ECE254: Operating Systems and Systems Programming Spring 2015

Lecture 10 — Symmetric Multiprocessing (SMP)

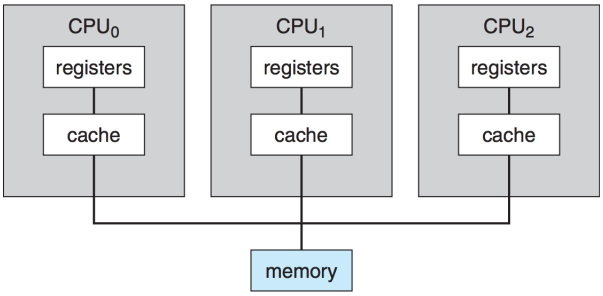
Jeff Zarnett

Multiprocessing

Not that long ago, a typical computer had one processor with one core. It could accordingly do exactly one thing at a time. When we say there is one processor, it's one general purpose processor that executes user processes. There may be additional special-purpose processors in the system (e.g., a RAID controller[1]) but there is only one general purpose processor so we call it a uniprocessor system.

Now, desktops, laptops, and even cell phones are almost certainly using multi-core processors. A quad-core processor may be executing four different instructions from four different threads at the same time. In theory, multiple processors may mean that we can get more work done in the same amount of (wall clock) time, but this is not a guarantee. Plus, if one of the CPUs in a two-CPU system fails, the system can likely carry on at a reduced performance level.

Multiple processor systems may be *asymmetric multiprocessing* systems or *symmetric multiprocessing* systems. If they are asymmetric, then there is a *boss* processor that is in charge of controlling the system: all other processors are *workers* and they take instructions from the boss or have a very specific task to perform. Most systems these days, however, are more collectivist: the workers are in charge and all processors are equal, comrade![2] Most of the systems we are familiar with (laptops, phones, etc) are symmetric multiprocessing (SMP), but in theory whether a system is symmetric or asymmetric can be configured in software; SunOS version 4 was asymmetric but version 5 (Solaris) is symmetric on the same hardware [SGG13].



Symmetric multiprocessing architecture [SGG13].

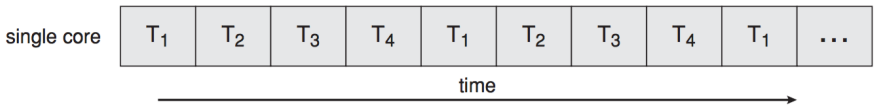
Terminology note: we often refer to a logical processing unit as a *core*. The term CPU may refer to a physical chip that contains one or more logical processing units. As far as the operating system is concerned, it does not much matter if a system has four cores in four physical chips or four cores in one chip; either way, there are four units that can execute instructions.

If there is exactly one process with one thread running in the system, then it does not matter how many cores are available: at most one core will be used to execute this task. If there are multiple processes, each process can execute on a different core. But what do we do if there are more processes and threads than available cores? We can hope that the processes get blocked frequently enough and long enough so that all processes get to run, but this is not something we can count on.

[1] The graphics card seems like a more obvious example, but these days there are various programs that can make use of the powerful GPU to do general purpose computation.

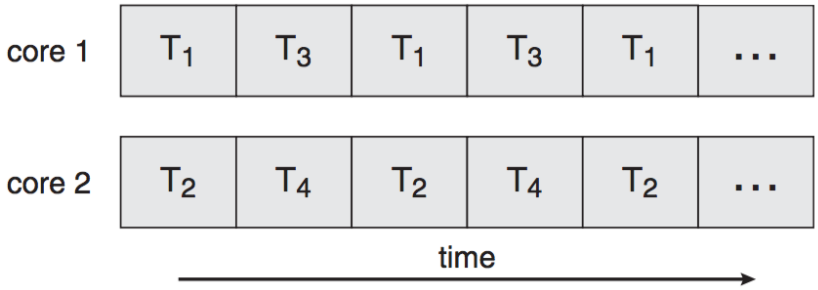
[2] History lesson: this idea did not work out so well for the Soviet Union.

