Jist

An Operating System for MIPS

Daniel DeCovnick

Tim Henderson

Steve Johnson

Problem Statement

SPIM is lacking as an operating system for MIPS. It has excellent I/O facilities, but poor memory management and idiosyncrasies that make it very difficult to code for. To remedy this, we created the Jist OS with a proper memory manager, memory-mapped I/O and a standard library to go with it, a stack manager, interrupt handler, and preprocessor. The sum of these parts is a programming platform that provides a large number of high-level features as well as cooperative multitasking. With these features, we were able to provide a better programming experience. As a result, we are able to code programs for MIPS in much less time than before.

Major Challenges

The biggest challenge we faced when writing an operating system was that it had to run on top of SPIM, but use as few SPIM facilities as possible, as we do not have real MIPS hardware. While this meant we could take advantage of some of the faculties of SPIM such as the sbrk system call for memory allocation, it also meant dealing with SPIM's bugs and idiosyncrasies. For instance, SPIM is actually a MIPS assembly interpreter, rather than a strict emulator; it does not load, run, or provide access to compiled MIPS object code, and does not support loading more than one assembly program at a time. As a result, we had to create a mechanism for loading multiple programs at startup, while keeping each in its own address space.

The second, related challenge was in memory management. While SPIM has the sbrk system call to allocate memory, it has no mechanism to free the memory claimed. So we wrote our own compacting memory manager, which handles a heap for each program.

Key Components of Jist

The key components of Jist are as follows:

- Memory Manager
- Stack Manager
- Context Manager
- Interrupt Handler
- Standard Library
- Preprocessor
- Demonstration programs

The memory manager is a compacting heap. When a program asks for memory, the allocator will not return the address of the memory, but rather a memory ID whose value must be queried via a macro when used. When memory is freed, the heap is compacted. The address of the ID will change, but not its contents. To prevent corruption of the heap, there is no API for getting the address of the memory ID. At the top of the heap is a Heap Control Block. It climbs the heap when new memory is allocated, and climbs down when memory is compacted on free. Programs must keep track of their heap control blocks to access the memory contained by the heap. See Appendix A: Memory Manager Documentation for a more complete description of the HCB and the memory layout in Jist.

Due to SPIM limitations, context switching is done in a somewhat unusual way. When a process A is launched, its stack is copied to the stack pointer at the top of memory. When a process B is launched, the save_stack function copies A's stack to a special heap, moves the stack pointer back to the top of memory, and moves B's stack to the stack pointer (restore_stack). When A is switched to again, A's stack is restored, and the copy on the heap is freed, which causes the heap to compact. For a graphical explanation, see Appendix B: Jist Context Switching. This forms the basis of the OS, using kernel.s to determine the nature of an interrupt or exception, and the interrupt manager, stack manager, and context manager.

There are three levels of interrupts in Jist: Hardware-level, software-level (clock-based) and OS-level. The interrupt handler is the state machine that drives the context manager and stack manager, based on OS-level interrupts. Hardware-level interrupts are not used in Jist. Clock interrupts are currently not enabled in Jist, but this can be changed to enable pre-emptive multitasking; this is not done because the edge cases of pre-emptive multitasking make supporting it, in the words of one group member, "tricky, difficult, and hard." Although we didn't enable them for our presentation, we actually have a branch in our version control system with a working prototype, though it is not stable enough to show off.

The stack manager is composed of three functions for manipulating the entirety of a program's stack: save_stack, restore_stack, and zero_stack. Save_stack copies a stack in its entirety onto the heap. Restore_stack copies a stack from the heap back to the stack pointer. Zero_stack zeros out the contents of the stack.

The context manager is also the process scheduler. It implements round-robin cooperative multitasking using a circularly linked list. The linked list is stored in a heap from memory manager.

The stack manager and context manager are both driven by the interrupt handler, which is essentially a state machine that calls the APIs of low-level kernel libraries.

