# Scott Hanselman's Tiny Virtual Operating System and Abstract Machine in C#

## What is this?

This was the final project for my CST352-Operating Systems class at [OIT](http://www.oit.edu) (Oregon Institute of Technology). The requirements, exactly as they were given to me by the teacher, are in Final\_Project.doc. The goal of this project was to write a small *virtual* operating system for an *abstract* machine that provides a number of basic OS-like services like:

* Virtual Memory, Demand Paging
* Input/Output
* Memory Protection, Shared Memory
* Registers, Stack, Data, Heap, etc
* Jump instructions for calling “Functions”
* And so on…

This is a cute, fun, interesting and *completely useless thing*. So, don’t get your hopes up that you’ll get actual work done with it. ­

It’s neat however, as it is a study on how I solved a particular problem (this assignment) given a 10 week semester. I was the only student to use C#, and I finished it in 4 weeks, leaving 6 weeks to chill and watch the other students using Java and C++ do their thing.

It’s also ironic because I used a high-level OO language like C# to deal with a minute concept like an “Operating System” the might have 256 bytes (bytes, not Kbytes) of memory.

During this process I exercised a good and interesting chunk of the C# language and the .NET Framework. There are some very silly things like the OS’s implementation of virtual memory swapping out memory pages as XML. I might take an array of 4 bytes and make an XML file to help them. I hope the irony isn’t lost on you. ☺ I also commented all the C# with XML and built an [ndoc](http://ndoc.sf.net) MSDN style help file. So, you might just use this as a learning tool and a pile of sample code.

One Note: I’m an ok programmer, but understand that I whipped this out in several 3am coding sessions, as well as during lectures in class. This is NOT fabulous code, and should be looked upon as interesting, not gospel. I’m sure I’ve done some very clever things in it, and for each clever thing, there’s *n* stupid things. I’ll leave the repair of these stupid things as an exercise to the reader.

Of course, you’ll need the .NET Framework to run this, but to really have fun debugging it and stepping through the code you’ll need VS.NET. Go read the code and final\_project.doc and enjoy!

Scott Hanselman

[scott@hanselman.com](mailto:scott@hanselman.com)

May 2002

## What’s the jist?

I started out with some basic OO constructs, a static CPU object, an OS object, objects for Processes, Instructions, etc. A lot of this information is in the Help (CHM) File. Basically it’s setup like this:

* **CPU Object** – responsible for physical memory (which may be smaller than addressable memory) holding CPU registers and fetching program opcodes and translating them into SystemCalls in the OS, etc.
* **OS Object –** Most of the work is here. It has a MemoryManager object who is responsible for the translations between Addressable (Virtual) memory, Process memory and Physical Memory. All Processes access memory from *their* point of view…they never talk to “physical memory” (Which is just an array of bytes in the CPU object)  
  The scheduler is in the OS object as well as locks, events, and all that good stuff. Each of the 36 opcodes are implemented here and treated as C# delegates…well, now I’m telling you all the good stuff…read the code!
* **Program Object –** represents a collection of instructions. Responsible for parsing the files off disk and holding them.
* **Process Object –** different than Program as it’s an actual running process in the TinyOS. There can be more than one instance of a Process for each Program. (You can run the same program twice, etc) It has a ProcessControlBlock containing Process specific registers, etc.
* **EntryPoint Class –** Contains main() and “bootstraps” the whole thing.

The OS is made up, the CPU is made up, the opCodes are made up. This is an exercise to learn about Operating System concepts, and it’s not meant to look or act like any specific OS or system.

## What does a Program look like?

Here’s an example program, specifically the “idle” loop. The idle loop is a process that never ends, it just prints out “20” over and over. I use it to keep the clock running when all the other processes are sleeping.

6 r4, $0 ;move 0 into r4

26 r4 ;lower our priority TO 0

6 r1, $20 ;move 20 into r1

11 r1 ;print the number 20

6 r2, $-19 ;back up the ip 19 bytes

13 r2 ;loop forever (jump back 19)

Programs consist of:

Opcode, [optional param], [other optional param] ;optional comment

So, if we look at:

6 r4, $0 ;move 0 into r4

We see that it’s operation 6, which is Movi or “Move Immediate.” It moves a constant value into one of our 10 registers. The first param is r4, indicating register 4. The second param is $0. The “$” indicates a constant. So we are the value 0 into register #4. Just like x86 assembly – only not at all. ☺

So if you look at the comments for this app you can see that it loops forever.

