

# **CSE 451: Operating Systems**

## **Winter 2022**

### **Module 4**

### **Processes**

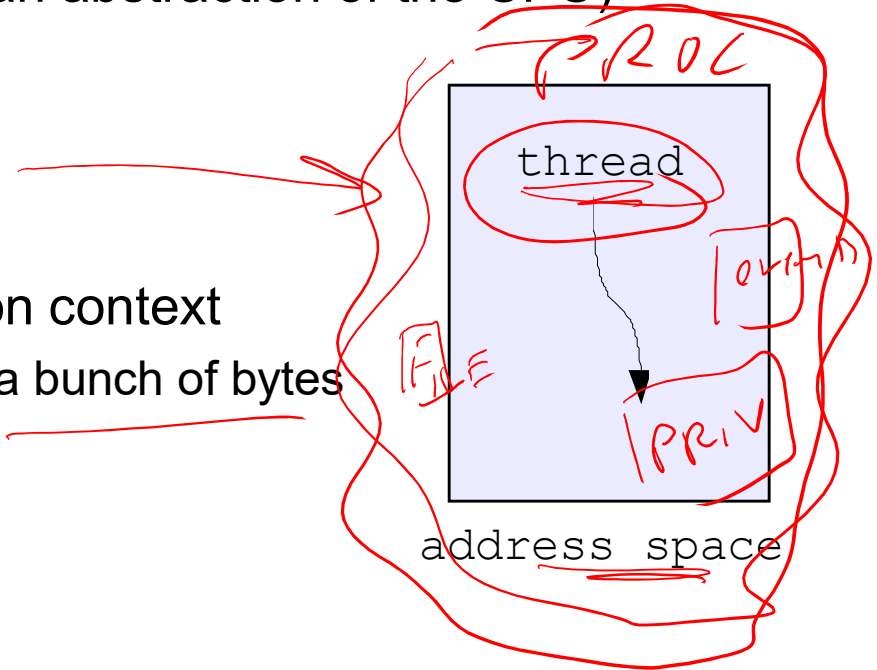
**Gary Kimura**

# Process management

- This module begins a series of topics on processes, threads, and synchronization
  - this is the most important part of the class
- In this module: processes and process management
  - What is a “process”? OWNERSHIP
  - What’s the OS’s process namespace?
  - How are processes represented inside the OS?
  - What are the executing states of a process?
  - How are processes created?
  - How can this be made faster?
  - Shells
  - Signals

# What is a “process”?

- The process is the OS’s abstraction for execution
  - A process is a program in execution
- Simplest (classic) case: a **sequential process**
  - An address space (an abstraction of memory)
  - A single thread of execution (an abstraction of the CPU)
- A sequential process is:
  - The unit of execution
  - The unit of scheduling
  - The dynamic (active) execution context
    - vs. the program – static, just a bunch of bytes

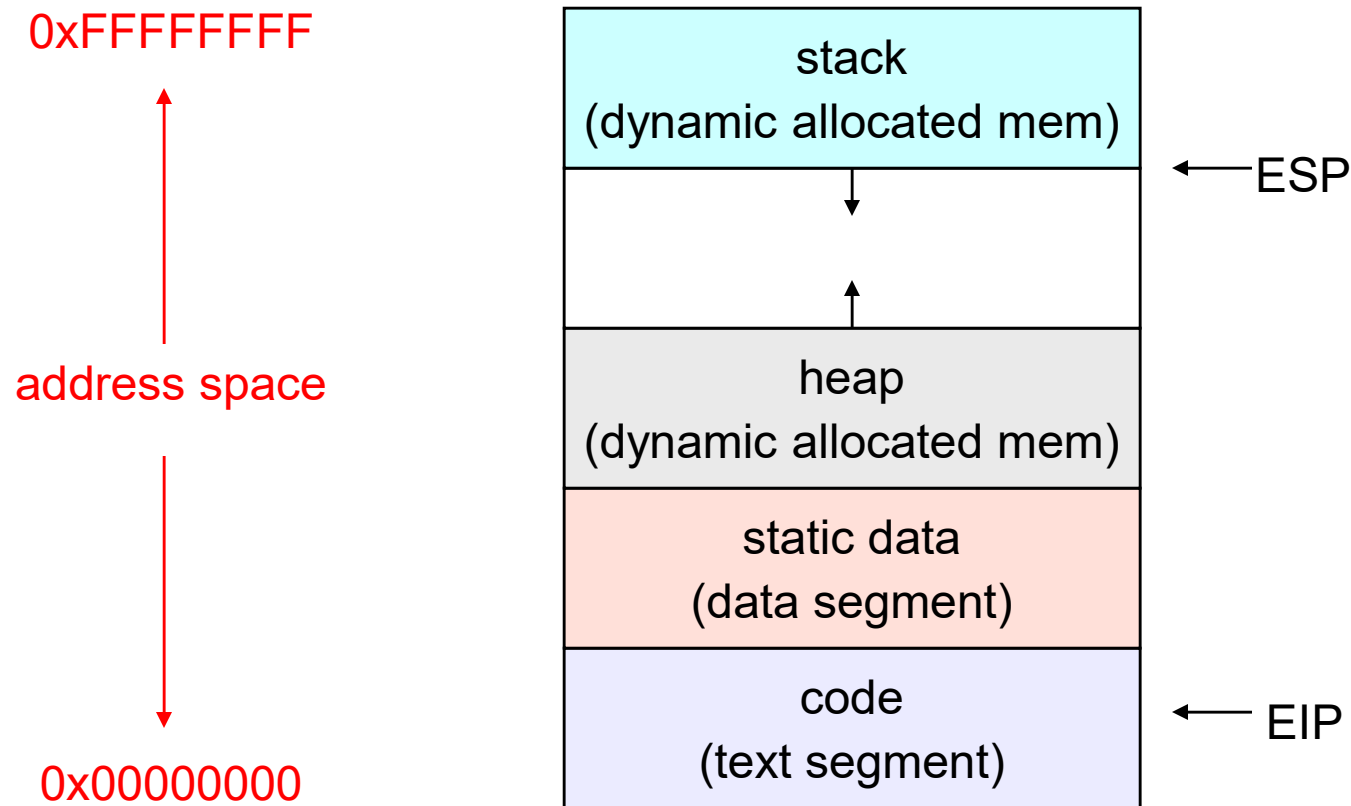


# What's "in" a process?

- A process consists of (at least):
  - An **address space**, containing
    - the code (instructions) for the running program
    - the data for the running program (static data, heap data, stack)
  - **(At least one) CPU state**, consisting of
    - The instruction pointer (EIP), indicating the next instruction
    - The stack pointer (ESP)
    - Other general purpose register values
  - A set of **OS resources**
    - open files, network connections, sound channels, ...
- In other words, it's all the stuff you need to run the program
  - or to re-start it, if it's interrupted at some point

