ECE459: Programming for Performance Winter 2018

Lecture 4 — Concurrency and Parallelism

Jeff Zarnett, based on original by Patrick Lam

Concurrency and Parallelism

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

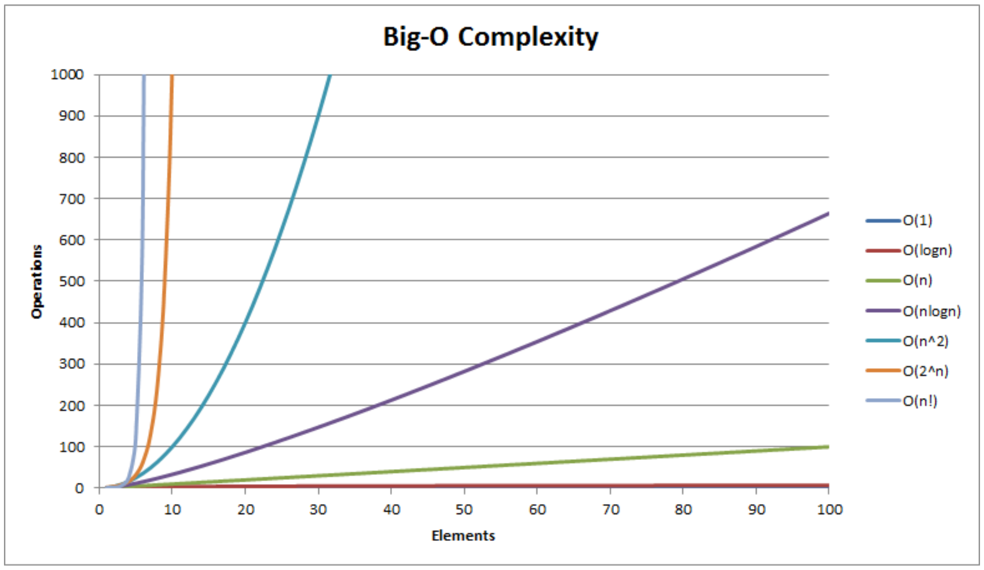
Concurrency. We'll refer to the use of threads for structuring programs as concurrency. Here, we're not aiming for increased performance. Instead, we're trying to write the program in a natural way. Concurrency makes sense as a model for distributed systems, or systems where multiple components interact, 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 of [McD05]:

Scalable Algorithms. Remember from ECE 250 that we often care about the worst case run-time performance of the algorithm. An algorithm that's $O(n^3)$ scales so much worse than one that's $O(n)$ that it's not even funny. Trying to do an insertion sort on a small array is fine (actually... recommended); doing it on a huge array is madness. Choosing a good algorithm is very important if we want it to scale.



Big-O Complexity comparison from [R+15]

Locking. Think back to the operating systems and systems programming course and the discussion of locking and concurrency. We'll be coming back to this subject before too long. But for now, suffice it to say, that the more locks and locking we need, the less scalable the code is going to be. You may think of the lock as a resource and the more threads or processes that are looking to acquire that lock, the more "resource contention" we have, and the more waiting and coordination are going to be necessary.

Cache Line Sharing. Multiprocessor (multicore) processors have some hardware that tries to keep the data consistent between different pipelines and caches (as we saw in the video). More processors, more threads means more work is necessary to keep these things in order.

Pools of Worker Threads. If we have a pool of workers, the application just submits units of work, and then on the other side these units of work are allocated to workers. The number of workers will scale based on the available hardware. This is neat as a programming practice: as the application developer we don't care quite so much about the underlying hardware. Let the operating system decide how many workers there should be, to figure out the optimal way to process the units of work.

