Lecture 5 — Concurrency and Parallelism

Jeff Zarnett & Patrick Lam
jzarnett@uwaterloo.ca patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

January 23, 2019

Part I

Limits

Writing Scalable Code

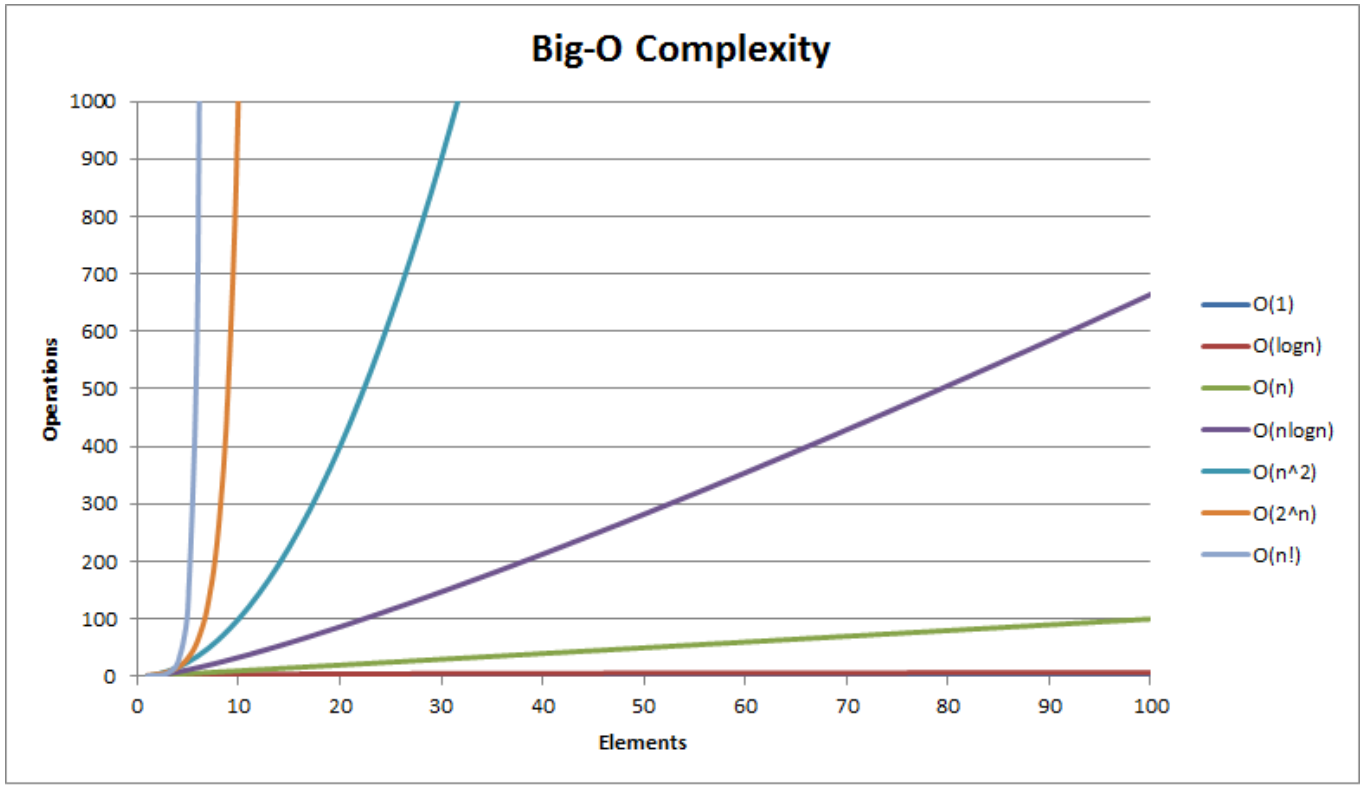
Think about the worst case run-time performance of the algorithm.

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



Parallelism versus Concurrency

Before we talk about parallelism, let's distinguish it from 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.

