Intro to node.js®

or: Kernels, Processes, Threads – Oh My!
or: Fun with Modules
or: The Kitchen Sink & Async
or: Six Degrees from Ryan Dahl

What is Node?

Node.js: the Essentials

- Node is a program (written in C, C++, and JS)
- It can run JavaScript via the Chrome V8 engine (C++)
- It provides APIs for your OS's file / network system (slow I/O)
- In other words, Node lets you run JS on a computer outside of a browser and interact with stuff on that computer
- Why?

Why use JS outside a browser?

- Firefox & Chrome & Safari & IE are all working to have the fastest JS engine
  - So JS engines are very fast
- You can share code between your backend & frontend
- Asynchronous IO has proven powerful

Upcoming

- How computers multi-task
  - CPUs / processes / threads
  - Scheduling system
- History / context
- Node as process
- Node internals
- Node asynchronicity & the event loop

Multi-tasking
Multi-tasking
Multi-tasking
Multi-tasking
Multi-tasking
Multi-tasking
Multi-tasking
Multi-tasking
Multi-tasking
Multi-tasking

Program vs. Process

"recipe"

- Program is data
  - machine code (pre-compiled)
  - bytecode
  - text file (can be interpreted)
- Inert – not doing anything
- Ready to be run as a process

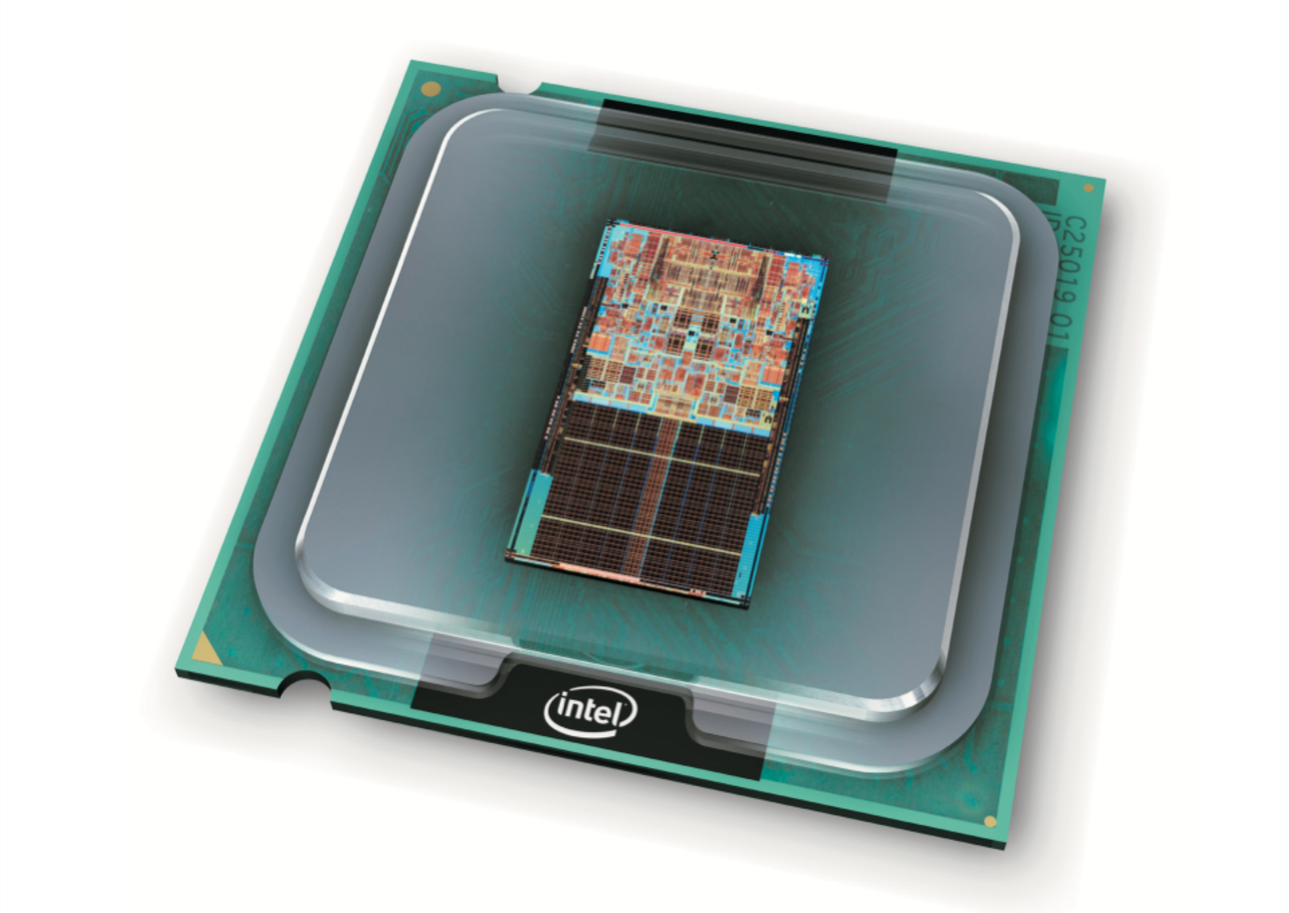
"cooking"

- Process is execution
  - memory allocated
  - CPU performing steps
- "Live"
- Produces results
- Interactive
- Can be started/stopped
- Multiple processes from one program…

"chef" — can only swear, drink, and speak machine code

CPU Core

- A big calculator
- Performs one instruction at a time
- Has a little bit of memory (registers)
- Can do basic math (`add, sub, mul...`)
- Can do comparisons (`cmp, test...`)
  - `cmp r0, 100`
- Can goto (jump: `jmp, jne, jeq`)
  - `jne start`



Grab File Edit Capture Window Help

Skype

Fullstack Academy
$9.52

Home
Contacts
Recent

Kyle Burke
Offline

en.wikipedia.org/wiki

"busywait" while waiting for requested input (such as disk, keyboard or network input). During this time, the process was not performing useful work, but still maintained complete control of the CPU. With the advent of interrupts and preemptive multitasking, these I/O bound processes could be "blocked", or put on hold, pending the arrival of the necessary data, allowing other processes to utilize the CPU. As the arrival of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

Although multitasking techniques were originally developed to allow multiple users to share a single machine, it soon became apparent that multitasking was useful regardless of the number of users. Many operating systems, from mainframes down to single-user personal computers and no-user control systems (like those in robotic spacecraft), have recognized the usefulness of multitasking support for a variety of reasons. Multitasking makes it possible for a single user to run multiple applications at the same time, or to run "background" processes while retaining control of the computer.

Time slice [edit]

The period of time for which a process is allowed to run in a preemptive multitasking system is generally called the *time slice*, or *quantum*. The scheduler is run once every time slice to choose the next process to run. The length of each time slice can be critical to balancing system performance vs process responsiveness - if the time slice is too short then the scheduler will consume too much processing time, but if the time slice is too long, processes will take longer to respond to input.

