What is Concurrency?

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Disclaimer

- There is content in the two sets of lecture notes in this module that overlaps with ICS332
- If you took ICS332 last semester, bear with us (or zone out)
- But if you took it a long(er) time ago, experience shows this is useful!

Concurrency

- **Definition**: Execution of multiple “tasks” at the “same” time
- You have mostly written non-concurrent, or sequential, programs
  - At any point, you could stop the program and say exactly which execution is being executed, what the calling sequence is, what the runtime stack looks like, etc.
  - And there is a single answer to all the above for any execution of your program
- In a concurrent program, you design the program in terms of tasks, where each task as a “life of its own”
  - Each task has a specific job to do
  - Tasks may need to “talk” to each other
  - Tasks can be in different regions of the code or in the same region of the code a the same time
  - Tasks can be short-lived or last the whole program’s execution
- A different way of thinking/programming

A brief history of concurrency (1)

- First machines were used in “single-user mode”
  - I declare: “I am going to use the machine for 2PM till 4PM”
  - I go in the special machine room and sit there for 2 hours
    - I try the punch cards that I have prepared in advance
    - I find bugs
    - I debug
    - etc.
- Extreme lack of productivity
  - During my “thinking time”, this multi-million $ machine does nothing

A brief history of concurrency (2)

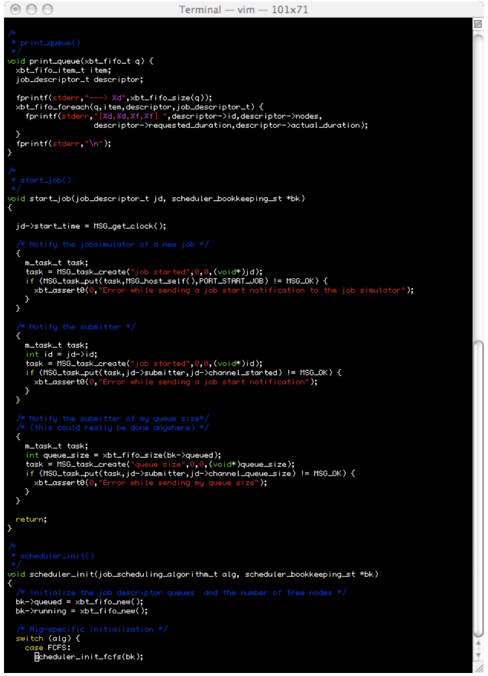
- Batch Processing!
  - Instead of reserving the machine for a lapse of time to do all my activities (including debugging), I “submit” requests to a “queue”
    - The queue serves requests in order (possibly with priorities)
    - When my program fails and stops, somebody else gets the machine immediately
  - Great but: CPU idle during I/O!
    - And I/O takes foreeeeeever

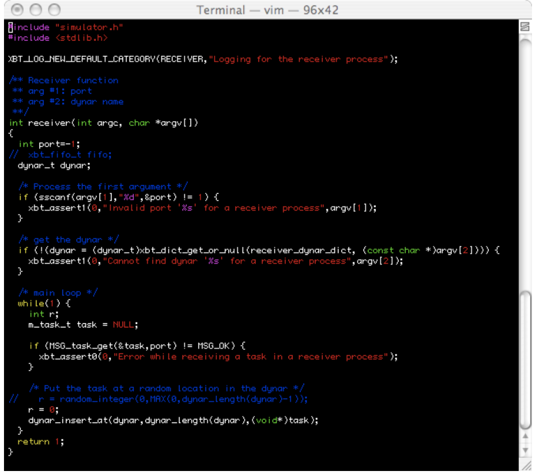
A brief history of concurrency (3)

- Multi-programming (the 60's)
  - Multiple programs reside in memory at once
  - Made possible due to increased memory size
  - Requires interrupts and memory protection
- Time-sharing (the 70's)
  - Multi-programming but rapid alternation between programs
  - Provides the illusion of programs all running simultaneously on the machine
- This is all in ICS332, and is what we have today
  - Virtual memory, fast context switching, etc.
- Eventually this has led to concurrency in user applications!
  - My application is "logically" multiple concurrent tasks
  - I can now implement it as concurrent tasks and the OS will run them simultaneously!
  - This is a main topic of this course