Limitations of Speedups

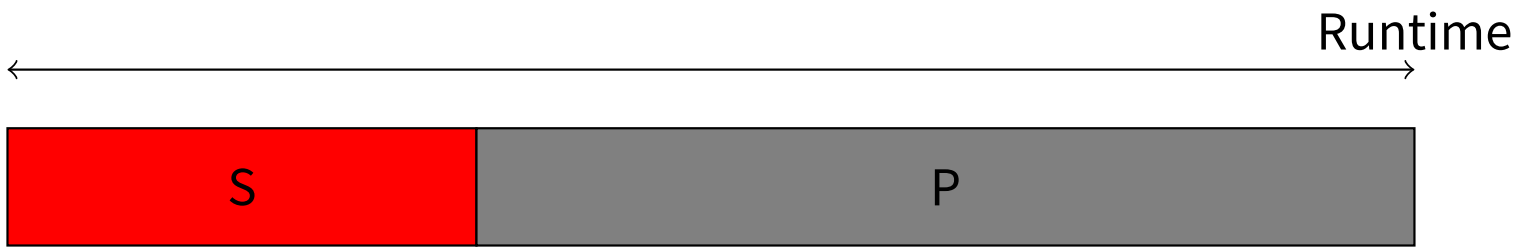
Our main focus is parallelization.

- Most programs have a sequential part and a parallel part; and,
- Amdahl's Law answers, "what are the limits to parallelization?"

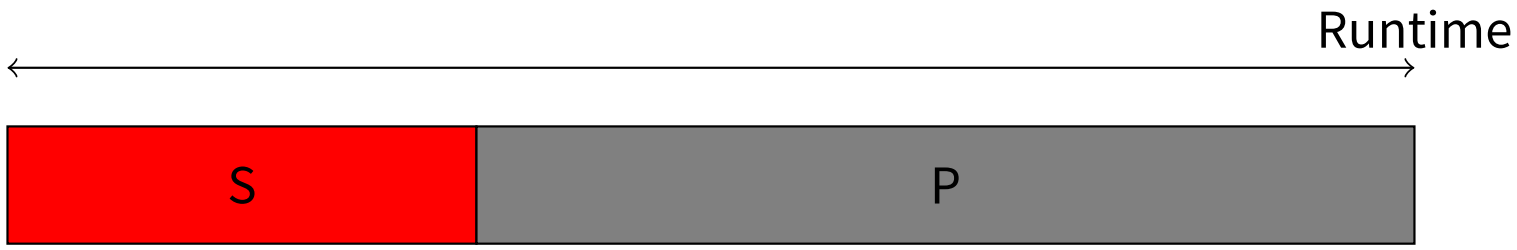
Visualizing Amdahl's Law

S: fraction of serial runtime in a serial execution.
P: fraction of parallel runtime in a serial execution.
Therefore, $S + P = 1$.

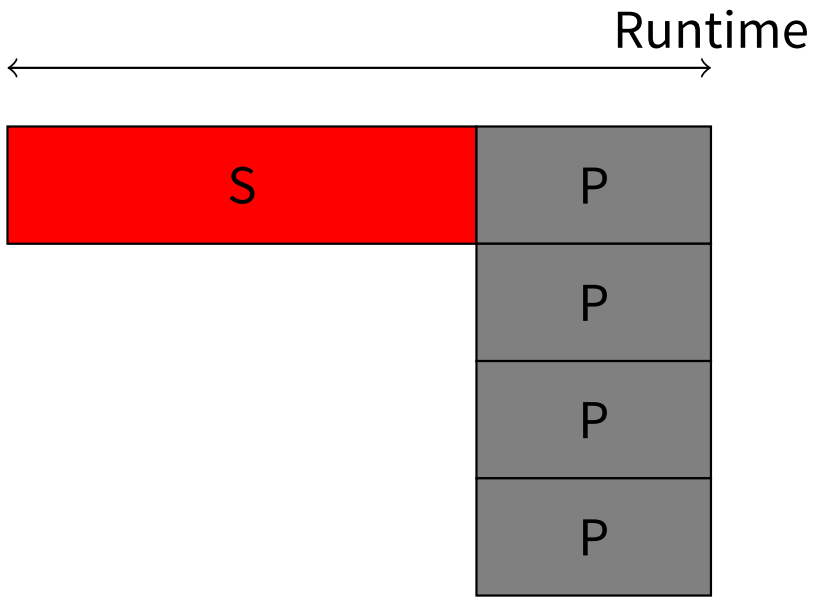
With 4 processors, best case, what can happen to the following runtime?



