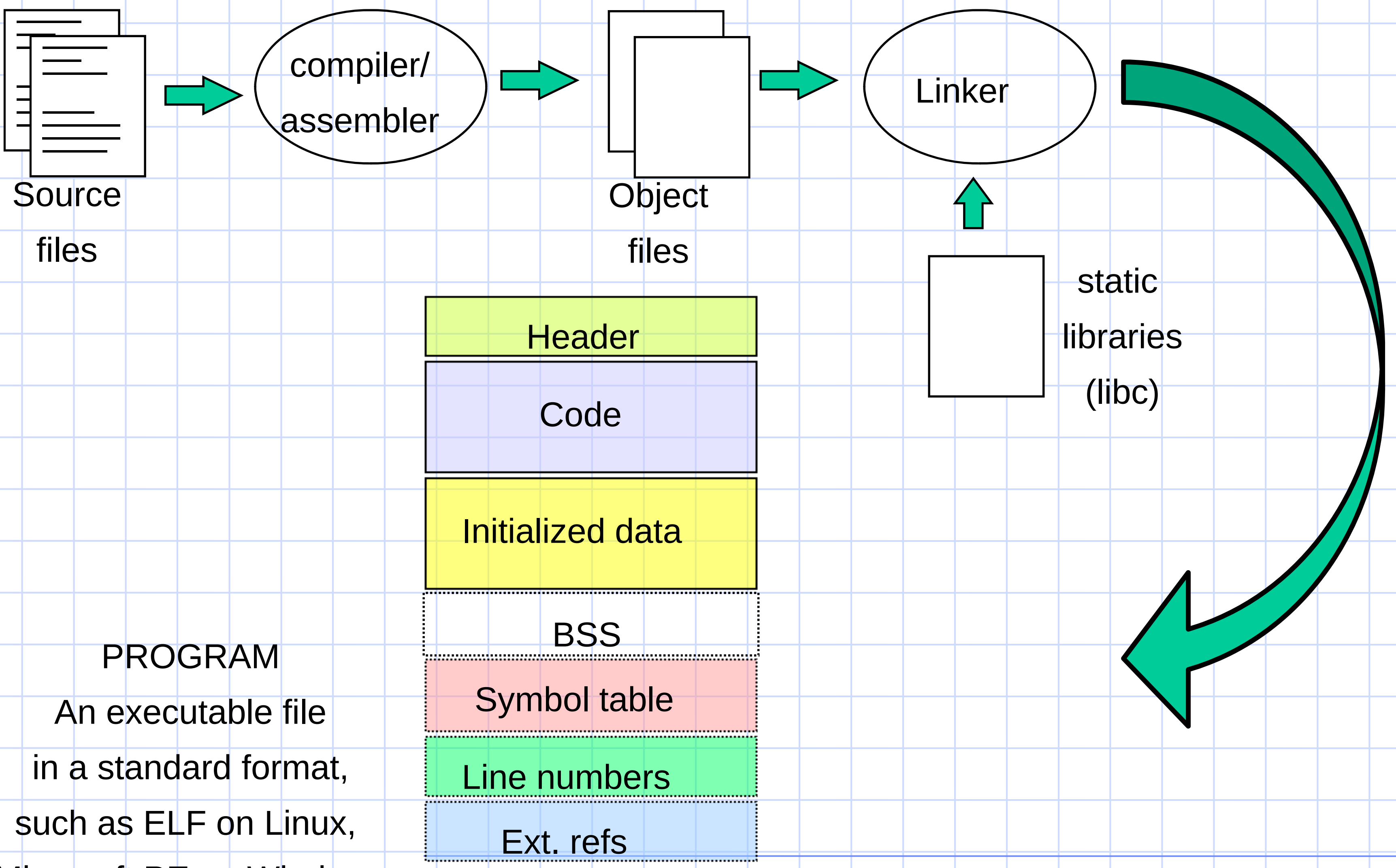
Processes and Threads

Prof. Sirer
CS 4410
Cornell University

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

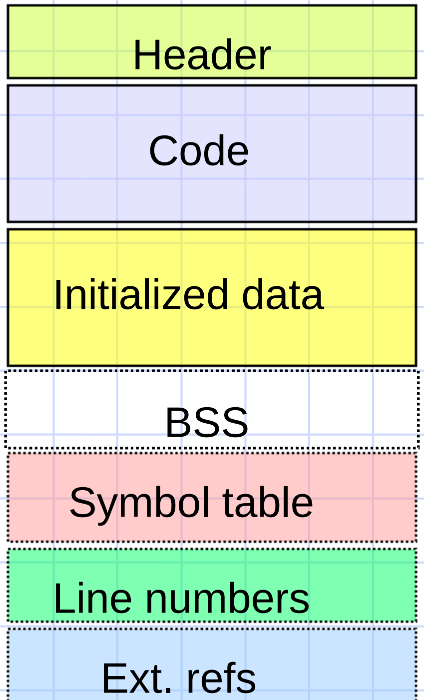


PROGRAM
An executable file
in a standard format,
such as ELF on Linux,
Microsoft PE on Windows