Concurrent Programs

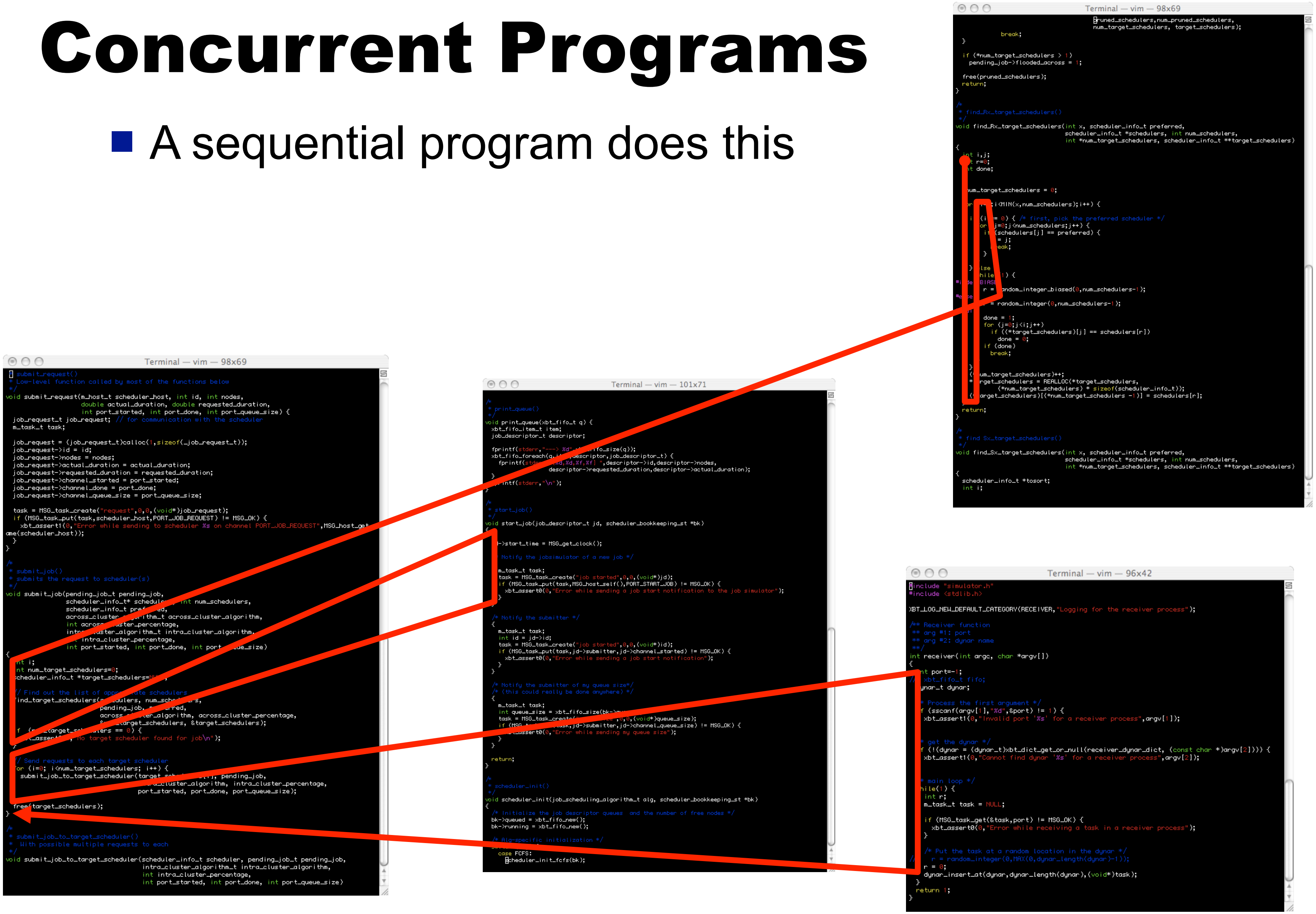
- A program consists of multiple files/modules/classes/functions