Visualizing Amdahl's Law



We want to split up the parallel part over 4 processors



Obey the Law

T_s: time for the program to run in serial

N: number of processors/parallel executions

T_p: time for the program to run in parallel

- Under perfect conditions, get N speedup for P

$$T_p = T_s \cdot (S + \frac{P}{N})$$

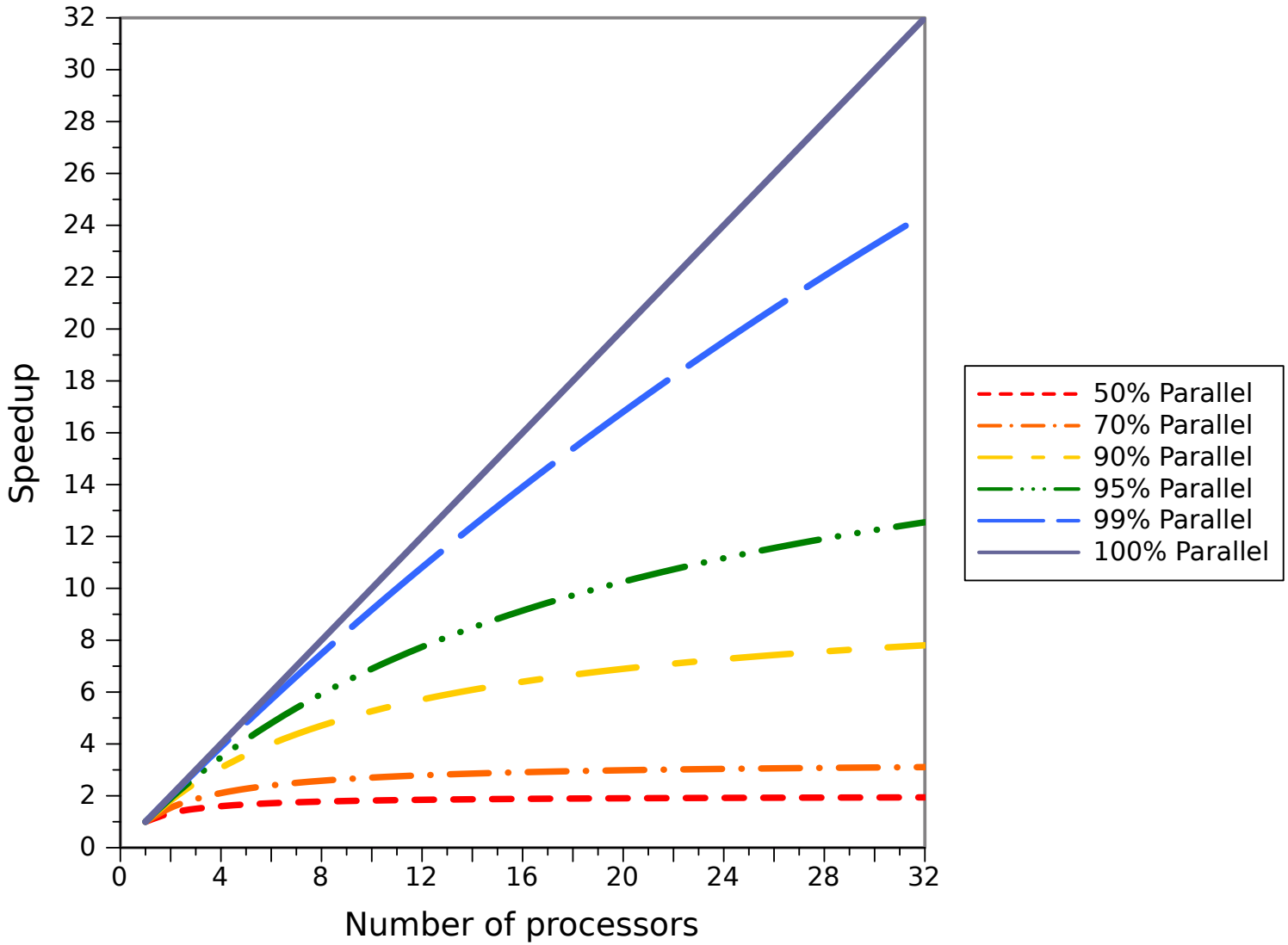
Obey the Law

How much faster can we make the program?

$$
\begin{aligned}
speedup &= \frac{T_s}{T_p} \\
&= \frac{T_s}{T_S \cdot (S + \frac{P}{N})} \\
&= \frac{1}{S + \frac{P}{N}}
\end{aligned}
$$

(assuming no overhead for parallelizing; or costs near zero)

Fixed-Size Problem Scaling, Varying Fraction of Parallel Code



Replace S with $(1 - P)$:

$$speedup = \frac{1}{(1-P)+\frac{P}{N}}$$

$$maximum\ speedup = \frac{1}{(1-P)}, \text{since } \frac{P}{N} \to 0$$

As you might imagine, the asymptotes in the previous graph are bounded by the maximum speedup.

Speedup Example

Suppose: a task that can be executed in 5 s, containing a parallelizable loop.