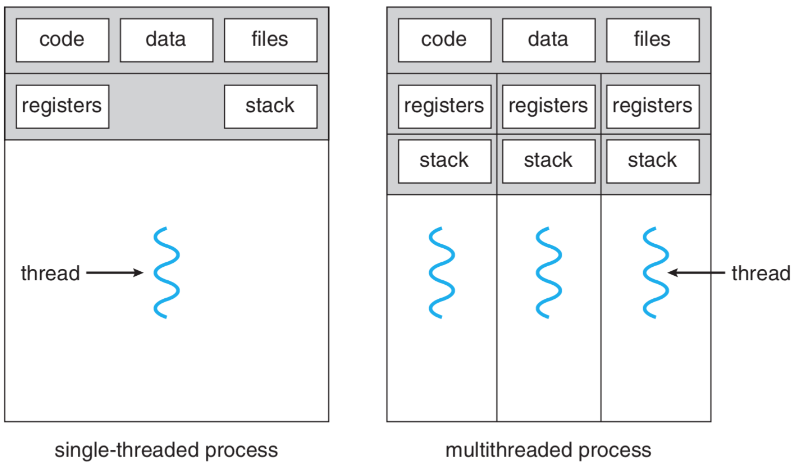
Memory Allocators. Assuming we're not working with an embedded system where all memory is statically allocated in advance, there will be dynamic memory allocation. The memory allocator is often centralized and may support only one thread allocating or deallocating at a time. This means it does not necessarily scale very well, though there are some techniques for dynamic memory allocation that allow these things to work in parallel.

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. Each thread has its own [Sta14]:

1. Thread execution state (like process state: running, ready, blocked...).
2. Saved thread context when not running.
3. Execution stack.
4. Local variables.
5. Access to the memory and resources of the process (shared with all threads in that process).

Or, to represent this visually:



A single threaded and a multithreaded process compared side-by-side [SGG13].

All the threads of a process share the state and resources of the process. If one thread opens a file, other threads in that process can also access that file.

The way programs are written now, there are few if any that are not in some way multithreaded. One common way of dividing up the program into threads is to separate the user interface from a time-consuming action.

Consider a file-transfer program. If the user interface and upload method share a thread, once a file upload has started, the user will not be able to use the UI anymore (and Windows will put the dreaded "(Not Responding)" at the end of its dialog title), even to click the button that cancels the upload. For some reason, users hate that.

We have two options for how to alleviate this problem: when an upload is ready to start, we can call `fork` and create a new process to do the upload, or we can spawn new thread. In either case, the newly created entity will handle the upload of the file. The UI remains responsive, because the UI thread is not waiting for the upload method to complete.

Motivation for Threads

Why choose threads rather than creating a new process? The primary, but not sole, motivation is performance:

1. Creating a new thread is much faster than creating a new process. In fact, thread creation is on the order of ten times faster [TRG+87].
2. Terminating and cleaning up a thread is faster than terminating and cleaning up a process.
3. It takes less time to switch between two threads within the same process (because less data needs to be stored/restored). In Solaris, for example, switching between processes is about five times slower than switching between threads [SGG13].
4. Because threads share the same memory space, for two threads to communicate, they do not have to use any of the IPC mechanisms; they can just communicate directly.
5. As in the file transfer program, use of threads allows the program to be responsive even when a part of the program is blocked.

This last advantage, background work, is one of four common examples of the uses of threads in a general purpose operating system [Let88]:

1. **Foreground and Background Work:** as already examined, the ability to run something in the background to keep the program responsive.
2. **Asynchronous processing**: for example, to protect against power failure or a crash, a word processor may write the document data in main memory to disk periodically. This can be done as a background task so it does not disrupt the user's workflow.
3. **Speed of Execution:** a multithreaded program can get more done in the same amount of time. Just as the OS can run a different program when the executing program gets blocked (say, on a disk read), if one thread is blocked, another thread may execute.
4. **Modular Structure:** a program that does several different things may be given structure through threads.

There are some drawbacks, however: there is no protection between threads in the same process: so one thread can easily mess with the memory being used by another thread. This once again brings us to the subject of co-ordination, which will follow the discussion of threads.

Also, if any thread encounters an error (such as a division by zero or Segmentation Fault), the whole process might be terminated by the operating system. If the program has multiple processes for different parts, then the other processes will not be affected; if the program has multiple threads and they all share the same process, then any thread encountering an error might bring all of them to a halt.

You can find another explanation of processes versus threads here:

`http://www.purplealienplanet.com/node/50`

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.

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.

