Parallel processing

YO DAWG, I HEARD YOU LIKED PROCESSORS...
...SO WE PUT PROCESSORS IN YOUR PROCESSOR SO YOU CAN PROCESS WHILE YOU PROCESS!

Highlights

- Making threads

```
thread another = thread(foo);
// foo() is a function!
```

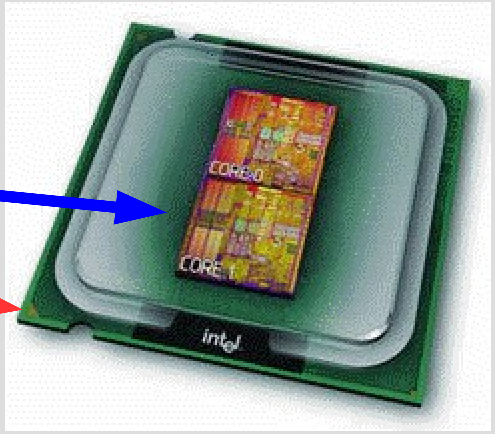
- Waiting for threads

```
another.join()
```

Terminology

CPU = area of computer that does thinking

Core = processor = a thinking unit



Cores

CPU

front/back

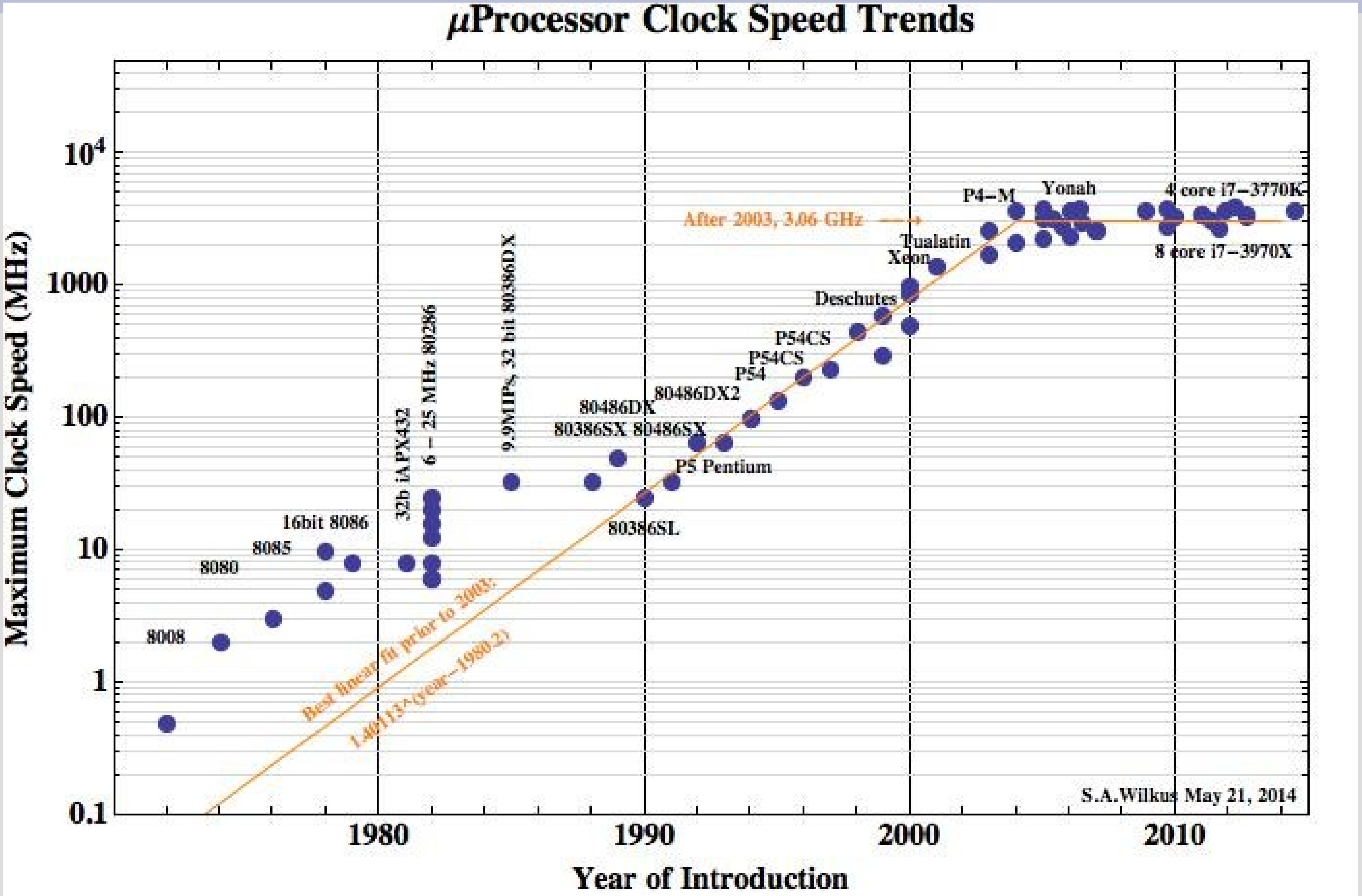
Program = code = instructions on what to do