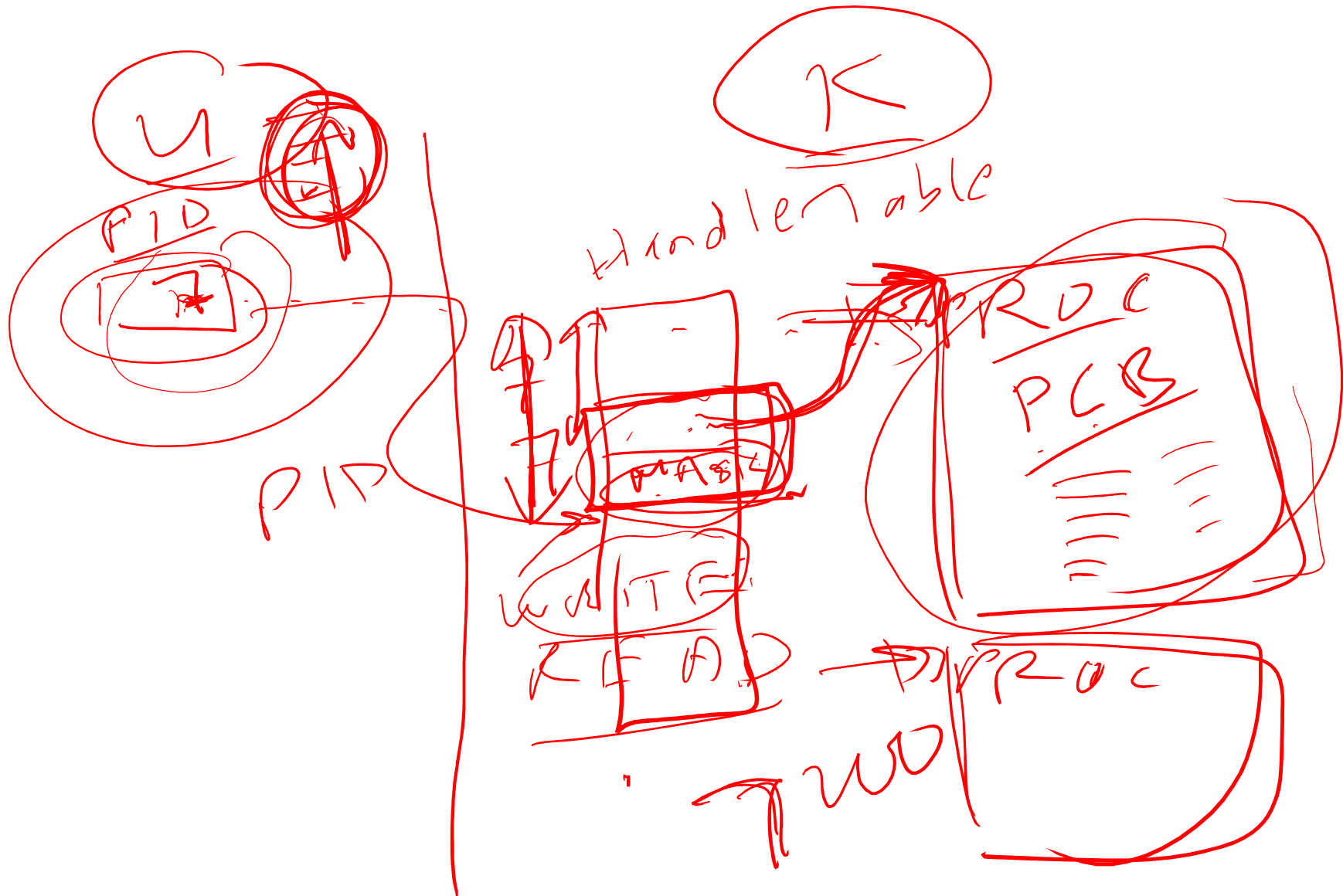
Threads  
EBP  
FRAME  
PTRNTR

# A process's address space (idealized)



# The OS's process namespace

- (Like most things, the particulars depend on the specific OS, but the principles are general)
- The name for a process is called a **process ID (PID)**
  - An integer
- The PID namespace is global to the system
  - Only one process at a time has a particular PID
- Operations that create processes return a PID
  - E.g., fork()
- Operations on processes take PIDs as an argument
  - E.g., kill(), wait(), nice()



# Representation of processes by the OS

- The OS maintains a data structure to keep track of a process's state
  - Called the **process control block** (PCB/KPROCESS/proc) or **process descriptor**
  - Identified by the **PID**
- OS keeps all of a process's execution state in (or linked from) the proc when the process isn't running
  - EIP, ESP, registers, etc.
  - when a process is unscheduled, the execution state is transferred out of the hardware registers into the proc
  - (when a process is running, its state is spread between the proc and the CPU)
- Note: It's natural to think that there must be some esoteric techniques being used
  - fancy data structures that you'd never think of yourself

***Wrong! It's pretty much just what you'd think of!***



# The proc

- The proc is a data structure with many, many fields:
  - process ID (pid)
  - pointer to parent proc
  - execution state ↵
  - Instruction pointer, stack pointer, registers }—
  - address space info ↵
  - pointers for state queues ↵
- In Linux:
  - defined in `task_struct` (`include/linux/sched.h`)
  - over 95 fields!!!