An interrupt is scheduled to allow the operating system kernel to switch between processes when their time slices expire, effectively allowing the processor's time to be shared between a number of tasks, giving the illusion that it is dealing with these tasks simultaneously, or concurrently. The operating system which controls such a design is called a multi-tasking system.

Systems supporting preemptive multitasking [edit]

Today, nearly all operating systems support preemptive multitasking, including the current versions of Windows, Mac OS,

Slack

Fullstack Academy
Gabriel Lebec

general
random
help-ticket-test
+51 More...

DIRECT MESSAGES
slackbot
Alex Castillo
Alexander Holman
Ben Neiswander
Chelsea Du
Chris Carrick
Cristina Colon
Jay Lee
Jenny Cui
Jimmy Farrell
Michael Abraham
Mike Lee
Roger Beaman
Ryan Dwyer

#random 287

July 30th, 2015

WHAT'CHU TALKIN' 'BOUT, WILLIS?

Stephen Colacrucio 10:28 PM
left #random, along with @tmoreton

July 31st, 2015

Christian Cueto 1:11 PM
Slack Update

1 Update Available
Slack
Slack Technologies, Inc.
Version 1.1.2
Released Jul 31, 2015
- Fixed: Keeping everything locked down.

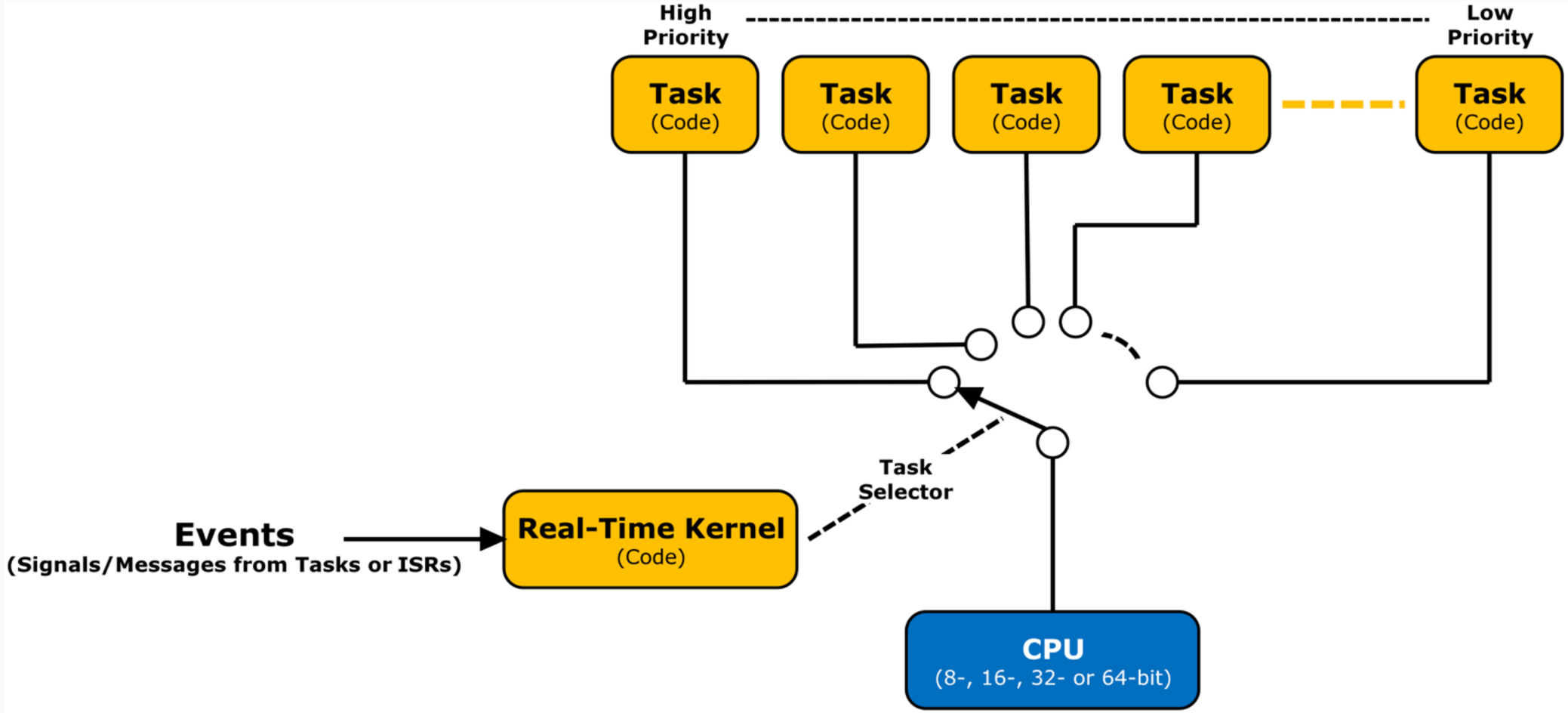
40KB PNG in #random • Add comment • Open original

