BSOS

Alexander Gould, William Woodruff, Paweł Czarnecki, Reyna Shaskan

Abstract

This paper details the key concepts and components of basic Operating System development, and applies them to the creation and application of BSOS, an experimental multiarchitecture system. In particular, the paper describes the innerworkings of the basic input-output system (BIOS), the standard bootstrapping procedure on x86 and ARM-based microprocessors, principles for interaction with system and external hardware, and proper resource management. BSOS itself can be applied, through the methodologies listed above, to a variety of limited-resource environments, most importantly the ARM-based VEX Cortex-M3 microcontroller used on a number of popular robotics kits. Ultimately, in combination with a framework based upon the already extant OpenVEX library, BSOS aims to become a complete replacement for simple operations on VEX robots.

Part I. Operating System Basics

Theory An Operating System has several different components. All software has to go through the OS in order for the user to be able to utilize any of the hardware. The kernel offers a basic level of control over doing this. The kernel manages memory access for programs in the RAM, determines which programs connect with which hardware, handles the CPU's operating states, organizes data in the long run, and orders processes in multi-tasking systems. Essentially, the kernel assigns memory space.

Interrupts Interrupts are also very important. Interrupts create an efficient way for the OS to interact with its environment. Older systems used to monitor different sources of input for events. Interrupts help automatically save local register contexts and run particular code in response to particular events. Interrupts cause the current program to pause for a moment, save it, and then run code associated with that interrupt. In other words, an interrupt interrupts the program-hence the name. Interrupts are handled by the kernel.

Types Three different types of interrupts are hardware interrupts, software interrupts, and traps. Hardware interrupts may come from a key on a keyboard being hit or the click of a mouse. Software interrupts are generated by programs. Traps come from the CPU itself due to errors or conditions which require the OS for assistance.

Boot When a computer initially turns on, it runs in supervisor mode. The first programs to run have unlimited access to hardware. When the OS allows another program to take control, the computer transitions into protected mode, where programs' access to the CPU's set of instructions

is limited. To leave protected mode, an interrupt has to be triggered so that way the kernel will be able to take charge once again.

Summary An OS manages the interactions between hardware and software on the computer. The most common Operating Systems are Microsoft Windows, Mac OS X, and Linux.

Part II. Booting Up

BSOS, as a portable and abstracted system, must be capable of bootstrapping on a variety of diverse and divergent architectures. Although hundreds exist, ranging from giants like x86, POWER, and ARM to lesser known embedded architectures like AVR and MIPS, the vast majority fall into one of two families: *CISC* and *RISC*.

CISC, or Complex Instruction Set Computing, began as a side effect of early computer development. As the (comparatively primitive) computers and calculators of the 1950s and 1960s became increasingly complex and multifunctional, they required new paradigms and methods of automation. Because languages like FORTRAN and ALGOL required a certain amount of boilerplate in order to operate, early system architects build complex control structures into the instruction sets of their machines, bridging the semantic gap between microcode-assembly and full-features programming languages. Because this approach to abstraction is relatively simple to implement, CISC quickly became the go-to approach for architecture design, eventually resulting in seminal processors like the MOS 6502, Zilog z80, and Intel 8086/8088.

Although CISC architectures are simple to implement, they have a number of downsides. Nowhere is this more apparent than in x86 family. AMD64, the most recent iteration in a long line of x86-based instruction sets, contains a truly massive number of opcodes, registers, and control structures. While this makes it very powerful and often very convenient, it also poses a major problem for programming and efficiency. Programmers who wish to write assembly for modern x86 platforms must memorize and utilize a vast array of confusing mnemonics, many of which are sparsely documented. Even compilers, which can take advantage of modern instructions and features through sheer force, have trouble fully exploiting x86's full feature set. Consequently, although programs can be specialized or drastically simplified through advanced structures within the architecture, they often perform poorly or inefficiently on CISC systems.

RISC (Reduced Instruction Set Computing), which began formal development in the 1980s, is a solution to many of CISC's problems. Instead of providing a large collection of instructions for programmers to utilize, RISC systems strip the instruction set down to a bare minimum of operation, leaving only a few dozen simple opcodes known as *primitives*. Additionally, those opcodes that remain follow strict cycle restrictions, rarely taking more than a single DMC (data memory cycle) to complete. This has two major side effects: simpler assembly programs, and the necessity of more abstraction between the user environment and the bare hardware. Although RISC CPU development is more recent than comparable CISC development, it is by no means unimportant. Architectures like DEC Alpha, SPARC, POWER, and ARM have become adopted in wide range of capacities, including supercomputers, mainframes, and mobile devices.