Initialization and recombination code in this routine requires 400 ms.

So with one processor executing, it would take about 4.6 s to execute the loop.

Split it up and execute on two processors: 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%.

Amdahl's Law on the 5 s Task

Applying this formula to the example:

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

Observations on the 5 s Task

1. Diminishing returns as we add more processors.

2. Converges on 0.4 s.

The most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$.

But that would require infinite processors (and therefore infinite money).

Assumptions behind Amdahl's Law

We assume:

- problem size is fixed (we'll see this soon);
- program/algorithm behaves the same on 1 processor and on N processors;
- that we can accurately measure runtimes—
  i.e. that overheads don't matter.

Amdahl's Law Generalization

The program may have many parts, each of which we can tune to a different degree.

Let's generalize Amdahl's Law.

$f_1, f_2, \ldots, f_n$: fraction of time in part n

$S_{f_1}, S_{f_n}, \ldots, S_{f_n}$: speedup for part n

$$speedup = \frac{1}{\frac{f_1}{S_{f_1}} + \frac{f_2}{S_{f_2}} + \ldots + \frac{f_n}{S_{f_n}}}$$

Application (1)

Consider a program with 4 parts in the following scenario:

		Speedup	
Part	Fraction of Runtime	Option 1	Option 2
1	0.55	1	2
2	0.25	5	1
3	0.15	3	1
4	0.05	10	1

We can implement either Option 1 or Option 2.
Which option is better?

"Plug and chug" the numbers:

Option 1

$$speedup = \frac{1}{0.55 + \frac{0.25}{5} + \frac{0.15}{3} + \frac{0.05}{5}} = 1.53$$

Option 2

$$speedup = \frac{1}{\frac{0.55}{2} + 0.45} = 1.38$$

Empirically estimating parallel speedup P

Useful to know, don't have to commit to memory:

$$P_{\text{estimated}} = \frac{\frac{1}{speedup} - 1}{\frac{1}{N} - 1}$$

- Quick way to guess the fraction of parallel code
- Use $P_{\text{estimated}}$ to predict speedup for a different number of processors

Summary of Amdahl's Law

Important to focus on the part of the program with most impact.

Amdahl's Law:

- estimates perfect performance gains from parallelization (under assumptions); but,
- only applies to solving a fixed problem size in the shortest possible period of time

Gustafson's Law: Formulation

n: problem size
$S(n)$: fraction of serial runtime for a parallel execution
$P(n)$: fraction of parallel runtime for a parallel execution

$$\begin{aligned} T_p &= S(n) + P(n) = 1 \\ T_s &= S(n) + N \cdot P(n) \end{aligned}$$

$$speedup = \frac{T_s}{T_p}$$

Gustafson's Law

$$speedup = S(n) + N \cdot P(n)$$

Assuming the fraction of runtime in serial part decreases as n increases, the speedup approaches N.

Yes! Large problems can be efficiently parallelized. (Ask Google.)

Amdahl's Law

Suppose you're travelling between 2 cities 90 km apart. If you travel for an hour at a constant speed less than 90 km/h, your average will never equal 90 km/h, even if you energize after that hour.

Gustafson's Law

Suppose you've been travelling at a constant speed less than 90 km/h. Given enough distance, you can bring your average up to 90 km/h.

Part II

Parallelization Techniques

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.

“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 (using locks to ensure this).

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.

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.

Part III

Parallelization Technique: Threads vs Processes

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

Developers mostly used `fork` before there was a standardized way to create threads (`clone` was non-standards-complient).

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

`fork` 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.

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.

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`.

More Detail: 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.

