Project 12

An Operating System (OS) is a collection of software services, designed to close gaps between the computer's hardware and software hierarchy. For example, if you use a high-level language to write a program that prompts the user to enter some data using the keyboard, and you then plot this data on the screen, the code generated by the compiler will include (among other things) calls to OS routines that handle keyboard inputs and graphics outputs. In this last project in the Nand to Tetris journey we will implement such an OS. The Jack OS consists of eight .Jack class files, each realizing a library of commonly used OS services such as input / output drivers, math operations, string processing, and memory management.

Objective

Implement the operating system described in the lecture.

Contract

Implement the operating system in Jack, and test it using the programs and testing scenarios described below. Each test program uses a subset of OS services. Note: Each one of the eight OS classes can be implemented and unit-tested in isolation, and in any order.

Resources

The main required tool is Jack – the language in which you will develop the OS. You will also need the supplied Jack compiler, for compiling your OS implementation, as well as the supplied test programs, also written in Jack. Finally, you'll need the supplied VM emulator, which is the platform on which all the tests will be executed.

OS API: Here is the Jack OS API, and the list of OS error codes and their meaning.

Your projects/12 folder includes eight skeletal OS class files named Math.jack, String.jack, Array.jack, Memory.jack, Screen.jack, Output.jack, Keyboard.jack, and Sys.jack. Each file contains the signatures of all the class subroutines. Your task is completing the missing implementations.

The VM emulator: Operating system developers often face the following chicken and egg dilemma: How can we possibly test an OS class in isolation, if the class uses the services of other OS classes, not yet developed?

As it turns out, the VM emulator allows doing just that, since the emulator’s software includes a “builtin” implementation of the OS. During program execution, whenever a VM command "call foo" is found in the loaded VM code, the emulator proceeds as follows. If a VM function named foo exists in the loaded code base, the emulator executes its VM code. Otherwise, the emulator checks if foo is one of the built-on OS functions. If so, it executes foo's built-in implementation. This convention is ideally suited for supporting the testing strategy that we now turn to describe.

Testing Plan

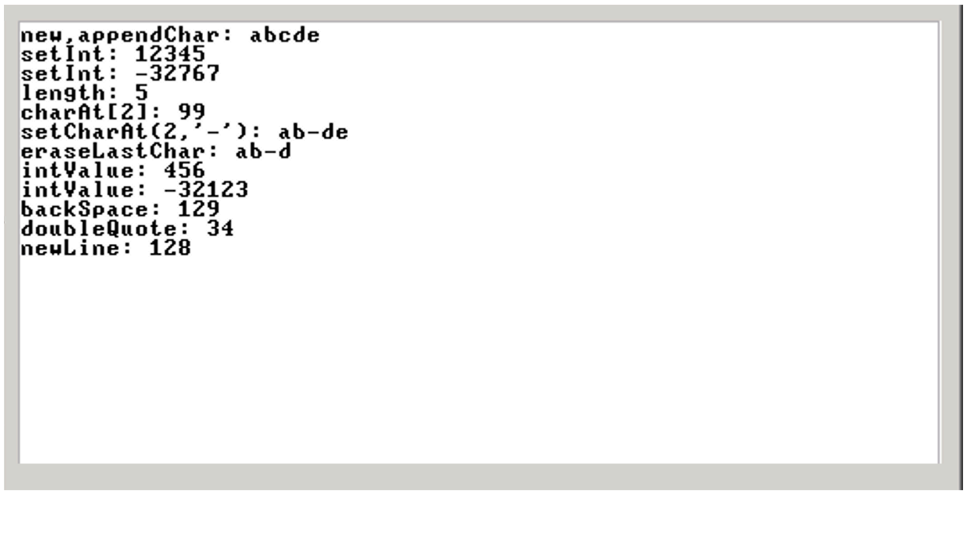
Your projects/12 folder includes eight test folders, named MathTest, MemoryTest, ..., etc., for testing each one the eight OS classes Math, Memory, ..., etc. Each folder contains a Jack program, designed to test (by using) the services of the corresponding OS class. Some folders contain test scripts and compare files, and some contain only .jack files. To test your implementation of the OS class Xxx.jack, proceed as follows:

1. Inspect the supplied XxxTest/*.jack code of the test program. Understand which OS services are tested, and how they are tested.
2. Put your OS class Xxx.jack in the XxxTest folder;
3. Compile the folder using the supplied Jack compiler. This will result in translating both your OS class file, as well as the .jack file or files of the test program, into corresponding .vm files, stored in the same folder.
4. If the folder includes a .tst test script, load the script into the VM emulator. Otherwise, load the folder into the VM emulator;
5. Follow the specific testing guidelines given below for each OS class.

