Intro to Threads and Multithreading

Advanced Embedded Linux Development

with **Dan Walkes**

University of Colorado **Boulder**

Learning objectives:

Thread overview/review

Multithreading Tradeoffs

Threading models

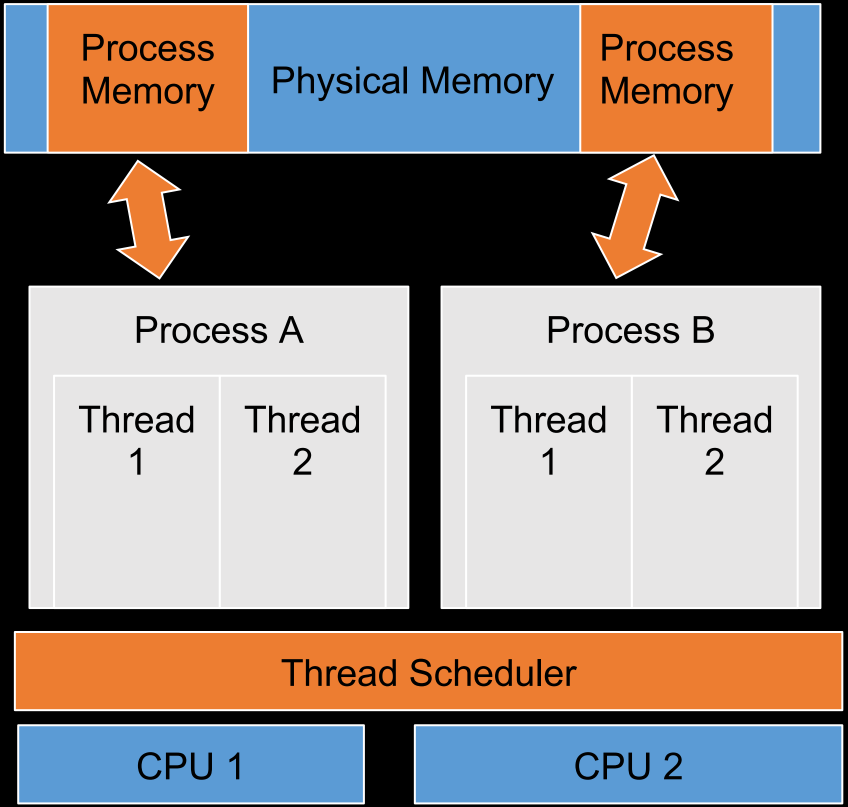
Threads

- Binaries: dormant programs on storage
- Processes: Binaries in action, running on the system
  - Contains one or more threads
  - One thread = single threaded
  - More than one thread = multithreaded
  - Has dedicated virtualized memory

Threads

- Threads: Units of execution within a process
  - Virtualized processor
  - Stack
  - Program state
  - **Shares** virtualized memory

Threads



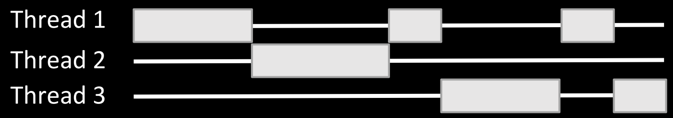
- Process memory maps to different physical address
- Threads share same memory space

https://www.javamex.com/tutorials/threads/how_threads_work.shtml

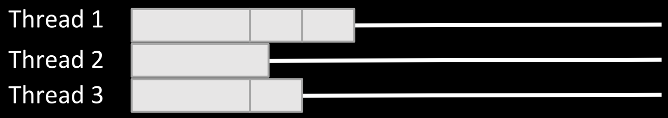
Why Use Multithreading?

- Alternative to state machines
- Parallelism - scales your process application to multiple processors.
- Improve responsiveness
  - Long running operations don’t belong in UI handling threads.
  - Blocking I/O doesn’t belong in UI threads.

Concurrency and Parallelism



- Concurrency
  - Ability of two or more threads to execute in overlapping time periods
  - Programming pattern



- Parallelism
  - Ability to execute two or more threads simultaneously
  - Requires multiple processors (hardware)

Why Use Multithreading?

- Context switching
  - Thread context switching is “cheaper than” process context switching.
- Memory savings
  - Share memory across multiple execution units.

Why *Not* Use Multithreading?

- Complexity
  - writing, understanding, debugging
- Alternatives to Multithreading
  - Non-blocking I/O
  - Multiple processes