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

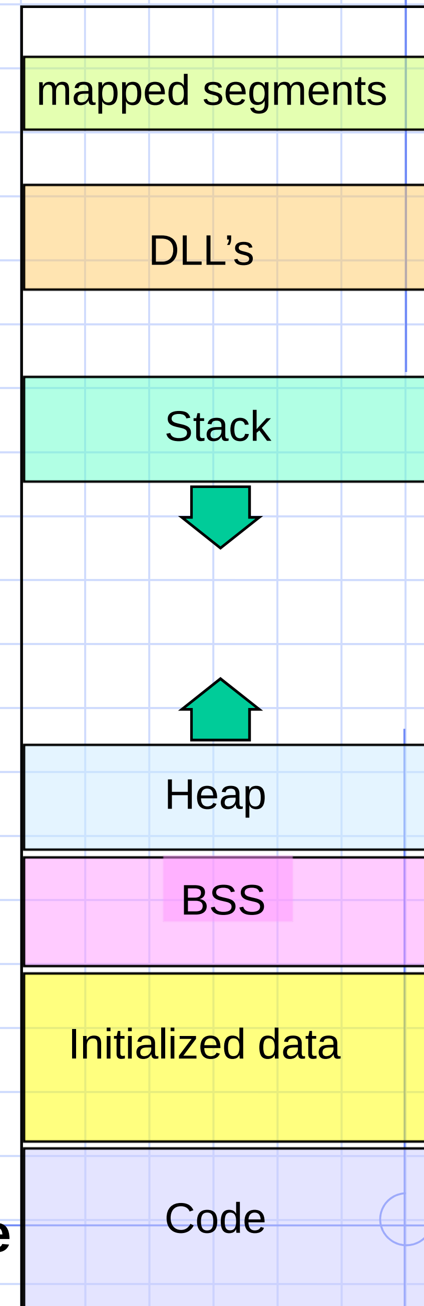
Process is running program

- stack, regs, program counter

Example:

We both run IE:

- Same program
- Separate processes



Process address space

Process Management

- Process management deals with several issues:
  - what are the units of execution
  - how are those units of execution represented in the OS
  - how is work scheduled in the CPU
  - what are possible execution states, and how does the system move from one to another

The Process

- A process is the basic unit of execution
  - it’s the unit of scheduling
  - it’s the dynamic (active) execution context (as opposed to a program, which is static)
- A process is sometimes called a *job* or a *task* or a *sequential process.*
- A sequential process is a program in execution; it defines the sequential, instruction-at-a-time execution of a program.

