Name: _____

IB Computer Science 2
Virtual memory worksheet

Download `UnsafeExtension.zip` to the folder where you usually write activity code, and extract it. To make macOS trust the library, run the command:
`xattr -d com.apple.quarantine libUnsafeExtension.dylib`

Memory addresses

The first two lines of the `Addr.java` program find and display the memory address of a string literal. Run it using the following command:
`java --add-exports java.base/jdk.internal.misc=ALL-UNNAMED Addr`

Let it sit at the `Hit enter to continue...` prompt. (Do not enter anything.)

Now split your terminal and run the same command again, so that you can see both running copies at the same time.

1. Compare the memory addresses. If they are different, hit enter and rerun the program. Repeat as necessary. What do you eventually notice?

2. Do you think it is safe for two running programs to share the same memory addresses? Why or why not?

3. With the memory addresses matching, instead of merely hitting enter at the prompt, enter some text of your choice. What happens to the program's final `println()`? Why?

4. Predict what would happen if you now hit enter at the other running copy.

5. Test your prediction. What have you learned about memory addresses?

Virtual memory

We just saw that memory addresses are remappable! Programs only see virtual addresses, which might correspond to completely different physical addresses in RAM. This feature of modern computers is called **virtual memory**, and is managed by the operating system. (For performance reasons, the processor has a *memory management unit* that translates virtual addresses into physical ones, according to the mappings configured by the operating system).

One consequence of virtual memory is that one running program cannot access (or even address) memory belonging to any other program. Another is that the operating system can play some tricks when allocating memory to programs.

The `Bench.java` program is a benchmark that measures the time required to request and traverse large memory allocations. To avoid measuring the Java runtime environment instead of the virtual memory system, it does not use arrays. Instead, it allocates memory from two low-level sources: first from the C language library, then directly from the operating system.

Run the program using the command:
`java --add-exports java.base/jdk.internal.misc=ALL-UNNAMED Bench.java`
Check that the two *total* times are similar (within tens of thousands of microseconds). If not, rerun until they are.

6. Look at the time taken to *allocate* memory from the two sources. Which allocation performs more work?

7. Look at the time taken to *traverse* the memory from the two sources. Which memory is more expensive to write to? Why do you suppose this is?

8. Predict what would happen if the program traversed the second allocation a second time.

9. Test your prediction by copying and pasting the second loop and printing its measurement. What do you notice? What have you learned about accessing memory?

Name: _____

Unused memory

We just saw that, thanks to virtual memory, the operating system can postpone allocation *until the program actually accesses the memory it has requested*!

10. Examine the program `UnusedJava.java`, which creates numerous large Java arrays. How much total memory does the program allocate? What do you expect will happen?

11. Test your prediction by running the program. Then run `UnusedC.java`, which instead allocates memory from the C library. What do you notice?

12. Examine the program `UnusedOS.java`, which allocates memory directly from the operating system. What do you expect will happen?

13. Test your prediction by running the program. What happens and why?

Paging

Virtual memory also gives the operating system options when the system is low on RAM. Start the `Paging.java` benchmark using the command:

```
java --add-exports java.base/jdk.internal.misc=ALL-UNNAMED
Paging.java
```

14. What do you notice happening as the program writes to more of the allocations?

15. What do you notice when the program finishes writing and switches to only reading? (If it refuses to make it this far, you may need to reduce the number of allocations.)

16. Notice that the program is successfully retrieving the values it saved. Where must all this data be stored? Once you have a prediction, use the macOS Activity Monitor to check.

We just caught the operating system **paging** out memory to disk. This is another benefit of virtual memory: when the system is low on memory, the operating system can unmap certain virtual addresses and move the underlying data from RAM to the SSD. If the program later accesses those addresses, the operating system moves the data back and possibly pages out other data instead.

17. The CPU does not support remapping an individual virtual memory address; instead, the mappings are in units called *pages*. The number of bytes per page determines the minimum data the operating system must transfer when paging. Check your system's page size with the command `sysctl -a | grep -F vm.pages`

If you have extra time…

In a new program `ExtraTime.java`, add the line

```
long address = UnsafeExtension.mapFileBackedMemory("ExtraTime.java");
```

18. Print the expression `(char) Unsafe.getUnsafe().getByte(address)`. You will need to `import jdk.internal.misc.Unsafe` and run with `--add-exports` as before. What do you see?

19. Write a loop that walks forward one byte at a time until it reads a 0, printing each byte as a character. What data is present in memory starting at `address`?

20. In your loop, call `Character.toLowerCase()` on each byte that you read from the file, then pass the result to `Unsafe`'s `.putByte()`. What happens? Describe how the phenomenon you just observed relates to paging, and what is different compared to the paging caused by the last program.

(Adapted from an activity co-authored for CMU 15-213 Introduction to Computer Systems)