ECE459: Programming for Performance Winter 2013

Lecture 3 — January 15, 2013

Patrick Lam *version 1*

Concurrency and Parallelism

Concurrency and parallelism both give up the total ordering between instructions in a sequential program, for different purposes.

Concurrency. We'll refer to the use of threads for structuring programs as concurrency. Here, we're not aiming for increased performance. Instead, we're trying to write the program in a natural way. Concurrency makes sense as a model for distributed systems, or systems where multiple components interact, with no ordering between these components, like graphical user interfaces.

Parallelism. We're studying parallelism in this class, where we try to do multiple things at the same time in an attempt to increase throughput. Concurrent programs may be easier to parallelize.

Processor Design Issues

Recall that we listened to Cliff Click describe characteristics of modern processors in Lecture 2. In this lecture we'll continue our quick review of computer architecture and how it relates to programming for performance. Here's another reference about chip multi-threading; we are going to study some of the techniques in the "Writing Scalable Low-Level Code" section.

`http://queue.acm.org/detail.cfm?id=1095419`

Processes and Threads. Let's review the difference between a process and a thread. A *process* is an instance of a computer program that contains program code and its own address space, stack, registers, and resources (file handles, etc). A *thread* usually belongs to a process. The most important point is that it shares an address space with its parent process, hence variables and code as well as resources. Threads have their own stack, registers, and thread-specific data.

Threads and CPUs. In your operating systems class, you've seen implementations of threads ("lightweight processes"). We'll call these threads *software threads*, and we'll program with them throughout the class. Each software thread corresponds to a stream of instructions that the processor executes. On a old-school single-core, single-processor machine, the operating system multiplexes the CPU resources to execute multiple threads concurrently; however, only one thread runs at a time on the single CPU.

On the other hand, a modern chip contains a number of *hardware threads*, which correspond to the virtual CPUs. These are sometimes known as *strands*. The operating system still needs to multiplex the software threads onto the hardware threads, but now has more than one hardware thread to schedule work onto.

What's the term for swapping out the active thread on a CPU?

Implementing (or Simulating) Hardware Threads. There are a number of ways to implement multiple software threads; for instance, the simplest possible implementation, **kernel-level threading** (or 1:1 model) dedicates one core to each thread. The kernel schedules threads on different processors. (Note that kernel involvement will always be required to take advantage of a multicore system). This model is used by Win32, as well as POSIX threads for Windows and Linux. The 1:1 model allows concurrency and parallelism.

Alternately, we could make one core execute multiple threads, in the **user-level threading**, or N:1, model. The single core would keep multiple contexts and could 1) switch every 100 cycles; 2) switch every cycle; 3) fetch one instruction from each thread each cycle; or 4) switch every time the current thread hits a long-latency event (cache miss, etc.) This model allows for quick context switches, but does not leverage multiple processors. (Why would you use these?) The N:1 model is used by GNU Portable Threads.

Finally, it's possible to both use multiple cores and put multiple threads onto one core, in a **hybrid threading**, or M:N, model. Here, we map M application threads to N kernel threads. This is a compromise between the previous two models, which both allows quick context switches and the use of multiple processors. However, it requires increased complexity; the library provides scheduling services, which may not coordinate well with kernel, and increases likelihood of priority inversion (which you've seen in Operating Systems). This method is used by modern Windows threads.

Multicore Processors

As I've alluded to earlier, multicore processors came about because clock speeds just aren't going up anymore. We'll discuss technical details today.

Each processor *core* executes instructions; a processor with more than one core can therefore simultaneously execute multiple (unrelated) instructions.

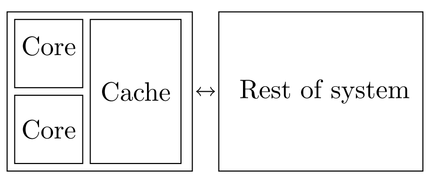
Chips and cores. Multiprocessor (usually SMP, or symmetric multiprocessor) systems have been around for a while. Such systems contain more than one CPU. We can count the number of CPUs by physically looking at the board; each CPU is a discrete physical thing.

Cores, on the other hand, are harder to count. In fact, they look just like distinct CPUs to the operating system:

```
plam@plym:~/courses/p4p/lectures$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
...
processor : 1
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
```

If you actually opened my computer, though, you'd only find one chip. The chip is pretending to have two *virtual CPUs*, and the operating system can schedule work on each of these CPUs. In general, you can't look at the chip and figure out how many cores it contains.

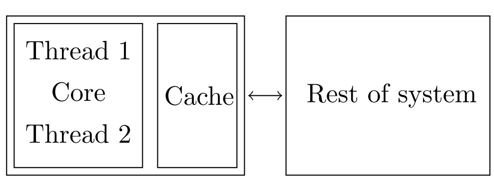
Hardware Designs for Multicores. In terms of the hardware design, cores might share a cache, as in this picture:



(credit: *Multicore Application Programming*, p. 5)

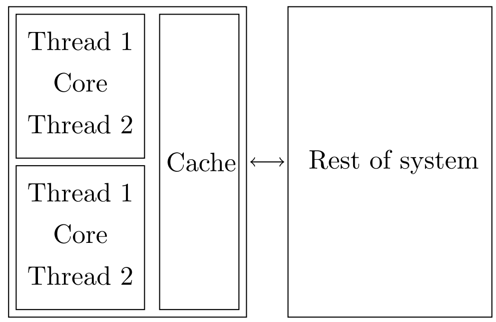
This above Symmetric Multithreading (SMP) design is especially good for the 1:1 threading model. In this case, the design of the cores don't need to change much, but they still need to communicate with each other and the rest of the system.

Or, we can have a design that works well for the N:1 model:



One would expect that executing two threads on one core might mean that each thread would run more slowly. It depends on the instruction mix. If the threads are trying to access the same resource, then each thread would run more slowly. If they're doing different things, there's potential for speedup.

Finally, it's possible to both use multiple cores and put multiple threads onto one core, as in the M:N model:



Here we have four hardware threads; pairs of threads share hardware resources. One example of a processor which supports chip multi-threading (CMT) is the UltraSPARC T2, which has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

Non-SMP systems. The designs we've seen above have been more or less SMP designs; all of the cores are mostly alike. A very non-Smp system is the Cell, which contains a PowerPC main core (the PPE) and 7 Synergistic Processing Elements (SPEs), which are small vector computers.

Non-Uniform Memory Access. In SMP systems, all CPUs have approximately the same access time for resources (subject to cache misses). There are also NUMA, or Non-Uniform Memory Access, systems out there. In that case, CPUs can access different resources at different speeds. (Resources goes beyond just memory).

In this case, the operating system should schedule tasks on CPUs which can access resources faster. Since memory is commonly the bottleneck, each CPU has its own memory bank.

Using CMT effectively. Typically, a CPU will expose its hardware threads using virtual CPUs. In current hardware designs, each of the hardware threads has the same performance.

However, performance varies depending on context. In the above example, two threads running on the same core will most probably run more slowly than two threads running on separate cores, since they'd contend for the same core's resources. Task switches between cores (or CPUs!) are also slow, as they may involve reloading caches.

Solaris "processor sets" enable that operating system to assign processes to specific virtual CPUs, while Linux's "affinity" keeps a process running on the same virtual CPU. Both of these features reduce the number of task switches, and processor sets can help reduce resource contention, along with Solaris's locality groups[1]

[1]Gove suggests that locality groups help reduce contention for core resources, but they seem to help more with memory.