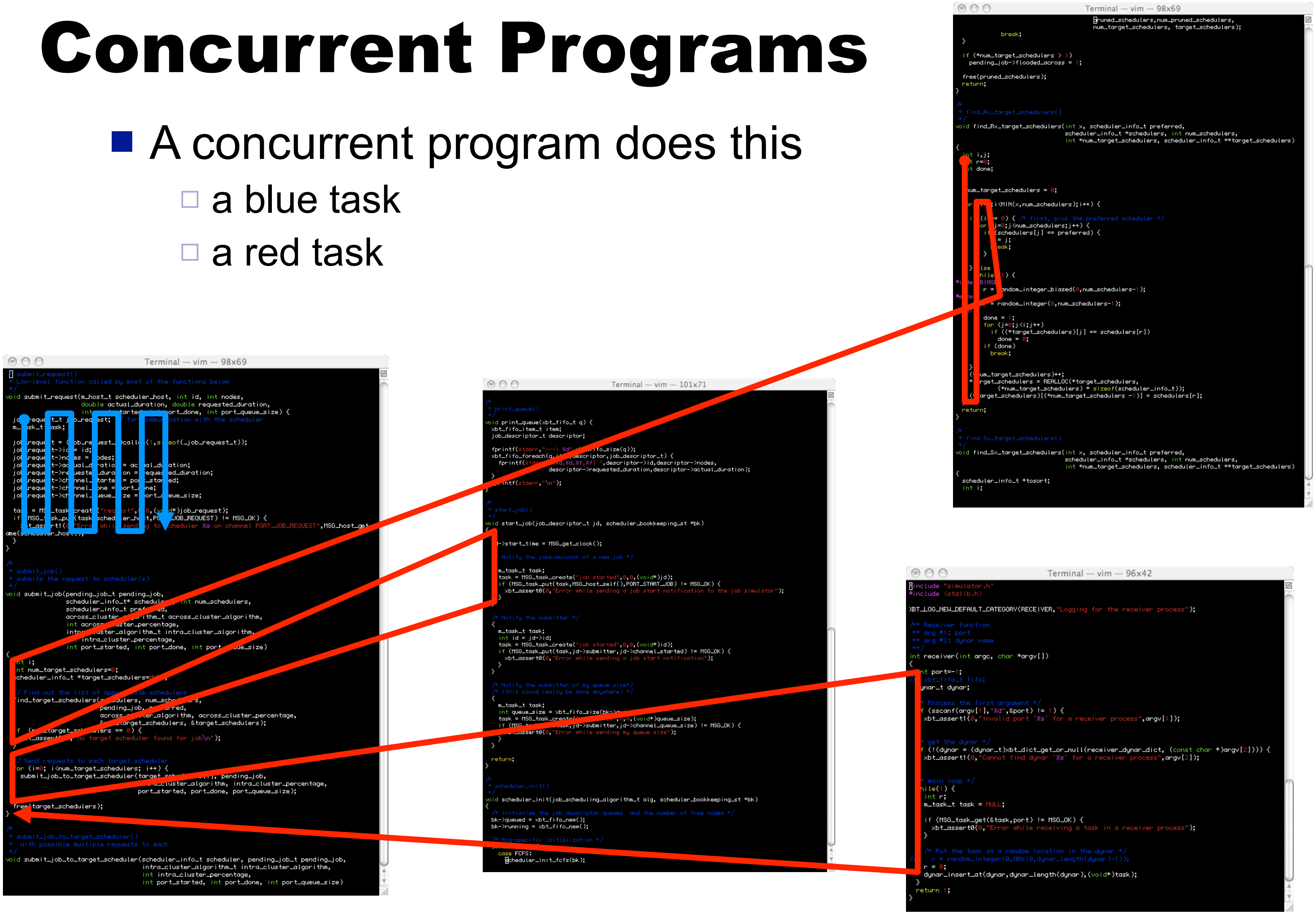
Concurrent Programs

- A sequential program does this



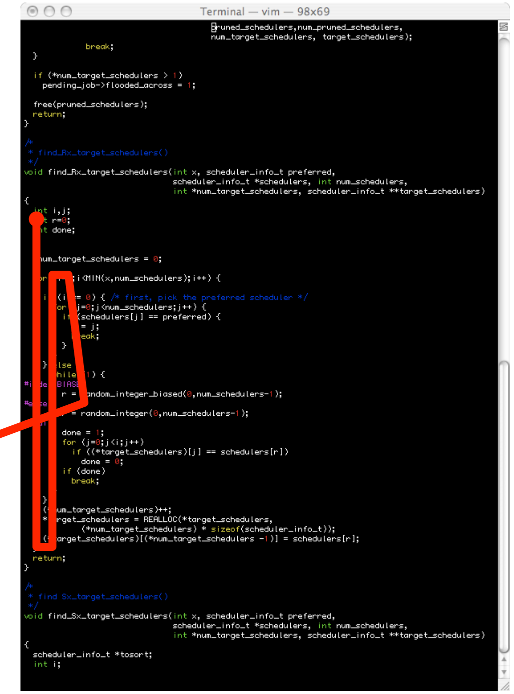
Concurrent Programs

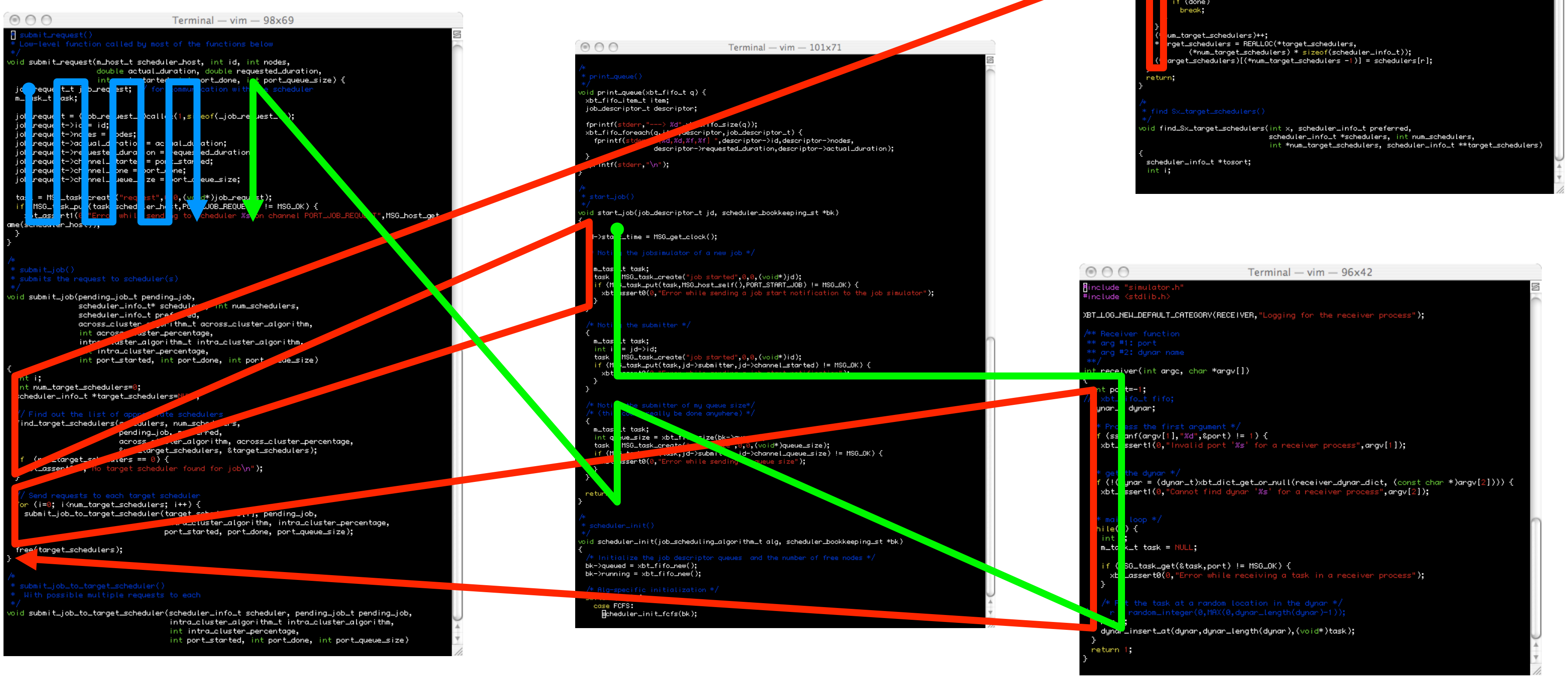
- A concurrent program does this
  - a blue task
  - a red task