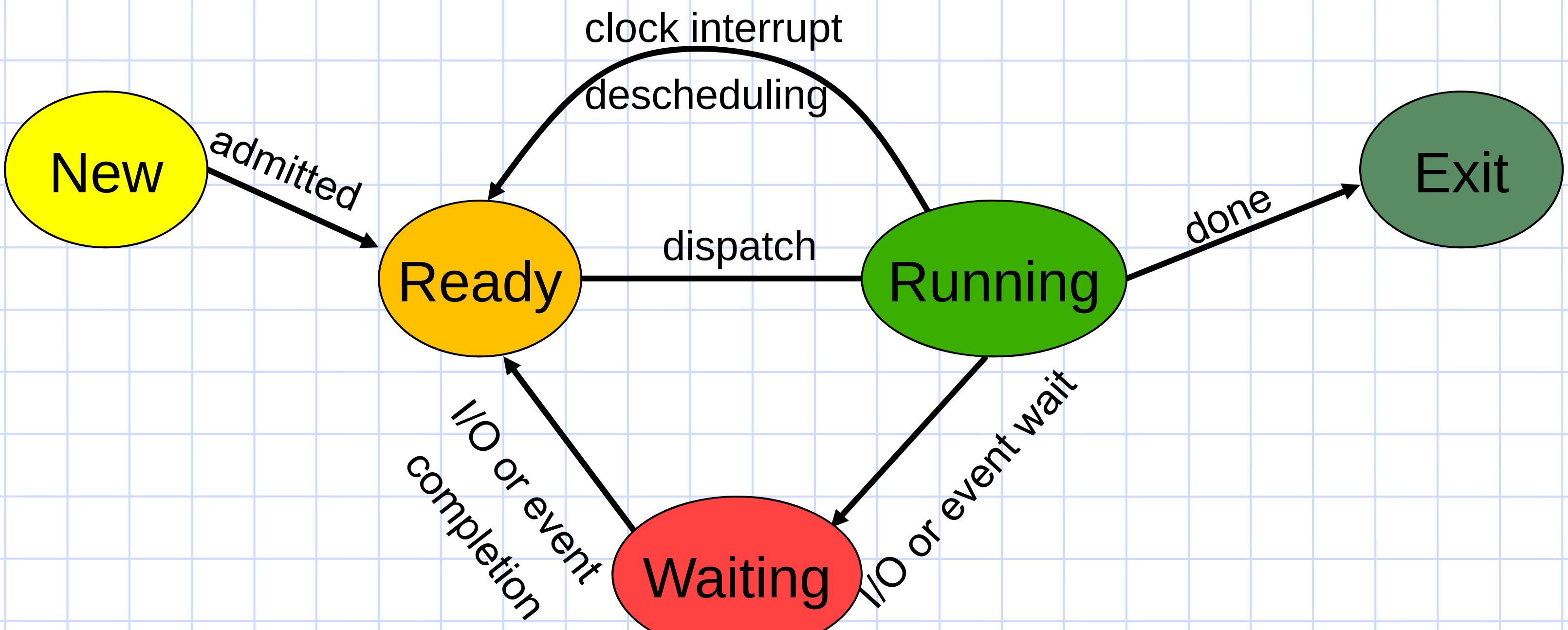
What's in a Process?

- A process consists of at least:
  - the code for the running program
  - the data for the running program
  - an execution stack tracing the state of procedure calls made
  - the Program Counter, indicating the next instruction
  - a set of general-purpose registers with current values
  - a set of operating system resources (open files, connections to other programs, etc.)
- The process contains all the state for a program in execution.

Process State

- There may be several processes running the same program (e.g. multiple web browsers), but each is a distinct process with its own representation.
- Each process has an *execution state* that indicates what it is currently doing, e.g.,:
  - ready: waiting to be assigned to the CPU
  - running: executing instructions on the CPU
  - waiting: waiting for an event, e.g., I/O completion
- As a program executes, it moves from state to state

Process State Transitions



Processes hop across states as a result of:

- Actions they perform, e.g. system calls
- Actions performed by OS, e.g. rescheduling
- External actions, e.g. I/O

Process Data Structures

- At any time, there are many processes in the system, each in its particular state.
- The OS must have data structures representing each process: this data structure is called the PCB:
  - *Process Control Block*
- The PCB contains all of the info about a process.
- The PCB is where the OS keeps all of a process' hardware execution state (PC, SP, registers) when the process is not running.

PCB

The PCB contains the entire state of the process

PCB
Process state
Process number
Program counter
Stack pointer
General-purpose registers
Memory management info
Username of owner
Scheduling information
Accounting info

Time Multiplexing (PCBs and Hardware State)

- When a process is running its Program Counter, stack pointer, registers, etc., are loaded on the CPU (I.e., the processor hardware registers contain the current values)
- When the OS stops running a process, it saves the current values of those registers into the PCB for that process.
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process' PCB.
- The process of switching the CPU from one process to another is called a *context switch*. Timesharing systems may do 1000s of context switches a second!

Context Switch

- For a running process
  - All registers are loaded in CPU and modified
    - E.g. Program Counter, Stack Pointer, General Purpose Registers
- When process relinquishes the CPU, the OS
  - Saves register values to the PCB of that process
- To execute another process, the OS
  - Loads register values from PCB of that process
- **Context Switch**
  - Process of switching CPU from one process to another
  - Very machine dependent for types of registers