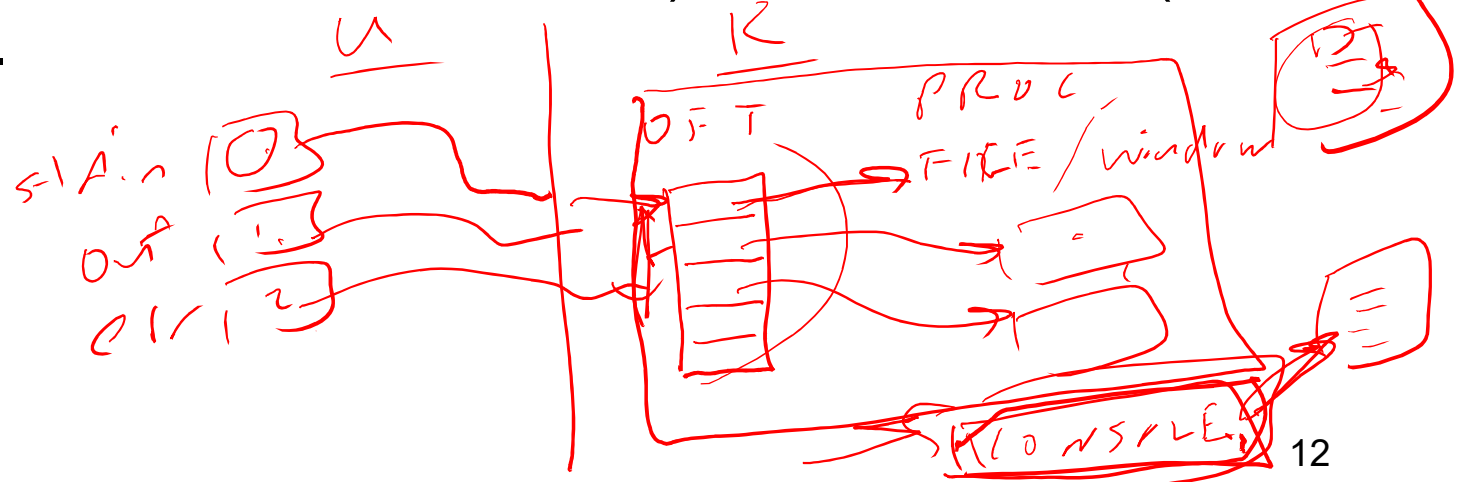
# procs and CPU state

- When a process is running, its CPU state is inside the CPU
    - EIP, ESP, registers
    - CPU contains current values
  - When the OS gets control because of a ...
    - Trap: Program executes a syscall
    - Exception: Program does something unexpected (e.g., page fault)
    - Interrupt: A hardware device requests service
- the OS saves the CPU state of the running process in that process's proc

- When the OS returns the process to the running state, it loads the hardware registers with values from that process's proc – general purpose registers, stack pointer, instruction pointer
- The act of switching the CPU from one process to another is called a **context switch**
  - systems may do 100s or 1000s of switches/sec.
  - takes a few microseconds on today's hardware
- Choosing which process to run next is called **scheduling**

# The OS kernel is not a process

- It's just a block of code!
- (In a microkernel OS, many things that you normally think of as the operating system execute as user-mode processes. But the OS kernel is just a block of code.)
- Remember: **the CPU is always executing code in the context of a process**. That code may be in user mode (restricted access to hardware) or kernel mode (free-for-all).