Concurrent Programs

- or this
  - a blue task
  - a red task
  - a green task





Concurrent Programs

- Thinking about what a concurrent program does is more difficult than for a sequential program
  - One may have to keep a mental picture of what each task is doing at all time (we try not to)
  - Questions like “While task #1 is in function f where is task #2?” are often difficult (and we try not to have their answers matter so that we don’t have to ask them)
  - Two executions of the same program may not be identical
    - We’ll explain this in more details
- As a result, concurrent programs are
  - Almost always more difficult to design for correctness
  - Almost always more difficult to read
  - Always more difficult to debug
- So, why do we bother at all?

Concurrency for Interactivity

- One of the oldest uses for concurrency is to make programs more interactive
- While a program is running and doing stuff, the user should still be able to interact with it
- Example:
  - What if in your Web browser you couldn't click "back" before the browser has finished loading the page you immediately realized is the wrong one?
  - What if in iTunes you couldn't look at your play list while you're playing a song because the program is busy playing the song?
- One wants to avoid the "frozen because I am working" problem as much as possible
- Let's look at a made-up example…

Designing a Concurrent GUI

- A common application of concurrent programming is for designing Graphical User Interfaces (GUIs)
- Example application
  - Say you want to write a program that renders 3-D objects on the screen
  - You have a clickable button to launch the rendering
  - But rendering takes a long time
  - You don’t want the GUI to appear “frozen” while rendering the objects
  - For instance, you want the “Quit” and “Cancel” buttons to still work

Concurrent GUI?

- One way to avoid the "frozen" problem without using concurrency is to write your code with breaking down a task into sub-tasks
  - Typically, I'll write code fragments in C/C++-like pseudo-code or in Java, without declarations, etc.

```
void render(...) {
    for (step=0; step<100; step++) {
      this.doSomeRendering(...);
      if (gui.cancelButton.clicked())
        break;
    }
}
```

That was (often) a bad idea

- It’s cumbersome:
  - What if you want to do 7 tasks?
  - Sprinkle “interaction checks” throughout your code will make is unreadable and annoying to maintain
- It’s no always doable:
  - What if rendering is not breakable into multiple calls??
    - Perhaps you call some library that you didn’t write and cannot modify
- What if some tasks have some real-time requirements?
  - e.g., you want to have an animated symbol that changes every t milliseconds but a call to doSimpleRendering() takes longer than that?

Task-based design

- Instead of thinking of your application as one task that has to juggle many things at once, you think of your application as a bunch of tasks that run concurrently
- Each task does one thing and perhaps doesn't even know that there are other tasks
- Assuming that we have a programming language that allows us to define tasks we can rewrite our application…

Concurrent GUI

- In *horrible* pseudo-code:

```
renderer = new Task(render)
mousewatcher = new Task(watchmouse)

renderer.start();
mousewatcher.start();
renderer.wait_until_finished()

Renderer::render() {
  // do rendering on screen
}

Mousewatcher::watchmouse() {
  // whenever mouse clicked, kill renderer
}
```

Concurrent Tasks Abstraction

- Fortunately, our OSs support the concurrent tasks abstraction
  - After all, on our machines many programs can run simultaneously
  - So why not tasks within our programs?
- This can be done by
  - A special library
  - A virtual machine like the JVM
  - The Operating System
  - A combination of the above
- Almost all modern programming languages allow you to create “tasks” in your programs

“Simultaneous” tasks?

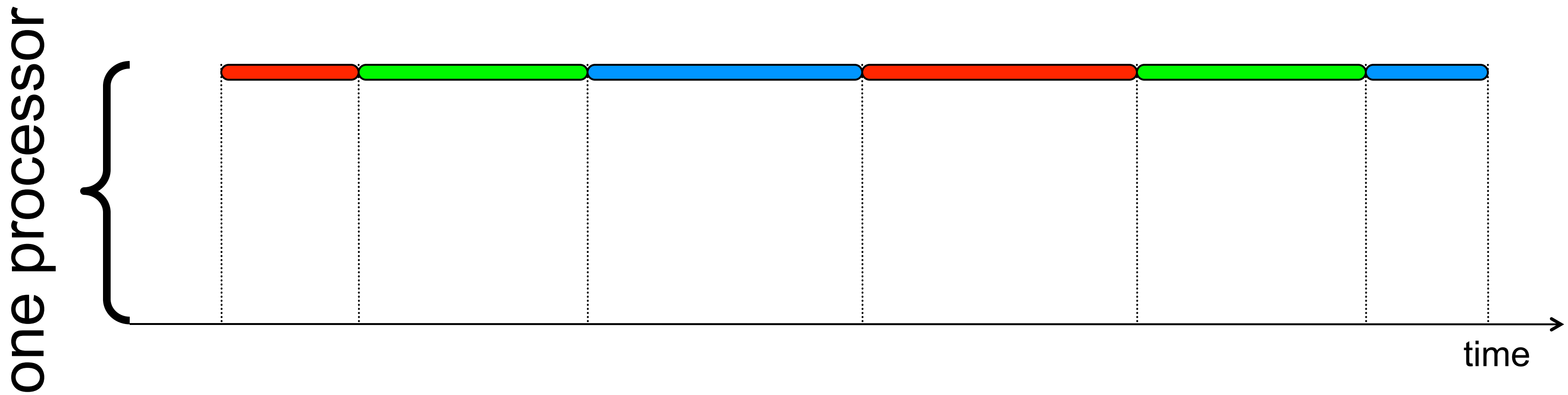
- Can we really have simultaneous concurrent tasks?