Memory, Array, Math: The three folders that test these classes include test scripts and compare files. Each test script begins with the command "load". This command loads all the .vm files in the current folder into the VM emulator. The next two commands in each test script create an output file, and load the supplied compare file. Next, the test script proceeds to execute several tests, comparing the test results to those listed in the compare file. Your job is making sure that these comparisons end successfully.

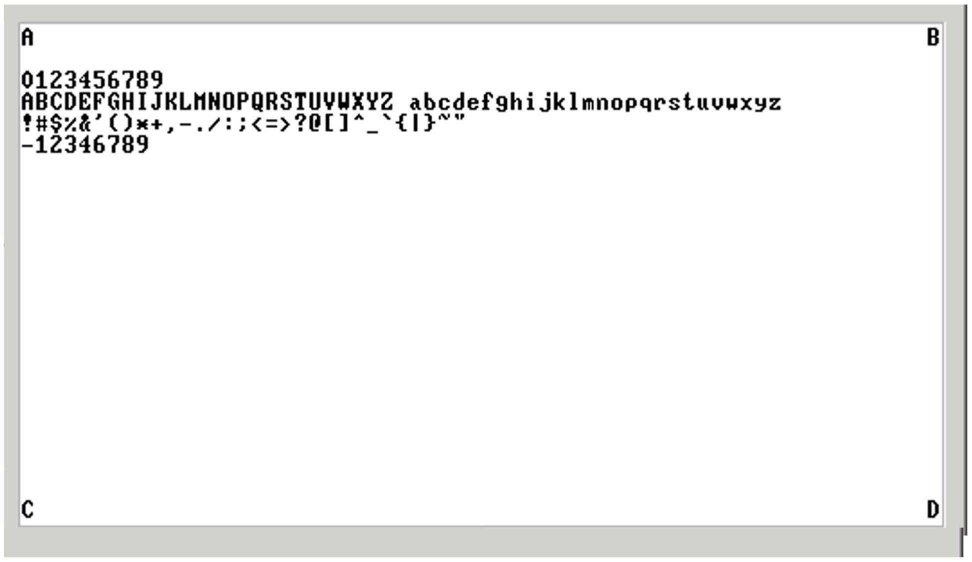
Note that the supplied test programs don't comprise a full test of Memory.alloc and Memory.deAlloc. A complete test of these memory management functions requires inspecting internal implementation details not visible in user-level testing. If you want to do so, you can test these functions using step-by-step debugging, and inspecting the affected locations in the host RAM.

String: Execution of the supplied test program should yield the following output:



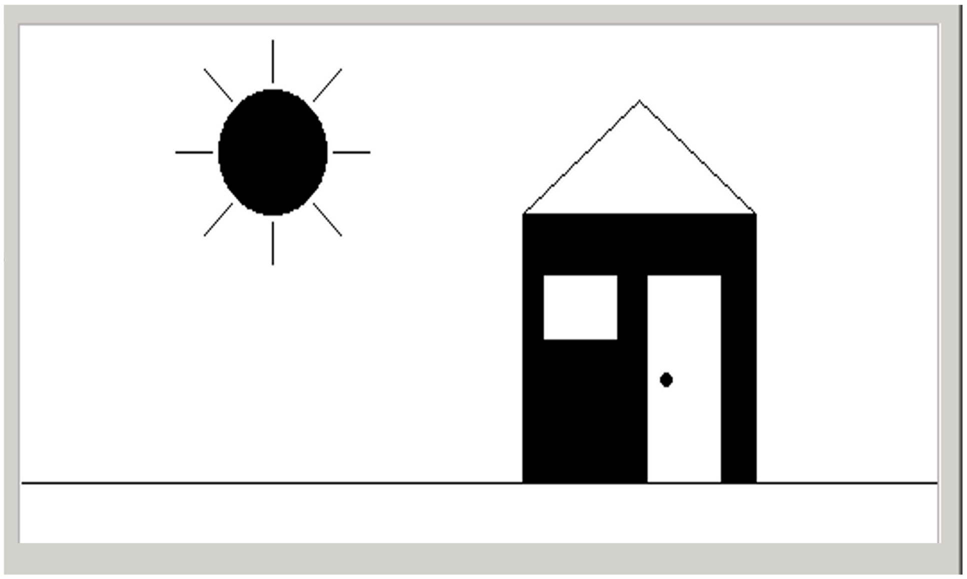
```
new,appendChar: abcde
setInt: 12345
setInt: -32767
length: 5
charAt[2]: 99
setCharAt(2,'-'): ab-de
eraseLastChar: ab-d
intValue: 456
intValue: -32123
backSpace: 129
doubleQuote: 34
newLine: 128
```

