# Processes and Threads

Prof. Van Renesse and Sirer

CS 4410

Cornell University

# Fun Starts Here!

- What involves “starting a program” or “running a program”?
  - which are misnomers…
- How can I run multiple processes on one computer?
- It’s all about “design and efficient implementation of abstractions”

# What is a Process?

- A process is an *abstraction* of a computer

# Abstractions

- A file is an abstract disk
- A socket is an abstract network endpoint
- A window is an abstract screen
- …

- Abstractions hide implementation details but expose (most of) the power

# Process Abstraction

STATE

ENVIRONMENT

ADDRESS SPACE

REGISTERS

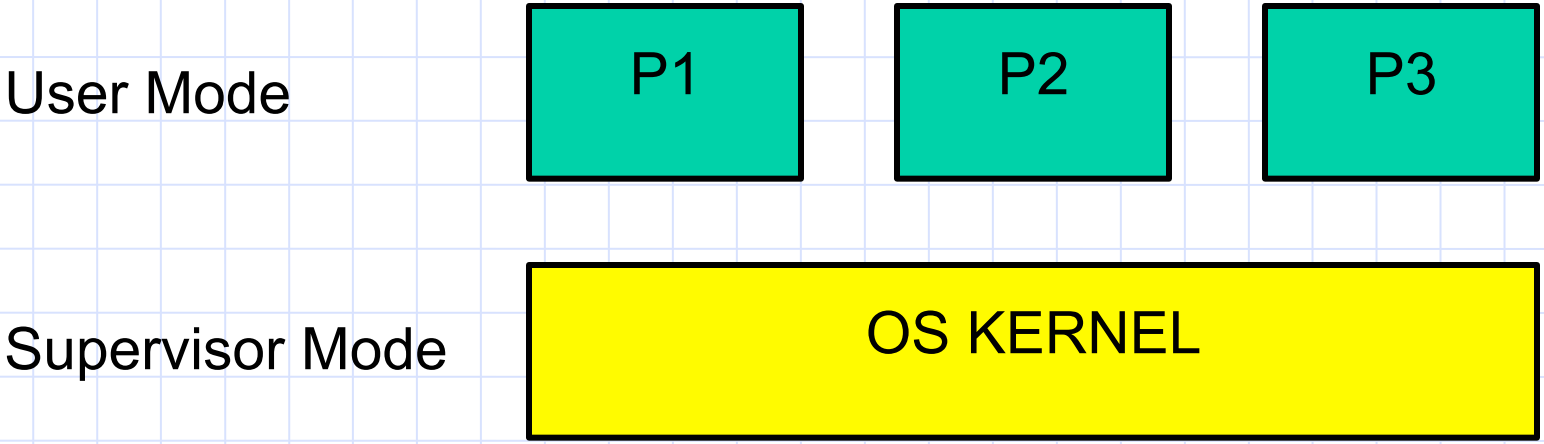
CONTROL

CPU

# Process Interface

- CreateProcess(initial state) → processID
- SendSignal(processID, Signal)
- GetStatus(processID) → runningStatus
- Terminate(processID)
- WaitFor(processID) → completionStatus
- ListProcesses() → [ pid1, pid2, … ]

# Kernel implements processes!




*Kernel is only part of the operating system*

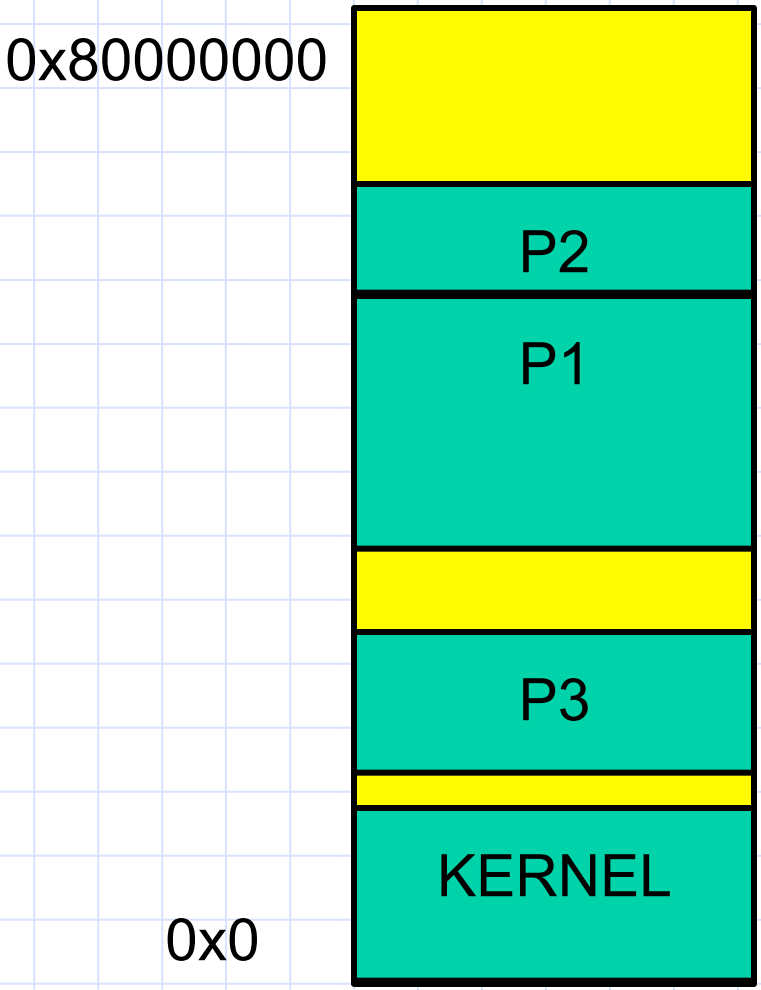
# Emulation…

- One option is for the hardware to simulate multiple instances of the same or other hardware
- Useful for debugging, emulation of ancient hardware, etc.
- But too inefficient for modern-day daily use

# CPU runs each process directly

- But somehow each process has its own
  - Registers
  - Memory
  - I/O resources
  - “thread of control”