Highscores Help

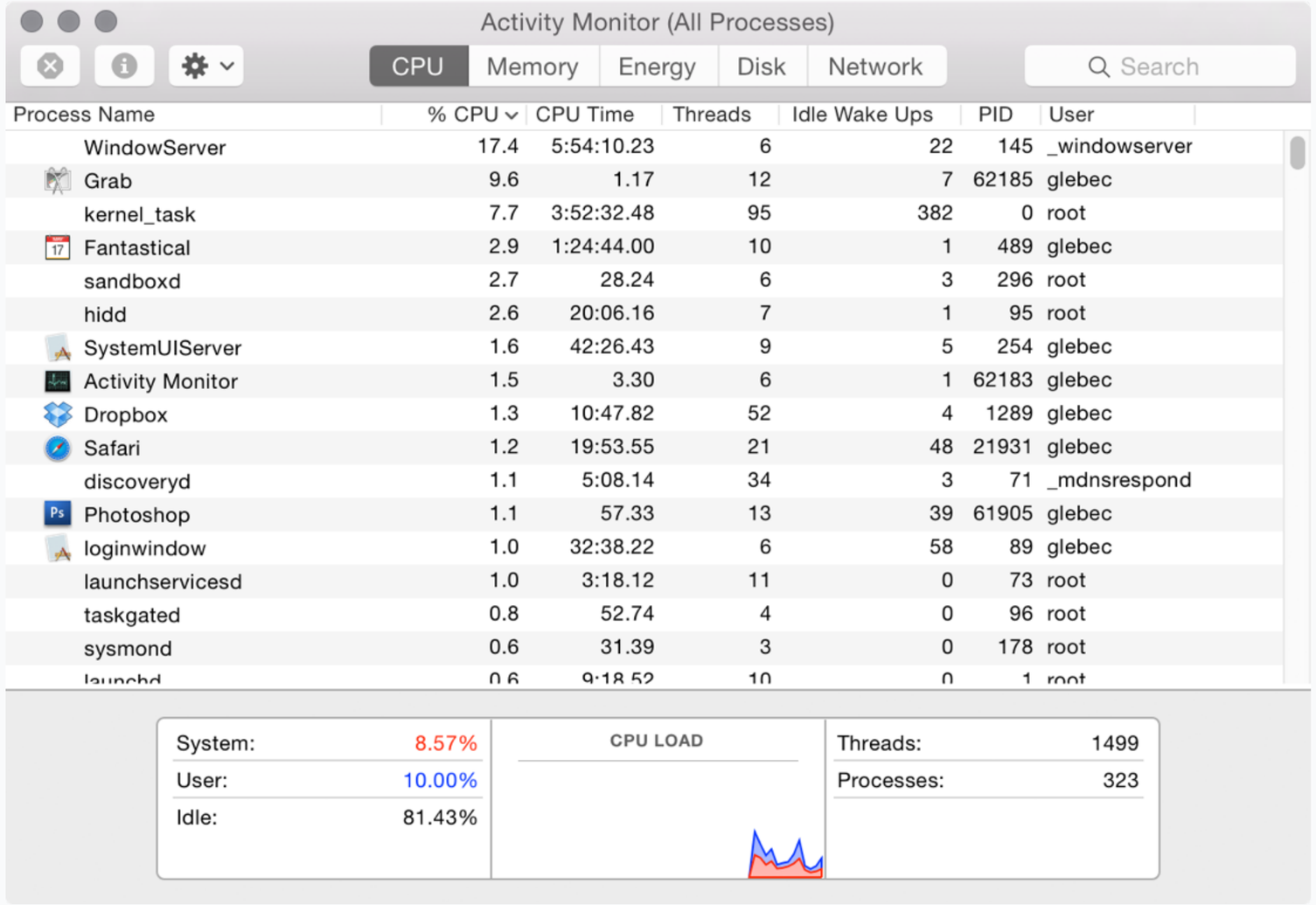
NEXT
SCORE 2
LINES 0
LEVEL 1
LPM 0.00

"manager"

Time Slice



The OS (Kernel) Manages Processes



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
WindowServer	17.4	5:54:10.23	6	22	145	_windowserver
Grab	9.6	1.17	12	7	62185	glebec
kernel_task	7.7	3:52:32.48	95	382	0	root
Fantastical	2.9	1:24:44.00	10	1	489	glebec
sandboxd	2.7	28.24	6	3	296	root
hidd	2.6	20:06.16	7	1	95	root
SystemUIServer	1.6	42:26.43	9	5	254	glebec
Activity Monitor	1.5	3.30	6	1	62183	glebec
Dropbox	1.3	10:47.82	52	4	1289	glebec
Safari	1.2	19:53.55	21	48	21931	glebec
discoveryd	1.1	5:08.14	34	3	71	_mdnsrespond
Photoshop	1.1	57.33	13	39	61905	glebec
loginwindow	1.0	32:38.22	6	58	89	glebec
launchservicesd	1.0	3:18.12	11	0	73	root
taskgated	0.8	52.74	4	0	96	root
sysmond	0.6	31.39	3	0	178	root
launchd	0.6	9:18.52	10	0	1	root

- Processes are **isolated**
- Each process thinks it has 256 TB of memory all to itself
- The kernel maintains this illusion (virtual address space)
- The kernel pre-empts processes when their time slice is up (task switching)

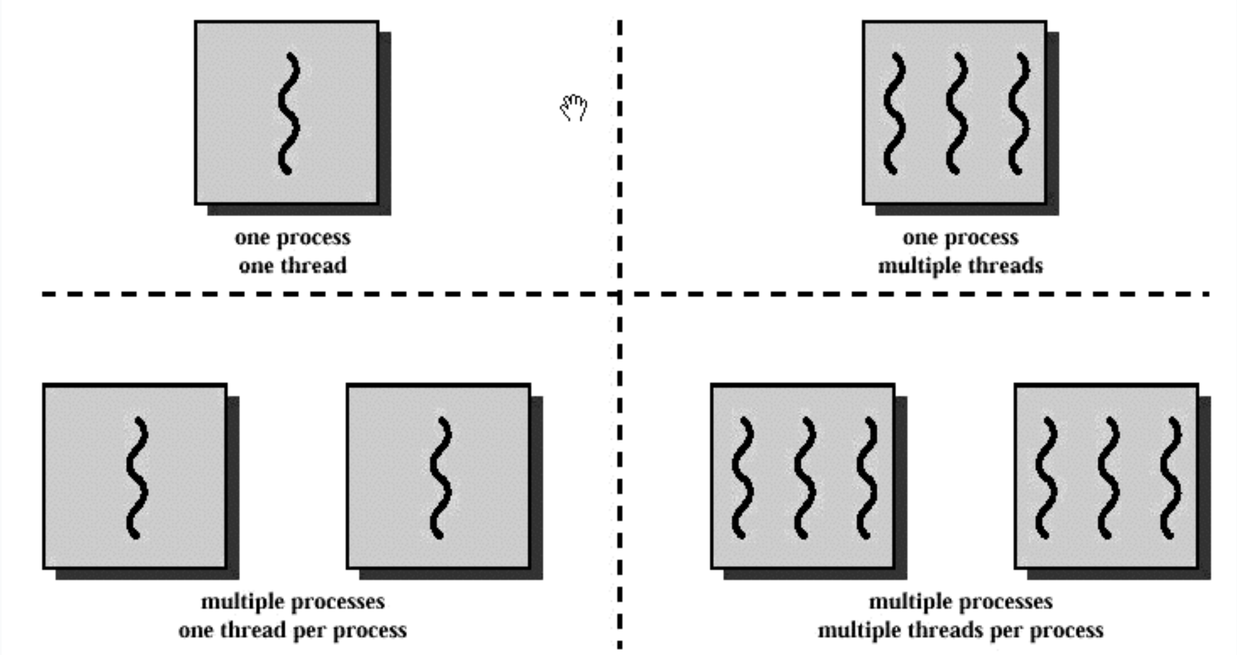
What if a process itself needs to multi-task?

What if a single process needs to multitask?