Output: Execution of the supplied test program should yield the following output:

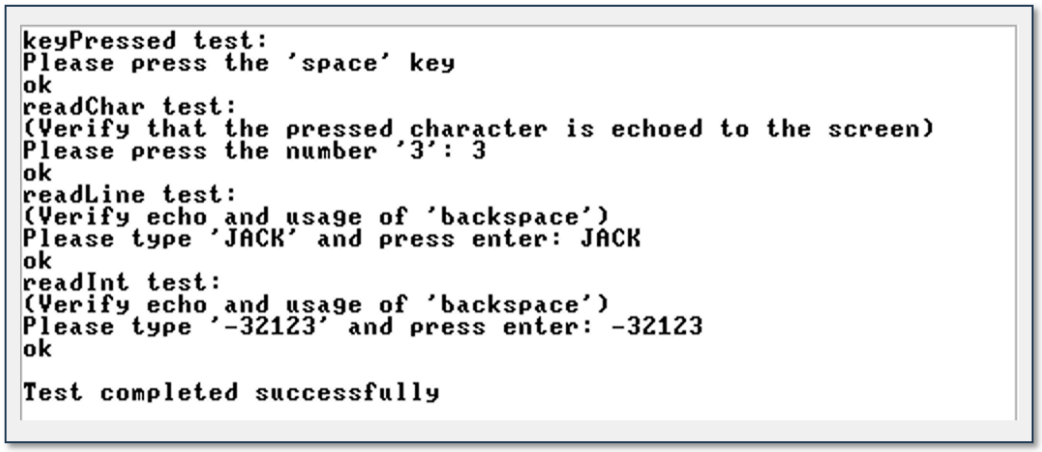


```
A                                                                      B

0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
!#$%&'()*+,-./:;<=>?@[]^_`{|}~"
-12346789




C                                                                      D
```

Screen: Execution of the supplied test program should yield the following output:



Keyboard: This OS class is tested by a test program that affects some user-program interaction. For each function in the Keyboard class (keyPressed, readChar, readLine, readInt) the program requests the user to press some keys. If the OS function is implemented correctly and the requested keys are pressed, the program prints "ok" and proceeds to test the next OS function. Otherwise, the program repeats the request. If all requests end successfully, the program prints "Test ended successfully". At this point the screen should look as follows:



```
keyPressed test:
Please press the 'space' key
ok
readChar test:
(Verify that the pressed character is echoed to the screen)
Please press the number '3': 3
ok
readLine test:
(Verify echo and usage of 'backspace')
Please type 'JACK' and press enter: JACK
ok
readInt test:
(Verify echo and usage of 'backspace')
Please type '-32123' and press enter: -32123
ok

Test completed successfully
```

Sys

The supplied .jack file tests the Sys.wait function. The program requests the user to press a key (any key), and then waits two seconds, using a call to Sys.wait. It then prints a message on the screen. Make sure that the time that elapsed from the moment the pressed key was released until the message was printed is about two seconds.

The Sys.init function is not tested explicitly. However, this function performs all the necessary OS initializations and then calls the Main.main function of each test program. Therefore, we can assume that nothing would work properly unless Sys.init is implemented correctly.

Complete test

After testing successfully each OS class in isolation, test your entire OS implementation using the Pong game introduced in earlier projects. The source code of Pong is available in projects/11/Pong. Put your eight OS .jack files in the Pong folder, and compile the folder using the supplied Jack compiler. Next, load the Pong folder into the VM emulator, execute the game, and verify that it works as expected. If for some reason you've implemented only some of the OS classes, you can still do this integrated test: Whenever the VM emulator will handle a call to a missing OS function, it will invoke instead the builtin implementation of that function.