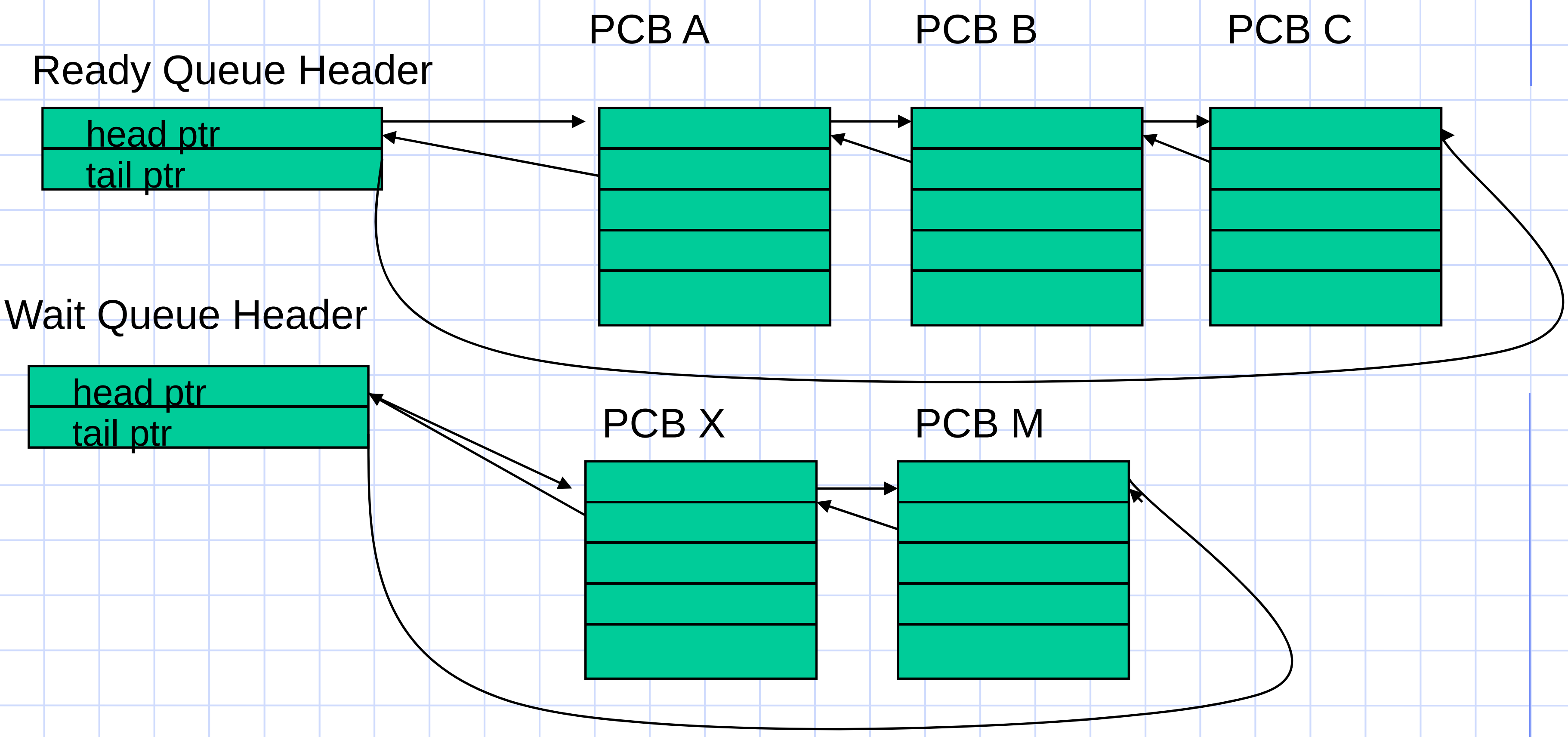
Details of Context Switching

- Very tricky to implement
  - OS must save state without changing state
  - Should run without touching any registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
      - Or way to save a register and then continue
- Overheads: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Wait for pipeline to drain in pipelined processors

State Queues

- The OS maintains a collection of queues that represent the state of all processes in the system
- There is typically one queue for each state, e.g., ready, waiting for I/O, etc.
- Each PCB is queued onto a state queue according to its current state.
  - As a process changes state, its PCB is unlinked from one queue and linked onto another.

State Queues



There may be many wait queues, one for each type of wait (specific device, timer, message,…).

PCBs and State Queues

- PCBs are data structures, dynamically allocated in OS memory.
- When a process is created, a PCB is allocated to it, initialized, and placed on the correct queue.
- As the process computes, its PCB moves from queue to queue.
- When the process is terminated, its PCB is deallocated.

