

Project 12: The Operating System

Background

An Operating System (OS) is a collection of software services, designed to close gaps between the computer's hardware and application software. For example, if you use a high-level language to write a program that prompts the user to enter some data using the keyboard, the code generated by the compiler will include (among other things) calls to OS routines that handle keyboard inputs. Therefore, the OS can also be viewed as an extension of a high-level programming language. This is a rather simplistic view of operating systems, but it will suffice for our purpose.

Objective

Implement the Jack OS specified in Chapter 12. In the process of building this OS, you will implement a modular collection of some beautiful algorithms in applied computer science. Thus, you will also be exposed, hands-on, to some cool computer science stuff.

Contract

Write a Jack OS implementation and test it using the programs and testing scenarios described below.

Resources

Chapter 12 includes the [Jack OS API](#), which has to be implemented in this project. The main tool needed is Jack, the high-level programming language with which you will build the OS (just like Unix is built in C). You will also need the supplied Jack compiler, to compile your OS implementation as well as the supplied test programs, and the supplied VM Emulator, to run and test the compiled code.

Testing

The OS is implemented as a collection of 8 Jack classes. Each class can be implemented and unit-tested in isolation, and in any desired order. To develop, compile, and test each `OSClass.jack` class in isolation, follow this procedure:

1. Put the `OSClass.jack` that you are developing in the same directory that includes the supplied test program designed to test it;
2. Compile the directory using the supplied Jack compiler. This will result in compiling your `OSClass.jack` as well as the supplied test class files. In the process, a new `OSClass.vm` file will be created;
3. Load the directory into the supplied VM Emulator;
4. Execute the code and check if the OS services are working properly, according to the guidelines given below.

Recall that the supplied VM Emulator features a built-in implementation of the entire Jack OS. With that in mind, the rationale of the above procedure is as follows. Normally, when the supplied VM Emulator encounters a call to an OS function, it handles the call by invoking a built-in implementation of that function. However, if the compiled directory contains a `.vm` file that includes a VM implementation of the function, this implementation will be executed, short-cutting the built-in implementation. This practice follows the reverse engineering spirit of GNU Unix and Linux: it allows you to build and test different OS modules in isolation, as if

all the other OS modules are implemented properly and operating side-by-side with the currently-developed module. That's important, since the OS class that you are presently developing may well include calls to the services of other OS classes.

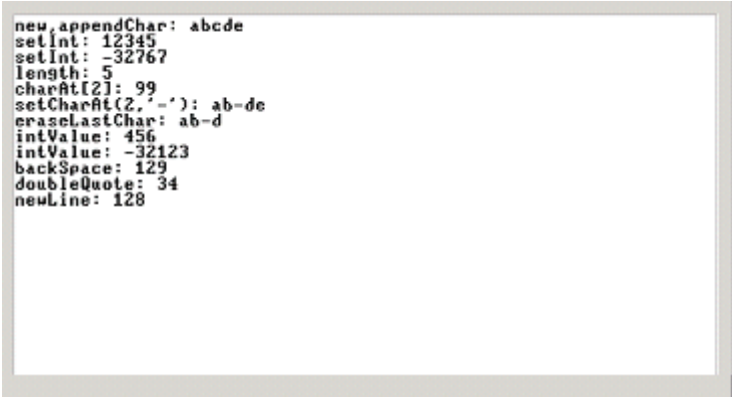
OS Classes and Test Programs

There are eight OS classes: Memory, Array, Math, String, Screen, Keyboard and Sys. For each OS class Xxx we supply a skeletal Xxx.jack class file with all the required subroutine signatures, corresponding test class named Main.jack, and related test scripts.

OS Module	Test Programs	Guidelines
<small>OS Module</small>	<small>Test Programs</small>	<small>Comments</small>
Memory.jack	Main.jack MemoryTest.tst MemoryTest.cmp	To test your implementation of this OS class, compile the directory, execute the test script on the supplied VM Emulator, and make sure that the comparison with the compare file ends successfully.
<small>OS Module</small>	<small>Test Programs</small>	<small>Comments</small>
Array.jack	Main.jack ArrayTest.tst ArrayTest.cmp	Same guidelines.
<small>OS Module</small>	<small>Test Programs</small>	<small>Comments</small>
Math.jack	Main.jack MathTest.tst MathTest.cmp	Same guidelines.

