Computer Science 2211b
Software Tools and Systems Programming
Winter 2024

Assignment 5 (8%)

Due: Monday April 08 at 11:55 PM

Objectives

In this assignment, you will consolidate what you've learned in this course by writing a more complex C program that involves:

- Working with a provided Abstract Data Type (stack)
- Reading from a file
- Working with C standard library functions -- some that you have seen and some that you will need to look up
- Further developing your skills working with arrays, pointers, structs, and dynamically-allocated memory
- Following style conventions and good programming practices

Implementing MiniJVM

A program written in Java is compiled into a machine-independent language called *bytecode*, output to a binary file with the extension `.class`. A Java Virtual Machine on your computer then interprets the bytecodes in the `.class` file, thereby executing the program on your machine.

A real `.class` file contains a lot of information in addition to the bytecodes for the methods in the file (e.g. header information, class inheritance information, table of constants, multiple methods, etc.).

In this assignment, we will be using mini versions of `.class` files that we will call `.mclass` files. Our `.mclass` files will contain only bytecodes, and only bytecodes for one method. However, the format of the bytecodes is the same as that found in real `.class` files. You will be writing part of a Java Virtual Machine (JVM) in C to read `.mclass` files into memory and execute them.

In CS 2208, Western CS students used to implement this assignment in SPARC assembly language, so implementing it in C should be a breeze, comparatively speaking!

Assignment Skeleton

We are providing an assignment skeleton that you will fill in. The files are available on compute.gaul.csd.uwo.ca in `/data/project/cs2211/asn5`.

Copy the contents of this directory into your `~/courses/cs2211/asn5` directory:

```
$ mkdir -p ~/courses/cs2211/asn5
$ cp -r /data/project/cs2211/asn5/* ~/courses/cs2211/asn5/
```

The skeleton is organized as follows:

`include`	This directory contains the header files that your program will include. These files are commented extensively and you should take time to examine them. Do not modify any header files in this directory -- your changes will be overwritten when we mark your code.
`lib`	This directory contains a provided library `libadt.a` that contains the stack ADT that MiniJVM will use. You will link your code with this library. A `.a` file is a *static library*: a collection of `.o` files
`minijvm.c`	You will implement MiniJVM in this file
`testfiles`	This directory contains files used to test your MiniJVM. More later.

Compiling Your Code

To compile `minijvm.c`, tell GCC to look in the `include` directory to find header files:

```
$ gcc -I include -c minijvm.c
```

This creates the object file `minijvm.o`. To create your executable, you must then link your object file with the provided library:

```
$ gcc -L lib -o mjava minijvm.o -ladt
```

This tells GCC to create an executable `mjava` by linking `minijvm.o` with the provided `adt` library. `-L lib` tells GCC to look for the `adt` library in the `lib` directory.

Note that the order here is important: `-ladt` must come at the end.

Understanding the Components of MiniJVM

The file `include/minijvm.h` contains the `minijvm` struct that you will use:

```
typedef struct
{
    char* bytecode;   // Bytecode array: program being executed
    char* pc;         // Program counter: points to next bytecode
    stack operands;   // Operand stack
    int locals[10];   // Local variables
    int return_value; // Return value: 0 = success.  1 = error
} minijvm;
```

- The `bytecode` member is a pointer to a dynamically-allocated array of signed bytes (i.e., `char`) containing the bytecode of the program being executed.

- The `pc` member is the *program counter*: a pointer to the next instruction in the `bytecode` array to be executed.

- The `operands` member is the operand stack used by the JVM. As we will discuss in class, the JVM is a stack-based machine. For instance, to add two operands, we push their values onto the operand stack and then issue the `iadd` bytecode. This pops the two operands off the stack, adds them, and pushes the result back onto the stack.

  The `stack` ADT has been provided for your use in `lib/libadt.a`. See `include/stack.h` for the functions you have available to you.