Processes Under UNIX

- Fork() system call to create a new process
- int fork() does many things at once:
  - creates a new address space (called the child)
  - copies the parent's address space into the child's
  - starts a new thread of control in the child's address space
  - parent and child are equivalent -- almost
    - in parent, fork() returns a non-zero integer
    - in child, fork() returns a zero.
    - difference allows parent and child to distinguish
- int fork() returns TWICE!

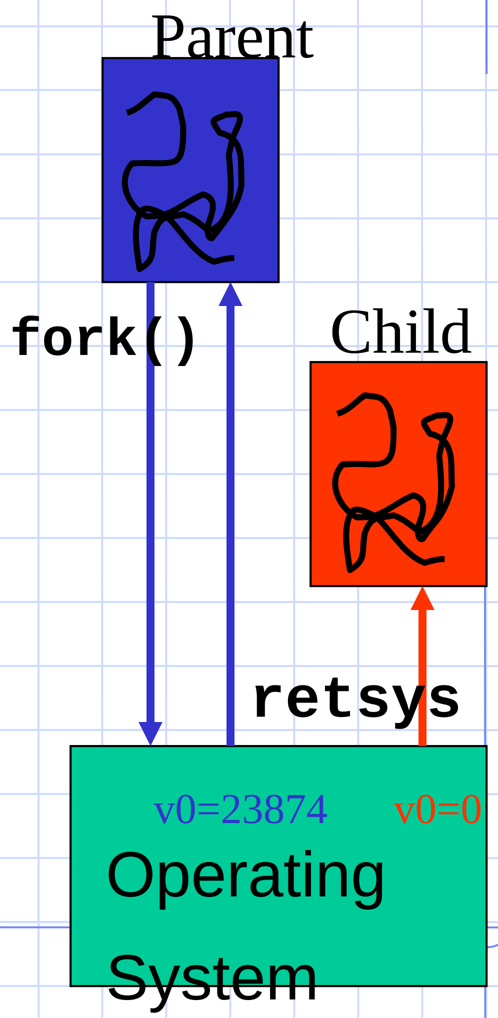
Example

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      exit(0);
   } else {
      printf("My child is %d\n", cpid);
      exit(0);
   }
}
```

What does this program print?

Bizarre But Real