- There are two kinds of concurrency:
  - True concurrency: two or more “things” happen at the same instant in time
  - False concurrency: only one thing happens at a time, but the illusion of concurrency is achieved because the OS performs rapid context switching

True/False Concurrency

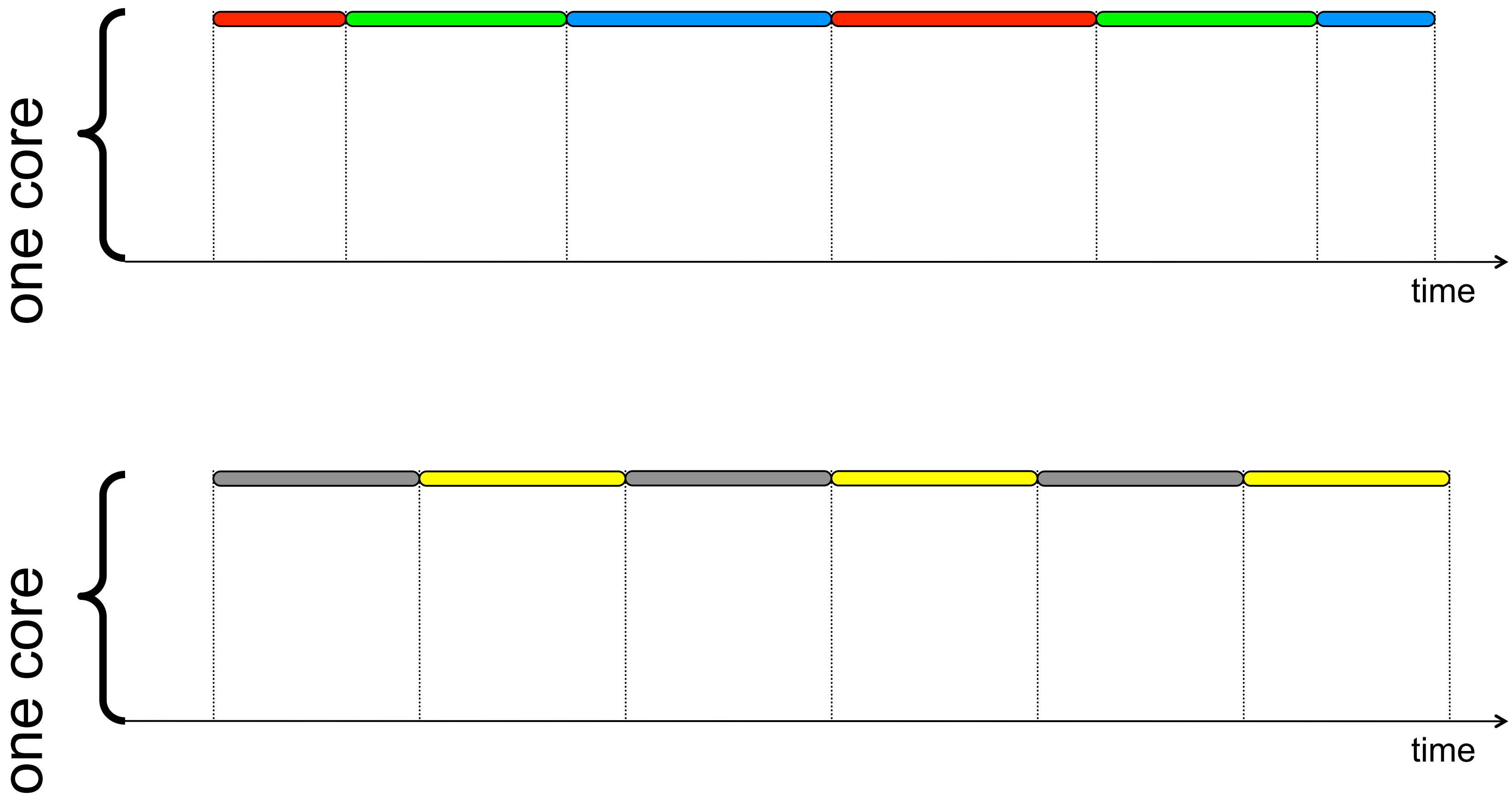
- Consider a program that defines two concurrent tasks, T1 and T2
- On a single core, only one task can use the CPU
  - The concurrent tasks use false concurrency
- On a multi-core system, each task can be on a different core
  - The concurrent tasks use true concurrency

False Concurrency on One Core



- False concurrency between the red task, the green task, and the blue task
- The OS context-switches back and forth between the three tasks
- Because this is very fast, we have the "illusion" of simultaneous execution

On Two Cores



- True concurrency between the yellow task and the green task, the grey and the blue, etc.