Unfortunately, RISC is not without its downsides as well. Because RISC instruction sets contain opcodes that are designed to execute within a single memory cycle, they often outpace their own hardware and are bottlenecked by latency when reading from and writing to RAM. Many RISC CPUs compensate for this by providing a large on-die L1 cache for compilers and programmers to optimize with, but RAM access is still inefficient in relative to comparable CISC CPUs. Another significant downside is *code expansion*. When high level languages are compiled down to assembly, their advanced control structures and concepts must be translated into the native structures of the system. For example, a high level procedure like this:

```
int a = 5, b = 3;
swap(a, b);
```

might be expanded to something like this:

```
mov ax, 5
mov bx, 3
xor ax, bx
xor bx, ax
xor ax, bx
```

The end result is an expansion of two high-level instructions into five low-level opcodes, an increase of 250%. While this happens on all architectures simply due to the nature of abstraction, it is particularly pronounced on RISC systems due to the lack of complex, comprehensive opcodes.

With these fundamental differences in mind, we can turn to BSOS and its booting procedure on two key systems: i386 (a descendant of x86), and Cortex-M3 (a version of ARM).

On x86 machines, specifically i386 and later, BSOS begins in 16-bit real mode. The strapping sector, which is 512 bytes long, then loads a general descriptor table, or GDT, into memory. That descriptor table is then used to switch the CPU into 32-bit protected mode, whereupon the kernel is loaded.

On ARM-based machines, specifically those with Cortex-M3 processors, the bootstrapping process is remarkably similar. Despite the RISC nature of the Cortex-M3, a similar chain of events takes place: the stack is set, the kernel is loaded from disk, and the system performs a far jump to the first instruction of the kernel, allowing it to take control.

Part III. Memory Management

Introduction In all modern computer systems arises the necessity to manage resources, in particular memory. Because a typical general purpose computer may have many different programs running simultaneously it is crucial to have a subsystem that assigns memory regions to applications and keeps track of those regions. In the case of modern operating systems the implementation of such a system may be done so based on different principles. In a very simple Operating System which is responsible for managing resources for a small pool of applications the choice of paradigm does not have a large impact on system stability or performance. With simplicity taken into account, the BSOS memory management system assigns variable sized blocks of memory to applications and keeps track of their status.

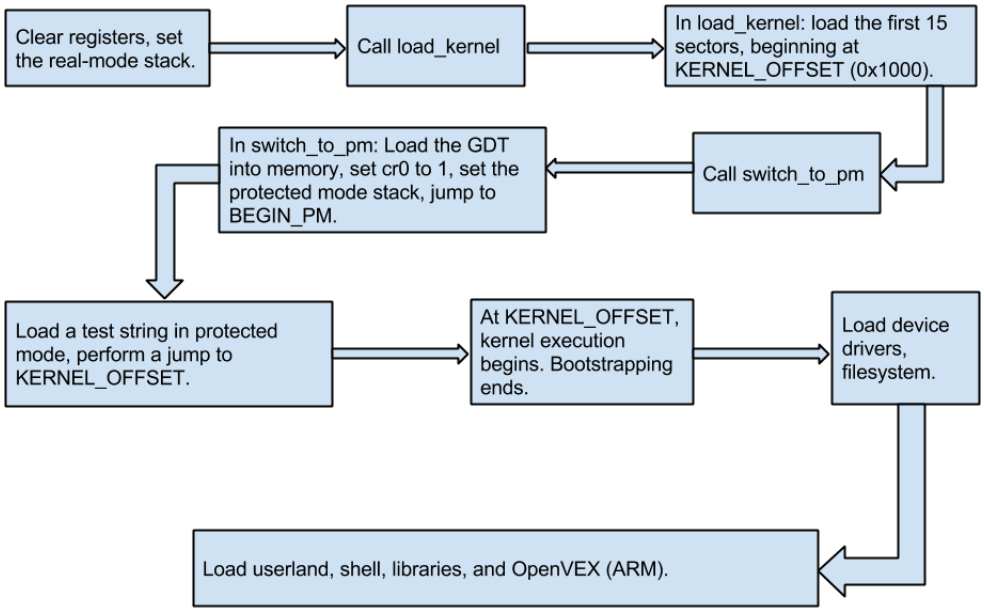

Fig. 1: Bootstrapping on i386.

Part IV. Interfacing With Hardware

There are 2 main components to input and output for this operating system, keyboard and screen, and motors and sensors.

Keyboard and Screen Information is sent from the keyboard to the processor via a standard protocol. Inside any commercial keyboard is a tiny computer chip called a *microcontroller*. When a key is pressed down, the microcontroller receives the signal and starts a short countdown timer. If your finger is still pressing down on the key after more than a millisecond, the microcontroller looks up a numerical value for the key and saves it in its small memory bank.