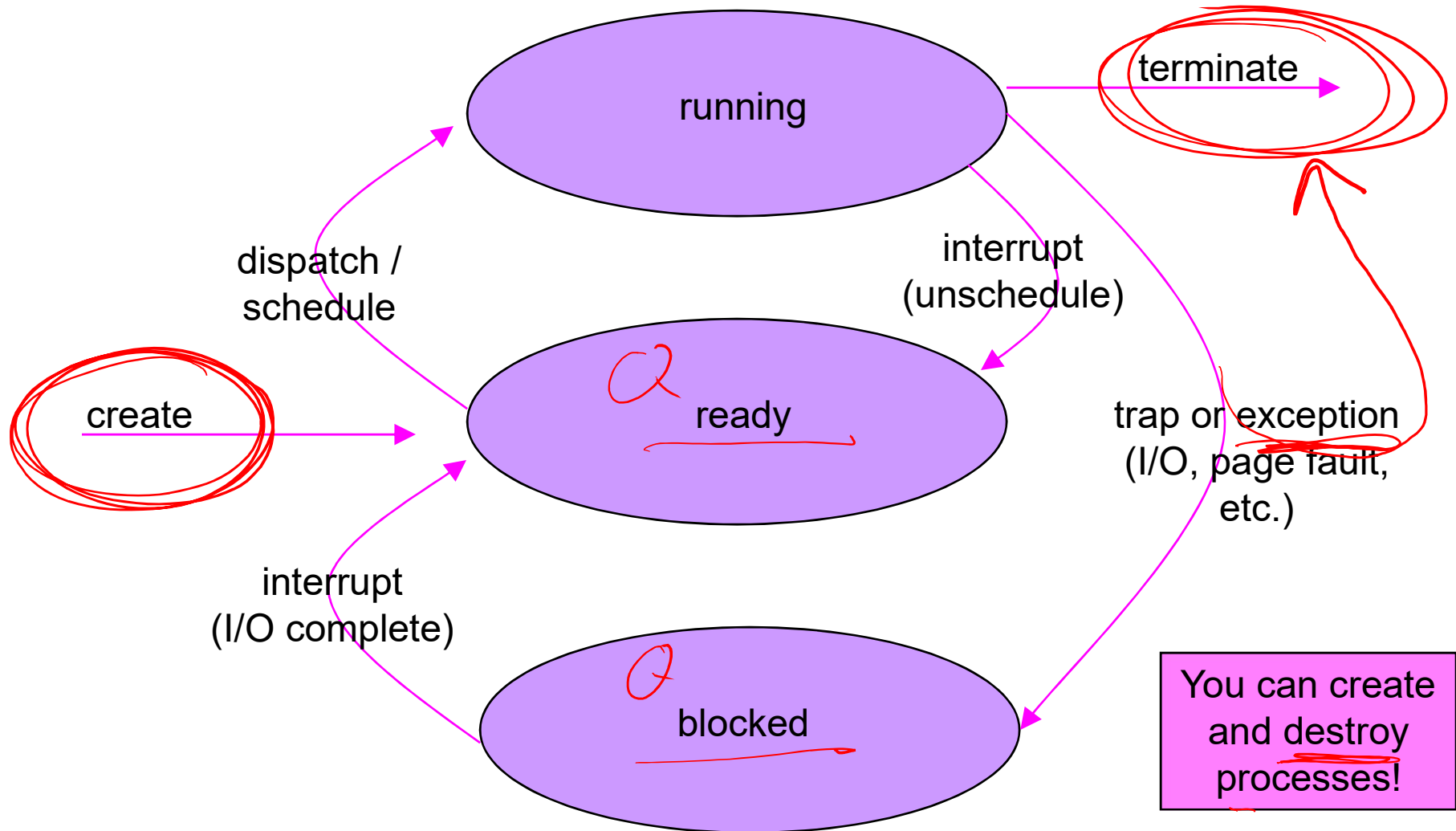




# Process execution states

- Each process has an **execution state**, which indicates what it's currently doing
  - **ready**: waiting to be assigned to a CPU
    - could run, but another process has the CPU
  - **running**: executing on a CPU
    - it's the process that currently controls the CPU
  - **waiting** (aka “blocked”): waiting for an event, e.g., I/O completion, or a message from (or the completion of) another process
    - cannot make progress until the event happens
- As a process executes, it moves from state to state
  - UNIX: run **ps**, STAT column shows current state
  - which state is a process in most of the time?

# Process states and state transitions



# State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
  - typically one queue for each state
    - e.g., ready, waiting, ...
  - each proc is queued onto a state queue according to the current state of the process it represents
  - as a process changes state, its proc is unlinked from one queue, and linked onto another
- Once again, *this is just as straightforward as it sounds!* The proc are moved between queues, which are represented as linked lists. *There is no magic!*

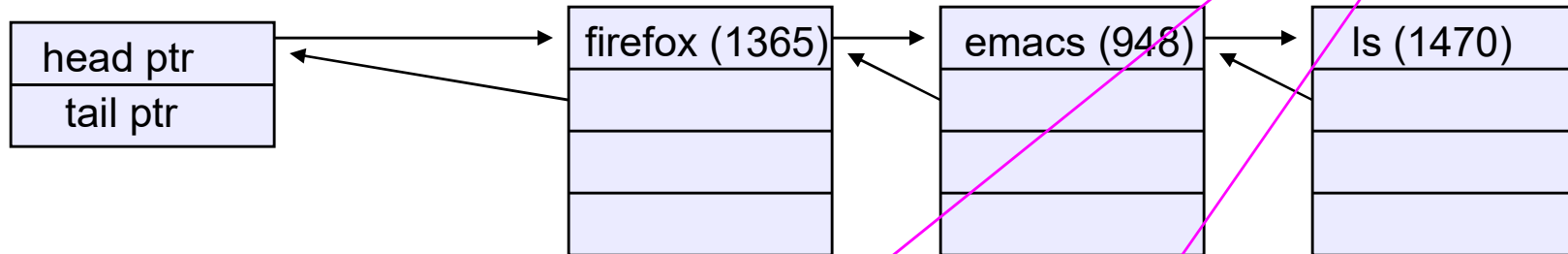




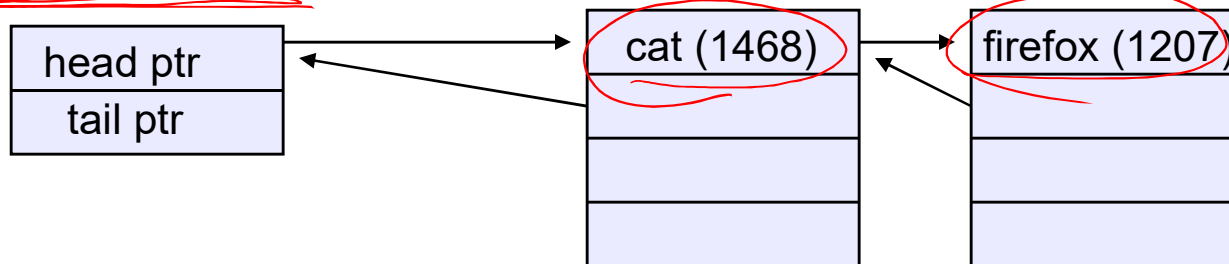
# State queues

These are procs!