## How do run Program(s)

Usage:

OS membytes [files]

For example:

OS 1568 prog1.txt prog2.txt prog3.txt prog1.txt idle.txt

This command line would run the OS with 1568 bytes of virtual (addressable) memory and start up 5 processes. Note that Prog1.txt is specified twice. That means it will have two separate and independent running processes. We also specified the forever running idle.txt. Use CTRL-C to break. The OS has operations for shared memory, locks, and events, so you can setup rudimentary inter-module communication.

## Why should I care?

There are 13 other sample apps you can play with. The OS is most interesting when you run it with multiple apps. You can mess with OS.config to setup the amount of memory available to each process, the whole system’s physical memory, memory page size etc.

It’s interesting from an OS theory point of view if you think about the kinds of experiments you can do like lowering physical memory to something ridiculous like 32 bytes. That will increase page faults and virtual memory swapping. So, the OS can be tuned with the config file.

## What are the options in OS.config?

It is certainly possible to give the OS a bad config, like more physical memory than addressable, or a page size that is the same size as physical memory. But, use common sense and play. I’ve setup it up with reasonable defaults.

Here’s a sample OS.config file. The Config has to be in the same directory as the OS.exe. Take a look at the included OS.config, as it has additional XML comments explaining each option.

<configuration>

<appSettings>

<add key="PhysicalMemory" value="384" />

<add key="ProcessMemory" value="384" />

<add key="DumpPhysicalMemory" value="true" />

<add key="DumpInstruction" value="true" />

<add key="DumpRegisters" value="true" />

<add key="DumpProgram" value="true" />

<add key="DumpContextSwitch" value="true" />

<add key="PauseOnExit" value="false" />

<add key="SharedMemoryRegionSize" value="16" />

<add key="NumOfSharedMemoryRegions" value="4" />

<add key="MemoryPageSize" value="16" />

<add key="StackSize" value="16" />

<add key="DataSize" value="16" />

</appSettings>

</configuration>

Of note are the Dumpxxx options. With all of these options set to false, the OS only outputs the bare minimum debug information. With them on (I like to have the all on) you’ll be able to see the results of each instruction and how they affect registers, physical memory, etc. You’ll see physical memory pages swap to disk, memory fragmentation, the instruction pointer increment, and processes take turns on the CPU.

Be sure to turn up the size of screen buffer (usually under “Layout” in the Properties Dialog of your command prompt) to something like 3000 or 9999. This way you’ll be able to scroll back and look at the details of your run.

OS 512 prog1.txt > output.txt

You might also try something like this to create a log file, or modify the source to include logging!

## What are the available OpCodes?

These are also printed and explain in the Help File as well as Final\_Project.doc. When writing a program you use the value, not the text name of the opcode. So, you’d say “2 r1, $1” and ***NOT*** “addi r1, $1”.

