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 Jist with a proper memory manager, memory-mapped I/O and a standard library to go with it, a stack manager, interrupt handler, and preprocessor. With these, 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. A demonstration program, Hunt the Wumpus, was done in roughly 3 days. Additionally, we have cooperative multitasking.

Major Challenges

The biggest challenge to writing an operating system was that it had to run on top of SPIM, but using as few SPIM facilities as possible, as we do not have MIPS hardware. While this meant we could take advantage of some of the facilities of SPIM (such as the sbrk system call for memory allocation, though we only ended up using it to get all the memory for the memory manager), 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. (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 copies A’s stack to the 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 a compaction. 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 preemptive multitasking; this is not done because the edge cases of preemptive multitasking make supporting it tricky, difficult, and hard (for more information, email Tim Henderson at [tah35@case.edu](mailto:tah35@case.edu)) (we were very close to supporting clock-based interrupts).

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 stack manager is driven by the interrupt handler.

The context manager is the Jist scheduler. It implements round-robin fully cooperative multitasking using a circularly linked list. The linked list is stored in a heap from memory manager. The context manager is driven by the interrupt handler.

The standard library is a relatively high-level I/O library which works by using memory-mapped I/O. It consists of several procedures and several macros. It implements the simple functions println(string\_address), print\_hex(hex\_int) print\_int(int), readln, read\_char, and print\_char(char), atoi(char), printf(format\_string,…). More complex, Jist-specific functions, such as print\_hcb(hcb\_address), print\_hcb\_item(address), println\_hex(string\_address, hex\_int), printand several others, are also implemented.

The preprocessor, MPP, was our way of bringing more high-level programming facilities to SPIM. Without it, the memory manager would have been practically impossible to write. 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 in, does 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).

We have written several programs to demonstrate the capabilities of Jist. The first is “Hunt the Wumpus,” a classic text adventure which makes heavy use of the standard library. The next is “Muckfips,” a Brainf\*ck 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.

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.

Conclusion

Jist is comprised of 15,000 lines (expanded) of saving and restoring stack frames, demonstrating how much higher-level programming constructs are all but indispensible. \*\*\*FIXME: A BETTER CONCLUSION\*\*\*