Ready queue header

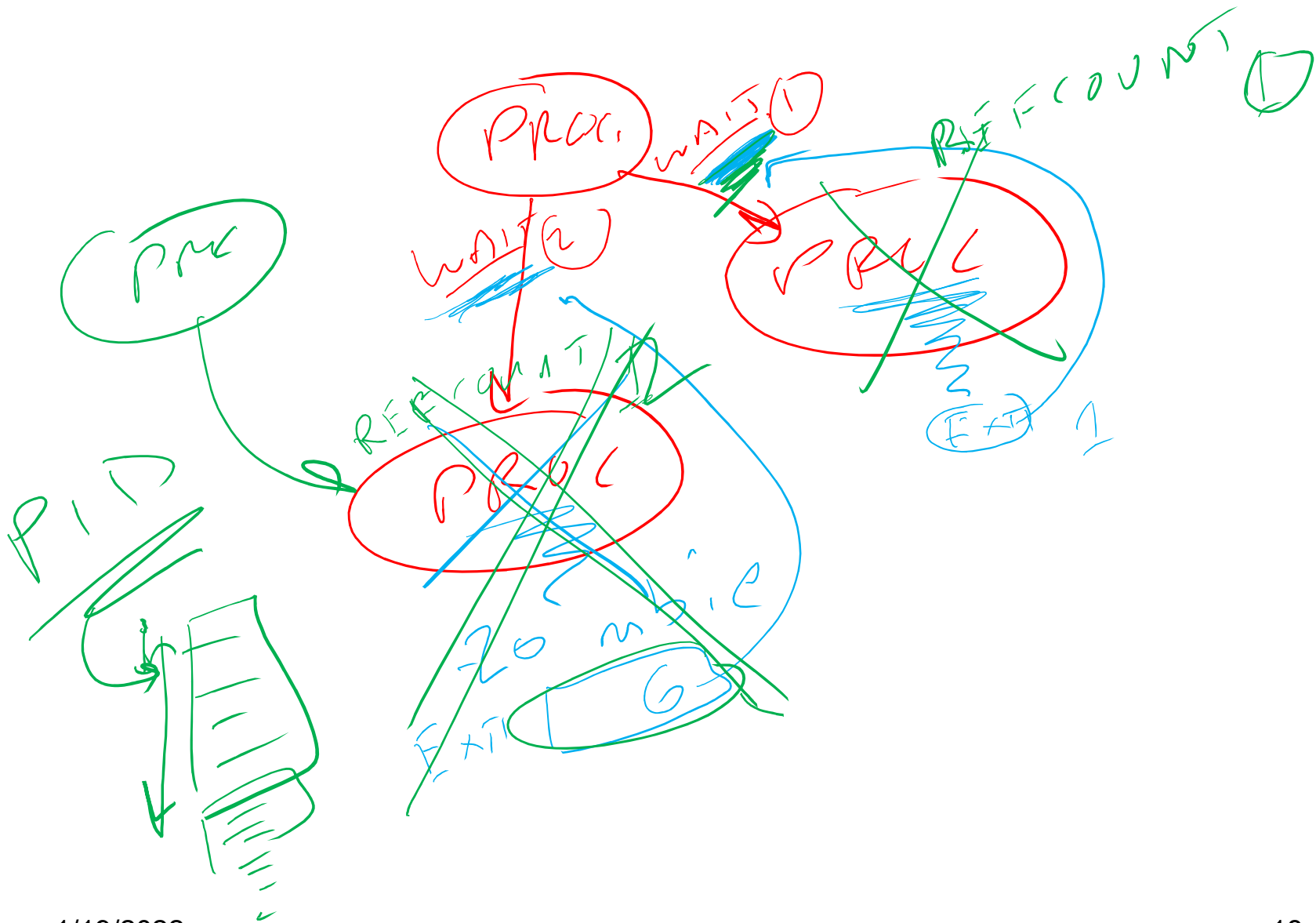


Wait queue header



wait  
SINGLE  
MULTIPLE  
ANY  
- TIMEOUT

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



# procs and state queues

- procs are data structures
  - dynamically allocated inside OS memory
- When a process is created:
  - OS allocates a proc for it
  - OS initializes proc
  - (OS does other things not related to the proc)
  - OS puts proc on the correct queue
- As a process computes:
  - OS moves its proc from queue to queue
- When a process is terminated:
  - proc may be retained for a while (to receive signals, etc.)
  - eventually, OS deallocates the proc

Make

h

make clean

# Process creation

- New processes are created by existing processes
  - creator is called the **parent**
  - created process is called the **child**
    - UNIX: do `ps`, look for PPID field
  - what creates the first process, and when?