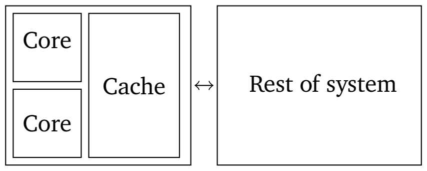
Cores, on the other hand, are harder to count. In fact, they look just like distinct CPUs to the operating system:

```
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.

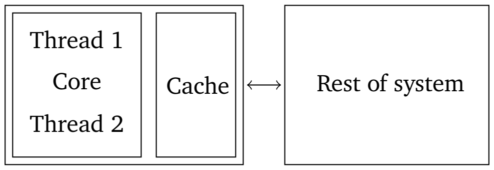
Hardware Designs for Multicores. In terms of the hardware design, cores might share a cache, as in this picture:



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

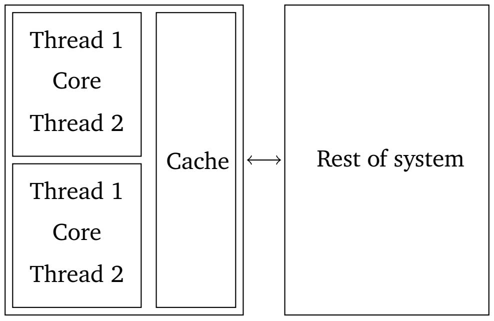
This above Symmetric Multithreading (SMP) design is especially good for the 1:1 threading model. In this case, the design of the cores don't need to change much, but they still need to communicate with each other and the rest of the system.

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]

Processes versus Threads

The first design decision that you need to solve when parallelizing programs is whether you should use threads or processes.

- Threads are basically light-weight processes which piggy-back on processes' address space.

Traditionally (pre-Linux 2.6) you had to use `fork` (for processes) and `clone` (for threads). But `clone` is not POSIX compliant, and its man page says that it's Linux-specific—FreeBSD uses `rfork()`. (POSIX is the standard for Unix-like operating systems).

When processes are better. `fork` is safer and more secure than threads.

1. Each process has its own virtual address space:
   - Memory pages are not copied, they are copy-on-write. Therefore, processes use less memory than you would expect.
2. Buffer overruns or other security holes do not expose other processes.
3. If a process crashes, the others can continue.

Example: In the Chrome browser, each tab is a separate process. Scott McCloud explained this: `http://uncivilsociety.org/2008/09/google-chrome-comic-by-scott-m.html`.

When threads are better. Threads are easier and faster.

1. Interprocess communication (IPC) is more complicated and slower than interthread communication; must use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library (or just memory reads and writes).
2. Processes have much higher startup, shutdown and synchronization costs than threads.
3. Pthreads fix the issues of `clone` and provide a uniform interface for most systems. (You'll work with them in Assignment 1.)

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

How to choose? If your application is like this:

- mostly independent tasks, with little or no communication;
- task startup and shutdown costs are negligible compared to overall runtime; and
- want to be safer against bugs and security holes,

then processes are the way to go. If it's the opposite of this, then use threads.

For performance reasons, along with ease and consistency across systems, we'll use threads. We will describe both Pthreads and C++ 11 threads, in particular.

Overhead of Processes vs Threads. The common wisdom is that processes are expensive, threads are cheap. Let's verify this with a benchmark on a laptop (included in the live-coding directory) which creates and destroys 50,000 threads:

```
jon@riker examples master % time ./create_fork
0.18s user 4.14s system 34% cpu 12.484 total
jon@riker examples master % time ./create_pthread
0.73s user 1.29s system 107% cpu 1.887 total
```

Clearly Pthreads incur much lower overhead than `fork`. Pthreads offer a speedup of 6.5 over processes in terms of startup and teardown costs.

References

[Let88] Gordon Letwin. *Inside OS/2*. Microsoft Press, 1988.

[McD05] Richard McDougall. Extreme software scaling, 2005. Online; accessed 14-November-2015. URL: `http://queue.acm.org/detail.cfm?id=1095419`.

[R+15] Eric Rowell et al. Know thy complexities!, 2015. Online; accessed 14-November-2015. URL: `http://bigocheatsheet.com/`.

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

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[TRG+87] Avadis Tevanian, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach threads and the UNIX kernel: The battle for control. In *Proceedings, Summer 1987 USENIX Conference*, 1987.