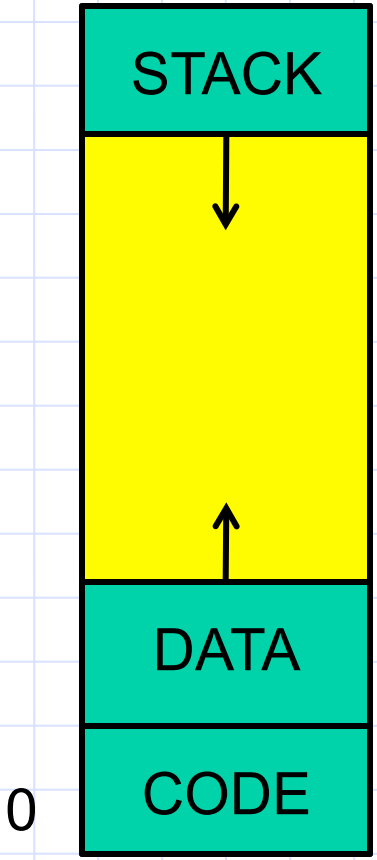
# (Simplified) RAM Layout




Base/Bound register

Supervisor mode

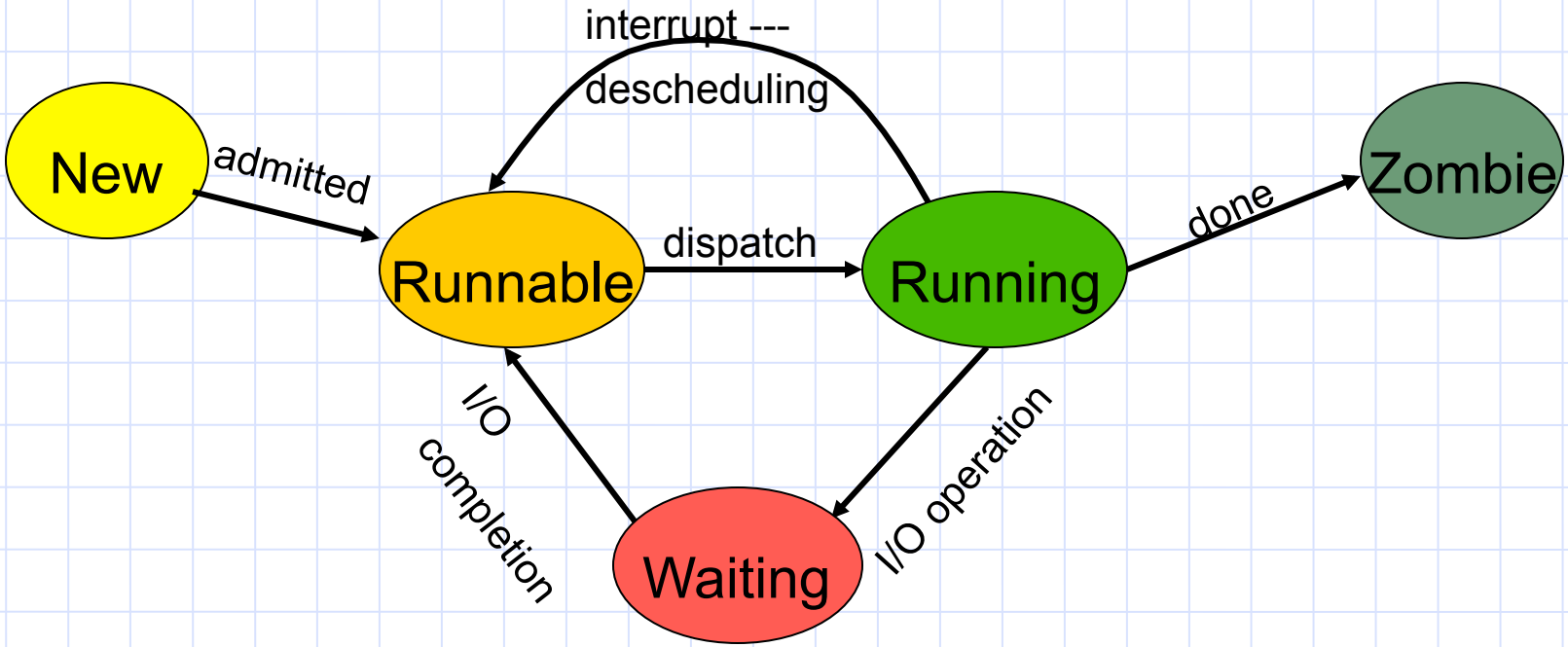
# Typical Address Space Layout
## (similar for kernel and processes)



# Process Control Block

- Process Identifier
- Process arguments (for identification)
- Process status (runnable, waiting, zombie, …)
- User Identifier (for security)
  - beware: superuser ≠ supervisor
- Registers
- Interrupt Vector
- Pending Interrupts
- Base / Bound
- Scheduling / accounting info
- I/O resources
- …

# Abstract life of a process



# createProcess(initial state)

- Allocate memory for address space
- Initialize address space
  - program vs fork
  - program ≠ process
- Allocate ProcessID
- Allocate process control block
- Put process control block on the *run queue*

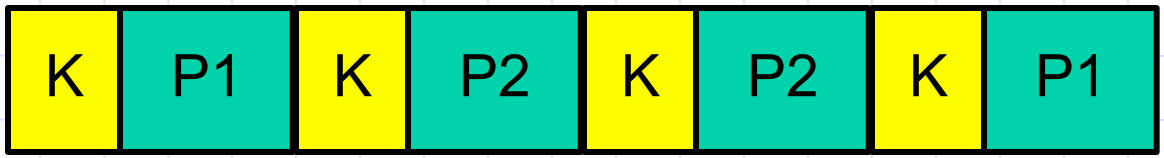
# How does a process terminate?

- External:
  - Terminate(ProcessID)
  - SendSignal(signal) with no handler set up
  - Using up quota
- Internal:
  - Exit(processStatus)
  - Executing an illegal instruction
  - Accessing illegal memory addresses

# For now: one process running at a time (single core machine)

- Kernel runs
- Switch to process 1
- Trap to kernel
- Switch to another (or same) process
- Trap to kernel
- etc.

*Context-switches*



# Processor Status Word

- Supervisor Bit or Level
- Interrupt Priority Level or Enabled Bit
- Condition Codes (result of compare ops)
- …

Supervisor can update any part, but user can only update condition codes

Has to be saved and restored like any other register!

# Time-Sharing

- Illusion: multiple processes run at same time
- Reality: only one process runs at a time
  - For no more than a few 10s of milliseconds
- But this can happen in parallel to another process waiting for I/O!
- Why time-share?

# Kernel Operation (conceptual)

- Initialize devices
- Initialize "First Process"
- For ever
  - while device interrupts pending
    - handle device interrupts
  - while system calls pending
    - handle system calls
  - if run queue is non-empty
    - select a runnable process and switch to it
  - otherwise
    - wait for device interrupt

# Invariants

- Supervisor mode → PC points to kernel code
- Equivalently: PC points to user code → user mode
- User code runs with interrupts enabled

- For simplicity: Kernel code runs with interrupts disabled (for now)

# Dispatch: kernel → process

- Software:
  - CurProc := &PCB of current process
  - Set user base/bound register
  - Restore process registers
  - Execute ReturnFromInterrupt instruction
- Hardware:
  - Sets user mode
  - Enables interrupts
  - Restores program counter

# Trap process → kernel

- Hardware:
  - Disables interrupts
  - Sets supervisor mode
  - Saves user PC and SP on *kernel* stack
    - why not on process stack?
  - Sets kernel stack pointer
  - Sets PC to kernel-configured position
- Software:
  - Save process registers in PCB of CurProc
  - Back to kernel main loop

# Causes for traps

- Clock interrupt
- Device interrupt
- System call
- Privileged instruction
- Divide by zero
- Bad memory access
- …

# System calls

- How does a process specify what system call to invoke and what parameters to use?
- How does the kernel protect itself and other processes?
- How does the kernel return a result to the process?
- How does the kernel prevent accidentally returning privacy sensitive data?