```
lace:tmp<15> cc a.c
lace:tmp<16> ./a.out foobar
The child of foobar is 23874
My child is 23874
```



Exec()

- Fork() gets us a new address space,
  - but parent and child share EVERYTHING
    - memory, operating system state

- int exec(char *programName) completes the picture
  - throws away the contents of the calling address space
  - replaces it with the program named by programName
  - starts executing at header.startPC
  - Does not return

- Pros: Clean, simple
- Con: duplicate operations

Process Termination

- 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*
- In either case, resources named in the PCB are freed, and PCB is deallocated

Processes and Threads

- A full process includes numerous things:
  - an address space (defining all the code and data pages)
  - OS resources and accounting information
  - a “thread of control”, which defines where the process is currently executing (basically, the PC and registers)
- Creating a new process is costly, because of all of the structures (e.g., page tables) that must be allocated
- Communicating between processes is costly, because most communication goes through the OS

Parallel Programs

- **Suppose I want to build a parallel program to execute on a multiprocessor, or a web server to handle multiple simultaneous web requests. I need to:**
  - **create several processes that can execute in parallel**
  - **cause *each* to map to the *same* address space (because they're part of the same computation)**
  - **give each its starting address and initial parameters**
  - **the OS will then schedule these processes, in parallel, on the various processors**
- **Notice that there's a lot of cost in creating these processes and possibly coordinating them. There's also a lot of duplication, because they all share the same address space, protection, etc……**

"Lightweight" Processes

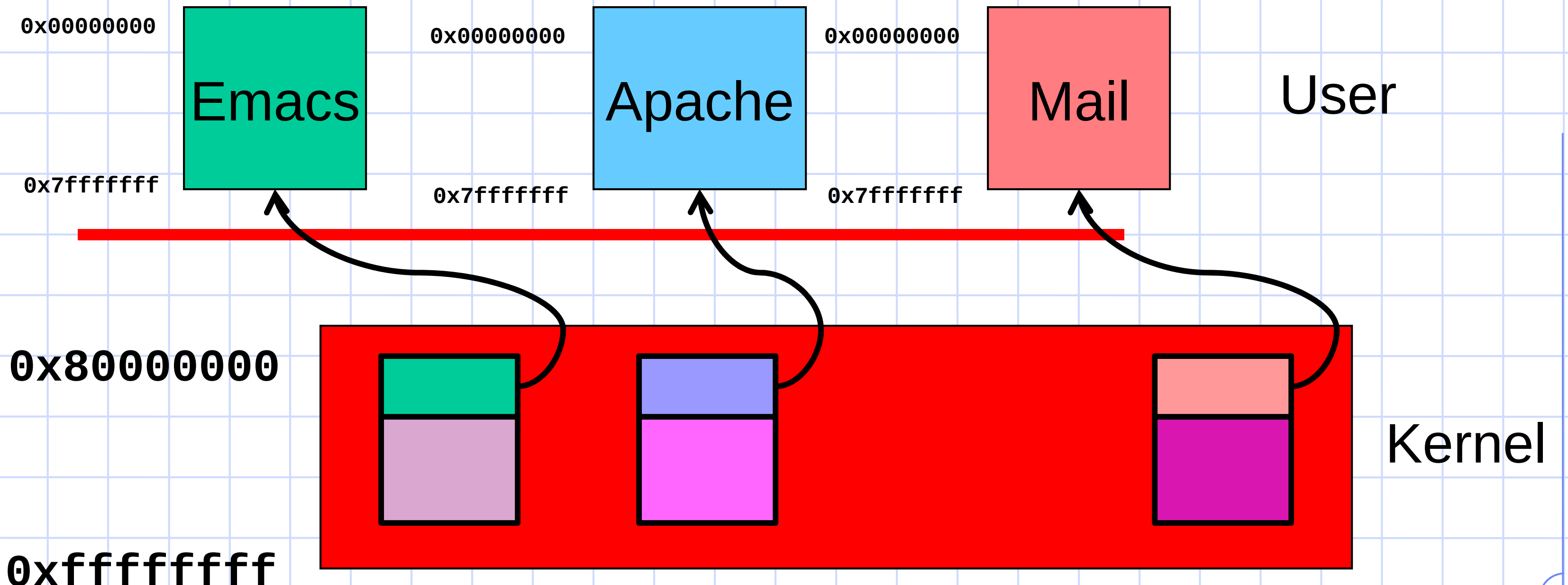
- **What's shared between these processes?**
  - **They all share the same code and data (address space)**
  - **they all share the same privileges**
  - **they share almost everything in the process**
- **What don't they share?**
  - **Each has its own PC, registers, and stack pointer**
- **Idea: why don't we separate the idea of process (address space, accounting, etc.) from that of the minimal "thread of control" (PC, SP, registers)?**

Threads and Processes

- **Modern operating systems therefore support two entities:**
  - **the <u>process</u>, which defines the <u>address space</u> and general process attributes**
  - **the <u>thread</u>, which defines a sequential execution stream within a process**
- **A thread is bound to a single process. For each process, however, there may be many threads.**
- **Threads are the unit of scheduling; processes are *containers* in which threads execute.**

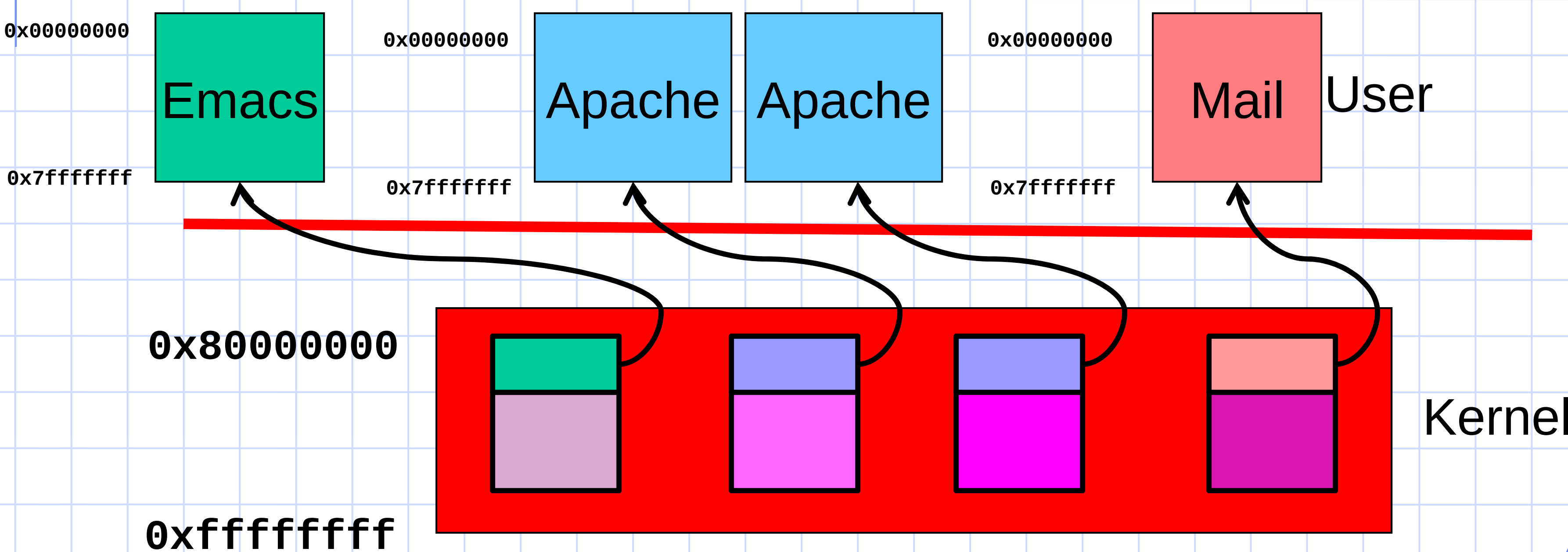
Processes and Address Spaces

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



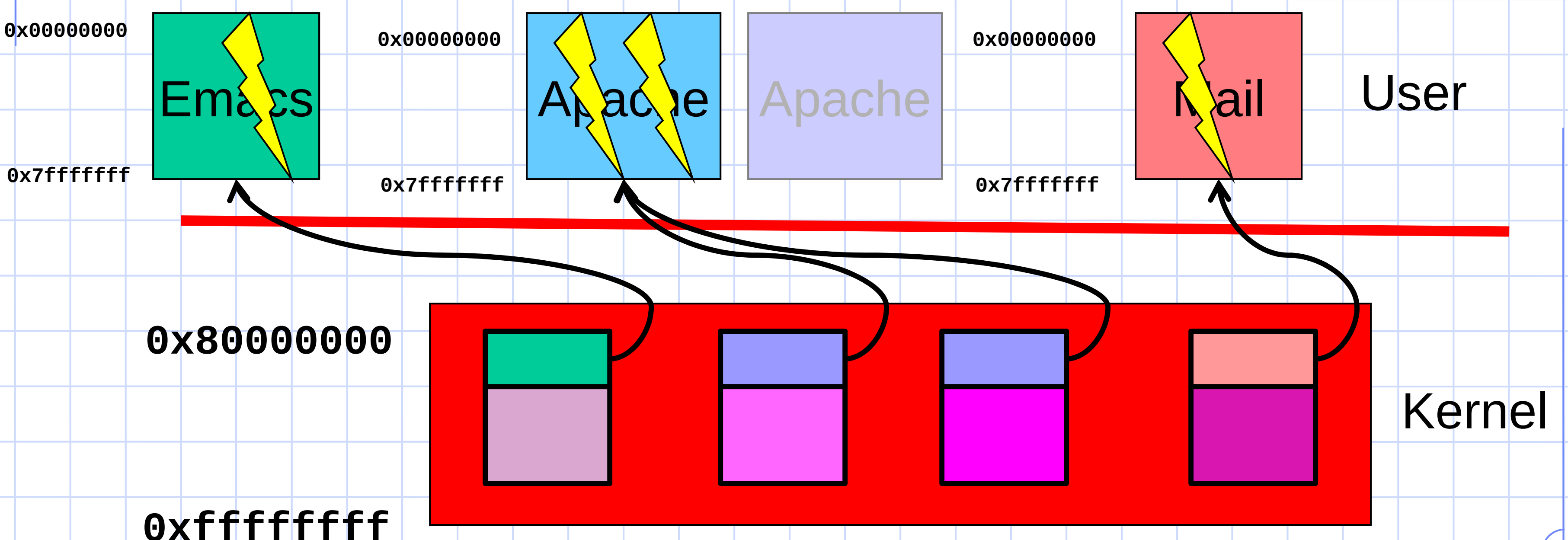