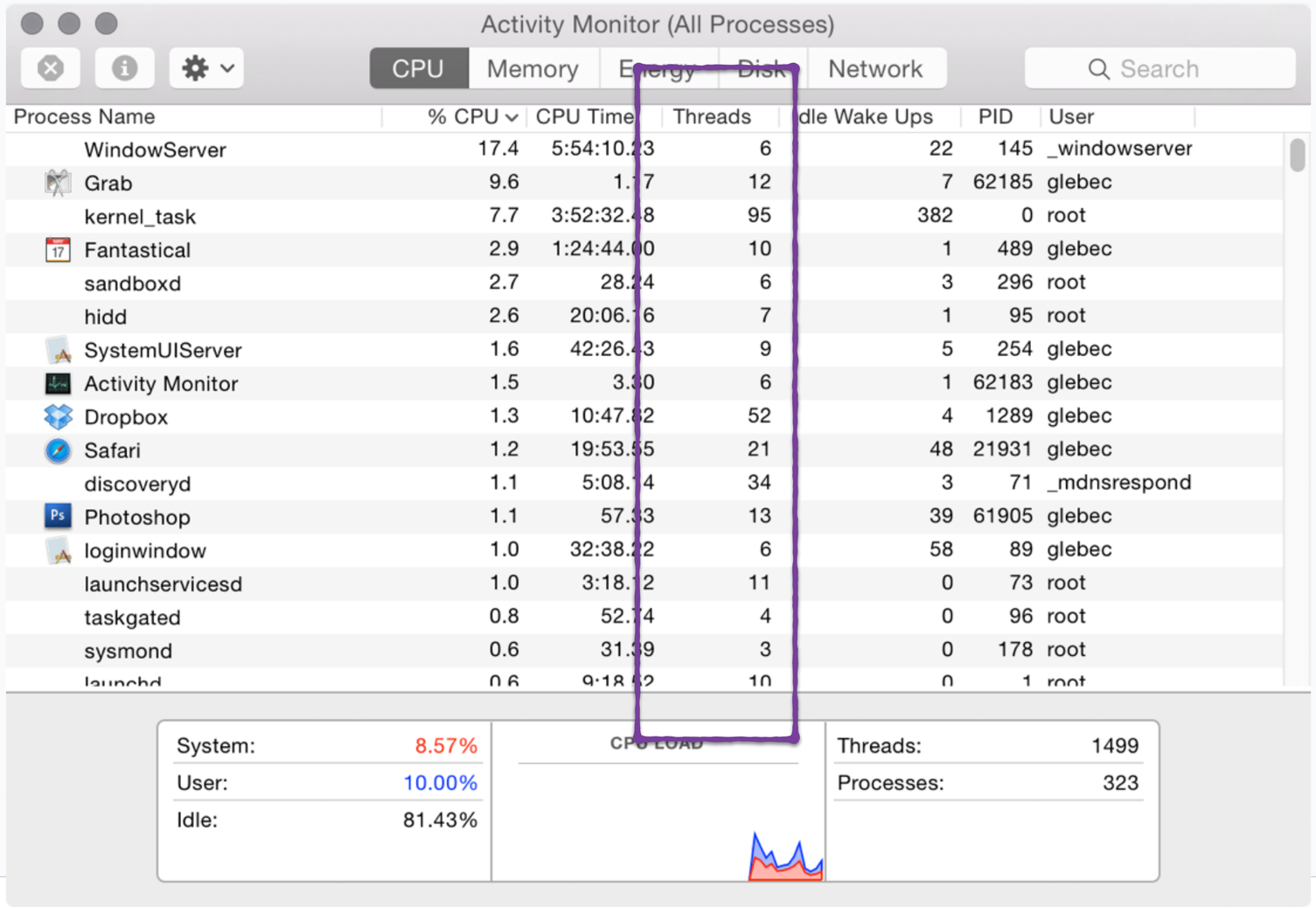
- Use case: we're writing Final Cut Pro
- We want to render a video while still responding to user input
- We could start another process to do the rendering
- Problem: Sharing data between processes is expensive
  - Your process can't see the data in the other process's address space
  - You have to ask the kernel to send it over somehow
- Solution: Threads!

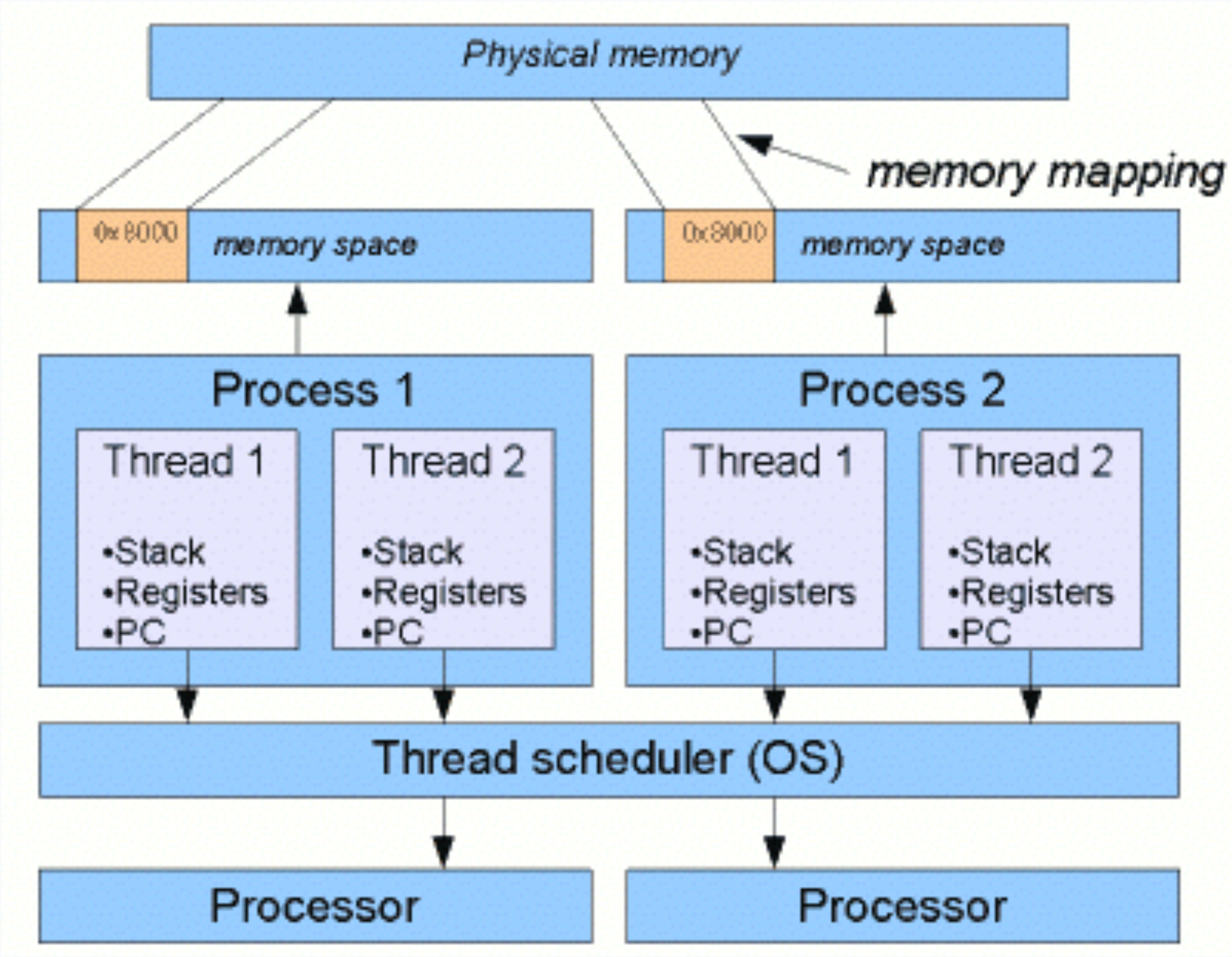
Threads

- When you step through your code in your head (or in a debugger), you are running a thread
- Every thread has its own call stack
- One *or more* threads per process – they share memory

