Lecture 4 — Concurrency and Parallelism

Jeff Zarnett & Patrick Lam
jzarnett@uwaterloo.ca p.lam@ece.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 28, 2016

Parallelism versus Concurrency

Parallelism

Two or more tasks are parallel
if they are running at the same time.

Main goal: run tasks as fast as possible.

Main concern: dependencies.

Concurrency

Two or more tasks are concurrent
if the ordering of the two tasks is not predetermined.

Main concern: synchronization.

Writing Scalable Code

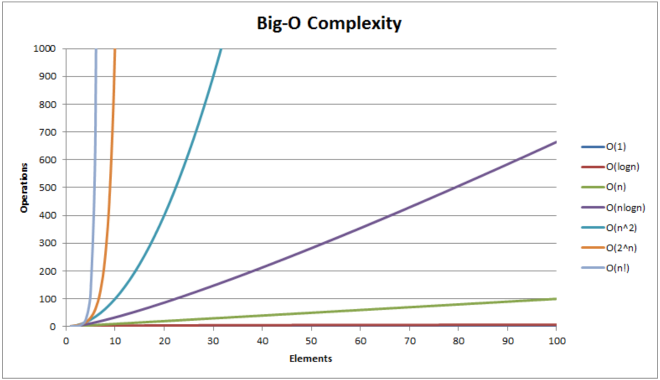
Think about the worst case run-time performance of the algorithm.

An algorithm that's $O(n^3)$ scales so much worse than one that's $O(n)$...

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



Lock It Down

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. The more threads or processes that are looking to acquire that lock, the more “resource contention” we have.

And thus more waiting and coordination are going to be necessary.

Doing Lines

Multiprocessor (multicore) processors have some hardware that tries to keep the data consistent between different pipelines and caches.

More processors, more threads means more work is necessary to keep these things in order.

Everybody Into the Pool

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.

“We have a new enemy...”

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.

There are some techniques for dynamic memory allocation that allow these things to work in parallel.

Processes and Threads

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.

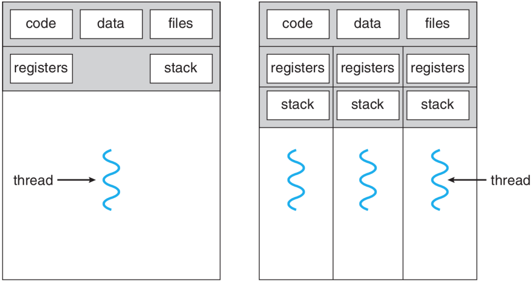
The most important point is that it shares an address space with its parent process, hence variables and code as well as resources.

Threads

Each thread has its own:

1. Thread execution state.
2. Saved thread context when not running.
3. Execution stack.
4. Local variables.
5. Access to the memory and resources of the process.

Threads and Processes



single-threaded process

multithreaded process

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.

My usual example: file transfer program...

Thread Motivation

Why threads instead of a new process?

Primary motivation is: performance.

1. Creation: $10\times$ faster.
2. Terminating and cleaning up a thread is faster.
3. Switch time: 20% of process switch time.
4. Shared memory space (no need for IPC).
5. Lets the UI be responsive.

Common Usage of Threads

1. **Foreground and Background Work**
2. **Asynchronous processing**
3. **Speed of Execution**
4. **Modular Structure**

Thread Drawbacks

There is no protection between threads in the same process.

One thread can easily mess with the memory being used by another.

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, the whole process might be terminated by the operating system.

Software and Hardware Threads

Software Thread:

What you program with (e.g. with `pthread_create()` or `std::thread()`).

Corresponds to a stream of instructions executed by the processor.

On a single-core, single-processor machine, someone has to multiplex the CPU to execute multiple threads concurrently; only one thread runs at a time.

Hardware Thread:

Corresponds to virtual (or real) CPUs in a system. Also known as strands.

Operating system must multiplex software threads onto hardware threads, but can execute more than one software thread at once.

Thread Model—1:1 (Kernel-level Threading)

Simplest possible threading implementation.

The kernel schedules threads on different processors;

- NB: Kernel involvement required to take advantage of a multicore system.

Context switching involves system call overhead.

Used by Win32, POSIX threads for Windows and Linux.

Allows concurrency and parallelism.

Thread Model—N:1 (User-level Threading)

All application threads map to a single kernel thread.

Quick context switches, no need for system call.

Cannot use multiple processors, only for concurrency.

- Why would you use user threads?

Used by GNU Portable Threads.

Thread Model—M:N (Hybrid Threading)

Map *M* application threads to *N* kernel threads.

A compromise between the previous two models.

