Operating Systems Principles

CPU Management
(Processes vs. Threads)

An Interlude

- Lets come back to our discussion on CPU scheduling after learning about the notion of threads in this lecture
  - Understanding the notion of threads is important for working on project 2

Flow of Ideas in this Lecture

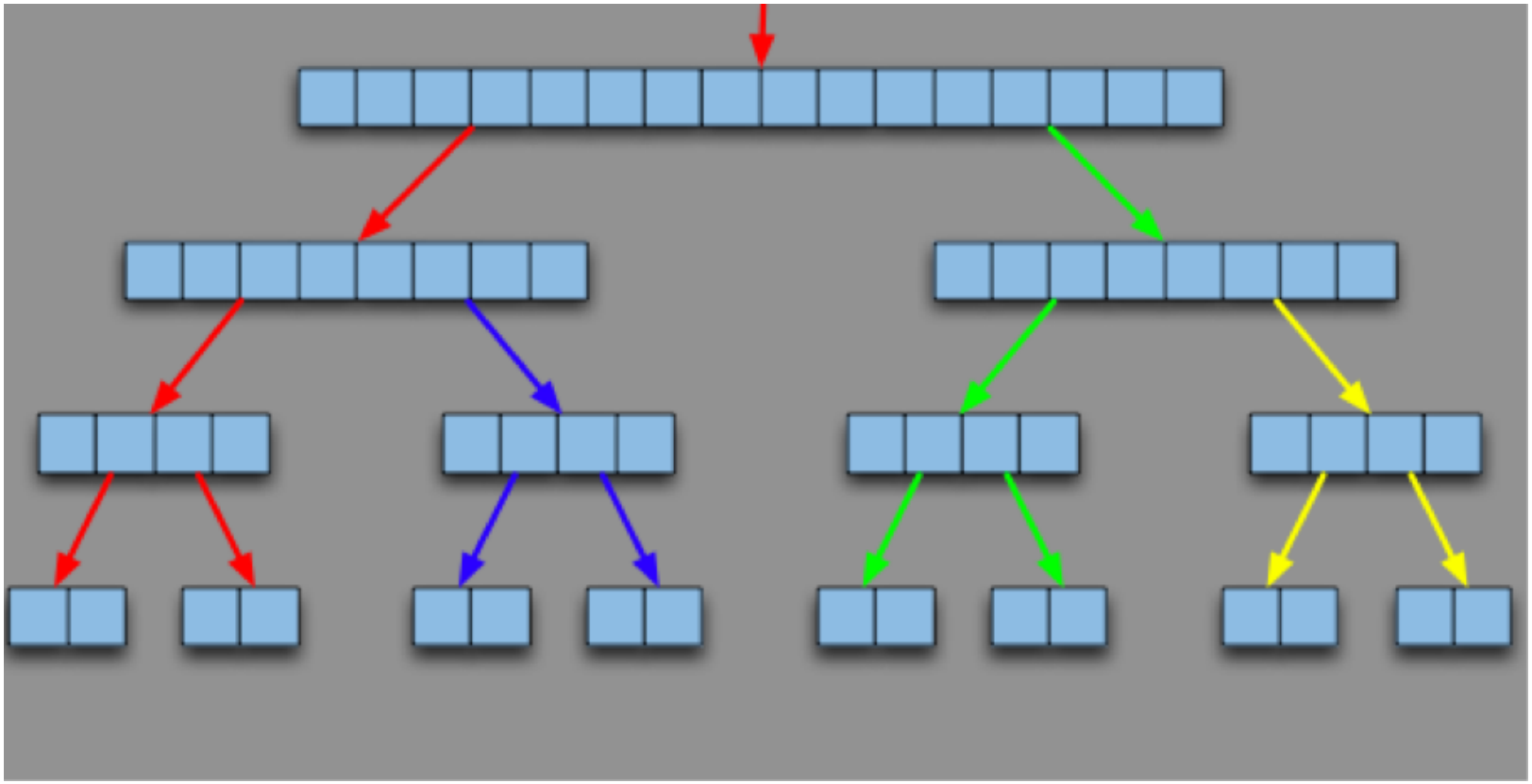
- We often want to write **concurrent** programs
- Concurrent programming needs OS support for
  - (i) data sharing and
  - (ii) synchronization that we will study later
- One way to write concurrent programs is by employing multiple processes that do (i)-(ii)
- However, processes can pose excessive overheads
- Threads are a more efficient alternative to processes for concurrent programming and we will study how the OS provides this abstraction

Introduction to Concurrent Programming

- Lets get a taste for concurrent programming by considering a simple **parallel** program
  - Example: a concurrent program for mergesort

- Our eventual goal: To understand the OS role in allowing/enabling concurrent programming

Parallel Mergesort



- The four colors correspond to four different processes cooperating with each other

Parallel Mergesort: How?