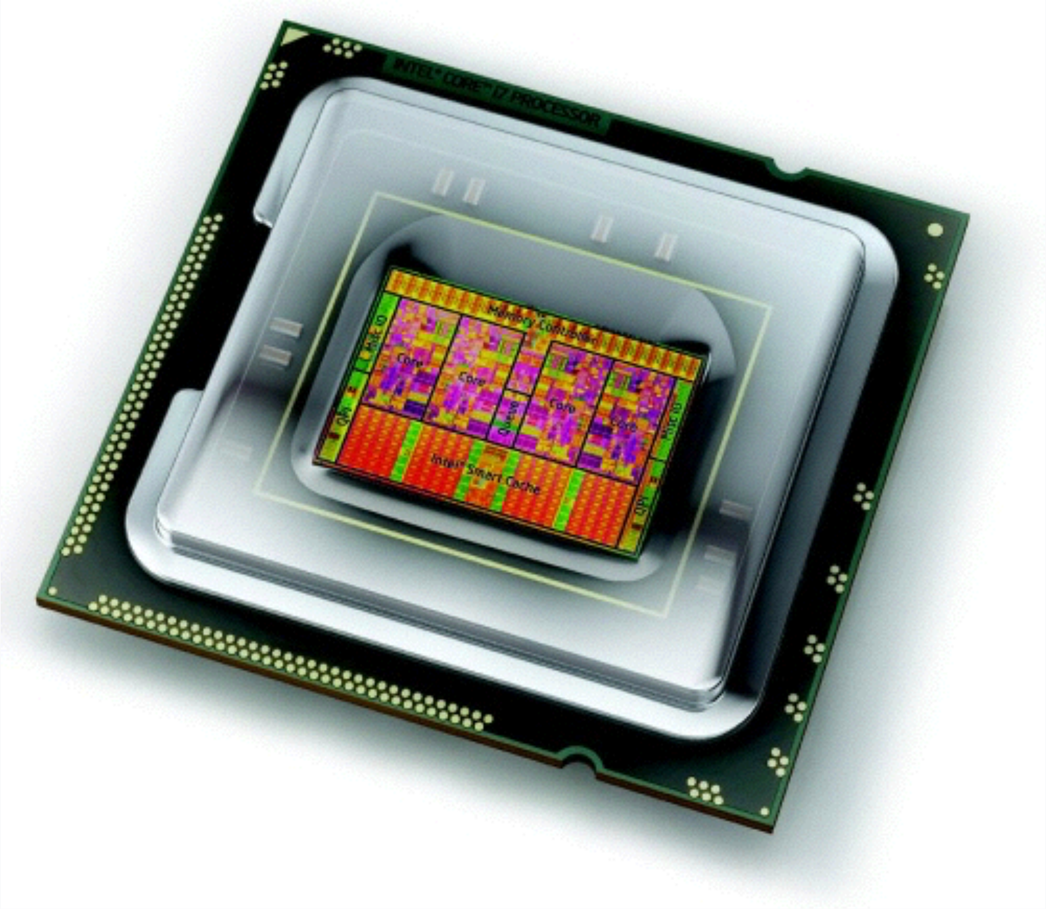


The OS Schedules Threads



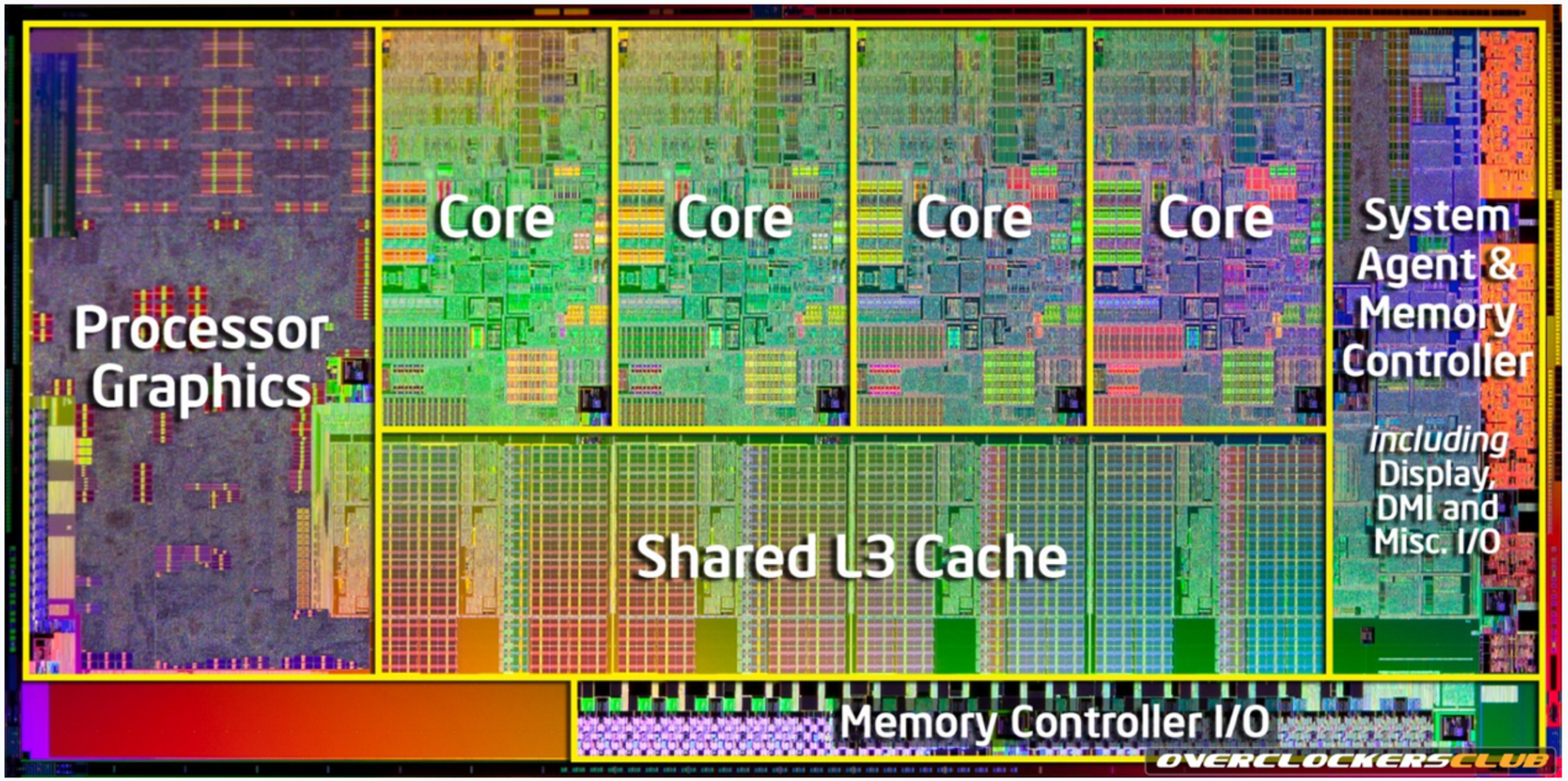

Physical memory
memory mapping
0x8000
memory space
0x8000
memory space
Process 1
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Process 2
Thread 1
•Stack
•Registers
•PC
Thread 2
•Stack
•Registers
•PC
Thread scheduler (OS)
Processor
Processor

…Wait – Multiple Processors?



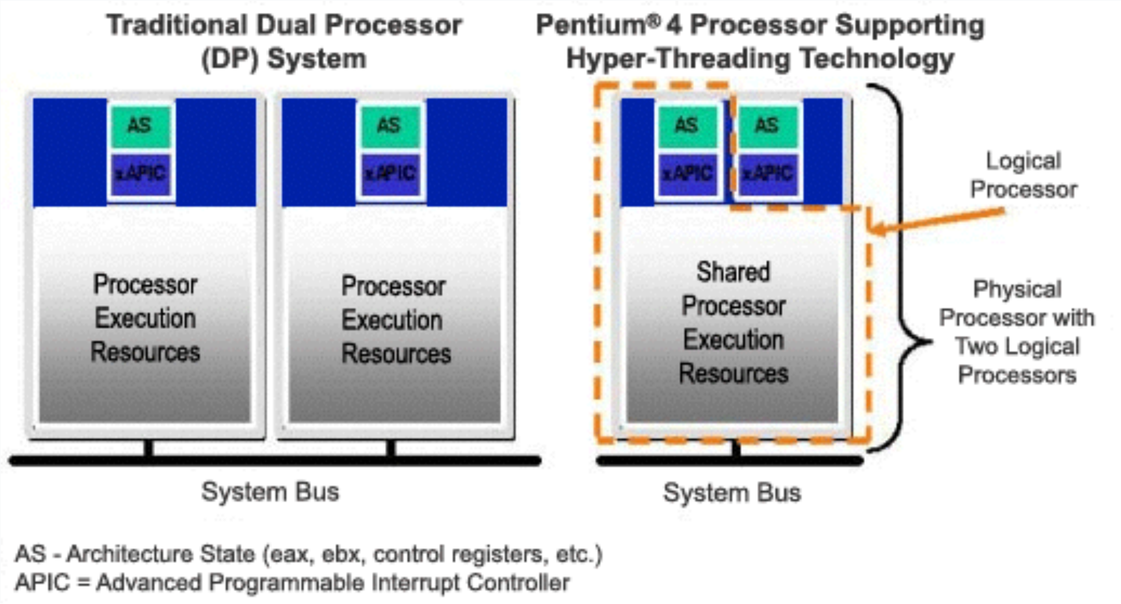
Multi-Core Processors

"line cooks"