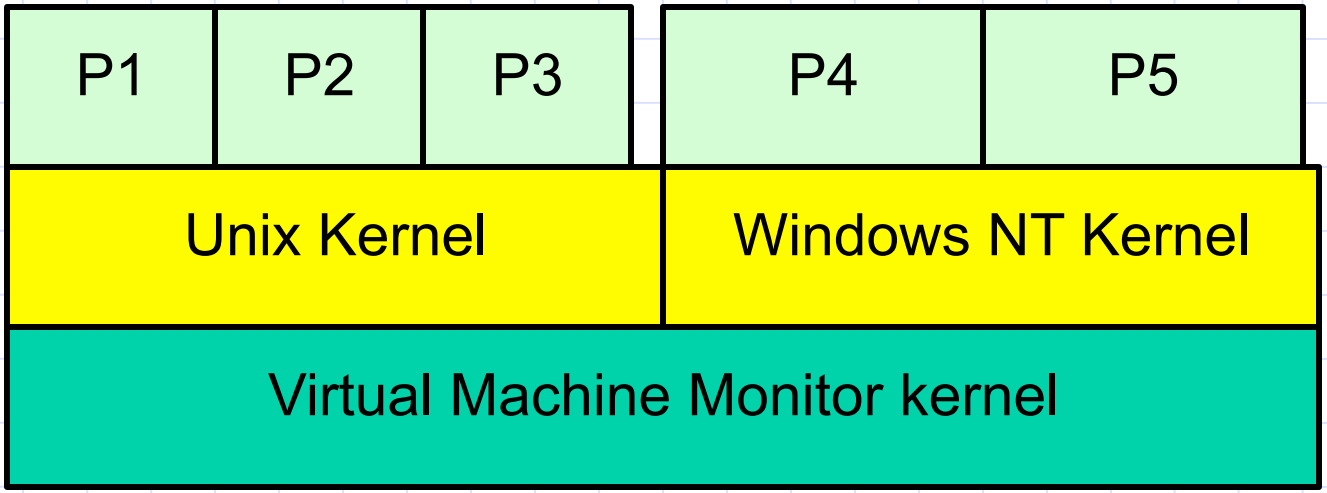
# Class Projects

- Implement sleep(delay) system call
- Implement a debugger
- Implement SendSignal(pid, signal)

# How Much To Abstract

- Unix and Windows provide processes that look like idealized machines, with nice looking file abstractions, network abstractions, graphical windows, etc.
- Xen, KVM, etc. provide processes that look just like real hardware
  - virtualization
- Requires different kinds of things from kernels
  - Unix/Windows: implement files, network protocols, window management
  - Xen/KVM/…: emulate hardware

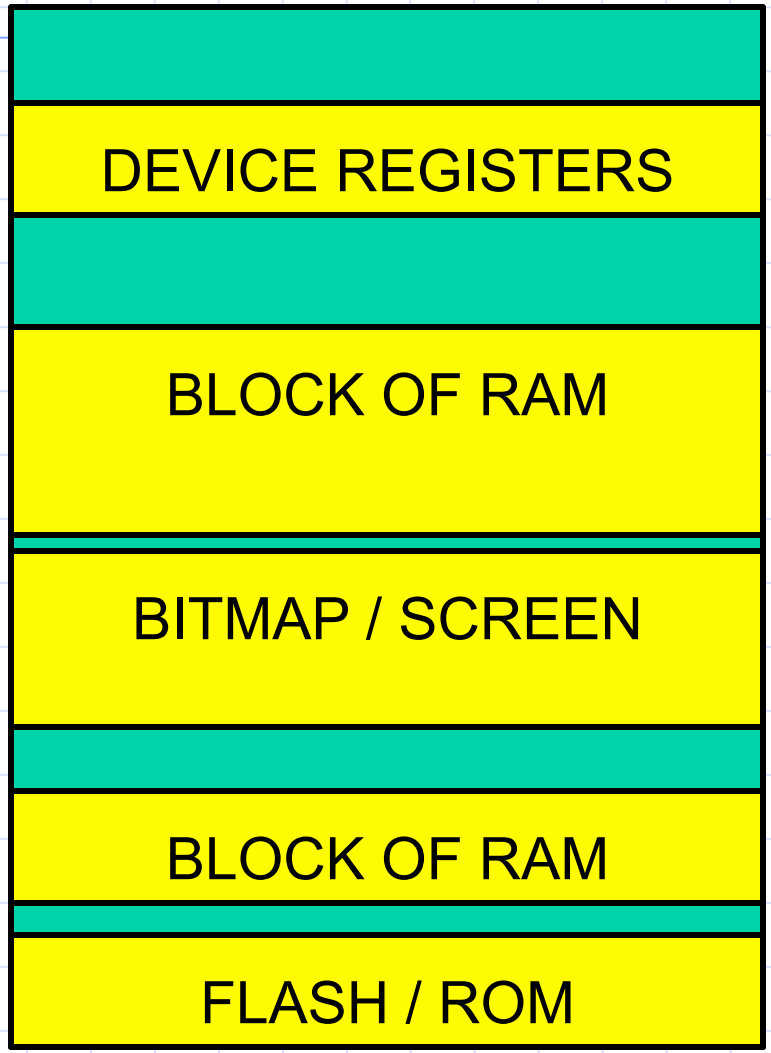
# Virtual Machine Abstraction



# Things to emulate

- Supervisor mode
- Base/Bound registers
- Device registers
- …

- Hardware can help
  - Multi-level supervisor
  - Multi-level base/bound
  - …



# Processes Under Unix/Linux

- Fork() system call to create a new process
  - Old process called parent, new process called child
- int fork() *clones* the invoking process:
  - Allocates a new PCB and process ID
  - Allocates a new address space
  - copies the parent's address space into the child's
  - in parent, fork() returns PID of child
  - in child, fork() returns a zero.
- int fork() returns TWICE!