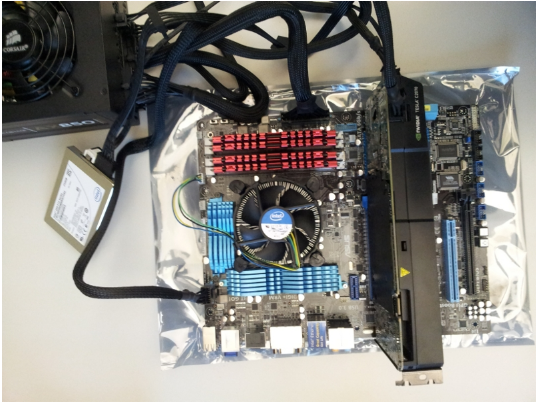
Allows quick context switches and the use of multiple processors.

Requires increased complexity:

- Both library and kernel must schedule.
- Schedulers may not coordinate well together.
- Increases likelihood of priority inversion
  (recall from Operating Systems).

Used by Windows 7 threads.

Example System—Physical View



- Only one physical CPU

Example System—System View

```
jon@ece459-1 ~ % egrep 'processor|model name' /proc/cpuinfo
processor : 0
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 1
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 2
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 3
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 4
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 5
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 6
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 7
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
```

- Many processors

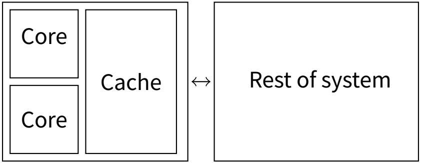
SMP (Symmetric Multiprocessing)

Identical processors or cores, which:

- are interconnected, usually using buses; and
- share main memory.

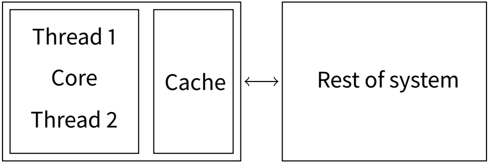
- SMP is most common type of multiprocessing system.

Example of an SMP System



- Each core can execute a different thread
- Shared memory quickly becomes the bottleneck

Executing 2 Threads on a Single Core



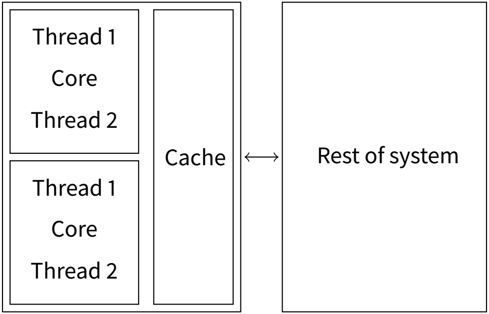
On a single core, must context switch between threads:

- every *N* cycles; or
- wait until cache miss, or another long event

Resources may be unused during execution.

Why not take advantage of this?

Executing M Threads on a N Cores



Here's a Chip Multithreading example.

UltraSPARC T2 has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

SMT (Simultaneous Multithreading)

Use idle CPU resources (may be calculating or waiting for memory) to execute another task.

Cannot improve performance if shared resources are the bottleneck.

Issue instructions for each thread per cycle.

To the OS, it looks a lot like SMP, but gives only up to 30% performance improvement.

Intel implementation: Hyper-Threading.

PlayStation 3 contains a Cell processor:

- PowerPC main core (Power Processing Element, or "PPE")
- 7 Synergistic Processing Elements ("SPE"s): small vector computers.

NUMA (Non-Uniform Memory Access)

In SMP, all CPUs have uniform (the same) access time for resources.

For NUMA, CPUs can access different resources faster (resources: not just memory).

Schedule tasks on CPUs which access resources faster.

Since memory is commonly the bottleneck, each CPU has its own memory bank.

Processor Affinity

Each task (process/thread) can be associated with a set of processors.

Useful to take advantage of existing caches (either from the last time the task ran or task uses the same data).

Hyper-Threading is an example of complete affinity for both threads on the same core.

Often better to use a different processor if current set busy.

Background

Recall the difference between processes and threads:

- 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).

History

`clone` is not POSIX compliant.
Developers mostly used `fork` in the past, which creates a new process.

- Drawbacks?
- Benefits?

Benefit: `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, uses 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.

Drawback of Processes: Threads are Easier and Faster

- Interprocess communication (IPC) is more complicated and slower than interthread communication.
  - Need to use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library (or just memory reads and writes).
- Processes have much higher startup, shutdown and synchronization cost.
- And, Pthreads/C++11 threads fix issues with `clone` and provide a uniform interface for most systems **(Assignment 1)**.

Appropriate Time to Use Processes

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.
- Want to be safer against bugs and security holes.

Then processes are the way to go.

For performance reasons, along with ease and consistency, we'll use Pthreads.

Results: Threads Offer a Speedup of 6.5 over `fork`

Here's a benchmark between `fork` and Pthreads on a laptop, creating and destroying 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`.