Name: _____

IB Computer Science 2
Concurrency worksheet

Download `Concurrency.zip` to the folder where you usually write activity code, and extract it.

Preemption

Immediately execute the `Preemption.java` program. Let it run while you discuss and answer the following questions.

1. Examine the program, which consists of a busy loop that repeatedly measures the elapsed time between loop iterations. A minute or two after you started it, what is the highest measurement that it has printed?

2. Open a second terminal. Do you see any effect on the running program?

3. Open a second terminal and check your CPU clock speed by running the command:
   `sysctl -a | grep -F hw.tb`
   The result is in 100s of Hz. Divide by 10 million to convert to GHz and record the result.

4. A CPU clocked at 1 GHz can execute one billion machine language instructions per second, or one million instructions per millisecond. How many instructions could your CPU have executed during the time that vanished during the long loop iterations?

5. A single iteration of the loop should not have taken that long! What do you suppose was happening during that time?

6. Check your CPU's number of cores by running: `sysctl -a | grep -F cpu.cores`

7. Open the macOS Activity Monitor application. What do you notice about the number of running processes? Does this confirm or refute your explanation for the lost time?

Concurrency

You just discovered that processes do not necessarily execute simultaneously. Instead, they run **concurrently**, or overlapping in time. Two processes that start and end at the same time might not each be running for the entire duration: one might pause for a while to allow the other to run.

One scenario that prevents truly simultaneous multitasking is when there are more running processes than processor cores. In this case, the operating system **preempts**, or forcibly interrupts, a process. It then schedules some other process onto its core, during which time the original is paused. The last program's measurements revealed the time it lost while paused. Notice that the program's ability to observe the lost time is one example of a hole in the illusion that each program is executing on its own private computer!

Another scenario where simultaneous execution is impossible is when the result of a calculation is needed to perform the next step of the program.

8. Examine `Pi.java`, which approximates the mathematical constant pi by throwing virtual "darts" at a square "dartboard" and observing the ratio landing within a circle. Try running it with 10 million darts. What do you notice about the program's performance?

The operating system is not the only program that can use concurrency. A single process can ask the operating system to start additional concurrent **threads** of execution. Each thread has its own independent control flow, but unlike a separate process, it shares memory (other than local variables!) with the rest of the process.

9. Examine `PiConcurrent.java`, which concurrently performs the calculation and awaits user input. Predict how it will behave with the same input. Then test your hypothesis.

10. Examine `Concurrency.java`, which concurrently concatenates together 1000 characters. Predict the length of the resulting string.

11. Run the program to test your hypothesis. What happens and why?

12. Now run `ConcurrencyFixed.java`. Looking at its source code, what different class does it use? Consult the Java documentation and explain how to choose which to use.

Name: _____

If you have extra time…

13. Examine `ConcurrencyPrimitive.java`, which concurrently increments an integer variable 1000 times. Predict the resulting counter value.

14. Run the program to test your hypothesis. What happens?

15. Compile the program with the command `javac ConcurrencyPrimitive.java`, then disassemble it with `javap -c -p ConcurrencyPrimitive`. Copy down the instructions in the `increment()` method, ignoring the `return` instruction.

16. Translate the instructions into three lines of pseudocode.

    a.

    b.

    c.

17. Recall that concurrent execution means that two tasks can interleave. That is, one gets preempted at an arbitrary point in its execution, and another runs while it is paused. Looking at the instructions, why is it unsafe to execute `increment()` concurrently?

In a concurrent program, we must consider another property called **atomicity**. An operation is atomic if it happens all at once; that is, if no concurrent task can interleave partway through it.

18. Which expression in the Java source code is not atomic? What makes it non-atomic?