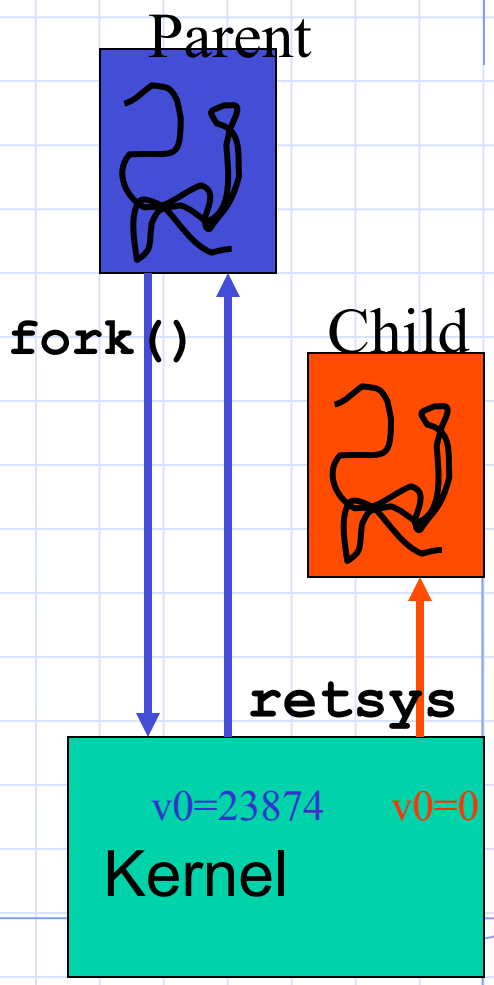
# Example

```
int main(int argc, char **argv)
{   int parentPid = getpid();
    int pid = fork();
    if (pid == 0) {
        printf("The child of %d is %d\n", parentPid, getpid());
        exit(0);
    } else {
        printf("My child is %d\n", pid);
        exit(0);
    }
}
```

What does this program print?

# Bizarre But Real

```
$ cc a.c
$ ./a.out
The child of 23873 is 23874
My child is 23874
```



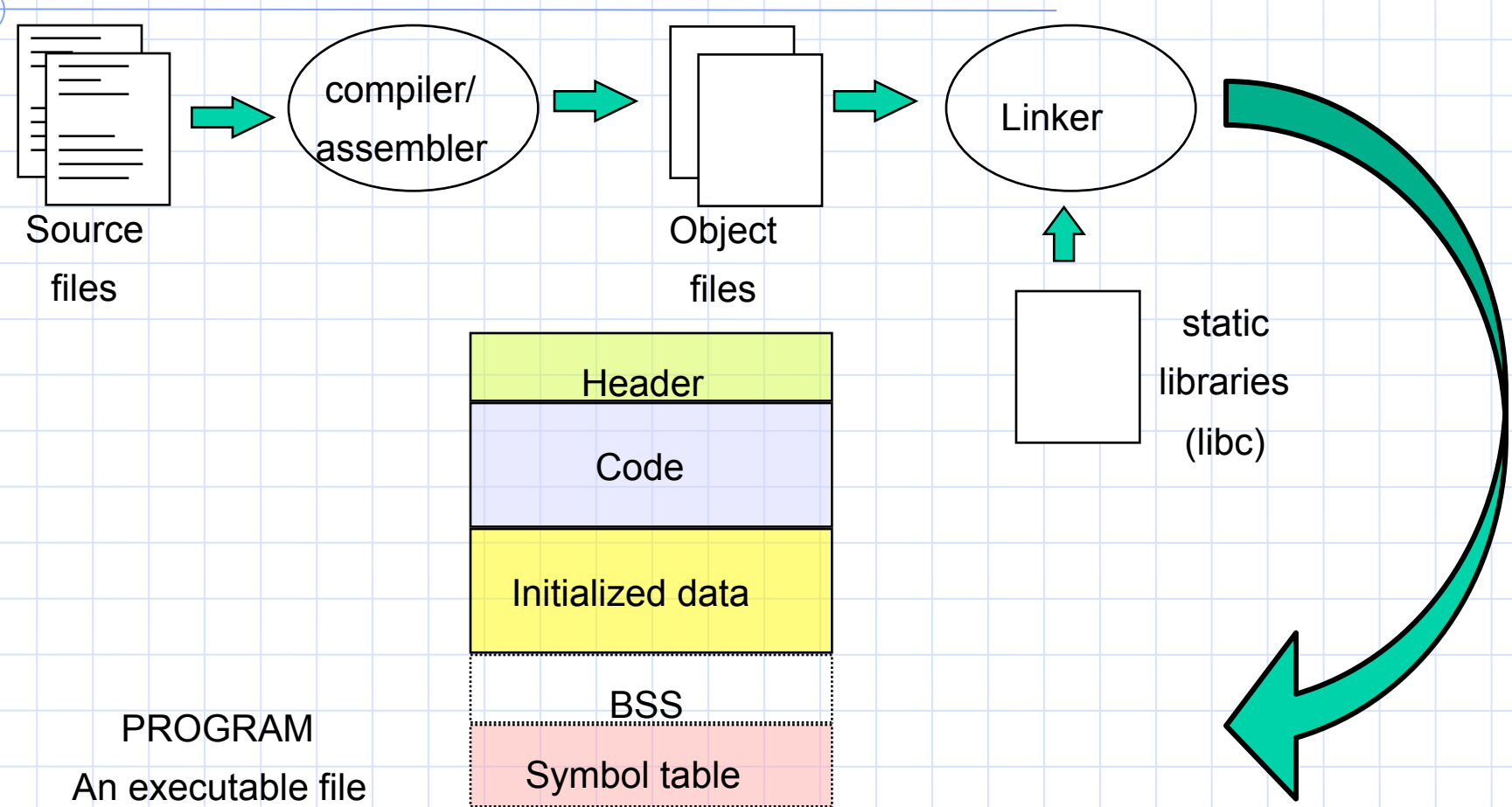
# Exec()

- Fork() gets us a new address space
- int exec(char *programName) completes the picture
  - throws away the contents of the calling address space
  - replaces it with the program in file named by programName
  - starts executing at header.startPC
  - PCB remains the same otherwise (same PID)
- Pros: Clean, simple
- Con: duplicate operations

# What is a program?

- A program is a file containing executable code (machine instructions) and data (information manipulated by these instructions) that together describe a computation
- Resides on disk
- Obtained through compilation and linking