- What would the processes need from the OS?
  - #1: Some way of sharing data across address spaces
  - #2: Mechanisms to ensure that processes modify and use these shared data correctly

Parallel Mergesort: How? (2)

- #1: Some way of sharing data across address spaces
  - Two generic ways
    - **Shared Memory:** OS ensures some portions of address spaces are shared
      - e.g., pipes or shmget (UNIX)
    - **Message Passing:** Sender process copies data to OS address space (via a system call), which then copies it to the recipient process address space
      - e.g., signals, messages over sockets

Parallel Mergesort: How? (3)

- #2: Mechanisms to ensure that processes modify and use these shared data correctly
  - Similar to the register sharing problem
  - We will study this in a few lectures: process synchronization

Parallel Mergesort: Why?

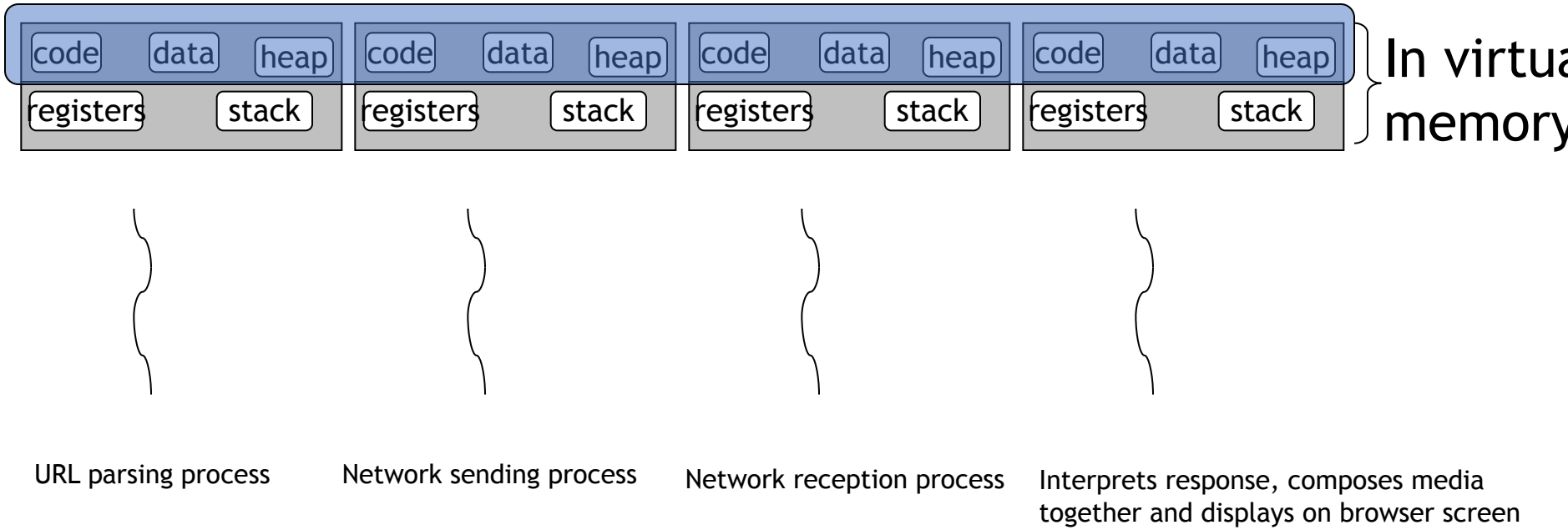
- Why would you do such a thing?
  - Is this faster than "serial" (single process) mergesort? When?
- Clearly, likely to benefit from multiple CPUs
  - E.g., each process could run on a separate CPU
- Even if only single CPU (as in this course), might improve CPU utilization
  - E.g., when process 1 is blocked on IO, process 2 might be able to run contributing to progress of the overall program

Concurrent Programs

- Many programs need to do multiple activities simultaneously
- Consider some activities a Web browser program needs to do
  - Parse requested URL and find IP address from DNS server
  - Use system calls to send request to some Web server
  - Receive response
  - Assemble response and display it

