I/O Bound vs CPU Bound

In this lesson, we delve into the characteristics of programs with different resource-use profiles and how that can affect program design choices.

I/O Bound vs CPU Bound

We write programs to solve problems. Programs utilize various resources of the computer systems on which they run. For instance a program running on your machine will broadly require:

- CPU Time
- Memory
- Networking Resources
- Disk Storage

Depending on what a program does, it can require heavier use of one or more resources. For instance, a program that loads gigabytes of data from storage into the main memory would hog the main memory of the machine it runs on. Another can be writing several gigabytes to permanent storage, requiring abnormally high disk I/O.

CPU Bound

Compute-intensive programs, i.e., where the program execution requires very high utilization of the CPU (close to 100%), are called CPU bound programs. Such programs primarily depend upon improving CPU speed to decrease program completion time. This could include programs such as data crunching, image processing, matrix multiplication, etc.

If a CPU bound program is provided a more powerful CPU, it can

potentially complete faster. Eventually, there is a limit on how powerful a single CPU can be. At this point, the recourse is to harness the computing power of multiple CPUs and structure your program code in a way that can take advantage of the multiple CPU units available. Say we are trying to sum up the first 1 million natural numbers. A single-threaded program would sum in a single loop from 1 to 1000000. To cut down on execution time, we can create two threads and divide the range into two halves. The first thread sums up the numbers from 1 to 500000 and the second sums up the numbers from 500001 to 1000000. If there are two processors available on the machine, then each thread can independently run on a single CPU in parallel. In the end, we sum up the results from the two threads to get the final result. Theoretically, the multithreaded program should finish in half the time that it takes for the single-threaded program. However, there will be a slight overhead of creating the two threads and merging the results from the two threads.

Multithreaded programs can improve performance in cases where the problem lends itself to being divided into smaller pieces that different threads can work on independently. However, this may not always be true.

I/O Bound

I/O bound programs are the opposite of CPU bound programs. Such programs spend most of their time waiting for input or output operations to complete while the CPU sits idle. I/O operations can consist of operations that write or read from main memory or network interfaces. Because the CPU and main memory are physically separate, a data bus exists between the two to transfer bits to and fro. Similarly, data needs to be moved between network interfaces and CPU/memory. Even though the physical distances are tiny, the time taken to move the data across is long enough for several thousand CPU cycles to go to waste. This is why I/O bound programs would show relatively lower CPU utilization than CPU bound programs.

Notes

Both types of programs can benefit from concurrent architectures. If a program is CPU bound, we can increase the number of processors and structure our program to spawn multiple threads that run individually on

a dedicated or shared CPU. For I/O bound programs, it makes sense to have a thread give up CPU control if it is waiting for an I/O operation to complete so that another thread can get scheduled on the CPU and utilize CPU cycles. Different programming languages come with varying support for multithreading. For instance, JavaScript is single-threaded, Java provides full-blown multithreading and Ruby is *sort* of multithreaded as you'll learn in later chapters. However, all three languages support asynchronous programming models, which is another way for programs to be concurrent (but not parallel).

For completeness, we should mention that there are also memory-bound programs that depend on the amount of memory available to speed up execution.