Threads

A Thread is…

- A portion of a program which runs independently from the other portions
- Shares resources (including main memory) with all portions
- Can run concurrently or synchronously

Thread Vs. Process

- Process: a unit of resource allocation and protection. Associated with process:
  - Virtual address space with process image
  - Protected access to processors, other processes.
- Thread has:
  - Thread execution state
  - Saved thread context when not running
  - Execution stack
  - Some per-thread static storage for local variables
  - Access to memory and resources of its process

Why Threads?

- Performance!
  - Takes less time to create a new thread in an existing process than to create a new process
  - Takes less time to terminate
  - Less time to switch between two threads of the same process
  - Enhance efficiency in communication between different executing programs
- Any problem can be solved without threads, it just wouldn’t be as efficient.

Multiprocessor vs. Uniprocessor

- On a system with multiple processors, threads can run concurrently (literally at the same time)
- On a uniprocessor system, only one thread can ever run since there is only one CPU
- If a program has multiple threads, the threads might be running on two different processors at the same time.
- Thread “safety” becomes key in these environments
- HyperThreading describes the ability for the CPU to “prepare” a second thread to run on the processor as soon as a different thread finishes

Where should/could we use Threads

- Foreground and background work
  - Spreadsheet: display vs updating
- Asynchronous Processing:
- Word processor: background thread to save to disk.
- Speed execution:
  - Compute while reading next batch of data
- Modular program structure

Thread States in the OS

- A Thread can be put into the following “states” in the Operating System
  - Running – This thread is, actively, running code on the CPU
  - Blocked – This thread is waiting for something to happen and cannot run until that occurs.
  - Ready – This thread can run if chosen.

Thread Priority

- Java and the Operating System choose which threads to run.
- The algorithm for choosing can be simple or complex.
- In Java, each Thread in a process is given a "Priority" number from 1 to 10 (10 means highest)
- In Java, of the threads that are ready (not blocked), the highest priority thread is chosen. If two threads have the same priority, FIFO dictates which runs first.

Thread Scheduling

- The Operating System controls when a thread runs
- Any thread can be started or stopped at any time
- Threads, like processes, will continue from where they last ran
- Your code isn’t “aware” of having been stopped and started again
- You do not have to, and in a lot of ways cannot, consider scheduling of your threads, but do have to take it into account.

Choosing to stop temporarily

- A thread has no way of choosing to “start” running again, since it’s not already running
- However, a thread CAN chose to stop running temporarily
  - Yield – The thread moves to the ready state and is chosen to run again soon
  - Sleep – A timer is set for a certain number of milliseconds, and the thread moves to the blocked state. After the timer expires, the thread moves to the ready state and is scheduled accordingly.

Thread class in Java

- The Thread class is the easiest way to implement a thread in Java.
- As it is a “normal” class, you can create constructors to pass “parameters”
- The “public void run()” method will be invoked when the Thread enters the running state for the first time
- Once invoked, the original thread (Main, maybe?) and the new thread are scheduled and run concurrently.
- The thread class includes methods like yield and sleep
- https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Construction/Execution

- To start execution of a new thread, construct one on your objects and call the “start” method.
  - Do not call “run” as that would not, first, create the thread. Start will call run for you
- Any thread can throw an “InterruptedException” which must be caught, or re-thrown

What if I’m already extending another class?

- Java has a “Runnable” interface. You can construct a new thread based on any Runnable object (pass the object to the thread constructor) and then run it.

How do I get “my” thread object?

- The Thread class has a static method “currentThread” which returns a reference to the thread that you’re in.
- This reference can be used to:
  - Change the priority of your own thread
  - Yield
  - Sleep
  - etc

Access Methods

- Public/Private/Protected/(Default) don’t change with respect to threads
- Static/Instance doesn’t change with respect to threads
- Nothing in the code is different when using threads

Downsides of using threads

- Concurrency control issues!
- Threads don’t know when they are running
- Threads need to, carefully, coordinate what they are doing with other threads when interacting on a common resource
  - Memory (buffers, maybe?)
  - Network Connections
  - Files
  - The output screen

Output to the screen

- Each thread is given an ID number (1-5) and is responsible for printing that number of tab characters and then printing it's ID number.
- If all works, each number is printed with the same amount of indentation

Producer/Consumer

- Imagine a shared, fixed size, memory buffer and multiple “producer” threads and “consumer” threads
    - Producer threads insert items into the buffer (if space is free)
    - Consumer threads remove items from the buffer (if items exist)
- Without coordinating their activity, anarchy can exist where two consumers try to consume the same item or two producers try to fill the last spot

Double update/Missing update

- Imagine a bank balance (shared variable)
- One person goes to the ATM and deposits money
- Another person goes to the ATM and withdraws money
- Each thread, initially, makes a local copy of the variable, does the calculation and then stores the result in the shared memory.
- The possibility exists that both actions occur at the same time and one is lost.

Semaphores

- Thread code is written with “Critical Sections” in mind
- No two threads can be in a critical section (for a given resource) at the same time
- A semaphore acts as a “traffic light” for the thread to allow the thread to enter its critical section or not.
- When entering the critical section, the thread calls a method to request access and inform the semaphore that it’s entering (wait or notify)
  - If another thread has already entered the critical section, this thread is blocked.
- When exiting the critical section, the thread calls a method to “release” the next blocked thread or “open” the resource

Java’s concurrency control solution

- https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
- Every object is a semaphore!

Synchronized

- Java allows for any method or block of code to be “synchronized” on an object
- Synchronized blocks cannot run concurrently so if one thread blocks in a synchronized section of code, it will prevent any other critical sections from starting