- The `locals` member is an array of 10 integers that a MiniJVM program can use for local variables.

- The `return_value` member stores the result of a MiniJVM program, which should be returned to the operating system from `main`. `0` = success, `1` = failure.

MiniJVM Algorithm

The general algorithm used by a JVM is as follows:

```
// Read all bytecodes into an array
// Point the program counter at the first bytecode

while (true)
{
    // Get next bytecode b pointed to by the program counter
    // (interpret it as an unsigned char since bytecode instructions
    // are unsigned -- that is, cast it)

    switch (b)
    {
        case INST_ILOAD:
            // Call function to execute iload bytecode
            break;
        …
        case INST_RETURN:
            // Stop execution and return to the operating system
            break;
    }

    // Update the program counter to point at the next bytecode
}
```

Note that performing the action for a particular bytecode may involve:

- Reading additional byte(s) from the bytecode array
- Pushing/popping values to/from the operand stack
- Storing/loading values to/from the local variables array

Example Program

You are provided with a number of `.mclass` files in the `testfiles` directory, along with their source code in `.mjava` files. Your program will run the `.mclass` files -- the `.mjava` files are provided so that you can trace the code to determine what each `.mclass` file is supposed to do.

For example, `test1.mclass` contains the following bytes:

```
16  5  16  -3  96  187  177
```

Its corresponding source code can be found in `test1.mjava`. We can interpret these bytes as follows:

Bytecodes	Instruction / Operands	Description
`16 5`	`bipush 5`	Push `5` onto the operand stack
`16 -3`	`bipush -3`	Push `-3` onto the operand stack
`96`	`iadd`	Pop and add the top two elements on the operand stack; push the result back onto the stack
`187`	`iprint`	Print the top element of the operand stack
`177`	`return`	Exit the program

Bytecode Summary

You will be implementing the following 17 bytecodes. While there are quite a few of them, many of them are *very* similar. For instance, `iadd`, `isub`, and `imul` are nearly identical, largely differing only in the arithmetic operator to use. The same is true for `idiv` and `irem`.

Each bytecode has a corresponding constant defined in `minijvm.h` that you can use.

Mnemonic	Byte	Arguments	Description
`iconst_0`	`3`		Push `0` onto the operand stack
`pop`	`87`		Pop the top item off the stack (discard it)
`dup`	`89`		Duplicate the top item on the stack
`iadd`	`96`		Pop and add the top two items on the stack; push the result back on the stack
`isub`	`100`		Pop and subtract the top two items on the stack: `(second from top) - (top)` Push the result back on the stack

`imul`	`104`		Pop and multiply the top two items on the stack; push the result back on the stack
`idiv`	`108`		Pop and integer divide the top two items on the stack: `(second from top) / (top)` Push the result back on the stack
`irem`	`112`		Pop the top two items on the stack and compute the remainder of integer division: `(second from top) % (top)` Push the result back on the stack
`ishr`	`122`		Arithmetic shift right: pop the top two items off the stack and shift right as follows: `(second from top) >> (top)` Push the result back on the stack
`return`	`177`		Return from method (exit the program)
`iprint`	`187`		Print the top item on the stack (without popping it off the stack). Not a real bytecode
`bipush`	`16`	`b`	Push signed byte `b` onto the stack (-128 ≤ `b` ≤ 127)
`iload`	`21`	`n`	Push local variable at index `n` onto the stack
`istore`	`54`	`n`	Pop the top of the stack and store it in local variable at index `n`
`iinc`	`132`	`n d`	Increment local variable at index `n` by `d`
`ifeq`	`153`	`offset`	Pop the top item off the stack; branch to `offset` if it is equal to zero. `offset` is a signed **2-byte** value stored in big-endian format (the most significant byte comes first in the bytecode array)
`goto`	`167`	`offset`	Branch to `offset`. `offset` is a signed **2-byte** value stored in big-endian format (the most significant byte comes first in the bytecode array)

For the `ifeq` and `goto` instructions, the jump offset is specified in the two bytes following the instruction bytecode. This offset is stored in big-endian format: the most significant byte (MSB) comes first in the bytecode array. To calculate the actual offset value, you need to combine these two bytes into a single integer using bitwise operations.