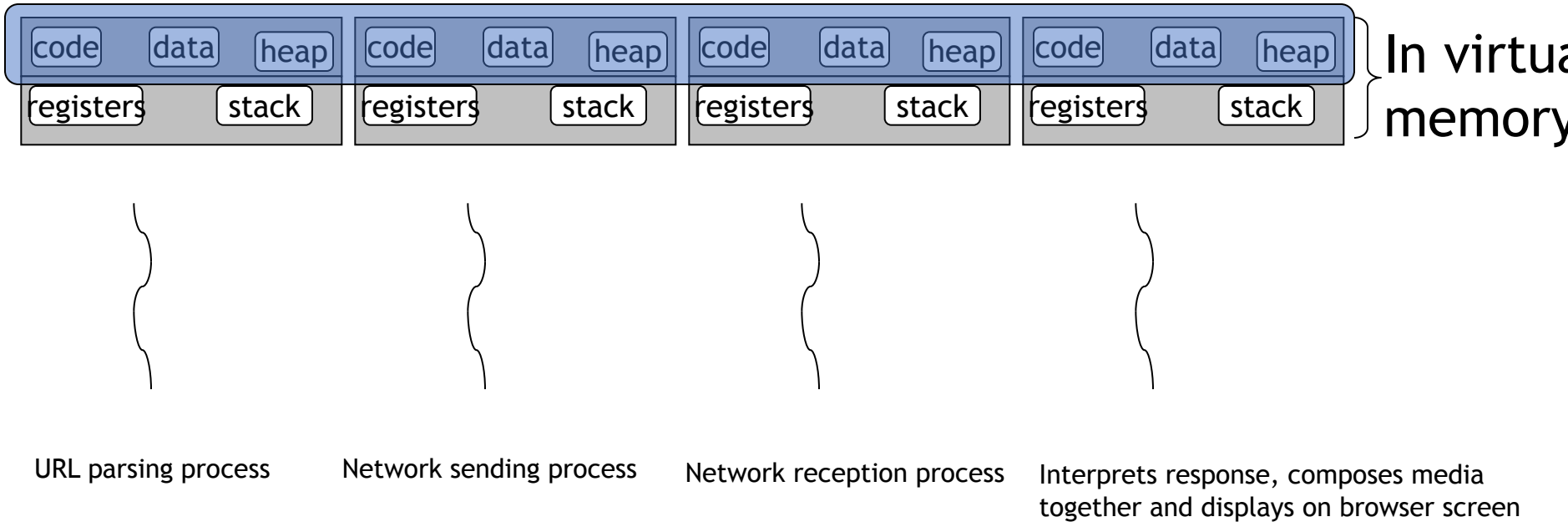
- **Performance:** Often speedup likely if we use multiple processes, one for each activity
- **Ease of design:** Often easier to write code for each "activity" as a separate process

Approach #1: Multiple Processes



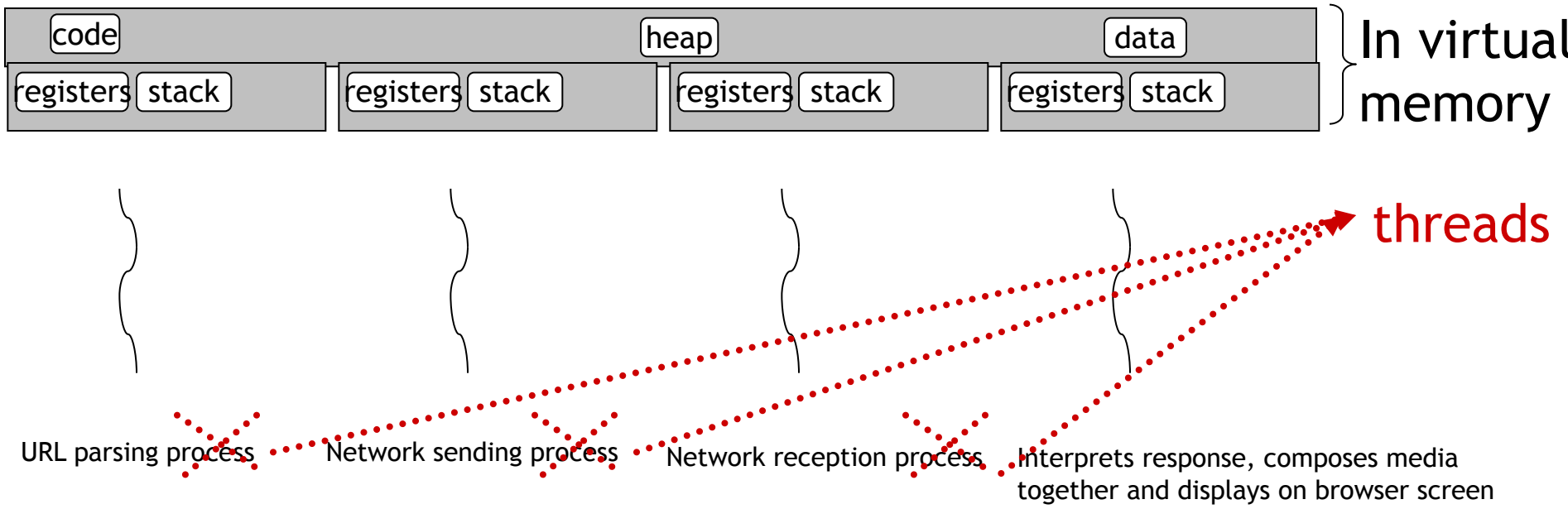
- Processes employ IPC to share data and synchronization techniques to ensure correct sharing
- **There's something terribly wrong (stupid?) with this approach!**

