CS/240/Lab/5

This lab has two major goals:

- Experimenting with the differences between Java and C.
- Writing working Makefiles.

We provide a set of Java programs, and you will reimplement them in C, meanwhile exploring some of the major differences between the two languages. If you are not familiar with Java, talk to a lab TA. Since one of the major goals of this lab is writing correct Makefiles, the command lines for compiling your C code will *not* be provided.

Refresher: Compiling Java

The command to compile a Java file (in this example, Hello.java) is:

```
javac Hello.java
```

The command to run a compiled Java program (in this example, Hello.java) is:

```
java Hello
```

These commands are general, we will be providing more specific ones for the programs below.

Since you are turning in Makefiles along with your C code, you will be turning in tar archives (often called "tarballs"), instead of turning in your C code directly. The `.tgz` files we've provided for various labs are examples of tar archives.

Making tar archives

The command to create a tar archive `turnin.tgz` from the files `Makefile`, `hello.c` and `hello.h` is:

```
tar zcf turnin.tgz Makefile hello.c hello.h
```

You can generalize this command to any number of files, just remember that the name of the tar archive to create comes first, after `tar zcf`.

Q1: Memory consumption

C gives the programmer fine-grained control over the size of objects in memory. As a result, it is often possible to handle more data in C than it is in Java. However, by *not* giving the user direct control over object sizes, Java can sometimes arrange them more cleverly than the programmer would, so in some cases it's possible to handle more data in Java than it is in C! To demonstrate this fact, you will create a very long linear linked list in both Java and C, and determine which runs out of memory first.

Both the Java and C programs will output one line for each member they add to the linked list, until they run out of memory and exit or crash.

Here is the Java code for you to reimplement in C, as a single C file `memory.c`:

ListMain.java:

```
class ListMain {
    public static void main(String[] args) {
        List cur = new List(null);
        while (true) {
            cur = new List(cur);
            if ((cur.value % 100000) == 0) {
                System.out.println(cur.value);
            }
        }
    }
}
```

List.java:

```
class List {
    public List(List next) {
        this.next = next;
        if (next == null) {
            this.value = 0;
        } else {
            this.value = next.value + 1;
        }
    }

    public List next;
    public long value;
}
```

Java will throw an exception when `new` fails. C's `malloc` will return `NULL` when it fails. Make sure you check for this case and stop the program gracefully, rather than causing a segmentation fault. You can use `exit` for this purpose: Make sure you include `stdlib.h`, then call `exit(0)` to exit immediately.

If you run these programs without restricting their memory, you may find that your system becomes unresponsive. Don't do this!

Restricting memory

To make this a fair fight, we must restrict both Java and C to the same amount of memory, and Java to not use memory compression. We will choose 500MB for this purpose. To restrict Java to 500MB, use the following command line (after compiling ListMain.java and List.java):

```
java -XX:-UseCompressedOops -Xmx500m ListMain
```

To restrict C to 500MB, use the following (arcane) command line:

```
sh -c 'ulimit -v 512000 ; ./memory'
```

Which program consumes less memory? That is, which is able to create more nodes in the list? How many more?

Q2: Performance

Although managing types and memory at C's low-level is often more difficult and error-prone than in Java, C is usually faster. Some types demonstrate the speed difference more than others; for instance, arrays are usually slower in Java than in C. Translate the following Java program into a C program named `perf.c`, then use the UNIX `time` command to compare their performance:

Arrays.java:

```
class Arrays {
    public static final int count = 10000000;

    public static void main(String[] args) {
        int i, rep, res;
        int[] ints = new int[count];
        for (i = 0; i < count; i++) {
            ints[i] = i;
        }
        res = 0;
        for (rep = 0; rep < 10; rep++) {
            for (i = 0; i < count; i++) {
                res += ints[i];
            }
        }
        System.out.println(res);
    }
}
```

Note that every `new` in Java should become a `malloc` in C. Java arrays are more similar to C pointers and `malloc` than C arrays.

The `time` command

To time how long a program takes to execute, use the `time` command. For instance, to time this Java program, use:

```
time java Arrays
```

To time your C program, use:

```
time ./perf
```

You can prefix `time` to any command to measure its runtime.
`time` reports several measures of the time, the one we're interested in is "real time".

Java will perform many optimizations. By default, `gcc` will perform none. To compile your C program with optimizations, add `-O2` to the `gcc` command line.

Which program is faster? Does using optimizations with `gcc` help? Why do you think one is faster than the other?

Q3: Makefile

You will only be turning in a single archive for this assignment. It must contain a `Makefile`, `memory.c` and `perf.c`. The Makefile will build the programs for both Q1 and Q2 above.

Your Makefile must support and properly build the following targets:

- `all`
- `clean`
- `memory.o`
- `perf.o`
- `memory`
- `perf`

The `all` target should be default, and should build everything. The `clean` target should remove all produced executables and `.o` files.

Your `Makefile` must also support the variables `CC` and `CFLAGS`. All non-pseudo targets should use these variables.

Turning in

Go to the submission web page, under "currently open projects" will be listed "lab 5". Click it, use the file selector to choose the tar archive you've made, then click "submit".

Lab is due Monday, February 20th before midnight. No late labs accepted.

Grading criteria

Memory

- source file is named `memory.c`
- code compiles
- code runs without error
- output is correct
- program consumes memory as described

Performance

- source file is named `perf.c`
- code compiles
- code runs without error
- output is correct

Makefile

- named `Makefile`
- correct syntax
- builds `memory` and `perf` properly and independently of each other
- builds `.o` files and links them into executables (not executables directly from `.c` files)
- all dependencies are correct (changes will cause rebuilds)
- `all` pseudo-target is default and works correctly
- `clean` pseudo-target cleans properly
- `CC` variable is used correctly in all cases
- `CFLAGS` variable is used correctly in all cases