For example, if the two bytes after the `ifeq` or `goto` instruction are `01` and `02`, these are combined into the single integer `258` (`0000 0001 0000 0010`).

You then add the offset to the address of the `ifeq` or `goto` instruction (not the address of an offset byte) to determine the new position of the program counter.

Part 1: Writing a Makefile (10%)

Your submission must contain a file `Makefile` that builds your code. This will also help you build and test your code, which is why we are starting with it.

To build your code, we should be able to type:

```
$ make
```

This must create an executable in the `asn5` directory called `mjava`.

We should also be able to type:

```
$ make clean
```

This should remove all `.o` files along with the `mjava` executable.

We should be able to type:

```
$ make test1

$ make test2

.

.

$ make test8
```

Each of these targets should run your `mjava` executable on `testfiles/testN.mclass`

For example, if we type:

```
$ make test1
```

Your `Makefile` should run:

```
./mjava testfiles/test1
```

Finally, we should be able to type:

```
$ make test
```

This should run all tests in order (`test1` through `test8`).

For each target in your `Makefile`, be sure that you are using the appropriate dependencies.

Part 2: Building MiniJVM (90%)

Complete `minijvm.c` to implement MiniJVM. The minimum functions to implement are as follows:

```
char* jvm_read(const char* filename)
```

Purpose	Reads the bytes from the specified file into an array
Parameters	• `filename`: Name of the `.mclass` file from which to read (**without** the `.mclass` extension)
What to Do	• Compute the target filename (`filename` + `".mclass"`) • If it doesn't exist, print an error and exit with `EXIT_FAILURE` • Dynamically allocate an array of bytes to store the bytecode. A file may contain up to `MAX_CLASS_SIZE` bytes • Open the specified file • Read it byte-by-byte into the array until `EOF` is reached • Close the file
Returns	A pointer to the array of bytecode
Note	• Look up the functions `fopen`, `fread`, `fclose`, and `stat` • `stat` could be used to determine if the file doesn't exist

```
minijvm* jvm_init(const char* filename)
```

Purpose	Initializes a new `minijvm` struct
Parameters	• `filename`: Name of the `.mclass` file from which to read (**without** the `.mclass` extension)
What to Do	• Dynamically allocate a new `minijvm` struct • Read the file and store the bytecode array in the struct • Set the program counter to point at the first bytecode in the array • Create the operand stack (see `stack.h` for the function to use)
Returns	A pointer to the initialized `minijvm`

```
void jvm_free(minijvm* jvm)
```

Purpose	Frees all memory allocated for the MiniJVM
Parameters	• `jvm`: The `minijvm` struct to deallocate
What to Do	• Deallocate all memory dynamically allocated for the MiniJVM
Note	Don't forget to free memory even if you exit early due to an error

`void jvm_run(minijvm* jvm)`

Purpose	Runs the program loaded into the MiniJVM
Parameters	• `jvm`: The `minijvm` initialized with the bytecode to run
What to Do	• In an infinite loop ○ Get the next bytecode pointed to by the program counter ▪ Interpret each bytecode as an `unsigned char` ○ Call a helper function to execute it ▪ If an error occurs in your helper function, set the `return_value` member of `jvm` to `1` and stop execution ▪ If the bytecode is `return`, set the `return_value` member of `jvm` to `0` and stop execution ○ Update the program counter to point to the next bytecode ▪ Note that some bytecodes take arguments in the bytecode array ▪ Your helper functions may also modify the program counter as needed
Note	This will likely be a big `switch` statement, but the code to execute the bytecodes **must** be implemented in helper functions. There are other ways to implement this rather than `switch`. For example, you could implement a lookup table with pointers to functions for each bytecode. However, keep things simple unless you're looking for a challenge.