The standard library is a high-level I/O library uses SPIM's memory-mapped I/O features. It consists of several procedures and several macros. It implements the functions println(string_address), print_hex(hex_int) print_int(int), readln, read_char, and print_char(char), atoi(char), and printf(format_string, arg_1, arg_2…). More complex, Jist-specific functions, such as print_hcb(hcb_address), print_hcb_item(address), println_hex(string_address, hex_int), and several others are also implemented.

The preprocessor, MPP, was our way of bringing high-level programming features to SPIM. Without it, the memory manager would have been practically impossible to write, and everything else would have been a major pain. MPP supports #includes, register aliasing to make code more self-documenting, mostly-recursive macros, and scoping, which makes labels and aliases local to their lexical scope. Due to a limitation of SPIM (the fact that it's a MIPS assembly interpreter which can't load more than one program at a time), MPP statically compiles all Jist programs into the kernel, does some introspection, and generates code that SPIM will understand. User programs can make use of all of these facilities as well. For examples of all of these see stdlib.s (lines 3, 6, 45, and 68) and Appendix C: MPP.

We have written several programs to demonstrate the capabilities of Jist. The first is "Hunt the Wumpus," a classic text adventure game which makes heavy use of the standard library. The next is "Muckfips," a Brainf*** interpreter for MIPS, demonstrating the ease of programming in Jist. "iMuckfips" is an interactive version of the same. "Multitask_demo" demonstrates the multitasking capabilities of Jist, as well as user-level heaps.

Interface

From within the Jist directory, run ./run.sh. To control which program runs initially, edit the "jistfile" (analogous to a makefile) "init-with:" line; the order of the programs listed above that determines the order in which the programs are loaded into memory. Please do not change the order or number of programs listed in the jistfile, because "eecs314demo.s" assumes their existence.

Conclusion

When we decided to write an OS as our project, none of us really expected it to ever reach this state of completion. Along the way, we ran into a few obstacles that looked like they would make our project utterly impossible, but we tackled them one at a time, and the pieces finally came together during the last week before the day of the presentation. All group members contributed important pieces to the design and to the code base, and we all understand operating systems *much* better than we did at the beginning of the semester.

The file generated by MPP with all macros expanded is 25,000 lines long without comments or blank lines. 15,000 of those lines are devoted to saving and restoring stack frames, and still more are devoted to smaller macros that we used repeatedly. These numbers demonstrate the necessity of high-level language features to OS development. Without macros, we would have spent a disproportionate amount of time manipulating the stack by hand, and there would probably be dozens of bugs scattered throughout the code base. The preprocessor allowed us to modularize and document our code with relative ease. All 25,000 lines of it.

Appendix A: Memory Manager Documentation

```
Structure of memory in spim
-------------------------
|      Kernel Data      | -> starts at 0x90000000
-------------------------
|     Kernel Program    | -> starts at 0x80000000
-------------------------
|                       | -> starts at 0x7fffffff
|         Stack         |
|                       | -> $fp denotes bottom of current stack frame
------------------------- -> $sp denotes top of stack
|            |          |
|           \|/         |
|                       |
|                       |
|       free space      |
|                       |
|                       |
|           /|\         |
|            |          |
-------------------------
|                       |
|         Heap          |
|                       |
------------------------- -> sbrk syscall allocates memory in the heap
|        Static         |
------------------------- -> $gp
|       User Data       | -> starts at 0x10000000
-------------------------
|      User Program     | -> starts at 0x40000000
-------------------------
|       Reserved        | -> starts at 0x00000000
-------------------------
```

HOW THE MEMORY ALLOCATOR WILL WORK
The memory allocator will be a way to manage the memory in the heap. While the sbrk syscall allocates memory in the heap, it cannot free memory. Thus the operating system needs a way to free and compact the heap when blocks of memory are released by either the OS or a user program.