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

Program = string,

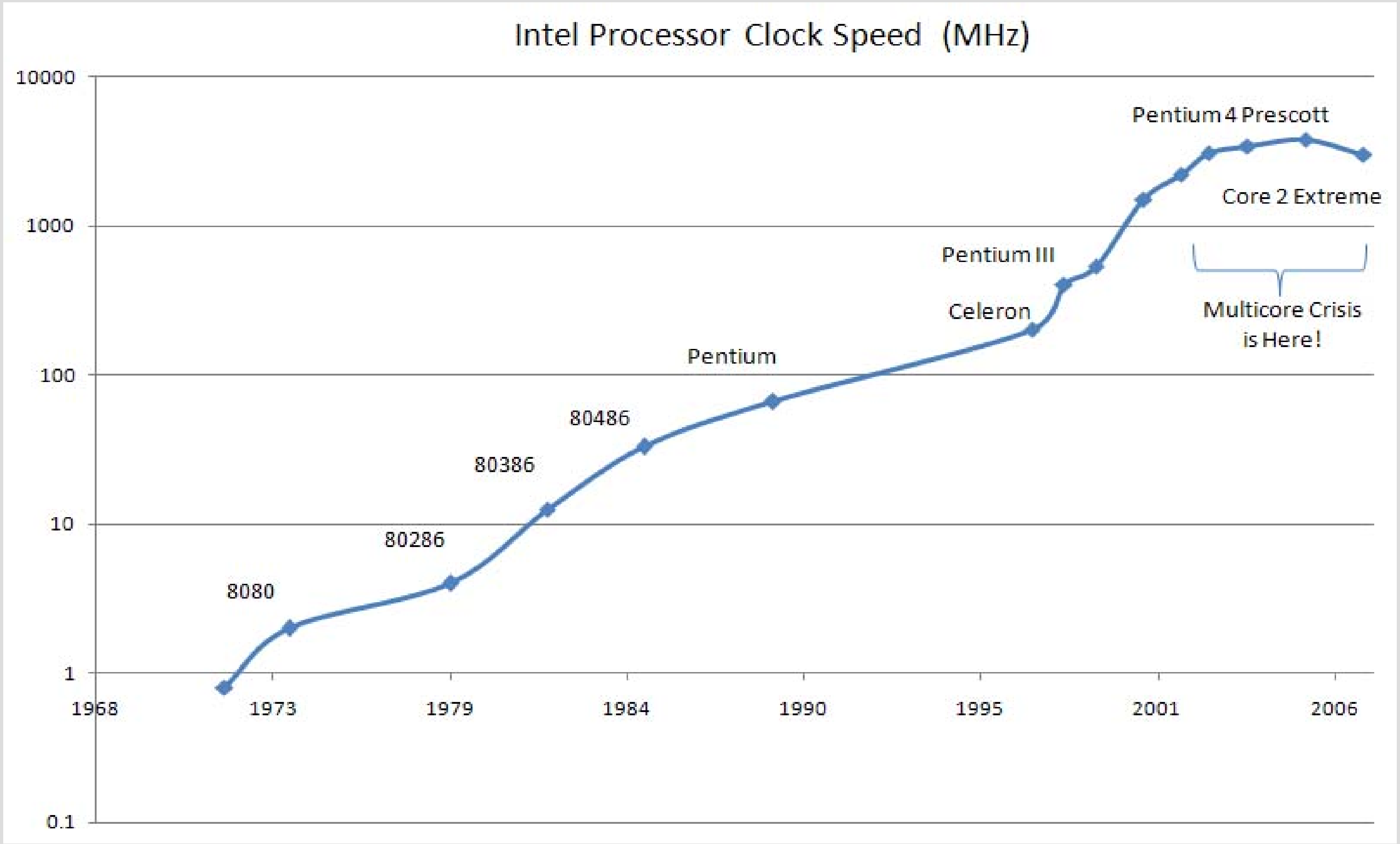
thread = 1 part of that

Review: CPUs

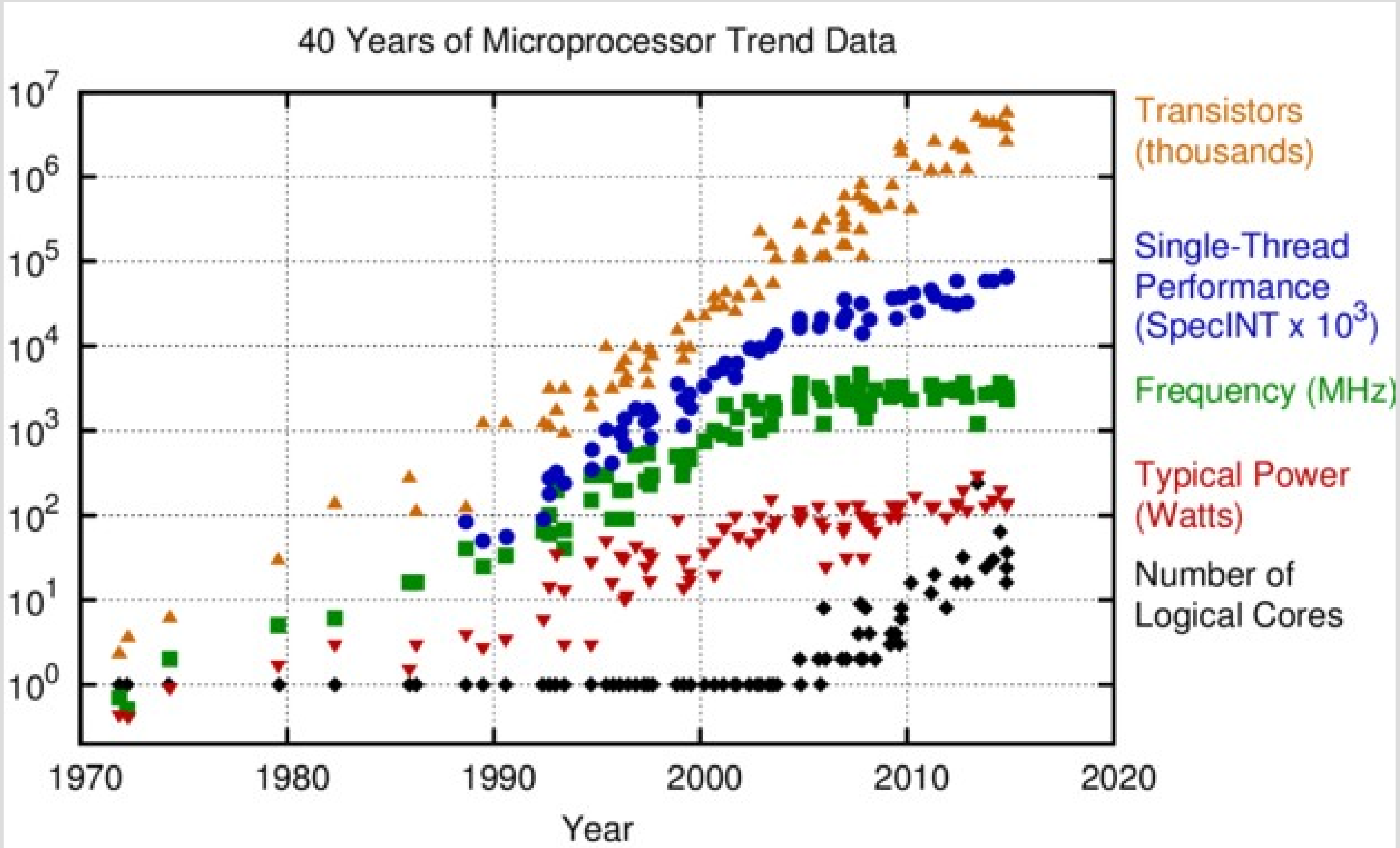

μProcessor Clock Speed Trends
Maximum Clock Speed (MHz)
Year of Introduction
0.1
1
10
100
1000
10^4
1980
1990
2000
2010
8008
8080
8085
16bit 8086
32b iAPX432
6 – 25 MHz 80286
9.9MIPs 32 bit 80386DX
80386SX 80486SX
80486DX
80486DX2
80386SL
P5 Pentium
P54
P54CS
P54CS
Deschutes
Xeon
Tualatin
P4–M
Yonah
4 core i7–3770K
8 core i7–3970X
After 2003, 3.06 GHz
Best linear fit prior to 2003:
1.40113^(year–1980.2)
S.A.Wilkus May 21, 2014

