Operating System Development

Creating a Minimal Operating System from Scratch

Eli Card, Scott Conner, and Elijah Wade
Computer Science Operating Systems: CS3800
Missouri University of Science and Technology
Rolla, United States of America
scott.conner@mst.edu

***Abstract*—Develop and run a minimal operating system from scratch that can run and manage user programs.**

Keywords—kernel; operating system; assembly; lisp; machine code; virtual processor

I. Introduction

This research project aimed to develop and run a minimal operating system from scratch with the goal of implementing low level processes and applications of different operating system design prospects. The operating system included multiple components, such as a personalized high level language called Lisp Plus; a compiler that interprets the Lisp Plus code and translates it into assembly code; an assembler that interprets the assembly code and translates it into machine code; a virtual processor that reads the machine code and runs calculations; and a kernel developed in Lisp Plus that manages virtual memory. These pieces worked in tangent with each other to run user programs and manage memory within a virtual main memory and virtual processor.

II. Kernel

The kernel design choices were the ones most closely related to the material discussed in the operating system lectures, involving multiple components necessary to manage main memory and create process control blocks.

A. Main Memory Blocks

The main memory uses a paging implementation where the memory is separated into equal sized frames and allocated based on how many pages are necessary to store the user program. Paging was used to resist external fragmentation, a symptom of memory allocation where the locations of used memory blocks separate portions of free space. If the free space is sliced up too much, large processes cannot be allocated. Because all pages are equal size and do not have to be allocated contiguously, external fragmentation can be avoided entirely. The problem with the paging implementation is internal fragmentation, where memory is allocated to a process but the process does not use all of the memory. Since the memory pages are allocated based on the size of the program, only the last page can contain internal fragmentation, but the severity of internal fragmentation varies depending on how much of the last memory block is wasted.

B. Process Control Block

The process control block (PCB) is created when a process is created and is stored in the first page of the process' allocated memory. The PCB stores information about the process at specific address locations referred to as "words".

- The first word of the PCB stores the process state. The process is termed as either ready to run, currently running, or blocked. Blocked means there isn't enough memory available to allocate the process.
- The second word stores the program counter, which is initialized to 0 when a process is created.
- Words three through eighteen are used to store information about the user registers. User registers are used by user programs and are separate from the dedicated kernel registers.
- Word nineteen determines the size of the function return stack.

Everything stored after the stack size is the stack itself, storing values that are pushed onto it during process execution.

C. Scheduling

The kernel uses a very basic round robin system to schedule the processes. During a context switch, the currently running process has its user registers and stack saved and the process state turned to ready. The kernel searches through the process queue starting with the process slot directly after the currently running process until it finds the next ready process. After it finds the process, it loads the user registers and the stack and runs the process for a set amount of processor ticks.

D. Interrupts

The kernel uses an interrupt based system instead of polling, which was ideal for both error handling and switching between kernel and user mode.

- Timer Tick: Triggers when the processor timer hits 0.
- Bad Ins: A bad instruction handler. Occurs either when an instruction could not be executed or a kernel instruction is executed in user mode. Activates "Kill Process".

- Stack Overflow: Occurs when the stack exceeds the length designated to it after a function call. Activates "Kill Process".
- Stack Underflow: Occurs when the stack attempts to return from a function when the function stack is empty. Activates "Kill Process".
- Bad Mem Access: A bad memory access handler. Occurs when a process attempts to use memory that was not allocated to it. Activates "Kill Process".
- User Defined 1: Allows a user program to utilize the kernel's string print functions. Might activate "Kill Process" if the value to print was not stored in the correct memory address.
- User Defined 2: Allows a user program to utilize the kernel's ability to print signed integers.
- User Defined 3: Allows a user program to utilize the kernel's ability to print unsigned integers.
- User Defined 4: Allows a user program to utilize the kernel's ability to print hexadecimal integers.
- User Defined 5: Allows a user program to utilize the kernel's ability to print a string label along with a signed int value. Might activate "Kill Process" if the value to print was not stored in the correct memory address.

Multiple interrupts have the ability to call "Kill Process" which deallocates the process memory and removes it from the process queue. A process that terminates successfully will also use the "Kill Process" function, but will not print out an error.

III. High Level Language

Lisp Plus (Also called Lispp) is the customized high level language used to program the kernel and user programs. The processor takes in machine code that was converted from assembly, however assembly is difficult to program long term, so a high level language was created. A custom compiler was built from scratch to translate the Lisp Plus code into assembly which allows for ease of programming and compiling. Similar to Lisp, a function is denoted by surrounding a function name and arguments in parenthesis.

Ex. (<Function Name> <arg1> <arg2> … <argN>)

Any function must be surrounded in parenthesis, including math operations, in order to create an incredibly disambiguous compiling interpretation.

Ex. (+ 1 2) returns 3

Creating nested functions works similarly, with all nested function calls surrounded in parentheses to explicitly denote a function call and what argument number it is within the parent function call.

Ex. (= (+ 1 2) (+ 0 3)) returns 1 which is True

This lisp-style syntax is easier to parse than many other programming languages because it has explicit operator precedence. However, since Lisp is traditionally an interpreted language, several changes were made to make it fully compilable. The first change is the addition of types to Lisp Plus. Types are defined by prefixing the variable name with the type of the variable and an underscore.

Ex. int_total is an integer type variable

The second change is the addition of multiple keywords to the language to facilitate the creation of different control structures. These include loops, conditionals, function definitions, and assignment statements. While this is not standard for a Lisp derivative, these control statements help make the structure as a whole more readable than normal lisp structure.