|  |  |  |
| --- | --- | --- |
| * + Opcode | Value(decimal) | * + Format |
| Incr | 1 | incr r1(increment value of register 1 by 1 ). |
| Addi | 2 | addi r1,$1 is the same as incr r1 |
| addr | 3 | Addr r1, r2( r1 <= r1 + r2 ). |
| Pushr | 4 | Pushr rx (pushes contents of register x onto stack. Decrements sp by 4 ). |
| Pushi | 5 | Pushi $x . pushes the constant x onto stack. Sp is decremented by 4 after push. |
| Movi | 6 | Movi rx, $y. rx <= y |
| Movr | 7 | Movr rx, ry ; rx <= ry |
| Movmr | 8 | Movmr rx, ry ; rx <= [ry] |
| Movrm | 9 | Movrm rx,ry; [rx] <= ry |
| Movmm | 10 | Movmm rx, ry [rx] <= [ry] |
| Printr | 11 | Printr r1 ; displays contents of register 1 |
| Printm | 12 | Printm r1; displays contents of memory whose address is in register 1. |
| Jmp | 13 | Jmp r1; control transfers to the instruction whose address is r1 bytes relative to the current instruction. R1 may be negative. |
| Cmpi | 14 | Cmpi rx, $y; subtract y from register rx. If rx < y, set sign flag. If rx > y, clear sign flag. If rx == y , set zero flag. |
| Cmpr | 15 | The same as cmpi except now both operands are registers. |
| Jlt | 16 | Jlt rx; if the sign flag is set, jump to the instruction whose offset is rx bytes from the current instruction. |
| Jgt | 17 | Jgt rx; if the sign flag is clear, jump to the instruction whose offset is rx bytes from the current instruction |
| Je | 18 | Je rx; if the zero flag is clear, jump to the instruction whose offset is rx bytes from the current instruction. |
| Call | 19 | Call r1; call the procedure at offset r1 bytes from the current instruction. The address of the next instruction to execute after a return is pushed on the stack. |
| Callm | 20 | Callm r1; call the procedure at offset [r1] bytes from the current instruction. The address of the next instruction to execute after a return is pushed on the stack. |
| Ret | 21 | Pop the return address from the stack and transfer control to this instruction. |
| Alloc | 22 | Alloc r1, r2; allocate memory of size equal to r1 bytes and return the address of the new memory in r2. If failed, r2 is cleared to 0. |
| AcquireLock | 23 | AcquireLock r1; Acquire the operating system lock whose # is provided in the register r1. If the lock is invalid, the instruction is a no-op. |
| ReleaseLock | 24 | Releaselock r1; release the operating system lock whose # is provided in the register r1; if the lock is not held by the current process, the instruction is a no-op. |
| Sleep | 25 | Sleep r1; Sleep the # of clock cycles as indicated in r1. Another process or the idle process must be scheduled at this point. If the time to sleep is 0, the process sleeps infinitely. |
| SetPriority | 26 | SetPriority r1; Set the priority of the current process to the value in register r1; See priorities discussion in Operating system design |
| Exit | 27 | Exit. This opcode is executed by a process to exit and be unloaded. Another process or the idle process must now be scheduled. |
| FreeMemory | 28 | FreeMemory r1; Free the memory allocated whose address is in r1. |
| MapSharedMem | 29 | MapSharedMem r1, r2; Map the shared memory region identified by r1 and return the start address in r2. |
| SignalEvent | 30 | SignalEvent r1; Signal the event indicated by the value in register r1. |
| WaitEvent | 31 | WaitEvent r1; Wait for the event in register r1 to be triggered. This results in context-switches happening. |
| Input | 32 | Input r1; read the next 32-bit value into register r1. |
| MemoryClear | 33 | MemoryClear r1, r2; set the bytes starting at address r1 of length r2 bytes to zero. |
| TerminateProcess | 34 | TerminateProcess r1; terminate the process whose id is in the register r1. |
| Popr | 35 | pop the contents at the top of the stack into register rx which is the operand. Stack pointer is decremented by 4. |
| popm | 36 | pop the contents at the top of the stack into the memory operand whose address is in the register which is the operand. Stack pointer is decremented by 4. |

## Known Issues/Things to Know

* The OS will look in the current directory for files if no directory is specified.
* Processes will be loaded and executed in the order they are specified on the command line.
* There are two idle processes included. idle.txt will run with lowest priority and will print out 20. It will run forever. idle-n.txt will run for n loops (not cycles), specified in the code. As shipped, idle-n will run for 100 loops.
* Redirecting OS output to a file is convenient: OS 1024 prog1.txt prog2.txt prog3.txt > output.txt
* OS.config contains many debug flags that can be included to provide insight into non-debug program execution.
* OS.config contains many options for configuration of memory. It IS possible to feed invalid values into this config file. For example, it's not valid to have a physical memory size larger than addressable memory, or a page size larger than addressable memory, etc. This behavior is appropriate, and is by design.
* Setting DumpInstruction=true in OS.config will output the a debug print of the executing instruction and the current process id.
* If an idle process (idle.txt, etc) isn't included, and all running processes sleep or block on a lock, the OS will spin forever looking for an eligible process to run, and will find none. CTRL-C will exit at this point. This behavior is per spec and by design.
* The OS will create XML swap files in the current for each memory page swapped. These are cleaned up at startup, rather than shutdown, for purposes of exploration.
* Batch files for testing purposes are included in this directory.