True/False Concurrency

- The programmer shouldn't have to care/know whether concurrency will be true or false
  - Besides the fact that true concurrency offers better performance than false concurrency
- Typically, the programmer doesn't know on which computer the program will run!
  - You have no idea how many cores your "customer" will have on their machines
- A concurrent program with 10 tasks will work on a single-core processor, a quad-core processor, a 32-core processor, etc
  - Your job as a developer is to create tasks
    - e.g., the program could easily discover that the machine it's running on has 8 cores, and thus decides to create 8 tasks
  - The job of the OS is to dispatch these tasks to the cores
    - e.g., the OS is smart enough to put each of the 8 tasks on its own core without you having to make those decisions

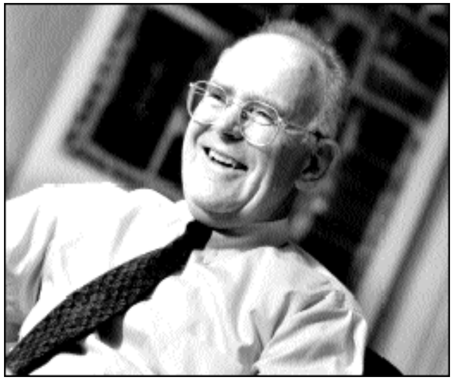
Performance!!!

- But wait, with true concurrency we can also go faster!!!
  - If you have to bake 2 cakes and you have 1 oven it will take you 2 hours
  - But if you have 2 ovens that can be on at the same time, it will take you only 1 hour
- This brings us to the second major reason why people want to use concurrency: compute stuff faster
- To summarize we have two motivations:
  - concurrency for interactivity
  - concurrency for performance

Multi-core Processors

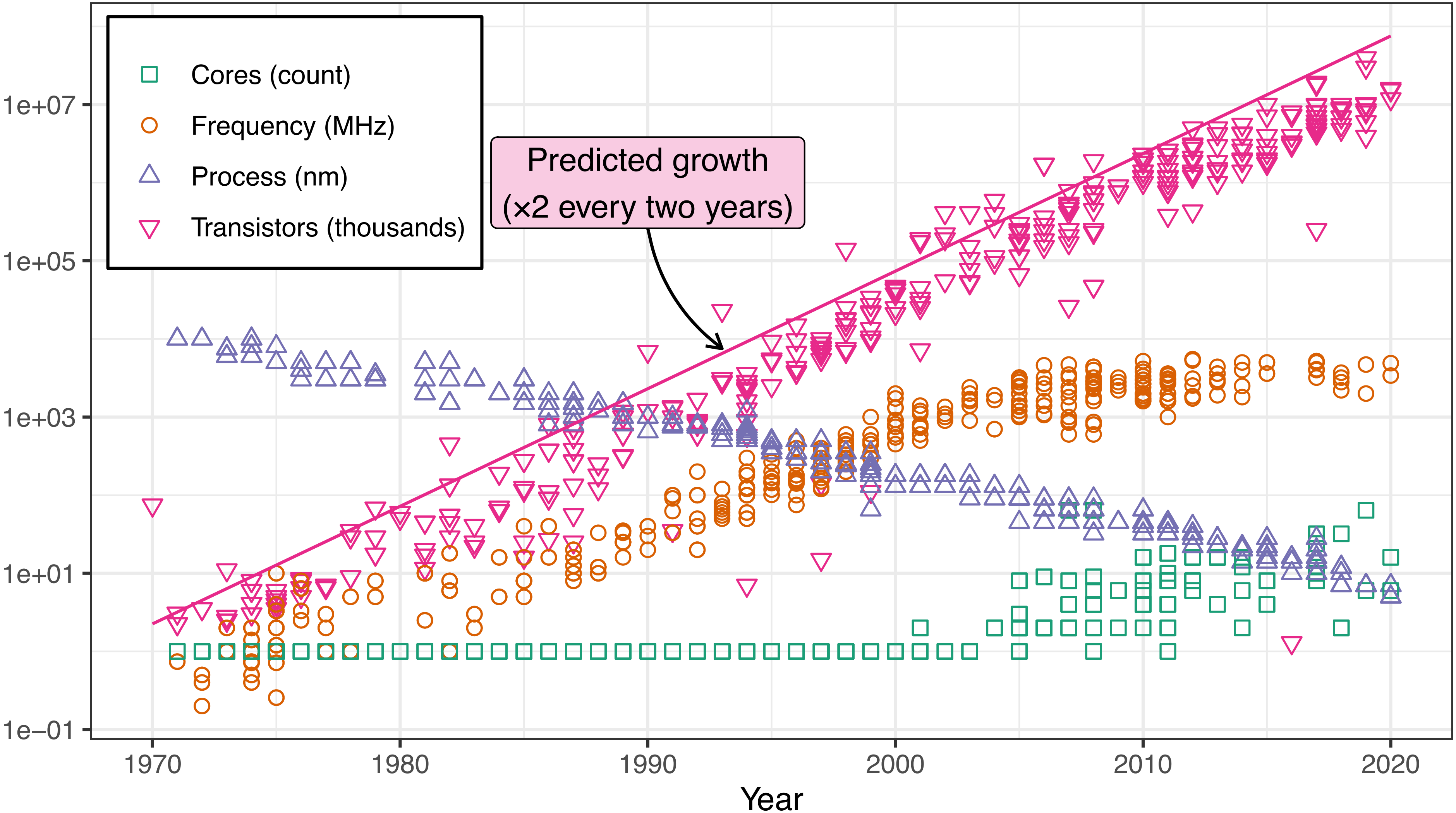
- There have always been different hardware resources to use concurrently to increase performance
  - e.g., the disk, the network, and the CPU can all be used at the same time because they are different pieces of hardware
- But the last decades have seen the advent of multi-core processors, which are now ubiquitous
- Many programs have been made concurrent so as to utilize multiple cores concurrently
  - It's become impossible to say "I am an employable software developer but I don't deal with concurrency"
- How come we have multi-core processors in the first place?