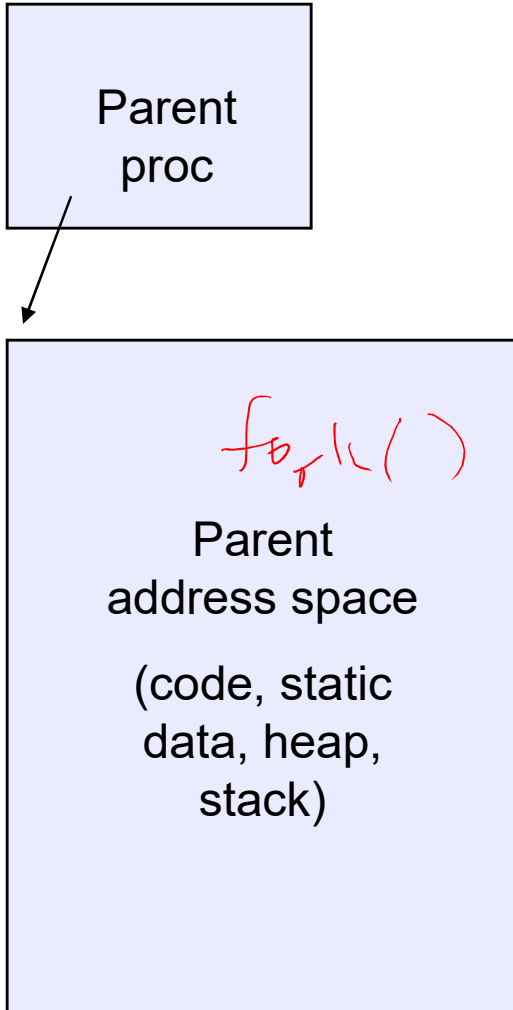
# Process creation semantics

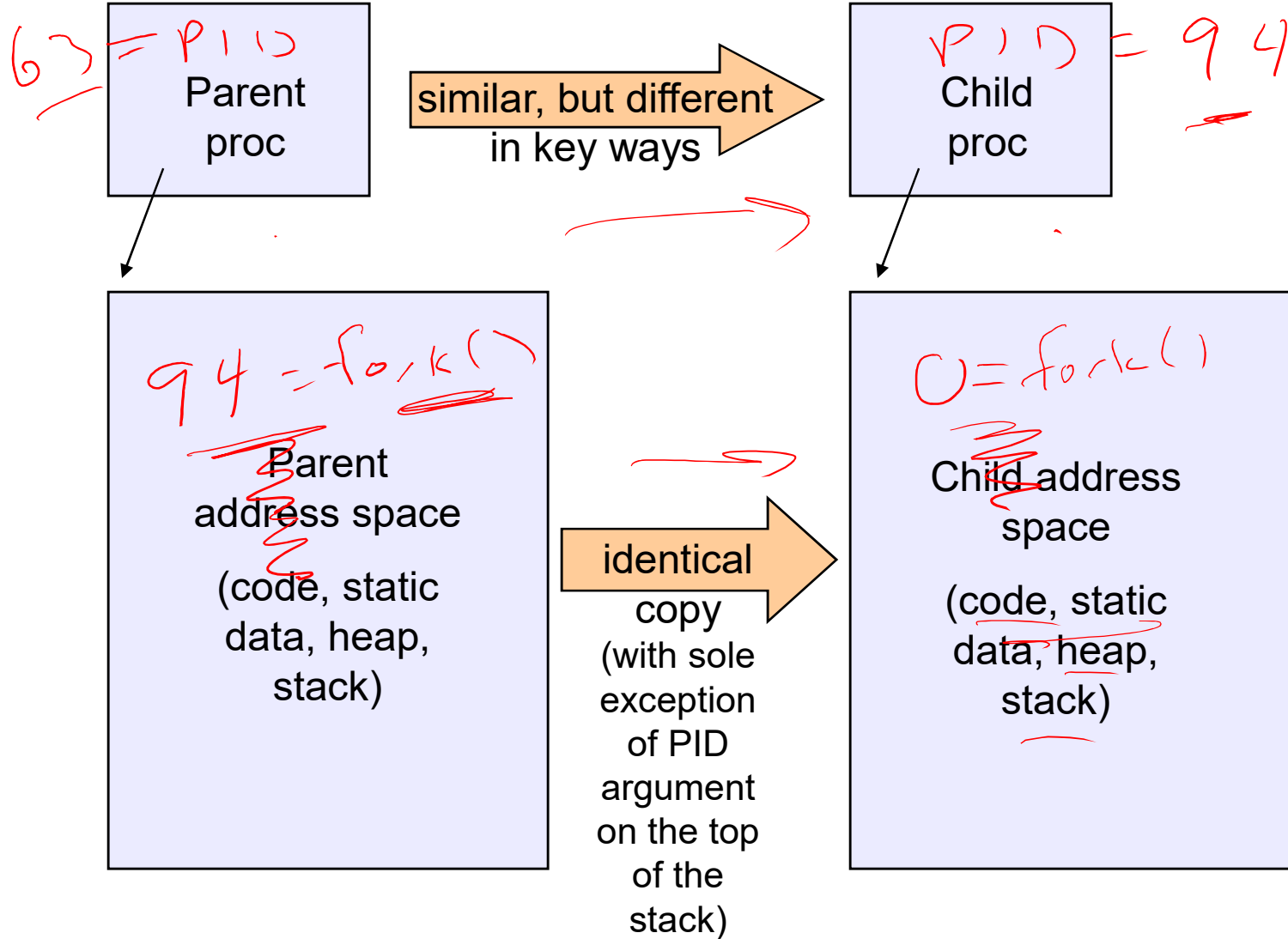
- (Depending on the OS) child processes inherit certain attributes of the parent
  - Examples:
    - Open file table: implies stdin/stdout/stderr
    - On some systems, resource allocation to parent may be divided among children
- (In Unix) when a child is created, the parent may either wait for the child to finish, or continue in parallel
- (In Unix) These are *policies implemented by the kernel*. In Windows, inheritance is done explicitly in user mode by the CreateProcess library routine (it's not a system call!)

# UNIX process creation details

- UNIX process creation through **fork()** system call
  - creates and initializes a new proc
    - initializes kernel resources of new process with resources of parent (e.g., open files)
    - initializes **EIP**, **ESP** to be same as parent
  - creates a new address space
    - initializes new address space with a copy of the entire contents of the address space of the parent
  - places new proc on the ready queue
- the **fork()** system call “returns twice”
  - once into the parent, and once into the child
    - returns the child’s PID to the parent
    - returns 0 to the child
- **fork()** = “clone me”







## testparent – use of fork( )

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    char *name = argv[0];
    int pid = fork();
    if (pid == 0) {
        printf("Child of %s is %d\n", name, pid);
        return 0;
    } else {
        printf("My child is %d\n", pid);
        return 0;
    }
}
```

# testparent output