The data is then sent to the computer along a cable, usually USB. At any given moment, power is passing through this cable to supply power to the keyboard, as well as to synchronize the keyboards internal timer. When the computer receives this data is sent to a program known as the *driver* for processing. The driver interprets this information and sends it to the input stream of whatever program is currently running in the form of bytes, hexadecimal numbers representing letters, symbols and punctuation marks. The program then processes the information in a way that makes sense, be that displaying the symbol on a screen, saving it locally in a document, or interpreting it as a *control code*.

A good example of this is the `CTRL+S` command. When you hold down the `CTRL` key, information is sent to the keyboard's on-board processor that that key is being held down. When you then press the `S` key, the numerical value for that key is sent for processing. The processor sees the code for the `S` key, but since the `CTRL` key is also held down, it changes the value sent to the computer to a control code. On your computer, the driver is monitoring continual packets of information sent in to the computer. Since the keyboard sends hundreds of these per second, most are empty. When one arrives with the control code you sent, the driver tells your operating system to process it. Your OS then tells your word processor. (Say it's your active window) When your word processor receives this information, (Some OS'es will relay the control code directly, others will interpret it and send their own message) it then takes appropriate action and saves your document.

The process works in reverse for display drivers. Since our OS uses a standard Video Graphics Array (VGA), I'll use that as an example.VGA monitors don't operate digitally like computers do, instead requiring a constant stream of electrons *analogous* to the image displayed, much like older TV's and radios. This poses a problem in that when programs send display data to your OS, your OS stores its images to output as digital files, long lists of discrete numbers. These two kinds of representing data don't play nice with each other, and it's the job of another driver to convert the signal.This driver takes the image, and splits it into red, green and blue elements, and stores it in its *video memory*. Think of the video memory as a canvas, a place where the driver draws what you see. After splitting the image into the three colors and drawing out each pixel as a combination of the three, the driver then sends the image as a set of analog signals, the same kind that travel to your cable box. For as long as the image in video memory remains unchanged, it keeps getting sent, and any changes to video memory are sent immediately. Sending these signals continuously means that the image always stays on your monitor.

Say we want to display a picture in the center of the screen. The program we use to do that, say ImageViewer or something, will take the picture data and send it to the OS in the form of digital values. How exactly depends on the image format and program you're using, but by the time the image gets to the OS, it's represented as a string of pixels. These pixels are then sent to the VGA driver, which breaks them down into red, green and blue values. Once that's done, it sends this information out through its pins in the form of analog signals, one color per pin, with extra pins used for time synchronization and error checking. These signals are directed via your monitor's on-board processor to their appropriate pixels, which light up in the colors you want. From a distance of a few inches, these colors blend to form your picture.

Sensors and Motors Information was originally sent from central control units to motors using a mechanism known as a *rheostat*. Basically, a rheostat was a long coil of wire. In order to get a motor to run at different speeds, current would be added at different points along the wire, and the amount of curent that wasn't dissipated by the wire drove the motor. While it was perfectly adequate in large machines, this approach doesn't work in robotics. While it's easy to set up, it takes up way too much space and dissipates way too much heat.

To solve this problem, modern servos use a system known as *Pulse Width Management* to regulate power sent to motors and sensors. The actual mechanics are somewhat complicated, but the principle itself is quite simple. Alternating current, as we know, can oscillate in many different patterns, triangle, square, sawtooth, what have you, but the most common form of oscillation is the simple square wave, easy to convert to binary and simple to generate. What PWM does is power the motor with a quickly oscillating square wave AC current. In order to change the power supplied to the motor, the microcontroller varies the width of the pulses. In order to make the

motor run faster, we make the "ON",or positive, portions of the wave longer, and the "OFF", or negative, portions shorter, raising the average value of the wave,which the motor interprets as more or less power. The opposite is true. By widening the "OFF" sections, we can slow the motor down. PWM also works for sensors. By sending back pulses of different widths, sensors can send us data about brightness, color or quantity. (Say, for a motor encoder.)

Once all the drivers are in place and running, it's a simple matter to call their functionality from within the OS kernel, making the data sent and received readily usable by all programs on the device.

References

[1] Blundell, Nick "Writing a Simple Operating System - From Scratch". Web. <http://www.cs.bham.ac.uk/ ẽxr/lectures/opsys/10_11/lectures/os-dev.pdf>. 2010.

[2] Bos, Herbert. "A Basic Kernel in C." Web. 22 May 2014. <http://www.cs.vu.nl/%7Eherbertb/misc/basickernel.pdf>.

[3] "Operating System Development Series." *BrokenThorn Entertainment.* Web. 08 May 2014. <http://www.brokenthorn.com/Resources/>

[4] Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. *Applied Operating System Concepts.* New York: John Wiley, 2000. Print.