# Preparing a Program

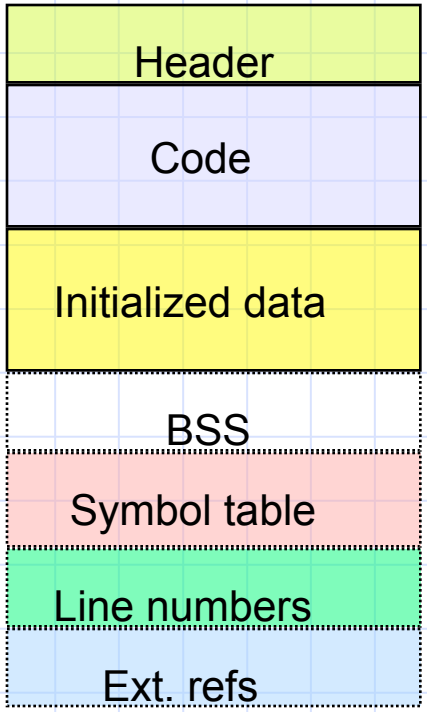



PROGRAM
An executable file
in a standard format,
such as ELF on Linux,
Microsoft PE on Windows

# Running a program

- Every OS provides a “loader” that is capable of converting a given program into an executing instance, a process
  - A program in execution is called a process
- The loader:
  - reads and interprets the executable file
  - Allocates memory for the new process and sets process’s memory to contain code & data from executable
  - pushes “argc”, “argv”, “envp” on the stack
  - sets the CPU registers properly & jumps to the entry point

# Process != Program




**Executable**

Program is passive

- Code + data

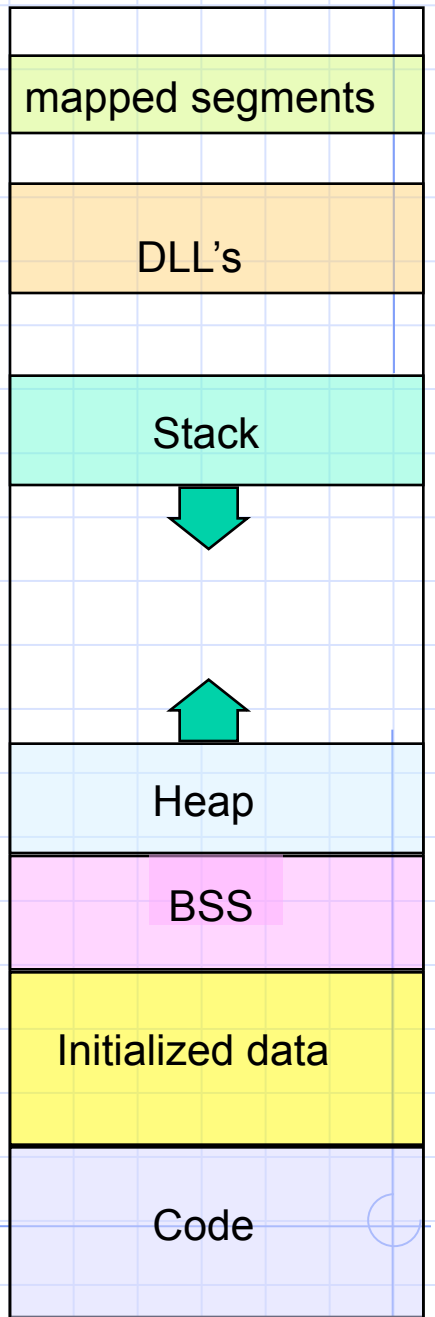
Process is running program

- stack, regs, program counter

Example:

We both run IE:

- Same program
- Separate processes




**Process address space**

# Process Termination, part 1

- Process executes last statement and calls **exit** syscall
  - Process' resources are deallocated by operating system
- Parent may terminate execution of child process (**kill**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some OSes don't allow child to continue if parent terminates
      - All children terminated - *cascading termination*

# Process Termination, part 2

- Process first goes into “zombie state”
- Parent can wait for zombie children
  - Syscall: wait() → (pid, exit status)
- After wait() returns, PCB of child is garbage collected