But wait, there's more – HyperThreading

- Each physical core can run *two* threads, doubling the number of "logical" cores!
- These logical cores are a lie
- But everything is a lie, so it's okay



"a line cook that is actually two children standing on each others' shoulders and wearing a chef coat

"they have more hands than usual, so you can give them the work of two people.

"this metaphor is getting really strained."

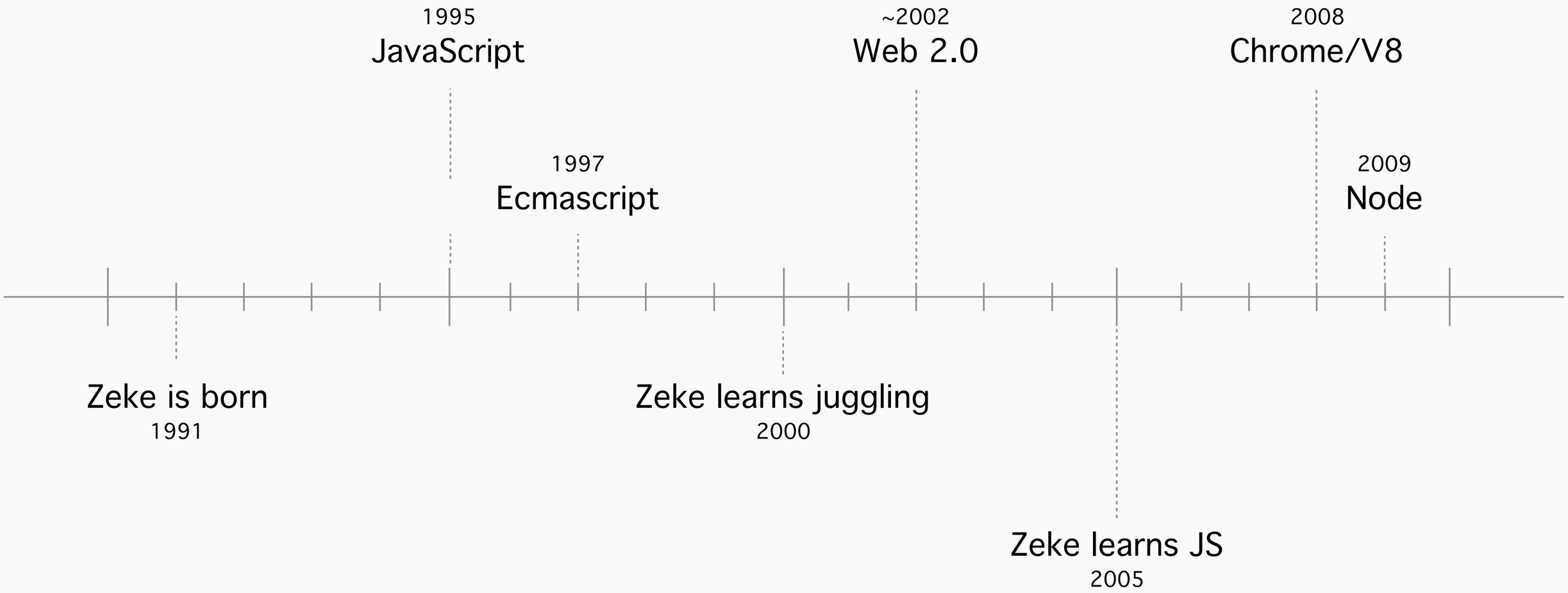
Takeaway:

- Program is a file (data) that can be loaded/executed as a process
- Process is state (memory/variables) and one or more threads
- Thread is a sequence of operations that the CPU performs
- Operating system (specifically, the kernel) schedules jumps between threads (loading and unloading state), as there are many more threads than logical cores
- CPU cores are the machines that actually perform steps; most processors have 2–4 physical cores / 4–8 logical cores

Node as Process

(Demo)

A Brief History of Everything



JS in Node vs Browser

fs

process

net

window

history

document

Browser Tool Examples

- Synchronous stuff:
  - console
  - window
  - document
- Asynchronous stuff:
  - setTimeout(callback, delay) (note: not in ECMAScript, "not JS" – an API!)
  - setInterval(callback, delay) – ditto above
  - new XMLHttpRequest() / .open, etc. – making HTTP requests
  - element.onclick(callback) // slightly different example

Node Tool Examples

- Synchronous stuff:
  - console (same name, but a wrapper for process.stdout)
  - global (no window!)
  - module (we'll get to this)
- Asynchronous stuff:
  - setTimeout(callback, delay) – same name for convenience's sake
  - setInterval(callback, delay) – ditto
  - http.request(options, callback) – uses built-in `http` JS module
  - etc.

Big Use Case: Server

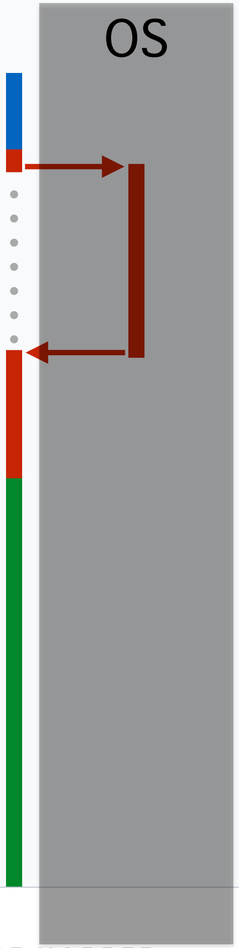
- A program running on a computer
- Listens on a *port* for *requests*
- Sends *responses* back
- Communication must follow agreed-upon *protocol* (e.g. HTTP)
- [DEMO]
- Problem: communication over network is super-slow! Should be able to field other requests / do other stuff in meantime.

Slow *blocking* I/O is handed off to a thread

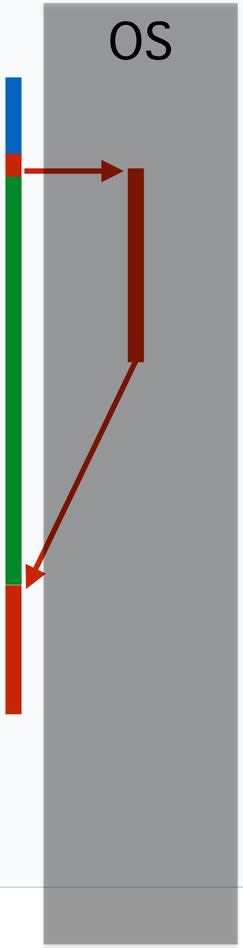
- CPU clock cycles per access level (*Dahl 2010):
  - Dynamic memory: "non-blocking"
    - L1 cache: 3
    - L2 cache: 14
    - RAM: 250
  - "Blocking" I/O:
    - Disk: 41,000,000
    - Network: 240,000,000

Concurrency

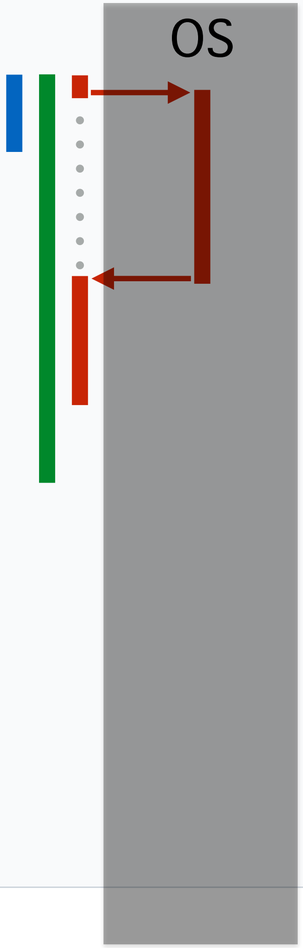
blocking



non-blocking



parallel (blocking)



"Threads are Hard"

- Every thread comes with performance overhead – memory, CPU, etc.
- Creating threads requires talking to the kernel: expensive
- Switching threads between processes consumes clock cycles AND waits for memory (have to load process state)
- Communicating between threads in the same process is very tricky – synchronizing results, being efficient

```
var counter = 0

function increment() {
   counter = counter + 1
}

var t1 = create_thread(increment)
var t2 = create_thread(increment)

t1.wait_until_done()
t2.wait_until_done()

// What's counter now?
```