Approach #1: Multiple Processes



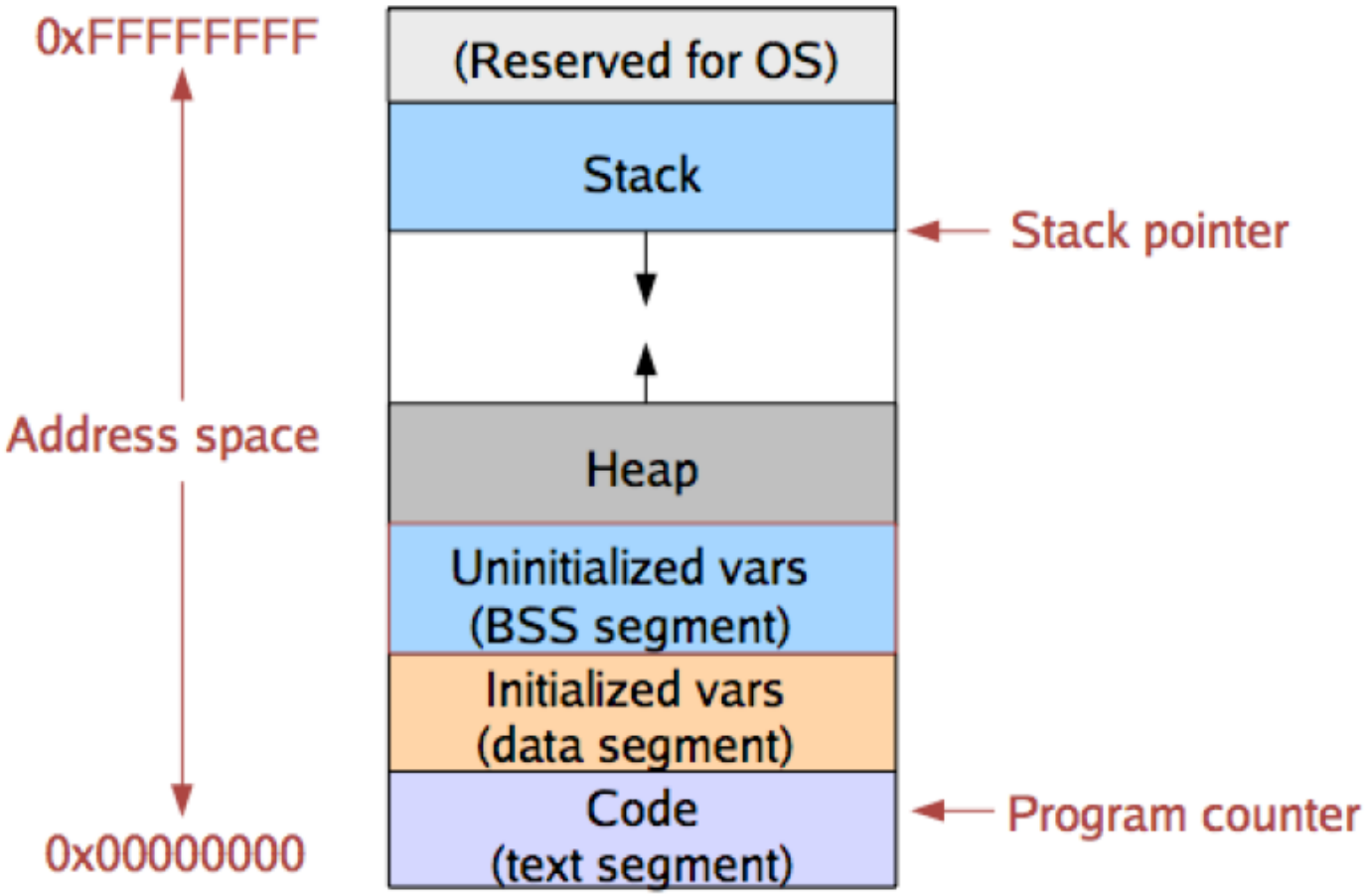
- Potentially, **lot of redundancy in code, data, and heap segments**!
  - Virtual memory wastage => More contention for precious RAM => More waiting for slow swap device => Reduction in computer's throughput
- Also, **TLB and cache contention** between processes of same program

Approach #2: Share code, data, heap!

