

Read Chapter 1 in the book. This will give you one perspective on the how and why of operating systems. I want to give you a somewhat different perspective.

The early designers of computers were faced with a number of problems that urgently needed to be solved. They developed the first operating systems in response to these problems. The process of solving these problems is straightforward, and the solution is reasonable. If you had been faced with the same sorts of problems, you would probably have come up with a solution that was identical, or at least very similar to the operating systems that exist today.

Here are the problems.

1. A computer was an extremely expensive piece of equipment. Even the smallest computer cost hundreds of thousands of dollars, and a computer of any reasonable size was several million dollars. For example one computer that I used extensively back in the early 1970's had a meg and a half of memory, could execute about half a million instructions per second (as opposed to billions for today's microprocessors) and cost three million dollars. It was important to use this very expensive hardware as effectively as possible.

2. Only one person could use the computer at a time. At one time, you would punch your program into a set of cards, then go to the computer room and wait in line for your turn to use the computer. When your turn came, you would go up to the front panel of the computer, bootstrap your program into the computer by using the front panel buttons, and then wait for the printout. Anyone else who wanted to use the computer had to wait until you were done.

3. Despite the slowness of early computers I/O operations were much slower than ordinary computation. The computer was capable of executing thousands of instructions while a single card was being read. Nevertheless, I/O operations were synchronous, so the instruction execution unit would interlock waiting for the I/O operations to complete, thus wasting valuable time doing nothing.

One solution to problem 3 was asynchronous I/O. I/O instructions were modified so that they operated independently of the instruction execution unit. An I/O instruction started the I/O operation, but the CPU continued to execute instructions while the I/O operation was taking place. The program had to be written in such a way as to request an I/O operation a long time before the data was actually required. Then, when the program reached the point where it really needed the data, it had to execute another instruction to wait for completion of the I/O operation. This wait instruction was synchronous. It interlocked the CPU until the I/O operation was complete.

Despite the use of asynchronous I/O, most programs spent most of their time waiting for I/O. In fact most programs spent over 90% of their time waiting for I/O. Thus you

have a multi-million dollar piece of equipment that is sitting idle 90% of the time, even though there are people standing in line waiting to use it. This is the sort of thing that is guaranteed to drive most computer scientists crazy.

Thus, we come to *the big idea*. *Why not run two programs at the same time?*

We have to be careful with this idea, because the CPU really isn't capable of doing this. The CPU can only execute one instruction at a time. Therefore, it can only run one program at a time. To make the big idea work, we have to do something clever. What we have to do is make the CPU jump around from one program to another. When one program is waiting for I/O, we don't interlock the CPU, we simply have the CPU jump to a different program and begin executing that program instead. Two programs might not be enough to keep the CPU busy all the time, but if we can run two programs at the same time, we can run three, or four, or a whole bunch. And sooner or later, we'll have enough programs running to keep the CPU busy most of the time, thus making the best use of our multi-million dollar piece of equipment.

Running two programs at a time poses several problems.

1. What if both programs want to use the same section of memory? Can we move one of them? What about address constants? What about absolute addresses?
2. What if one of the programs decides it doesn't want to give up the CPU while it is waiting for I/O? How do we force the program to give up the CPU?
3. What if one of the programs is malicious and decides to damage the other, or the operating system?
4. When an I/O operation completes, how do we find out about it? How do we restart a program that is waiting for I/O to complete?
5. If I have a choice between running program A or program B, which do I choose?

Once we have the answers to these questions we will have most of what we need to design an operating system.

Interestingly enough, we set out to solve one problem, but we have inadvertently solved another one at the same time. The problem of only one person being able to use the computer at a time is implicitly solved by allowing more than one program to run simultaneously. There are issues that pertain exclusively to this problem, and we will examine those when we get to CPU scheduling.