What Operating Systems Need from the Hardware

Bob Brown
Computer Science Department
Southern Polytechnic State University

About Operating Systems

It is possible to write effective operating systems for even very simple hardware. Early operating systems ran on computers with a few thousand words of memory and a few dozen instructions. The hardware of the original IBM PC was primitive for its time, and yet the PC and the MS-DOS operating system launched the personal computer revolution.

In this paper, we aren't talking about primitive operating systems. We will discuss what is required of the computer hardware for a *robust multiprogramming operating system.* Let's examine that description for a minute. A *multiprogramming operating system* is one that can run two or more programs at the same time, possibly for two or more users. A *robust* operating system is one where each user appears to have complete control of the machine, and where errors in a running program can't affect other programs running at the same time, or the operating system itself. In other words, the operating system is able to protect itself and the other programs it is running from malicious or defective programs.

There are three requirements that a robust multiprogramming operating system must make of the hardware. They are:

- Privileged instructions
- Protected memory, and
- A timer that can generate interrupts.

There are many more things a computer architecture can do to make things easier for writers of operating systems, but these three are absolute requirements. The other things can be embodied in either hardware or software, but these three must be in the hardware; the writer of the operating system cannot "program around" their absence.

Let's examine each of the requirements in turn.

Privileged Instructions

It is possible to build computer systems so that there are two classes of machine instructions. A "mode bit," often in the program status word, determines whether the CPU is in privileged mode or user mode. (Privileged mode is also called supervisor mode or kernel mode.) When the CPU is in privileged mode, both classes of instructions are considered to be valid. When the CPU is in user mode, only one class of instructions, the user mode instructions, is valid. Attempts to execute privileged instructions while the CPU is in user mode result in illegal instruction traps.

Why do we need privileged instructions? To keep user programs from doing things only the operating system should do. Recall that writing a block to disk means moving the disk head, waiting for the proper sector, then actually doing the write. Suppose some other program commanded the disk head to move just as your program started writing. There's no telling where your data would end up, but you probably couldn't find it again, and it would probably damage something else. Things like physical-level I-O need to be left to the operating system on a multi-user computer, and the only way to be certain that happens is to prevent user-mode programs from executing the I-O instructions.

There are other instructions besides I-O that belong in the privileged set. These include setting and removing memory protection, setting and resetting system timers, and many other things that should be handled by a single, well-tested program rather than by a collection of user programs.

Protected Memory

If all the memory in a computer is viewed as a single address space, and if any program can write to any location, some other program can write to memory used by your program, probably causing your program to crash or produce erroneous results. Even worse, user programs could write to memory owned by the operating system, causing the operating system itself to crash.

One way to prevent writes into another program's memory is to divide memory into allocation units similar to the page frames of a virtual memory system. When a program needs memory, it is allocated as one or more pages, and tag bits in the hardware of the memory subsystem record which program owns each page. Programs may not write into memory areas they don't own; attempts generate illegal address traps. In a refinement, other programs *may* be allowed to read pages they don't own depending upon the setting of a bit. In this way, a single copy of program code or static data may be shared among two or more running programs, conserving memory resources.

In computer systems with virtual memory, programs and the operating system may be protected from each other by giving each its own virtual memory address space. The virtual memory subsystem of the operating system handles mapping of virtual pages into real page frames transparently to the running program. Since every possible address a program can generate is within that program's address space, writing into another program's space is a logical impossibility.

Unfortunately, this approach makes sharing static program code and data a logical impossibility, too. Some operating systems divide the virtual address space in half, with shared code and data in one half, and "visible" to all running programs, and private storage in the other half, protected from both reading and writing.

A Timer That Can Generate Interrupts

So far we have protected against user programs executing privileged instructions and against writing to memory they don’t own. That isn’t quite enough. In ordinary circumstances, the operating system gets control of the CPU when the running user program needs an operating system service such as an I-O operating. The operating system starts the I-O operation, then lets some *other* program run while the requested operation completes. In this way, many programs appear to run simultaneously, and nearly as fast as if they really had the computer to themselves.

If a program got into an infinite (or very long) loop that did not contain any request for operating system services, the operating system would never get a chance to run, and so would never let any other programs run. The computer would appear to freeze up, although in reality the CPU would be executing instructions at full speed.

Infinite loops frequently happen by accident. It is easy to write a malicious program that consists only of an infinite loop, and certain numerical calculations, such as finding large prime numbers, can go on for a very long time without requesting services of the operating system.

What is needed to prevent these kinds of programs from freezing the computer system is a hardware timer that generates interrupts. Such a timer counts down a very small time interval, say 100 microseconds, independent of what the CPU is doing. At the end of the interval, the timer generates a hardware interrupt that stops the running program and gives control to the operating system. Even if the running program were in an infinite loop, the operating system would get control when the timer ran down. It could then transfer control to another user program that is ready to run.

Malicious programs would still be able to consume their full allotment of CPU cycles, but they can no longer freeze the entire system. Programs with errors that caused infinite loops would appear to freeze, but the rest of the system would keep running, and eventually someone would stop the erroneous program. And programs doing intensive number crunching will run normally, with minimal impact on the rest of the computer system.

As a refinement on this technique, the operating system could reset the timer (using a privileged instruction) just before transferring control to a user program. As long as operating system resources are being requested frequently enough, timer interrupts are not needed, and resetting the timer will prevent them from happening.

What About the Logical Equivalence of Hardware and Software?

You may be wondering why we are so insistent that these three functions be embodied in hardware. After all, we’ve said hardware and software are logically equivalent. Couldn't the writers of operating systems just "program around" these limitations?

The answer is no. When we say that hardware and software are logically equivalent, we make the assumption that both the hardware and the software are *correct.* We put requirements on the hardware to compensate for the possibility that some programs that will run on our computer will be incorrect. If we could assume that user programs would never try to execute forbidden instructions, never write to memory that doesn't belong to them, and give up control of the CPU at frequent intervals, then the hardware described here would be unnecessary. However, experience has shown that it is not safe to make any of those three assumptions. So, for the operating system to be robust, support from the hardware is a requirement.