Review: CPUs

In the 2000s, computing too a major turn: multi-core processors (CPUs)



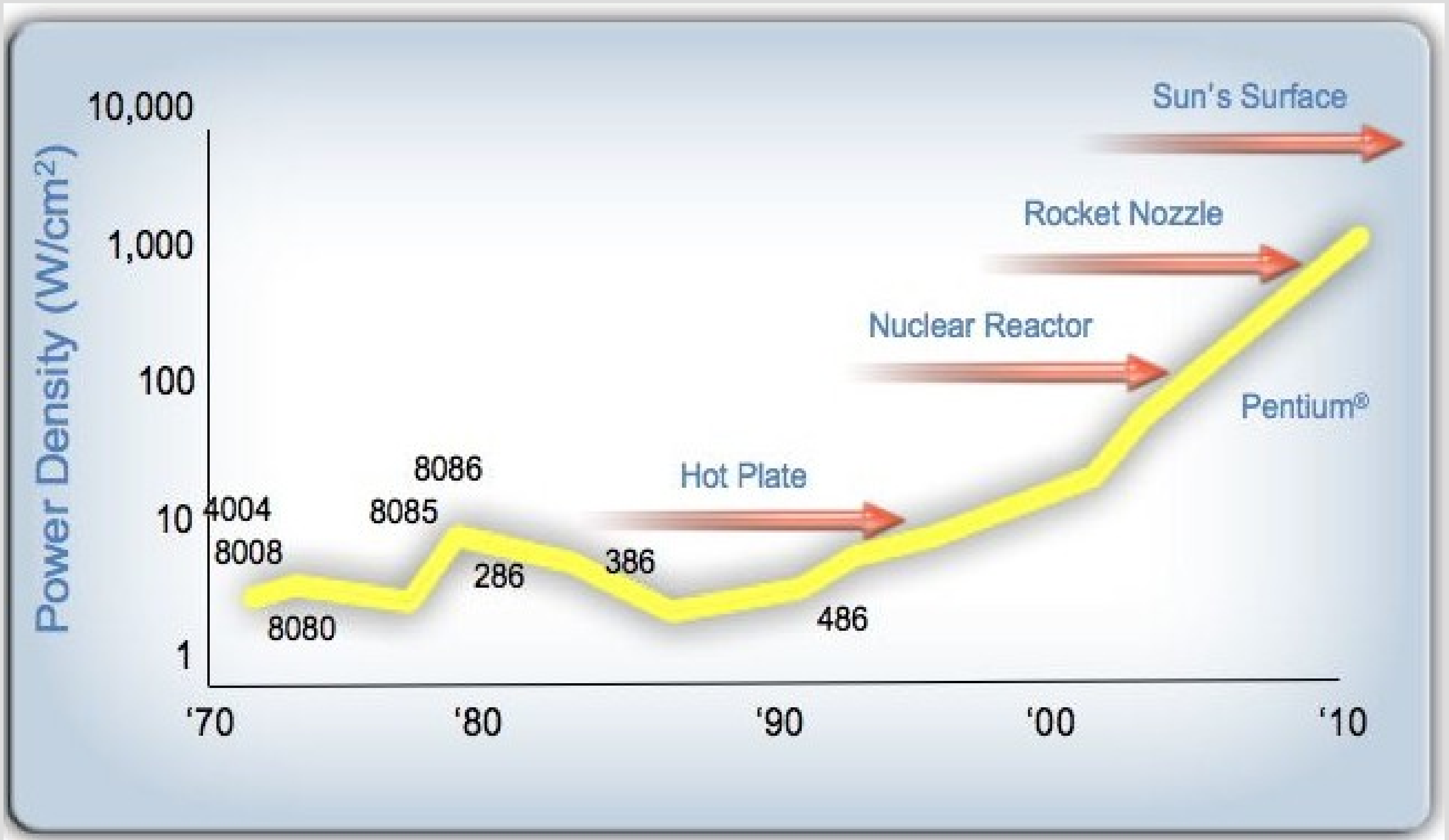
Review: CPUs



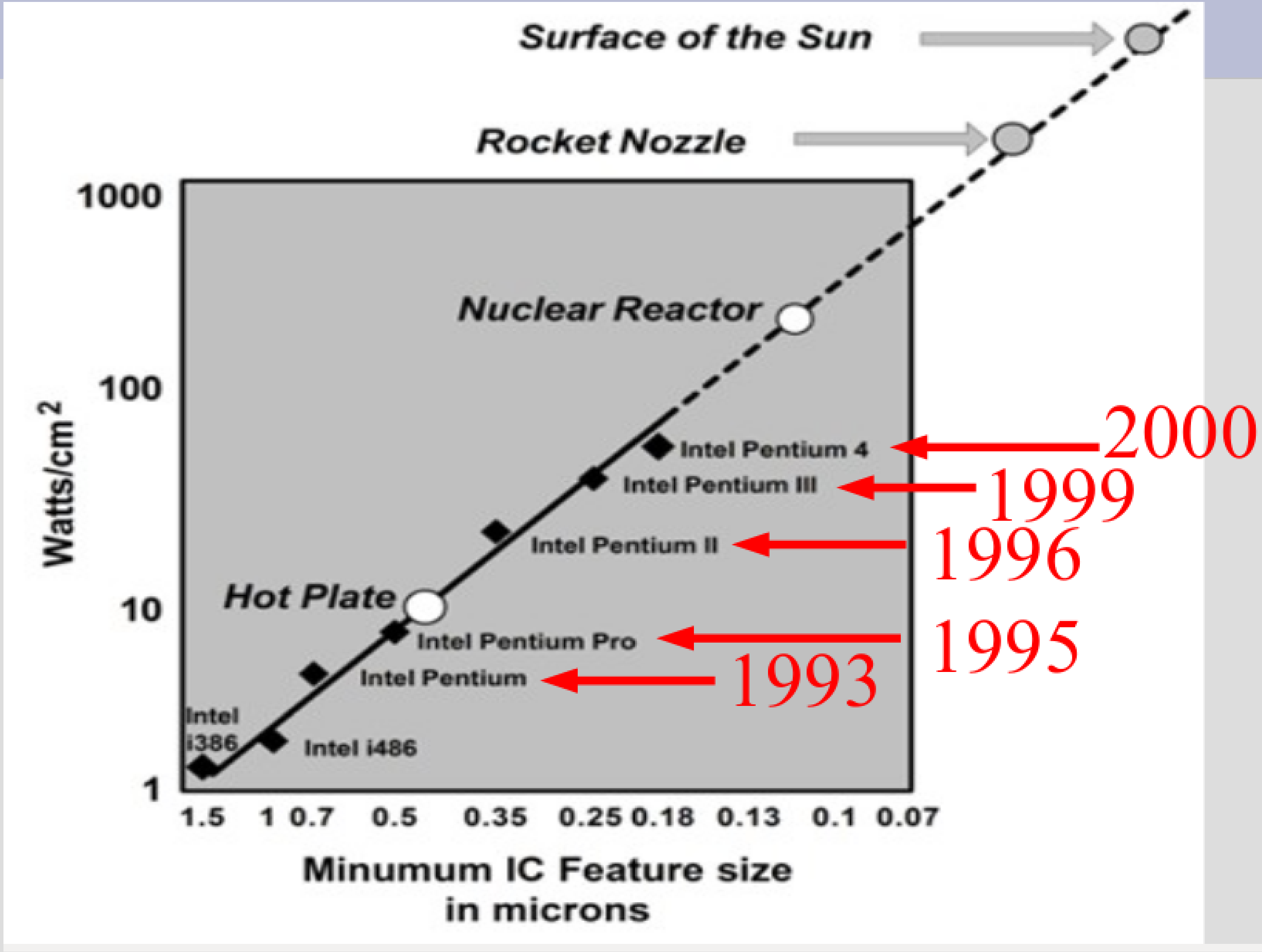
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Review: CPUs

The major reason is due to heat/energy density



Review: CPUs


Surface of the Sun
Rocket Nozzle
Nuclear Reactor
Hot Plate
Intel Pentium 4
Intel Pentium III
Intel Pentium II
Intel Pentium Pro
Intel Pentium
Intel i386
Intel i486
2000
1999
1996
1995
1993
1000
100
10
1
Watts/cm²
1.5
1
0.7
0.5
0.35
0.25
0.18
0.13
0.1
0.07
Minumum IC Feature size in microns

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:

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

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

starts another thread at foo

Need: #include <thread>

Parallel: how

If the function wants arguments, just add them after the function in the thread constructor:

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

This will start function "say" with first input as "hello"

```
void say(string s)
{
    cout << s << endl;
}
```

(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?