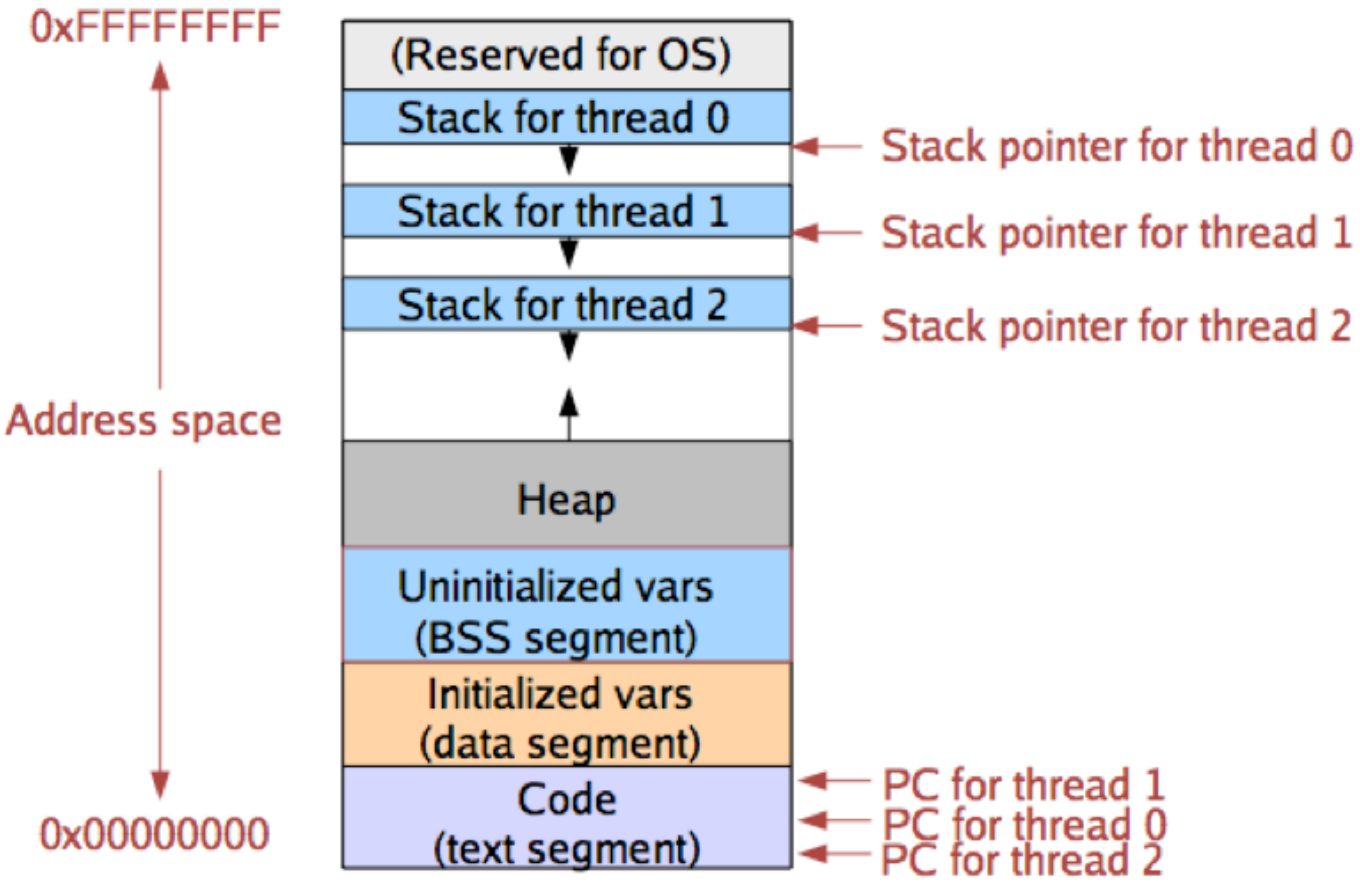


- Share code, data, heap via shared memory mechanisms
  - Make them part of the same address space
- Or let kernel or a user-library handle sharing of these parts of the address spaces and let the programmer deal only with synchronization issues
  - **Kernel vs. User threads**

(Old) Process Address Space



(New) Address Space with Threads



- All threads in a process share the same address space