A program will request memory by asks for x number of words. The programs will not be able to request by number of bytes. The user programs will not get back the address of the memory – instead,
they will get back a unique identifier for their memory. When they want a word from their memory, the programs will use a global macro that will be part of this library. ie they will pass the macro their memory id and the word they want (ie 0, 1, 2, 3 ... n) and the system will return the value of the word in from their memory block.

The memory manager will allow users to free memory when they are done with it. When a piece of memory is free the heap will be compacted by the memory manager so that there will be no empty space. This is the reason that the user will never be given the address of their memory. The address will not remain constant so the users cannot have them.

At the very top of the heap will be a control structure. In essence it will be an ascending sorted list of memory blocks in use sorted by the memory id number. It will look like this.

```
Structure of Heap Control Block(HCB)
-------------------------------------
| The Sorted List Inside the Block  |
| --------------------------------- |
| | Size in Words of the Block    | |
| | Address of the Memory Block   | |
| | Memory id N                   | |
| --------------------------------- |
| --------------------------------- |
| |                               | |
| |             ...               | |
| |                               | |
| --------------------------------- |
| --------------------------------- |
| | Size in Words of the Block    | |
| | Address of the Memory Block   | |
| | Memory id 0                   | |
| --------------------------------- |
-------------------------------------
| Length of List                    | -> the length of the sorted list
-------------------------------------
| Freed Space in Words              | -> Words of free space above the HCB
-------------------------------------
|                                   | -> farthest the heap can grow without
| Address of the Top of the Heap    |    another sbrk call, including space
|                                   |    that the Heap Control Block occupies
-------------------------------------
| Next Memory id                    |
-------------------------------------
| Size in Words of Control Block    | -> includes the first 5 words
-------------------------------------
```

The heap control block will grow in size as the number of blocks in the heap grows and it will shrink at as blocks of memory are free. There will be a special label called HCB_ADDR which will store the start of the heap control block. This will make it quicker to access the block. That way the memory manager doesn't have to walk the entire heap to get to the control block.

```
Structure of Heap
------------------------------ -> Top of Heap
|                            |
|        Freed Space         |
|                            |
------------------------------
| -------------------------- |
| | Heap Control Block(HCB)| |
| -------------------------- | -> HCB_ADDR
------------------------------
| Memory Block N             |
------------------------------
| Memory Block N-1           |
------------------------------
|                            |
|            ....            |
|                            |
------------------------------
| Memory Block 1             |
------------------------------
| Memory Block 0             |
------------------------------ -> Bottom of Heap
```

Appendix C: MPP – MIPS PreProcessor

#includes

The most basic feature of MPP is the `#include` directive:

```
#include stdlib.s
```

MPP does not check to see if a file has already been included.

Macros

Macros are defined like so:

```
	#define printchar_immediate
		li $a0 %1
		call print_char
	#end
```

This particular macro might be called like this:

```
	printchar_immediate 10 #prints a newline
```

The %1 token is replaced by MPP with the given parameter, in this case 10. The `call print_char` line executes the `call` macro on the `print_char` label, which is a function. As you can see, macros can be used within macros, as long as the inner macro is defined above the outer macro.

Register Aliases

To increase readability and maintainability, aliases can be defined for registers:

```
	@input = $s0
	read_char @input
```

This feature is particularly useful if the same register means the same thing over a few dozen lines of code, especially if that register conflicts with a value somewhere else and must be changed during development.

Scope

When writing programs of nontrivial complexity, it can be a pain to keep coming up with unique names for labels. To solve this problem, MPP introduces label and register alias scoping to MIPS assembly. The concept is simple: if a label is defined within a scope block, it cannot be accessed outside of that block. With MPP, this block of code executes with no errors whatsoever:

```
{
	@loopvar = $t9
	li @loopvar 20
	loop:
		addi @loopvar @loopvar -1
		bgtz @loopvar loop
}
{
	@loopvar = $s0
	li @loopvar 19
	loop:
		addi @loopvar @loopvar -1
		bgez @loopvar loop

}
```