The test programs supplied above don't entail a full test of the Memory.alloc and Memory.deAlloc functions. A complete test of these memory management functions requires inspecting internal implementation details not visible in user-level testing. This can be done via step-by-step debugging in the supplied VM Emulator.

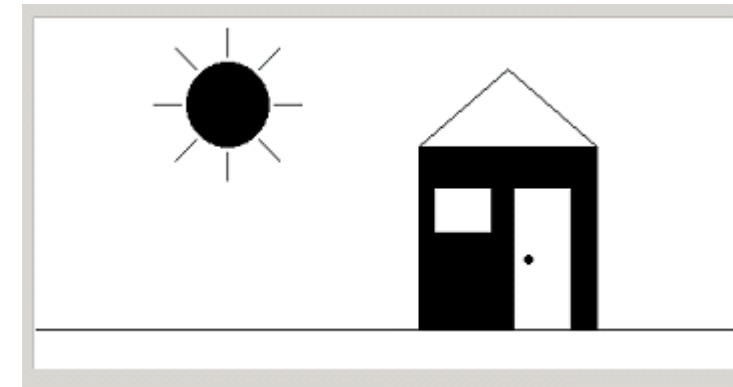
The remaining test programs include no test scripts. They should be compiled and executed on the supplied VM Emulator as follows:

OS Module	Test Programs	Desired Test Results (Actual Screen Shots)
<small>OS Module</small>	<small>Test Programs</small>	<small>Desired Test Results (Actual Screen Shots)</small>
String.jack	Main.jack	
<small>OS Module</small>	<small>Test Programs</small>	<small>Desired Test Results (Actual Screen Shots)</small>
Output.jack	Main.jack	

OS Module	Test Programs
Screen.jack	Main.jack



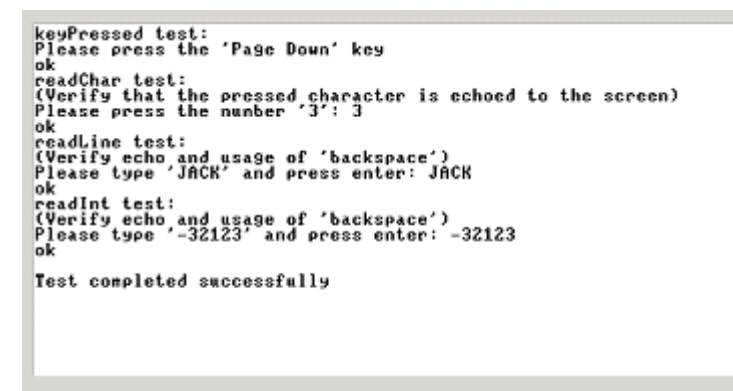
Desired Test Results (Actual Screen Shots)



Desired Test Results (Actual Screen Shots)

OS Module	Test Programs
Keyboard.jack	Main.jack

This OS class is tested via a test program that effects some user-program interaction. For each function in the Keyboard OS class (keyPressed, readChar, readLine, readInt), the program requests the user to press some keyboard keys. If the OS functions are implemented correctly and the requested keys are pressed, the program prints the text "ok" and proceeds to test the next function. If not, the program repeats the request for the same function. If all requests end successfully, the program prints "Test ended successfully", at which point the screen may look as follows:



OS Module	Test Programs
Sys.jack	Main.jack

Desired Test Results (Actual Test Results)

Only two functions in this class can be tested: Sys.init and Sys.wait. The supplied test program tests the Sys.wait function by requesting the user to press any key, waiting two seconds (using Sys.wait) and then printing another message on the screen. The time that elapses from the moment the key is released until the next message is printed should be two seconds.

The Sys.init function is not tested explicitly. However, recall that it 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. A simple way to test Sys.init in isolation is to run the supplied Pong game using your Sys.vm file.

| Complete test, and ...

After testing successfully each OS class in isolation, test your entire OS implementation using the Pong game, whose source code is available in your projects/11/Pong directory. Put all your OS .jack files in this directory, compile it, and execute the game in the supplied VM Emulator. If the game works, then congratulations: you are the proud owner of an operating system written entirely by you. And, by the way, you've just completed the construction of a complete general-purpose computer system. Go celebrate!