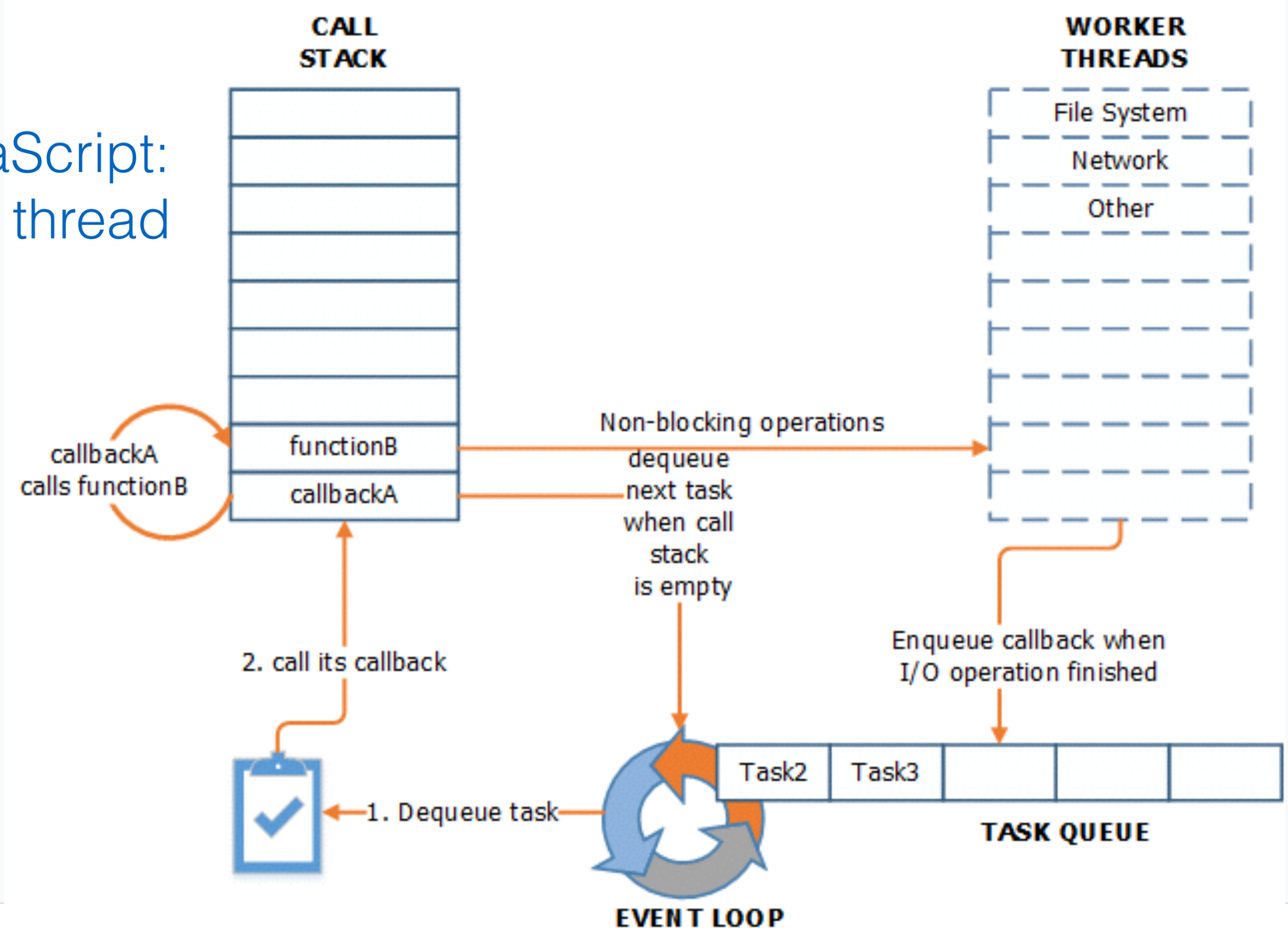
Server Strategies

- Nginx: an event loop which queues *callbacks* back into the main execution stack, once thread managing

- Apache: every new connection gets its own thread (wasteful of resources, doesn't scale well, lots of work to synchronize)

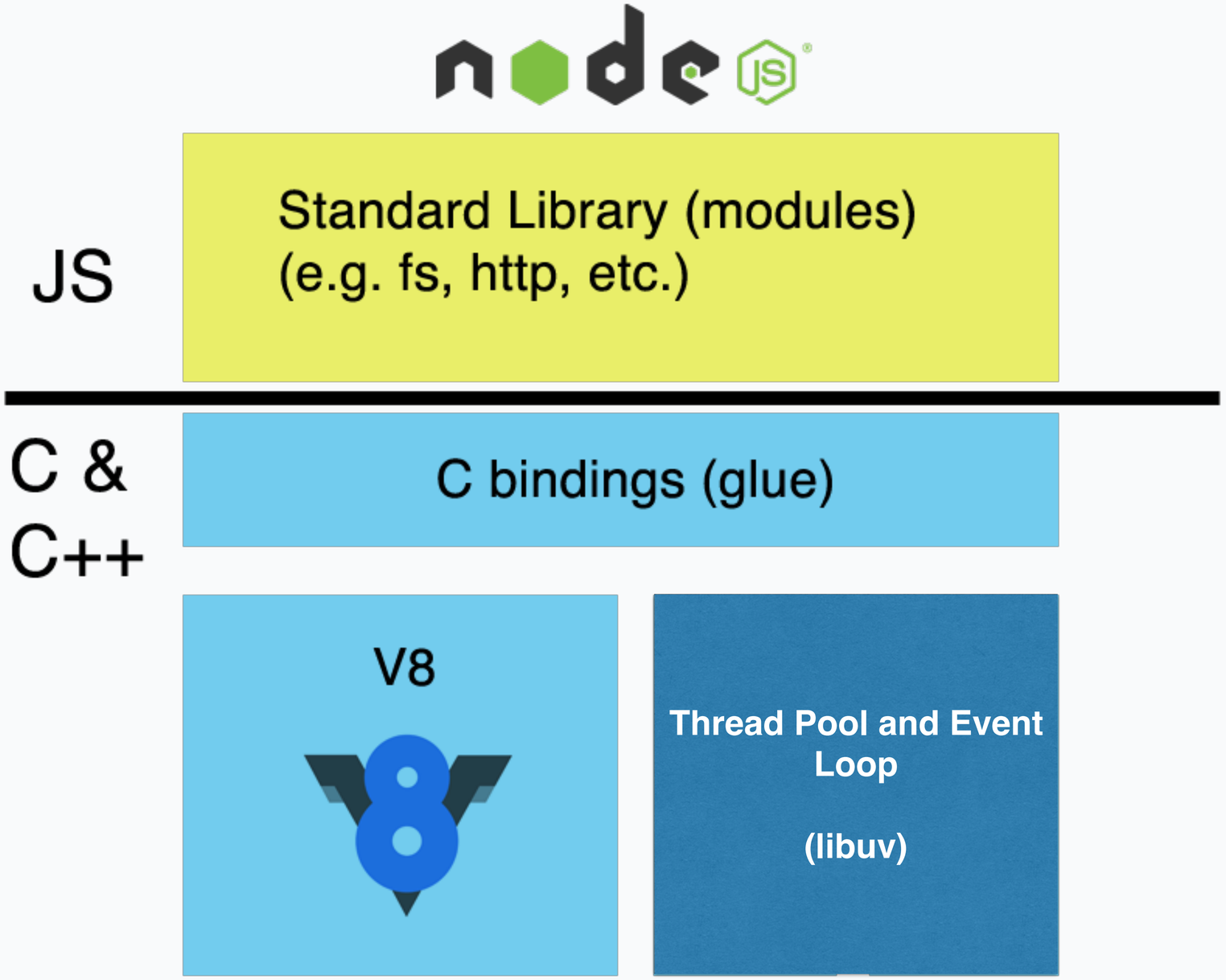
Event Loop

JavaScript:
One thread



Thread pool (libuv):
Slow stuff, multiple threads

Event loop (libuv):
One thread


node
JS
Standard Library (modules)
(e.g. fs, http, etc.)
C &
C++
C bindings (glue)
V8
Thread Pool and Event Loop
(libuv)