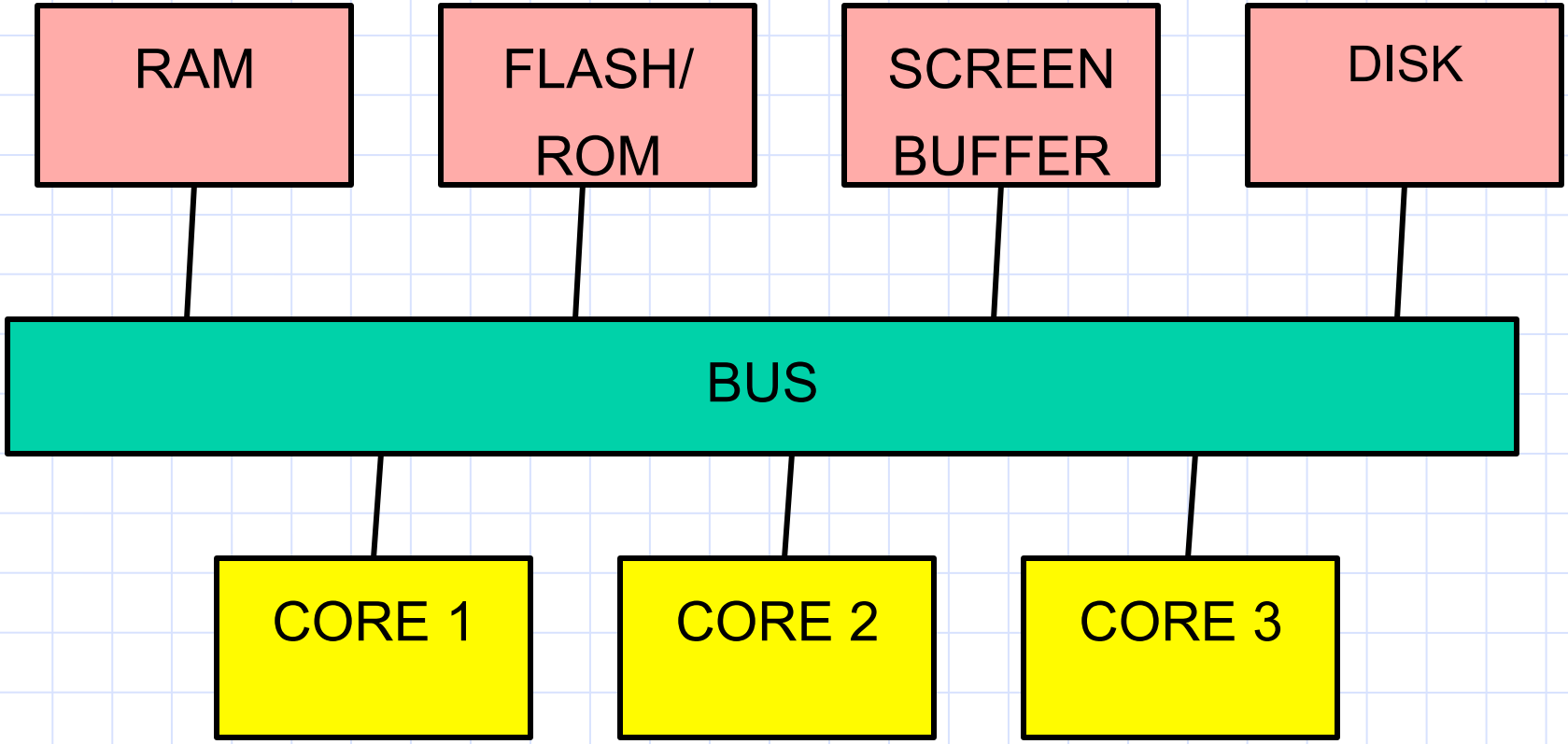
# Class Project

- Write a simple command line interpreter

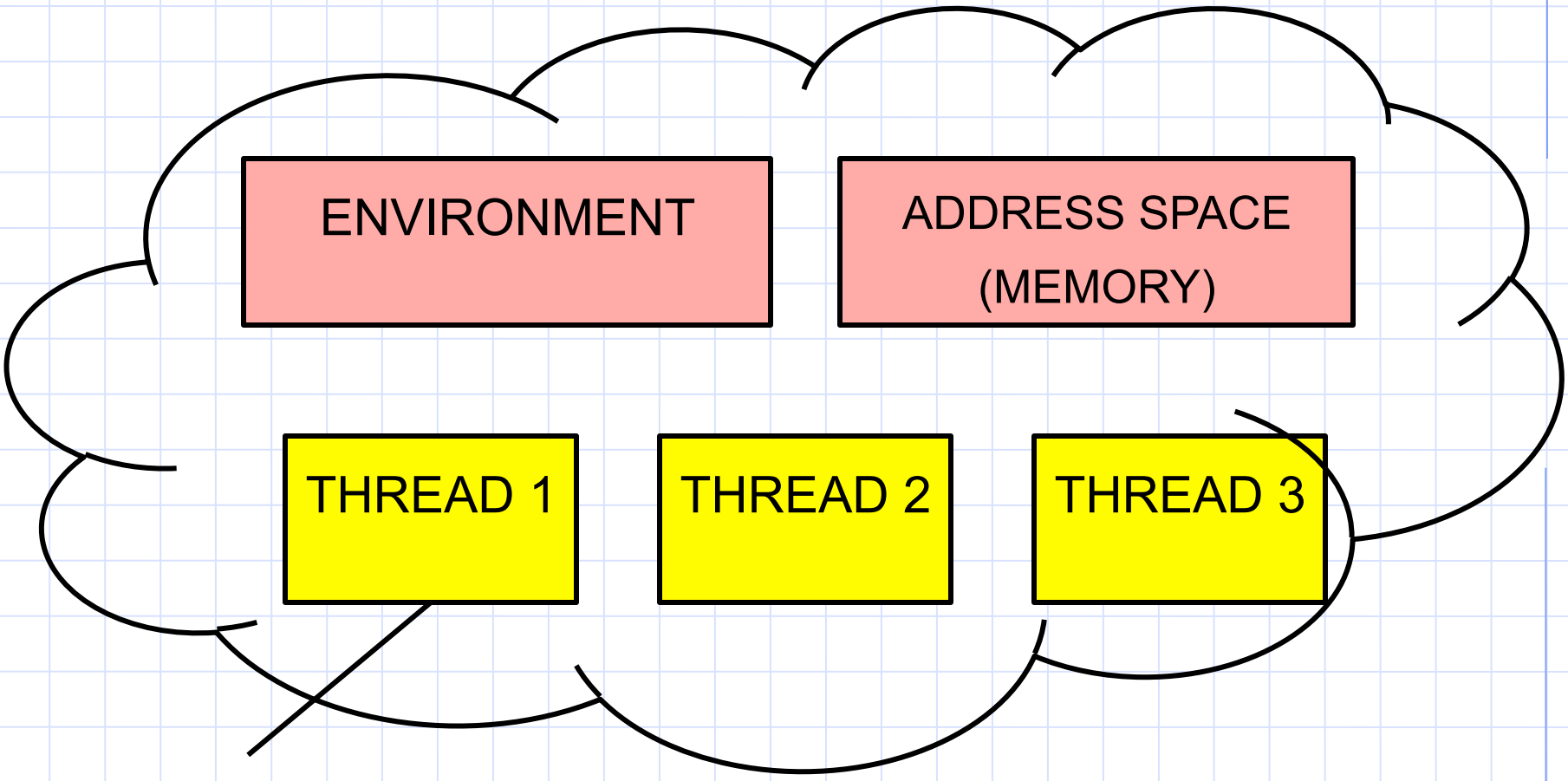
# Multiple Cores

- Modern computers often have several if not dozens of cores
- Each core has its own registers, but cores share memory and devices
- Cores can run user processes in parallel
- *Cores need to synchronize access to PCBs and devices*