```
spinlock% gcc -o testparent testparent.c
```

```
spinlock% ./testparent
```

```
My child is 486
```

```
Child of testparent is 0
```

```
spinlock% ./testparent
```

```
Child of testparent is 0
```

```
My child is 571
```

parent

←

child

parent

## exec() vs. fork()

- Q: So how do we start a new program, instead of just forking the old program?
- A: First fork, then **exec**
  - `int exec(char * prog, char * argv[])`
- **exec()**
  - stops the current process
  - loads program 'prog' into the address space
    - i.e., over-writes the existing process image
  - initializes hardware context, args for new program
  - places proc onto ready queue
  - note: does not create a new process!

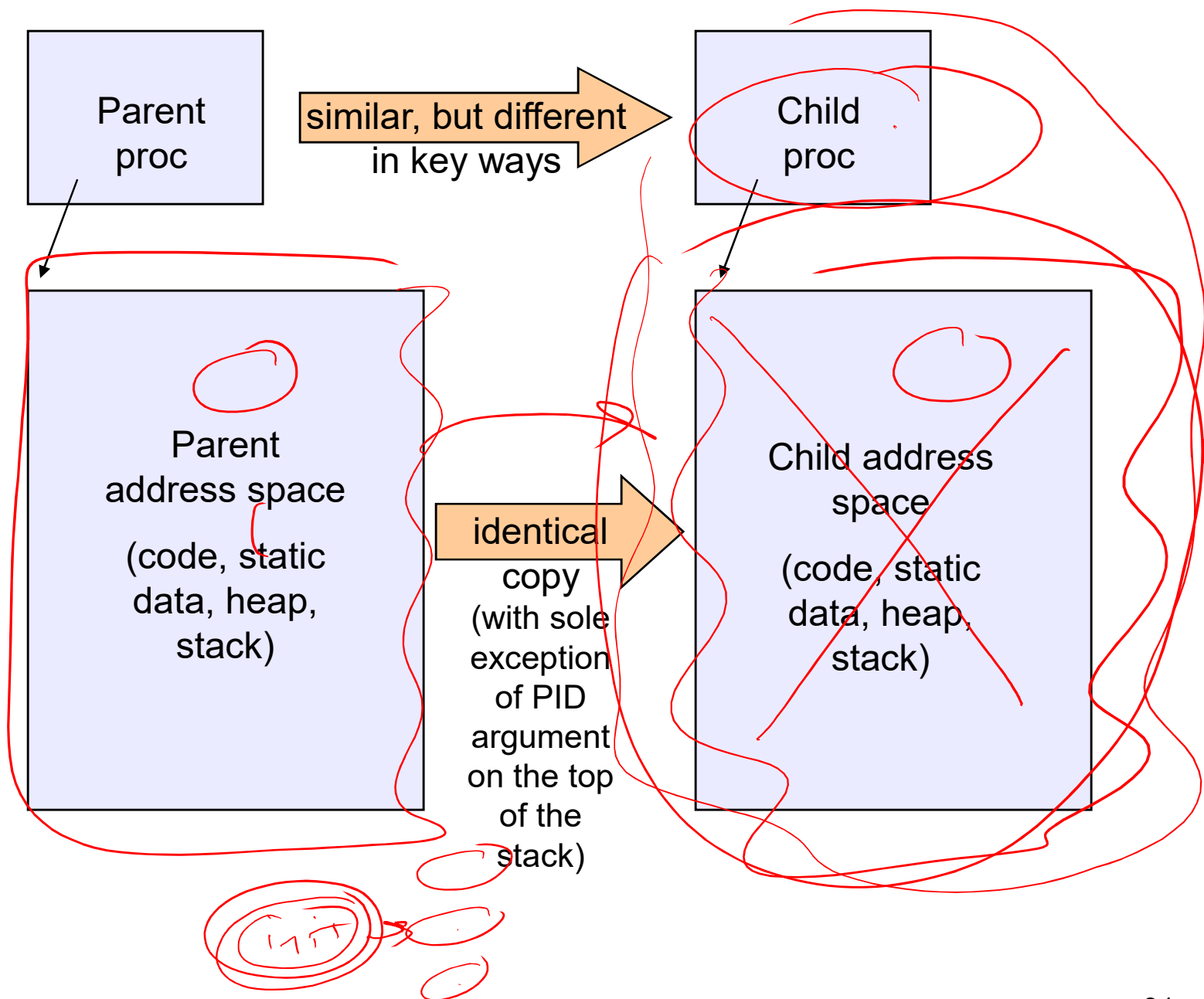
- So, to run a new program:

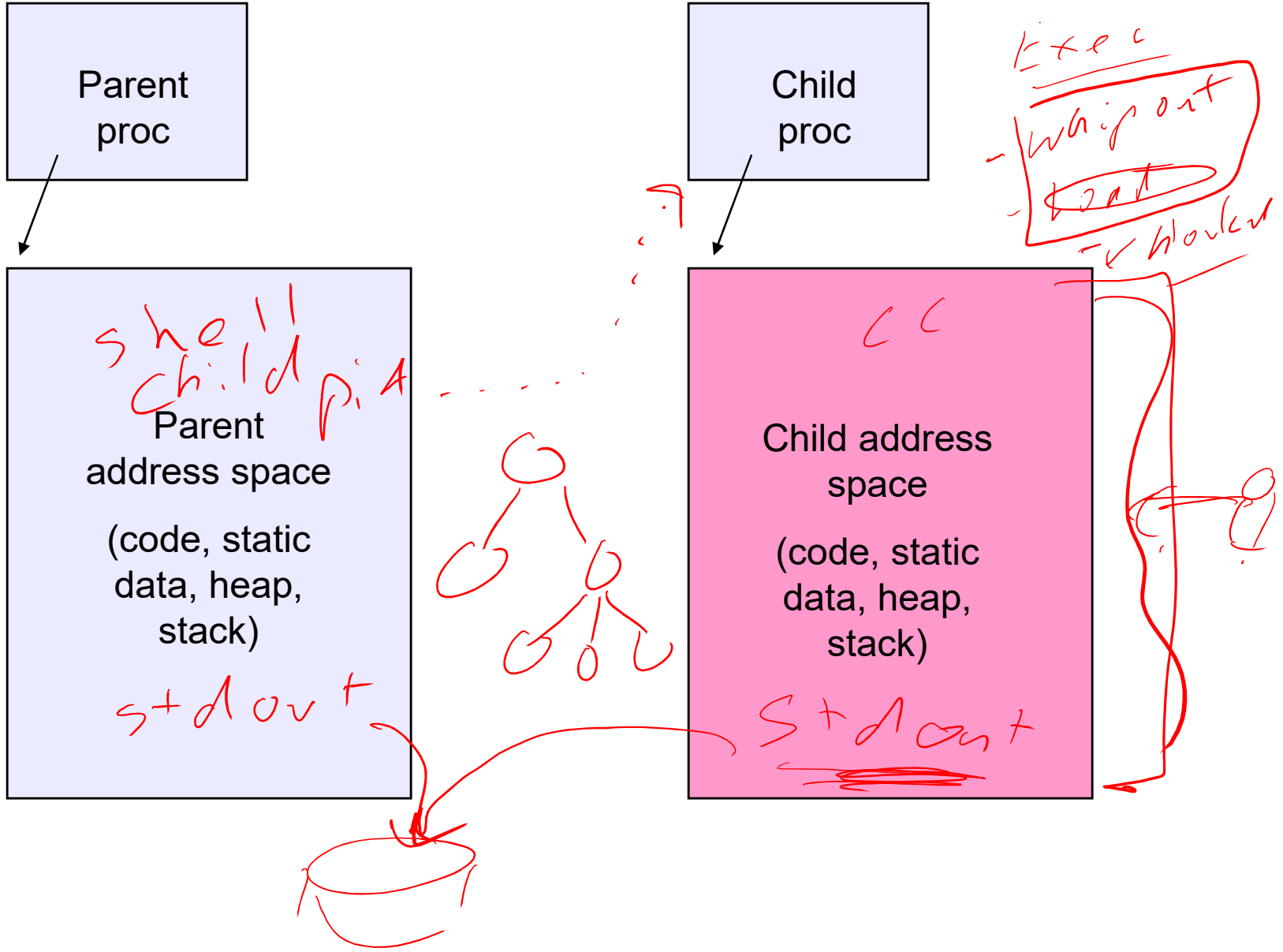
- fork()
- Child process does an exec()
- Parent either waits for the child to complete, or not



Shell