`int main(int argc, char** argv)`

Purpose	The main entry point to MiniJVM
Parameters	• `argc`: Number of parameters passed to the program • `argv`: Array of parameters passed to the program. `argv[0]` is always the name of the program. `argv[1]` is the first parameter
What to Do	• Call functions to initialize MiniJVM, run the program, and free dynamically allocated memory • If an incorrect number of arguments is passed, call `usage`
Returns	• `0`, upon successful termination of an `.mclass` file • `1`, if any error occurs in the program

Sample Output

Trying to run a file that does not exist:

```
$ ./mjava nonexistent
File 'nonexistent.mclass' not found
$ echo $?
1
```

`$?` Is a shell variable that stores the status code returned by the last command. You can `echo` this to check that your program is returning the correct values. Observe that the program returns `1` upon an error.

Successfully running a program:

```
$ ./mjava testfiles/test1
2
$ echo $?
0
```

Observe that the program returns `0` upon successful termination.

Running a program that has a division by zero error:

```
% ./mjava testfiles/test3
2
Division by zero
$ echo $?
1
```

Observe that the program returns `1` since an error occurred.

Running the program with invalid arguments:

```
$ ./mjava
Usage: mjvm FILENAME
$ echo $?
1
```

IMPORTANT: We will be using automated tests to grade your assignments and get them returned to you as soon as possible. If you do not wish to have marks deducted, it is important that your output is **exactly** as shown.

A Word on Memory Leaks

Don't forget to free the memory allocated for your MiniJVM before the program exits. If your program has a memory leak, marks will be deducted. Fortunately, there's a helpful program called `valgrind` that you can use to check for memory leaks. Here's an example of what `valgrind` reports when you have a memory leak:

```
$ valgrind ./mjava test/test1

==9006== Memcheck, a memory error detector
.
.
.
==9006== LEAK SUMMARY:
==9006==    definitely lost: 72 bytes in 1 blocks       <---
==9006==    indirectly lost: 1,056 bytes in 4 blocks
==9006==      possibly lost: 0 bytes in 0 blocks
==9006==    still reachable: 0 bytes in 0 blocks
==9006==         suppressed: 0 bytes in 0 blocks
```

And here's what it reports when you *don't* have a memory leak:

```
$ valgrind ./mjava test/test1
==9090== Memcheck, a memory error detector
.
.
.
==9090== All heap blocks were freed -- no leaks are possible
```

Test Files

The `testfiles` directory contains eight `.mclass` files with which you can test your program. Human-readable versions are available in eight corresponding `.mjava` files. You should trace the code in these files to ensure you understand what the code is doing and what your program should output.

Submitting Your Assignment

Please note that no exceptions will be made for this assignment if it is not submitted correctly. You only need to enter 5 Git commands, which are provided below. As university students, I trust you can manage this. Make sure to follow the instructions carefully.

Your programs must follow the **CS 2211 Style Guide** introduced in Topic 5.

After completing this assignment, your `asn5` directory should look as follows:

```
$ tree ~/courses/cs2211/asn5
[redacted]
├── accommodation.txt       (if you requested academic accommodation)
├── Makefile
├── include
│   ├── minijvm.h
│   └── stack.h
├── lib
│   └── libadt.a
├── minijvm.c
└── testfiles
    ├── test1.mclass
    ⋮
    └── test8.mjava

3 directories, 21 files
```

If not, go back and ensure that your directory matches this structure.

Be sure to run `make clean` before submitting!

For full details on submission, see the **Assignment Submission Instructions** from the course web site. As a quick overview, submitting will involve the following commands:

```
cd ~/courses/cs2211/asn5
git add .
git commit -m "Submitting assignment 5"
git push
git tag asn5-submission
git push --tags
```

[redacted]