# Multi-Core Architecture



# Abstraction… multi-threaded process



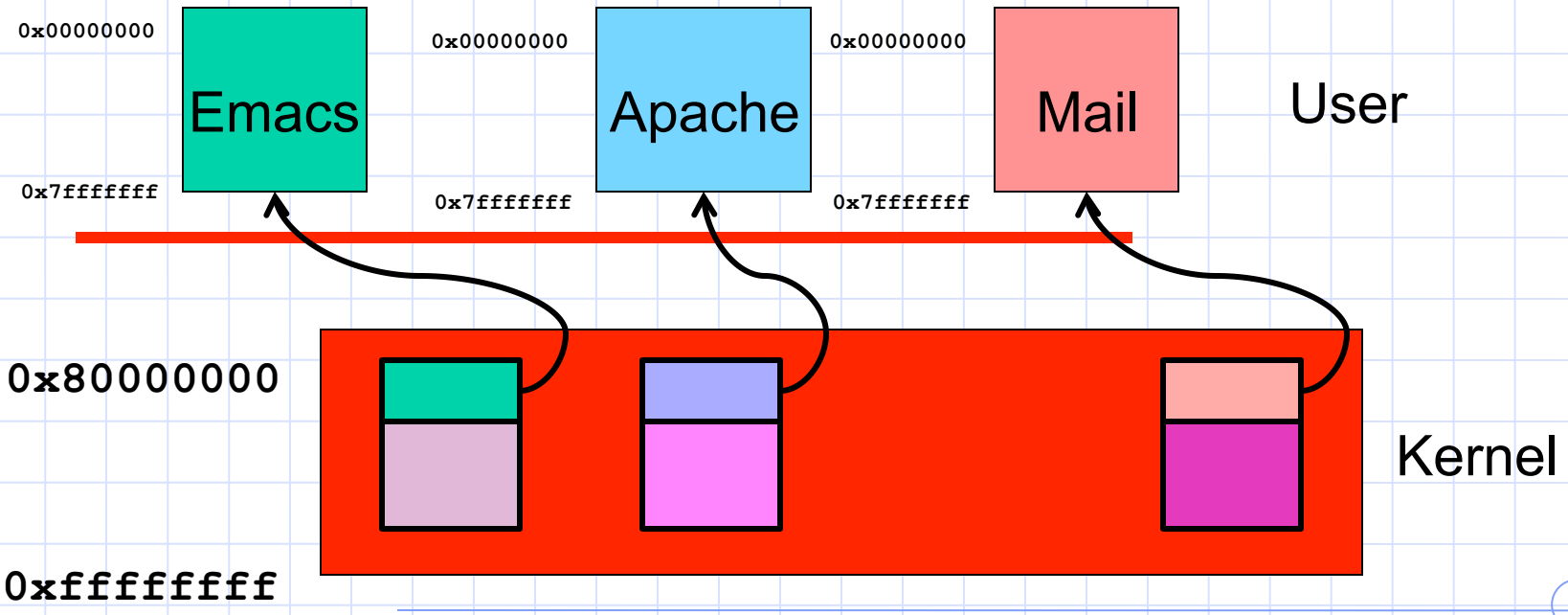

CPU + registers

# Why?

- Make it simpler and more efficient for a process to take advantage of multicore machines
  - Instead of starting multiple processes, each with its own address space and a single thread running on a single core
- Not just for CPU parallelism: I/O parallelism can be achieved even if I/O operations are “blocking”
- Program structuring: for example, servers dealing with concurrent incoming events
- Might well have more threads than cores!!

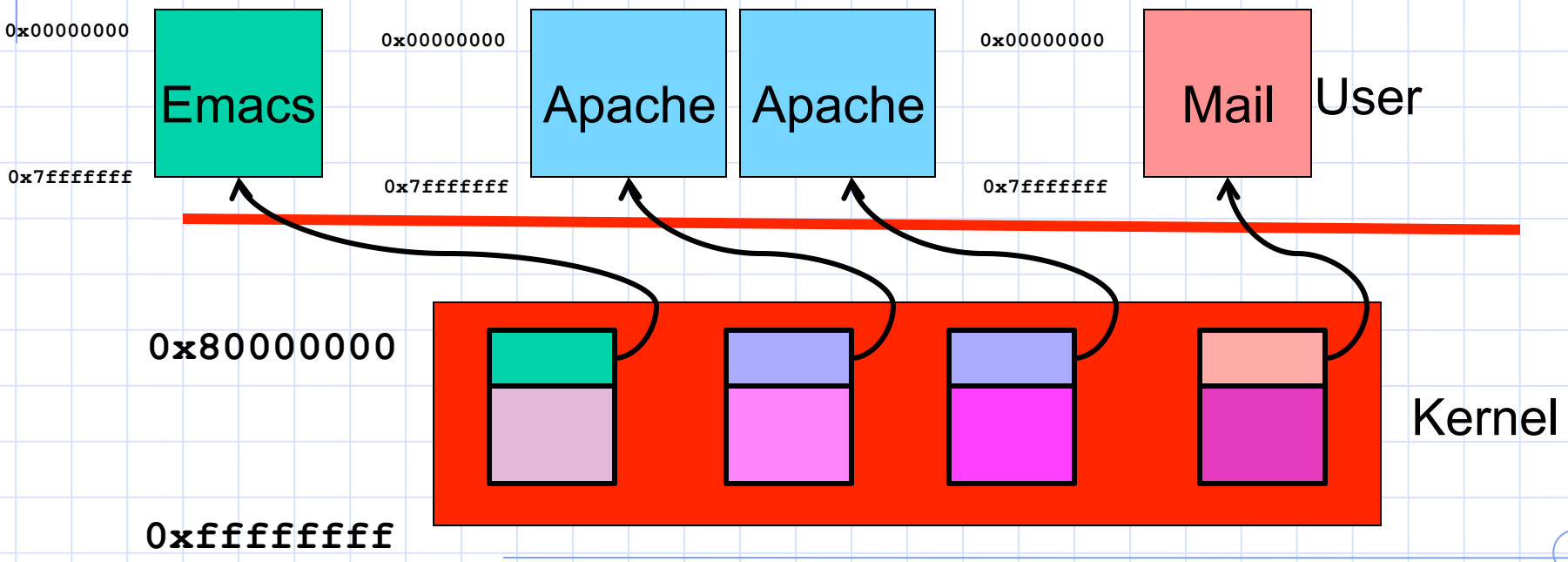
# Processes and Address Spaces

- What happens when Apache wants to run multiple concurrent computations ?



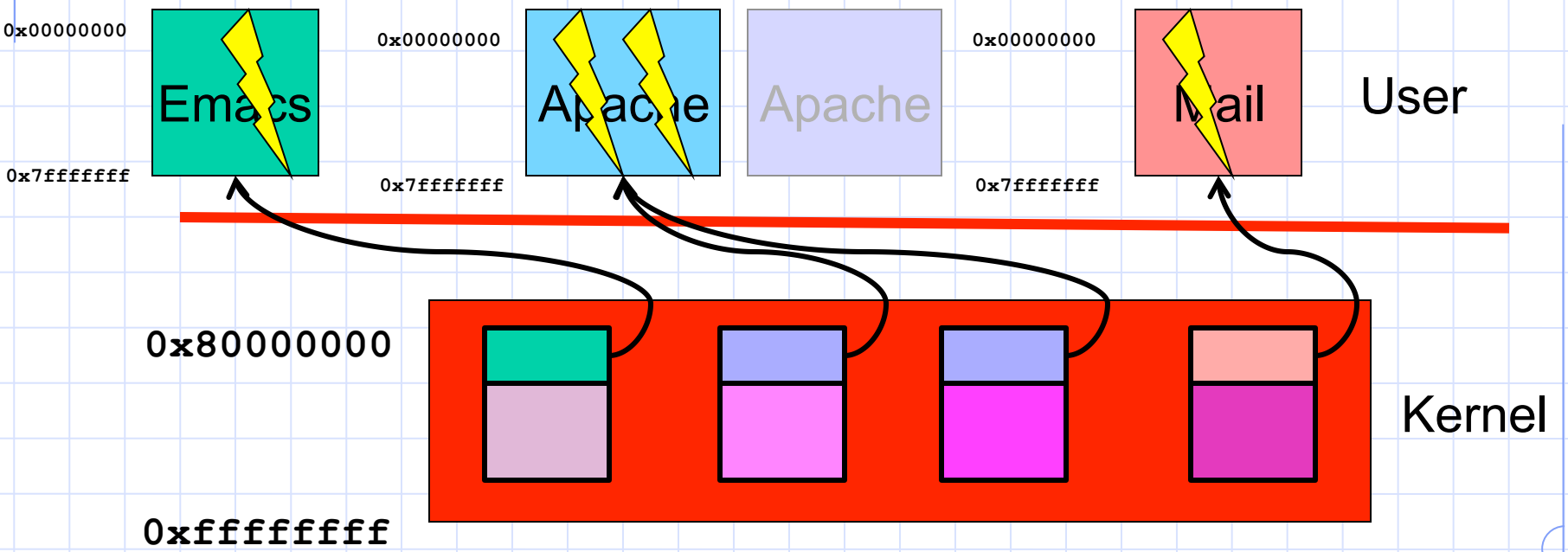
# Processes and Address Spaces

- Two heavyweight address spaces for two concurrent computations ?