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

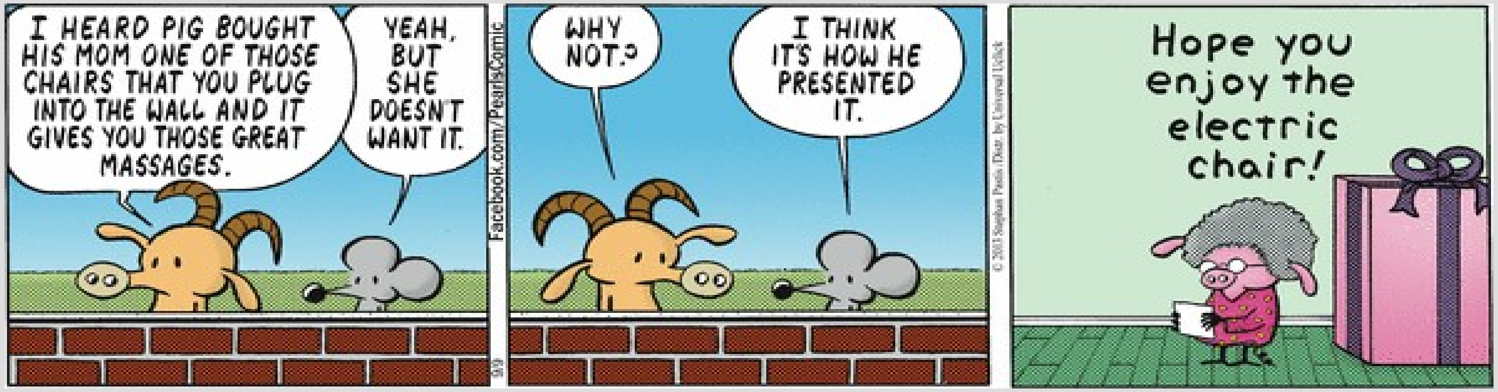
We will actually learn how to resolve minor resource conflicts to ensure no logic errors

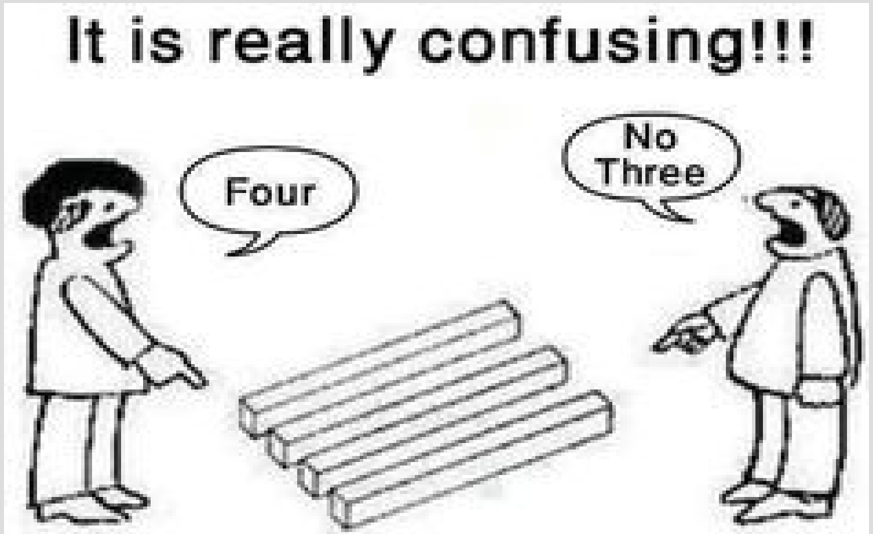
This is similar to a cost of calling your forgetful relative to remind them of something

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


I HEARD PIG BOUGHT HIS MOM ONE OF THOSE CHAIRS THAT YOU PLUG INTO THE WALL AND IT GIVES YOU THOSE GREAT MASSAGES.
YEAH, BUT SHE DOESN'T WANT IT.
WHY NOT?
I THINK IT'S HOW HE PRESENTED IT.
Hope you enjoy the electric chair!
Facebook.com/PearlsComic