Our solution is that the CPU should switch between the different tasks via a procedure we call *time slicing*. So thread 1 would execute for a designated period, such as 20 ms, then thread 2 for 20 ms, then thread 3 for 20 ms, then back to thread 1 for 20 ms. To the user, it seems like threads 1, 2, and 3 are being executed in parallel, because 20 ms is fast enough that the user does not notice the difference.



Execution of different Threads T_1 through T_4 on a single core [SGG13].

Time slicing of execution will still occur, if necessary. Continuing our example, if there are four threads running on a dual-core system, time slicing is necessary to run all those programs.



Execution of different Threads T_1 through T_4 on two cores [SGG13].

The subject of time slicing is something we will return to later on in our discussion of scheduling.

Parallelism and Speedup

No doubt it has occurred to you that if there are multiple threads running at the same time, it means a task will get completed faster, right? Well... maybe. It depends a lot on what the task is. There is some overhead involved in splitting a task up and re-combining the results (if necessary), but in most circumstances the overhead is negligible compared to the amount of time working on the task.

If a task can be fully parallelized, it means the task can be split up in such a way that adding a second executing thread would double the speed of execution. Imagine painting an apartment. It would take one person 12 hours to paint the whole apartment and two people could complete the job in 6 hours. The pattern continues: three people can complete the job in 4 hours, four people in 3 hours, et cetera. This is the ideal, but in the real world there will be a limit to how many additional workers you can add and continue to have this speedup characteristic. At some point, the overhead of adding more threads is no longer negligible. In theory, you could hire 720 painters and finish the job in 1 minute, but at some point you cannot physically fit any more painters into the room.

If a task can be partially parallelized, it means the task can be divided, but doubling the workers doesn't result in completing the job in half the time. Two chefs working together in a kitchen might take 75% of the time it would take one chef to cook a meal. Adding the extra worker to the kitchen improved the speed at which food was prepared, but it's not doubled. The chefs can work independently some of the time, but at other times one has to wait for the other; the sauce cannot be put on the chicken until the chicken comes out of the oven.

If a task cannot be parallelized at all, then no amount of extra workers will speed it up. Some tasks can only be done sequentially. As the folk saying goes, "Nine women can't have a baby in one month."

Let us consider an example from [HZM14]: Suppose we have a task that can be executed in 5 s and this task contains a loop that can be parallelized. Let us also say initialization and recombination code in this routine

requires 400 ms. So with one processor execution it would take about 4.6 s to execute the loop. If we split it up and execute on two processors it will take about 2.3 s to execute the loop. Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s. Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%.

A smart fellow by the name of Gene Amdahl came up with a formula for the general case of how much faster a task can be completed based on how many processors we have available. Let us define S as the portion of the application that must be performed serially and N as the number of processing cores available.

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

This is a math formula after all and you can do things like take the limit as N approaches infinity and you will find the speedup converges to $\frac{1}{S}$. So the limiting factor on how much additional processors help is, of course, the size of the S term in the equation.

Applying this formula to the example from earlier, we get the following run times:

Processors	**Run Time (s)**
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

There are two observations from this data immediately. The first is that we get diminishing returns as we add more processors. Going from 1 to 2 processors reduced the runtime dramatically, but going from 64 to 128 reduced the run time only a very small amount. The second is that as we continue to add more processors we are converging on a run time of 0.4 s, which fits our expectations of what would happen with infinite processors. The serial part will take 0.4 s no matter what, and with infinite processors the parallel part would be (effectively) instant. Again, applying the formula, the most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$. It is not possible to do better than this. In reality we will never be able to equal the limit either, because nobody has infinite processors available, considering that would take an infinite amount of space and an infinite amount of money...

References

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghami. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.