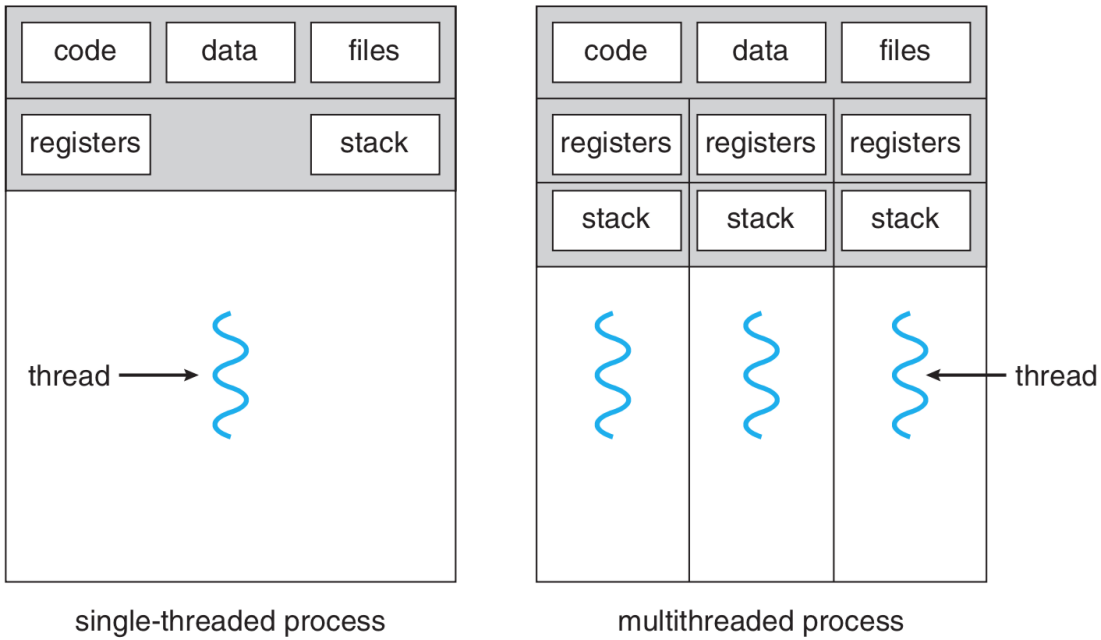
The most important point is that it shares an address space with its parent process, hence variables and code as well as resources.

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



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 implementations: 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.

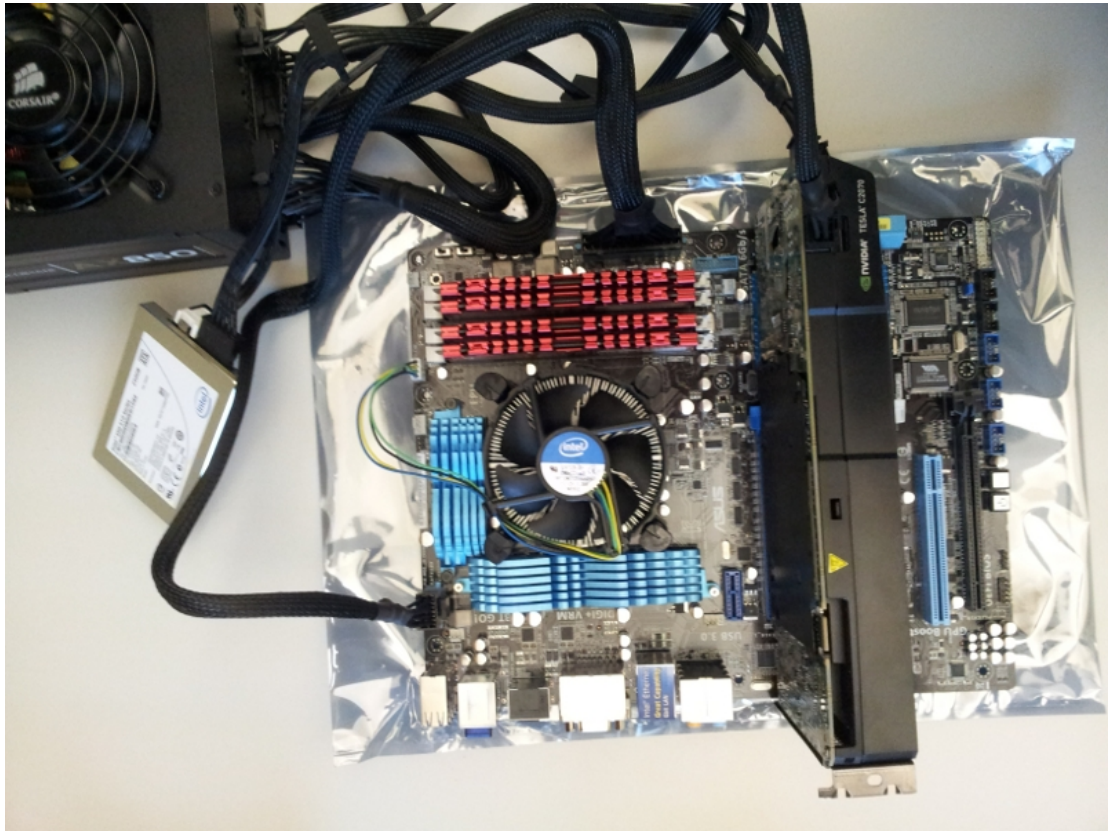
Allows quick context switches and the use of multiple processors.

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