Thread Per Connection Model

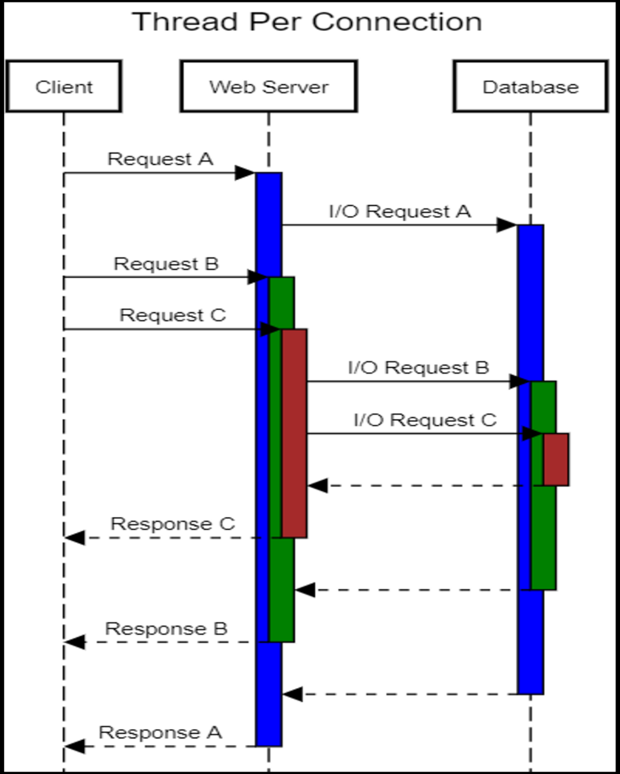
- Each unit of work (request, connection, etc) gets a thread.
- Thread performs blocking I/O
  - May need more threads than CPU processors for maximum parallelism.
- Breaks down as number of requests becomes very large.
- Apache web server is an example of this model

https://opensource.com/business/16/8/top-5-open-source-web-servers

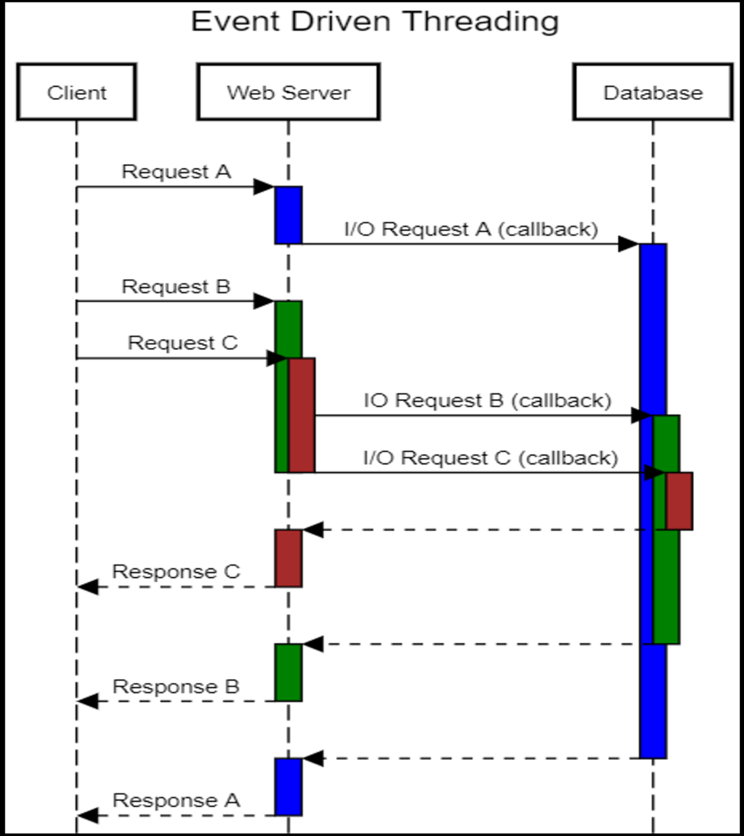
Event-Driven Threading Model

- Uses asynchronous I/O and callbacks
- Uses an event loop
- NGINX, Node.js are examples of this model
- No reason to have more threads than CPU processors
- No waiting in threads
- Author’s suggestion: try this pattern first for your application.

Thread Per Connection vs Event Driven



How does the max number of threads and thread duration compare?

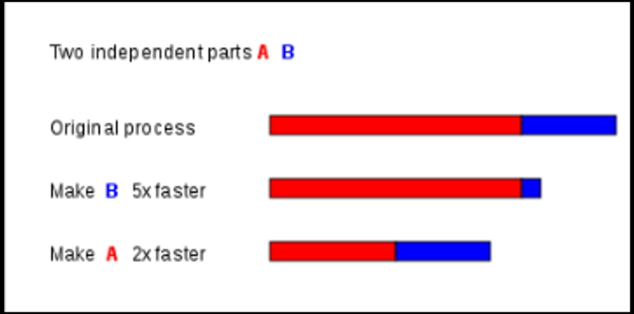


Linux System Programming Chapter 7
https://stackoverflow.com/questions/14189496/thread-in-an-event-driven-vs-non-event-driven-web-server

Amdahl’s Law

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

$$\begin{cases} S_{\text{latency}}(s) \le \dfrac{1}{1-p} \\ \lim\limits_{s\to\infty} S_{\text{latency}}(s) = \dfrac{1}{1-p}. \end{cases}$$



- S = Theoretical speedup (x multiplier)
- s = part which benefits from improved resources
- p = proportion of execution time the part benefiting from improved resources originally occupied
- How does this relate to Event Driven Threading and statement “no reason to have more threads than processors”?
  - Not doing any waiting in threads, can’t be parallelized

https://en.wikipedia.org/wiki/Amdahl%27s_law

Event Driven Threading and Parallelism

As with the thread-per-connection pattern, nothing about the event-driven pattern need be threaded. Indeed, the event loop could simply be the fall-through when a single-threaded process is done executing a callback. Threads need only be added to provide true parallelism. In this model, there is no reason to have more threads than processors.

- Second sentence is true because we can't add more threads to parallelize if all threads are non-blocking and we've already got a thread assigned to each processor.