Moore's Law



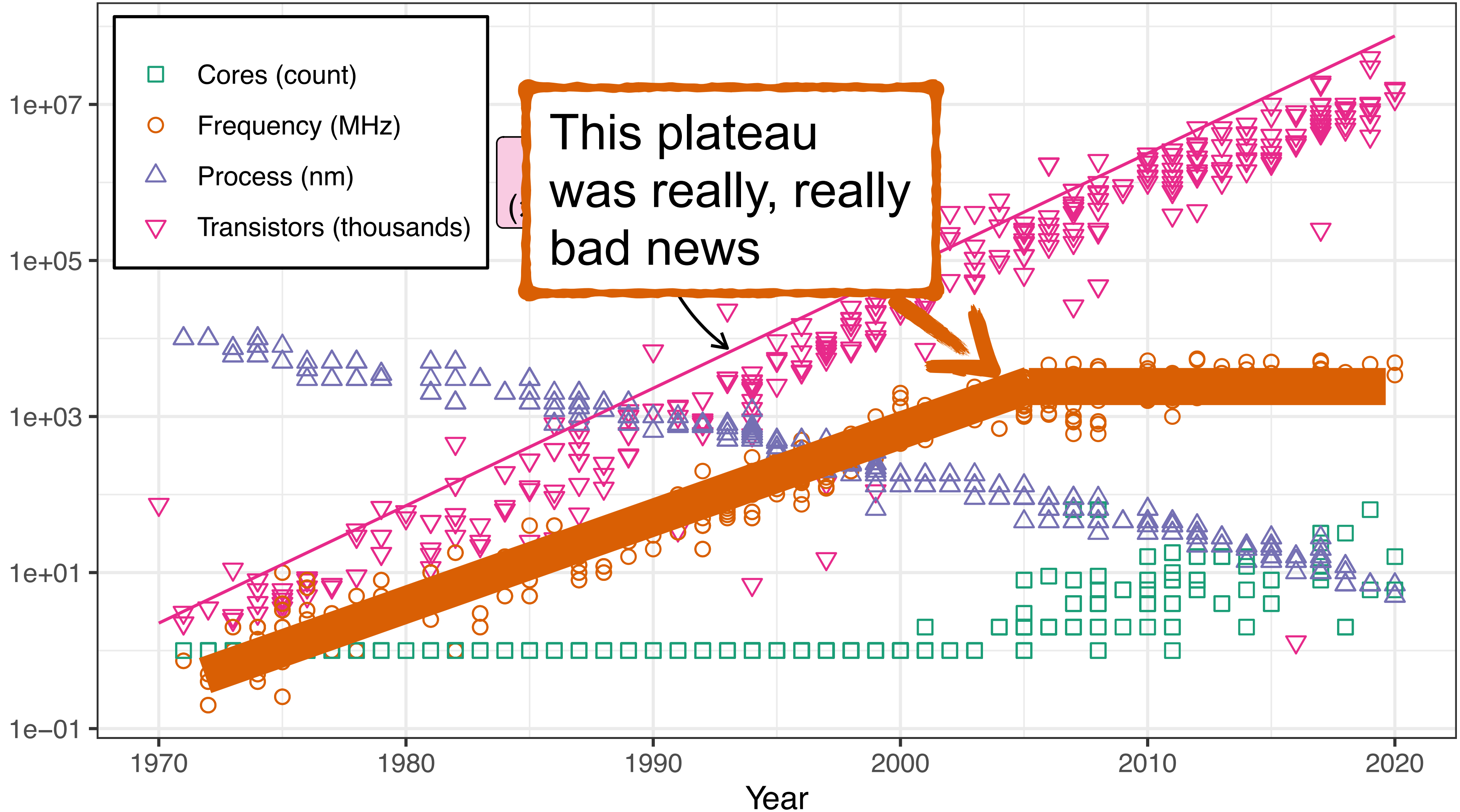
- In 1965, Gordon Moore (co-founder of Intel) predicted that transistor density of semiconductor chips would double roughly every 24 months (often "misquoted" as 18 months)
  - He was right
  - But, the law was often wrongly interpreted as: "Computers get twice as fast every 2 years"
  - This wrong interpretation was true for a while, but no longer...