IV. Lisp Plus Compiler

The Lisp Plus compiler is designed to read in a .lispp file and translate it into assembly code. While the Lisp Plus compiler could directly translate the language into bytecode, it was easier to debug the generated assembly code than machine code.

Ex. Machine Code: 1000 0111 0101 0000

Ex. Assembly Code: SETTIMER 80

While the rest of this project is coded in C++, the Lisp Plus compiler was written in Python to aid in rapid development. The compiler uses a recursive descent parser to construct the abstract syntax tree. Variables are assigned to various registers. Functions are either encoded by translating to equivalent assembly instructions, or by performing function calls in the case of user defined functions. The compiler does not maintain a function stack; rather, it relies on using dedicated registers to keep track of important variables for performing the current and future calculations. This has some side effects, most noticeably the lack of recursive functions in the language.

V. Assembly Compiler

The Assembly compiler is designed to read in a .asm file and translate it into machine code. The machine code is put into a .bin (Binary) file which can be fed to the virtual processor in order to run the compiled program. The assembly maps directly to the bytecode for most instructions.

Ex. Before Translation: SETTIMER 81

Ex. After Translation: 0x8751

There are some instructions that are somewhat difficult to encode properly. For example, jumping several instructions forward in assembly causes a token to be read that doesn't exist in the context yet, causing the assembler to have to create some empty space in the buffer and return to that point later in time to fill in the jump address.

VI. Processor

While this kernel could realistically be implemented for an ARM or x86 processor, the team elected to use a custom processor to better facilitate the creation of the kernel. Since this is a custom virtual processor, the team was able to implement many custom features, such as a hardware return stack, an extensive ALU, and many other custom processor components that are not commonly found in general purpose chips.

The processor utilizes a 16 bit architecture to perform all operations. The processor maintains an impressive 69 general purpose registers. 69 registers was a nice midpoint that allowed the kernel to store important variables in register space for fast access without wasting space. The 64kb 16-bit addressable main memory stores both code and data. Each word in the machine code is also 16 bits, and an instruction takes anywhere from 1-2 words. The instruction set is relatively simple, but powerful, supporting instructions for loading memory into registers, ALU operations, function calls using the hardware stack, conditional jumps, and more.

The processor can operate in one of two modes. In kernel mode, memory is treated as one continuous array, and the program counter maps exactly to this memory. Additionally, the kernel has access to many protected instructions, including printing to the console, stack manipulation, reading or writing from external memory, and shutting down the processor.

The RETI processor instruction pops the stack and enters user mode. User mode is designed to allow user programs to execute independently from each other without damaging other programs or the processor. In user mode, only a carefully selected subset of instructions are allowed to be executed by the processor. Additionally, only certain registers are allowed to be used by the processor and certain parts of memory are locked off. If the user program wishes to perform operations not available in the subset (such as printing information to the console), the user program may trigger an interrupt to request the operating system perform this task.

In user mode, a second hardware stack is used to create processor-level memory paging. This stack can be manipulated in kernel mode using dedicated instructions. Whenever an address is processed in user mode, the high byte is transformed using this stack. For example, if the stack is 0x05, 0x34, 0x65 and a program in user mode executes an instruction at 0x0113, the automatic translation algorithm will convert this address to 0x3413. This essentially creates a virtual address space that user programs may be placed into by the kernel. Attempting to access outside this address space triggers a bad memory access interrupt.

Interrupts on the virtual processor do two major things. The first is that triggering an interrupt enters kernel mode, enabling the full instruction set and disabling the virtual paging stack. The second thing that it does is move the program counter to a predetermined spot in memory to handle the interrupt, similar to how a CALL instruction works. There are many things that can trigger an interrupt, including overflows and underflows of the hardware stack, malformed instructions, and attempts to access memory outside of the program. User programs can also choose to trigger custom interrupts to allow the kernel to perform actions that would normally be disallowed in user mode. This design allows the kernel to specify how to deal with various events on the virtual processor.

The processor's memory can be initialized using a file which can be passed as a command line argument. Additionally, multiple external files can be attached to the processor via the command line. Each file can be read using processor instructions. The system uses this functionality to allow users to define which programs they want the operating system to run on their virtual processors.

VII. Limitations

A. *Operating System Specific Limitations*

Despite the ability to compile, load, and run user programs, there is no user input stream. The original design plan utilized an input stream to run certain programs based on the user's interests. However, no such input stream exists and the system is designed instead to run all user programs sent to it as an argument during run time.

B. *Kernel Specific Limitations*

The size of the program and the memory allocated to that program needs to be determined at the time the process is created. Memory cannot be additionally allocated based on a process' needs. This also means that while there is a blocked process state, it is underutilized because if a process is deemed ready, it won't require more memory after the fact.

C. *Lisp Plus Specific Limitations*

While the Lisp Plus is very powerful, the compiler for it is incomplete. The most glaring issue with the Lisp Plus compiler is its over-reliance on register space. On the one hand, this makes the kernel very fast, however since every variable in every function uses a register slot, the number of used registers balloons quickly. This was especially a problem when developing user programs, where only sixteen registers are accessible to the user. This also has the side effect of making it impossible to code recursive functions.

The other big issue is that the compiler does not support struct data types, so all data must be represented internally as a 16-bit integer. This can cause problems for developing high-level data structures such as linked lists, vectors, or even a Process Control block. The PCB was developed using an integer array to circumvent this problem.

References

For better or worse, the operating system was created using only the collective knowledge of the team members working on the project. This means that no references were used during the development process. The failures of the project, as well as successes, are a symptom of said research model.