seems that after constructing your top level object (App), your program should exit

- Swing automatically creates some threads for you and your main thread is joined with these threads
  - these threads are sitting in loops and your program (main thread) can not exit until these threads have exited.