```
o = fork()
if (o) Exec("cc");
else
    wait or child to
    finish
```







# Making process creation faster

- The semantics of fork() say the child's address space is a copy of the parent's *exec*
- Implementing fork() that way is slow
  - Have to allocate physical memory for the new address space and reserve swap space
  - Have to set up child's page tables to map new address space
  - Have to copy parent's address space contents into child's address space
    - Which you are likely to immediately throw away with an exec()!

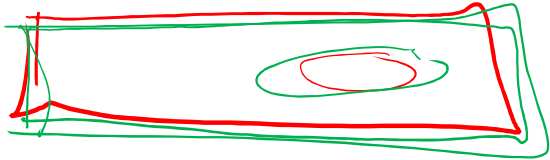
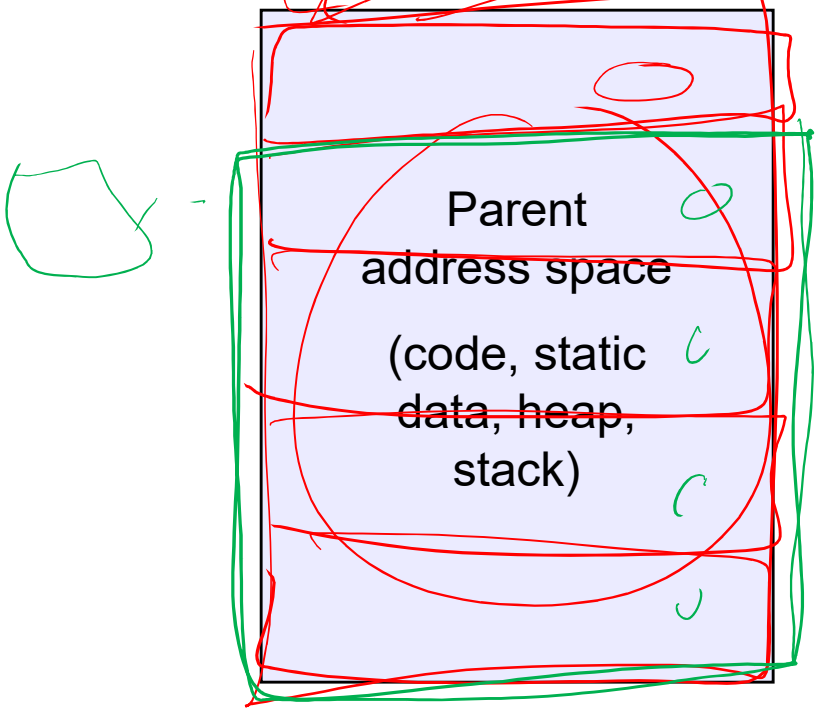
# Method 1: vfork()

- vfork() is the older (now uncommon) of the two approaches we'll discuss
- Instead of “child's address space is a copy of the parent's,” the semantics are “child's address space *is* the parent's”
  - With a “promise” that the child won't modify the address space before doing an `execve()`
    - Unenforced! You use vfork() at your own peril
  - When `execve()` is called, a new address space is created and it's loaded with the new executable
  - Parent is blocked until `execve()` is executed by child
  - Saves wasted effort of duplicating parent's address space, just to throw it away

Parent proc.

similar, but different  
in key ways

Child  
proc



Vfork()

## Method 2: copy-on-write

- Retains the original semantics, but copies “only what is necessary” rather than the entire address space
- On fork():
  - Create a new address space
  - Initialize page tables with same mappings as the parent’s (i.e., they both point to the same physical memory)
    - No copying of address space contents have occurred at this point – with the sole exception of the top page of the stack
  - Set both parent and child page tables to make all pages read-only
  - If either parent or child writes to memory, an exception occurs
  - When exception occurs, OS copies the page, adjusts page tables, etc.

# UNIX shells

```
int main(int argc, char **argv)
{
    while (1) {
        printf ("$ ");
        char *cmd = get_next_command();
        int pid = fork();
        if (pid == 0) {
            exec(cmd);
            panic("exec failed!");
        } else {
            wait(pid);
        }
    }
}
```

## Truth in advertising ...

- In Linux today, clone is replacing fork (and vfork)
  - clone has additional capabilities/options
- But you need to clearly understand fork as described here
  
- In Linux today, exec is not a system call; execve is the only “exec-like” system call
  - execve knows whether you have done a fork or a vfork by a flag in the PCB
- But you need to clearly understand exec as described here

# Input/output redirection

- `$ ./myprog < input.txt > output.txt # UNIX`
  - each process has an open file table
  - by (universal) convention:
    - 0: stdin
    - 1: stdout
    - 2: stderr
- A child process inherits the parent's open file table
- Redirection: the shell ...
  - copies its current stdin/stdout open file entries
  - opens input.txt as stdin and output.txt as stdout
  - fork ...
  - restore original stdin/stdout

# Old-school Inter-process communication via signals

- Processes can register event handlers
  - Feels a lot like event handlers in Java, which ..
  - Feel sort of like catch blocks in Java programs
- When the event occurs, process jumps to event handler routine
- Used to catch exceptions
- Also used for inter-process (process-to-process) communication
  - A process can trigger an event in another process using **signal**
  - Note that a signal is one bit; you can pass way more information using other signals... but why?



# Signals

| Signal  | Value    | Action | Comment   |
|---------|----------|--------|---|
| SIGHUP  | 1        | Term   | Hangup detected on controlling terminal or death of controlling process |
| SIGINT  | 2        | Term   | Interrupt from keyboard   |
| SIGQUIT | 3        | Core   | Quit from keyboard  |
| SIGILL  | 4        | Core   | Illegal Instruction   |
| SIGABRT | 6        | Core   | Abort signal from abort(3)  |
| SIGFPE  | 8        | Core   | Floating point exception  |
| SIGKILL | 9        | Term   | Kill signal   |
| SIGSEGV | 11       | Core   | Invalid memory reference  |
| SIGPIPE | 13       | Term   | Broken pipe: write to pipe with no read                                 |
| SIGALRM | 14       | Term   | Timer signal from alarm(2)  |
| SIGTERM | 15       | Term   | Termination signal  |
| SIGUSR1 | 30,10,16 | Term   | User-defined signal 1   |
| SIGUSR2 | 31,12,17 | Term   | User-defined signal 2   |
| SIGCHLD | 20,17,18 | Ign    | Child stopped or terminated   |
| SIGCONT | 19,18,25 |        | Continue if stopped   |
| SIGSTOP | 17,19,23 | Stop   | Stop process  |
| SIGTSTP | 18,20,24 | Stop   | Stop typed at tty   |
| SIGTTIN | 21,21,26 | Stop   | tty input for background process  |
| SIGTTOU | 22,22,27 | Stop   | tty output for background process                                       |

# Example use

- You're implementing Apache, a web server
- Apache reads a configuration file when it is launched
  - Controls things like what the root directory of the web files is, what permissions there are on pieces of it, etc.
- Suppose you want to change the configuration while Apache is running
  - If you restart the currently running Apache, you drop some unknown number of user connections
- Solution: send the running Apache process a signal
  - It has registered a signal handler that gracefully re-reads the configuration file