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)

Parallel: how

Consider this:

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

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

```
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 <u>mutex</u> (stands for "mutual exclusion")

Mutexes have two functions:

1. lock
2. unlock

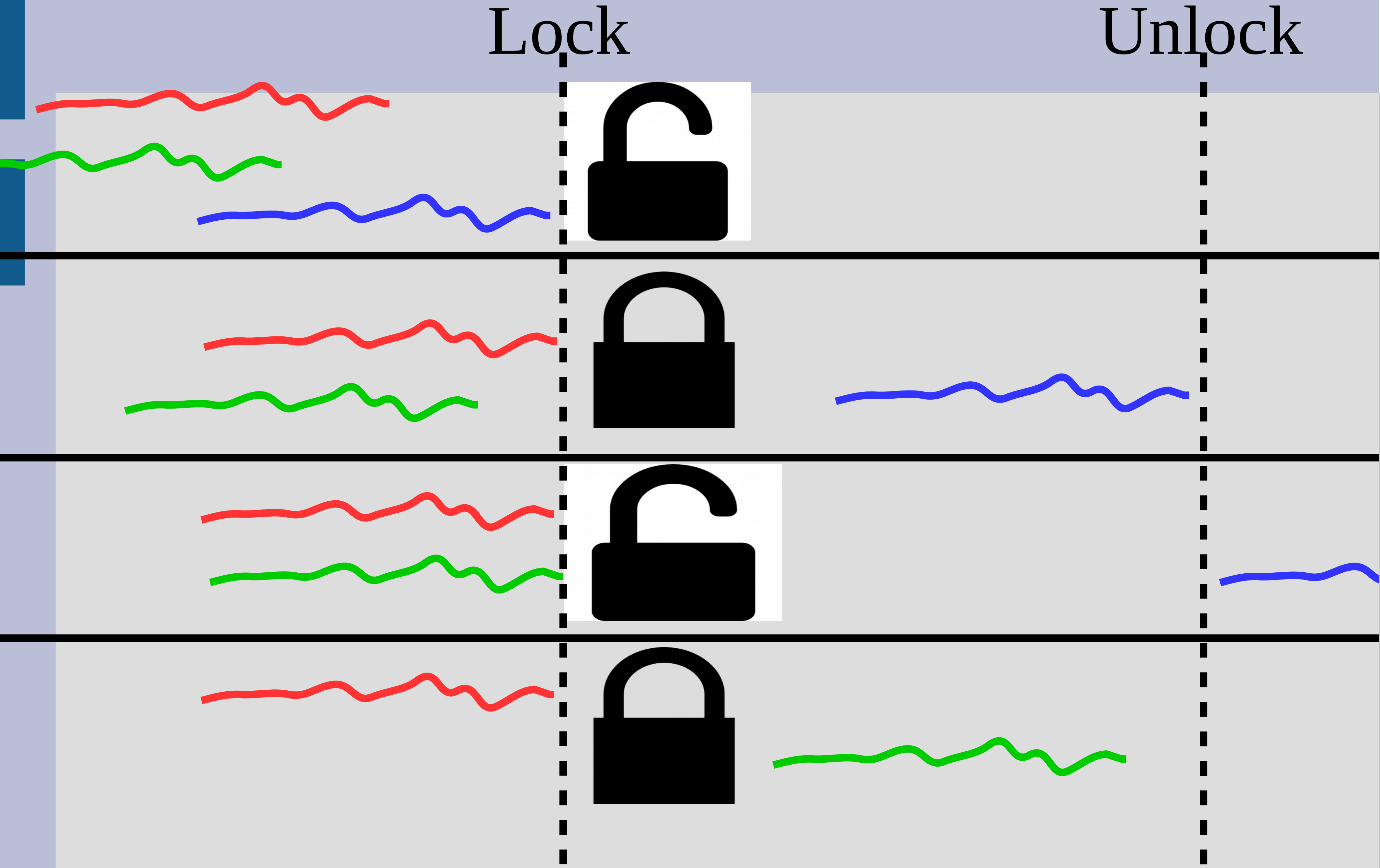
After one thread "locks" this mutex, no others can pass their "locks" until it is "unlocked"

Parallel: advanced

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



Parallel: advanced

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)

Parallel: advanced

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 way 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