50-year Trend



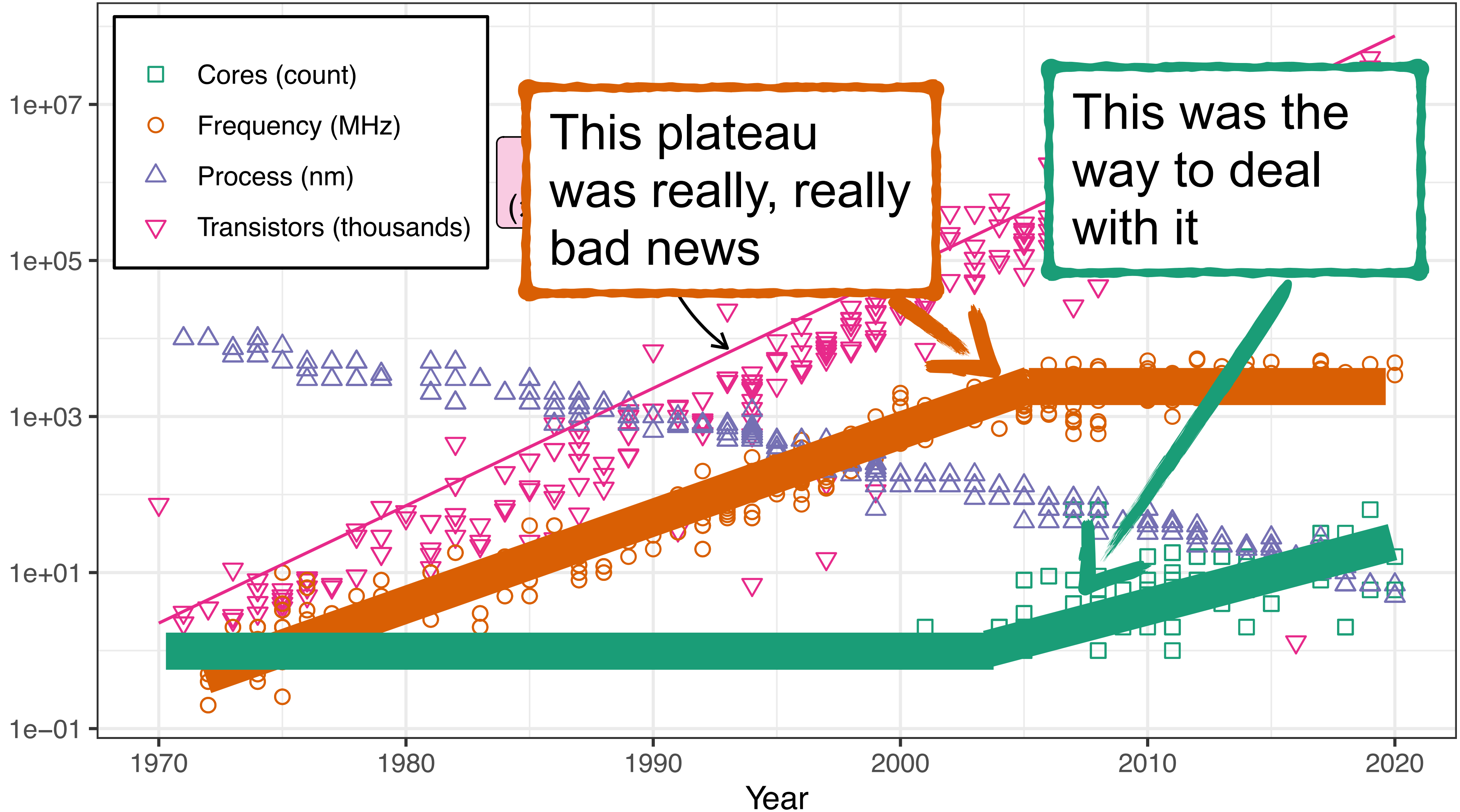
Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

50-year Trend



Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

50-year Trend



Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

Multi-core Chips

- Constructors cannot increase clock rate further
  - Power/heat issues
- They bring you multi-core processors
  - Multiple “low” clock rate processors on a chip
- It’s really a solution to a problem, not a cool new advance
  - Even though there are many cool/interesting things about multicore processors
  - Even though writing concurrent code is cool/interesting, as we’ll see in this course
- But most users/programmers would rather have a 100GHz core than 50 2GHz cores
  - In which case we would not need to write concurrent programs
  - When given the choice, if you can get by without concurrency, you’re likely better off (until you can’t avoid concurrency anymore)
  - i.e., in general no compiler will nicely take your sequential app and magically transform it into an efficient multi-threaded app

So, Multi-Core = High Performance?

- A big question is: how much performance benefit can we really get from concurrency?
- It’s a difficult question because the answer
  - depends on the application
  - depends on the computer
  - depends on the language / operating system
- In some cases, it’s very easy to achieve great performance via concurrency
- In others, it’s very difficult
- We’ll be exposed to this in this course

Take-away

- Concurrency is about structuring your programs as sets of tasks
- Typically done for interactivity and/or for performance
- Concurrency is supported by programming languages, by OSes, and by the hardware
- Issues for programmers:
  - Correctness (we’ll see this can be a tough one)
  - Performance (sometimes easy, sometimes not)

Task-based Thinking

- From now on, you should begin writing code with concurrency in mind (even if the code is not concurrent right away)
- You currently think of your programs as sets of classes/objects
  - Or data structures and functions
- But now, you also have to think of your programs as sets of tasks
  - And these tasks operate on the objects
- This requires a little bit of mental adjustment, and our programming assignments will help making that adjustment
  - You have been exposed to this a little bit in ICS332, which will help too
- Thinking of programs as “sets of tasks” will become second nature soon enough
  - And is very natural for many applications actually
  - e.g., each tab in your Web browser is a task
- When I write a sequential program, I typically think of it as a concurrent program that happens to have a single task

Conclusion

- In the next set of lecture notes, we'll look at processes and threads
  - What they are (super quick ICS332 "review")
  - How we can use them to make an application concurrent (beyond ICS332)