Parallel processing

YO DAWG, I HEARD YOU LIKED PROCESSORS...

...SO WE PUT PROCESSORS IN YOUR PROCESSOR SO YOU CAN PROCESS WHILE YOU PROCESS!
- Making threads
  thread another = thread(foo);
  // foo() is a function!

- Waiting for threads
  another.join()
**Terminology**

**CPU** = area of computer that does thinking

**Core** = processor = a thinking unit

**Program** = code = instructions on what to do

**Thread** = parallel process = an independent part of the program/code

**Program** = string,
**thread** = 1 part of that
Review: CPUs

μProcessor Clock Speed Trends

Year of Introduction

Maximum Clock Speed (MHz)

- 8086
- 8085
- 8080
- 8086
- 80286
- 80386
- 80486
- 80486DX
- 80486DX2
- P5 Pentium
- Xeon
- 2.5 GHz
- 3.06 GHz
- After 2003

S.A. Wilkens May 21, 2014
In the 2000s, computing took a major turn: multi-core processors (CPUs)
Review: CPUs

40 Years of Microprocessor Trend Data

- Transistors (thousands)
- Single-Thread Performance (SpecINT x 10^3)
- Frequency (MHz)
- Typical Power (Watts)
- Number of Logical Cores

Year


Review: CPUs

The major reason is due to heat/energy density
Review: CPUs

[Diagram showing the relationship between minimum IC feature size in microns and watts/cm², with points labeled for CPUs from 1993 to 2000.]
Review: CPUs

This trend will almost surely not reverse

There will be new major advancements in computing eventually (quantum computing?)

But “cloud computing”, which has programs that “run” across multiple computers are going nowhere anytime soon
Parallel: how

So far our computer programs have run through code one line at a time. To get multiple parts running at the same time, you must create a new thread and give it a function to start running:

```cpp
#include <thread>

int main()
{
    thread another = thread(foo);
}

void foo()
{
    // some function...
}
```

Need: #include `<thread>`
If the function wants arguments, just add them after the function in the thread constructor:

```cpp
int main()
{
    thread another = thread(say, "hello");
}
```

This will start function “say” with first input as “hello” (see: createThreads.cpp)
Parallel: basics

The major drawback of distributed computing (within a single computer or between) is resource synchronization (i.e. sharing info)

This causes two types of large problems:
1. Conflicts when multiple threads want to use the same resource

2. Logic errors due to parts of the program having different information
1. Resource conflict

Siblings anyone?

EVEry SHower stall in the BATHroom OCCUPIED?

BACK to bed it is
1. Resource conflict

Public bathroom?

All your programs so far have had 1 restroom, but some parts of your program could be sped up by making 2 lines (as long as no issues).
1. Resource conflict

We will actually learn how to resolve minor resource conflicts to ensure no logic errors.

This is similar to a cost of calling your forgetful relative to remind them of something.

This only needs to be done for the important matters that involve both of you (e.g. when the family get-together is happening).
2. Different information

If you and another person try to do something together, but not coordinated... disaster
2. Different information

Each part of the computer has its own local set of information, much like separate people

Suppose we handed out tally counters and told two people to count the amount of people
2. Different information

However, two people could easily tally the number entering this room...

Simply stand one by each door and add them.

Our goal is to design programs that have these two separate parts that can be done simultaneously (which tries to avoid sharing parts).
Parallel: how

However, main() will keep moving on without any regard to what these threads are doing.

If you want to synchronize them at some later point, you can run the join() function.

This tells the code to wait here until the thread is done (i.e. returns from the function).
Consider this:

```cpp
void peek()
{
    cout << "peek-a-";
}
```

The `start.join()` stops `main` until the `peek()` function returns.

```cpp
int main()
{
    thread start = thread(peek);
    start.join(); // YOU MAY NOT PASS
    cout << "boo!\n";
}
```

(see: `waitForThreads.cpp`
Parallel: advanced

None of these fix our counting issue (this is, in fact, not something we want to parallelize)

I only have 4 cores in my computer, so if I have more than 3 extra threads (my normal program is one) they fight over thinking time

Each thread speeds along, and my operating system decides which thread is going to get a turn and when (semi-random)
Parallel: advanced

We can force threads to not fall all over themselves by using a mutex (stands for “mutual exclusion”)

Mutexes have two functions:
1. lock
2. unlock

After one thread “locks” this mutex, no others can pass their “locks” until it is “unlocked”
You can think about a “muxtex” like a porta-potty or airplane lavatory indicator:

It is a variable (information) that lets you know if you can proceed or have to wait (when it is your turn, you indicate that this mutex is “occupied” by you now via “lock()”)
Parallel: advanced

Lock

Unlock
These mutex locks are needed if we are trying to share memory between threads

Without this, there can be miscommunications about the values of the data if one thread is trying to change while another is reading

A very simple example of this is having multiple threads go: x++
(see: sharingBetweenThreads.cpp)
You have to be careful when locking a mutex, as if that thread crashes or you forget to unlock... then your program is in an infinite loop.

There are ways around this:
- Timed locks
- Atomic operations instead of mutex

The important part is deciding what parts can be parallelized and writing code to achieve this.