# Processes and Address Spaces

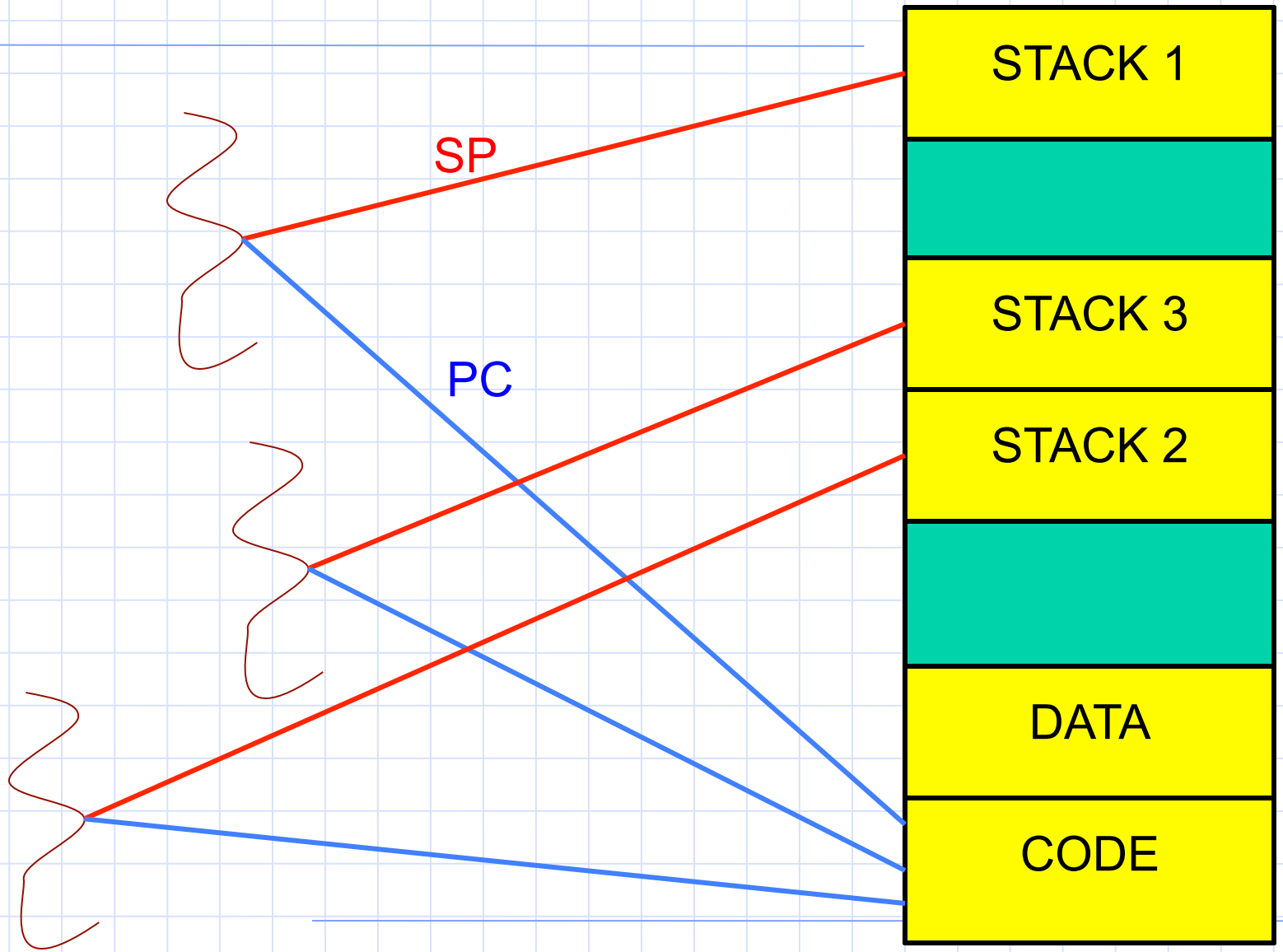
- We can eliminate duplicate address spaces and place concurrent computations in the same address space



# Architecture

- Process consists of
  - One address space containing chunks of memory
  - Shared I/O resources
  - Multiple threads
    - Each with its own registers, in particular PC and SP
    - Each has its own stack in the address space
    - Code and data is shared
- **Other terms for threads**
  - **Lightweight Process**
  - **Thread of Control**
  - **Task**

# Memory Layout



# Sharing

- **What's shared between threads?**
  - **They all share the same code and data (address space)**
  - **they all share the same privileges**
  - **they share almost everything in the process**
- **What don't they share?**
  - **Each has its own PC, registers, stack pointer, and stack**

# Threads

- Lighter weight than processes
- Threads need to be mutually trusting
  - Why?
- Ideal for programs that want to support concurrent computations where lots of code and data are shared between computations
  - Servers, GUI code, …

# Separation of Thread and Process concepts

- **Concurrency (multi-threading) is useful for:**
  - **improving program structure**
  - **handling concurrent events (e.g., web requests)**
  - **building parallel programs**
- **So, multi-threading is useful even on a uniprocessor**
- **To be useful, thread operations have to be fast**

# How to implement?

- Two extreme solutions
  - "Kernel threads":
    - Allocate a separate PCB for each thread
    - Assign each PCB the same base/size registers
    - Also copy I/O resources, etc.
  - "User threads":
    - Built a miniature O.S. in user space
- User threads are (generally) more efficient
  - Why?
- Kernel threads simplify system call handling and scheduling
  - Why?

# User Thread Implementation

- User process supports
  - "Thread Control Block" table with one entry per thread
  - "context switch" operations that save/restore thread state in TCB
    - Much like kernel-level context switches
  - yield() operation by which a thread releases its core and allows another thread to use it
    - Automatic pre-emption not always supported
  - Thread scheduler

# System calls

- With user threads, a process may have multiple systems calls outstanding simultaneously (one per thread)
- Kernel PCB must support this

# Things to Think about

- Scheduling
  - While runnable process / thread runs when?
- Coordination
  - How do cores / threads synchronize access to shared memory and devices?