Processes and Address Spaces

- Two heavyweight address spaces for two concurrent computations ?



Processes and Address Spaces

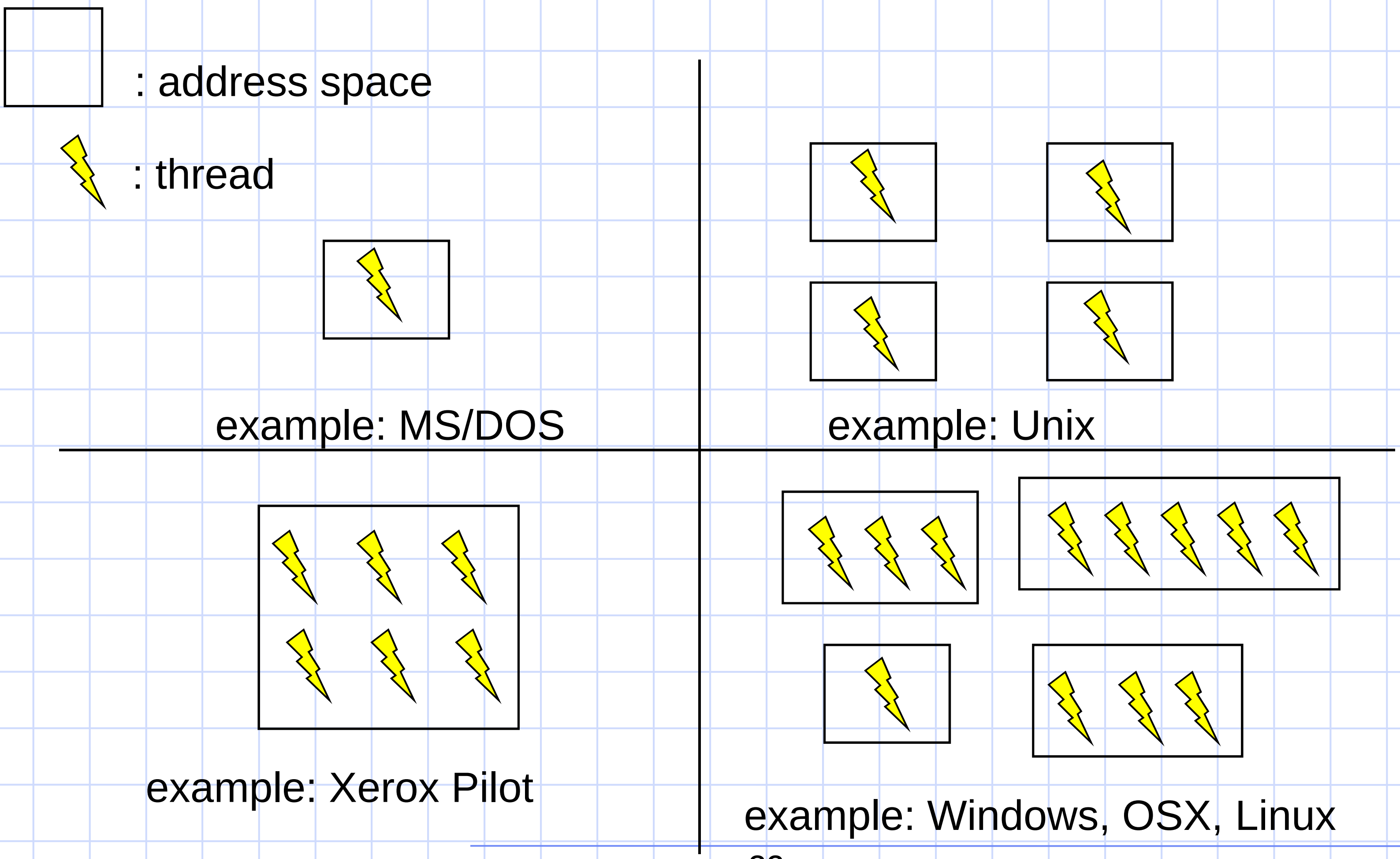
- We can eliminate duplicate address spaces and place concurrent computations in the same address space



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, ...

How different OSes support threads



Separation of Threads and Processes

- **Separating threads and processes makes it easier to support multi-threaded applications**
- **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 <u>uni</u>processor**
- **To be useful, thread operations have to be fast**

Kernel Threads

- Kernel threads still suffer from performance problems

- Operations on kernel threads are slow because:
  - a thread operation still requires a kernel call
  - kernel threads may be overly general, in order to support needs of different users, languages, etc.
  - the kernel doesn't trust the user, so there must be lots of checking on kernel calls

User-Level Threads

- To make threads really fast, they should be implemented at the user level
- A user-level thread is managed entirely by the run-time system (user-level code that is linked with your program).
- Each thread is represented simply by a PC, registers, stack and a little control block, managed in the user's address space.
- Creating a new thread, switching between threads, and synchronizing between threads can all be done without kernel involvement