CSE 451: Operating Systems
Spring 2020

Module 4
Processes

John Zahorjan

Lecture Questions

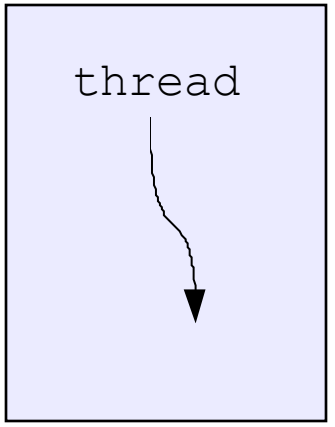
- What is as process?
- What lower level resources are hidden/simplified by the process abstraction?
- Which aspects of process semantics most commonly important to software design issues? to system management issues?
- Why is process creation broken into fork() and then exec()?
- How do you make fork() less expensive?
- What does a shell do?
- How can processes communicate with each other? why would you want them to?
- Why might you want additional abstraction built above processes? What's the relationship of "user" to "process"? Is "user" the fundamental abstraction?

Process management

- This module begins a series of topics on processes, threads, and synchronization
  - this is the most important part of the class
  - there definitely will be several questions on these topics on the midterm
- In this module: processes and process management
  - What is a “process”?
  - 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 unit of failure
  - The dynamic (active) execution context
    - vs. the program – static, just a bunch of bytes

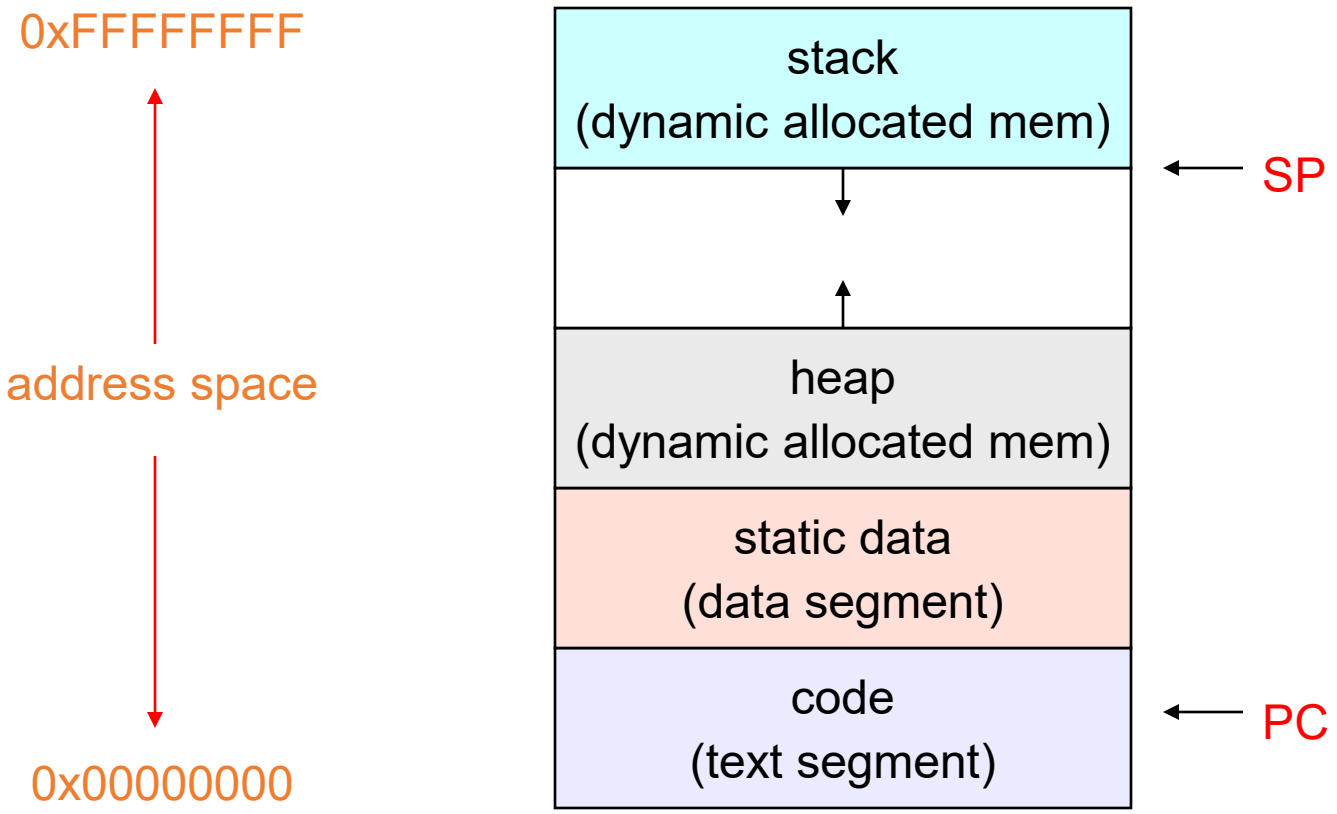


address space

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)
  - CPU state, consisting of
    - The program counter (PC), indicating the next instruction
    - The stack pointer (SP)
    - Other general purpose register (GPR) values
      - Each thread has its own PC, SP, and GPR 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

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) or process descriptor
  - Identified by the PID
- OS keeps all of a process's execution state in (or linked from) the PCB when the process isn't running
  - PC, SP, registers, etc.
  - when a process is unscheduled, the execution state is transferred out of the hardware registers into the PCB
  - (when a process is running, its state is spread between the PCB and the CPU)
- Note: It's natural to think that there must be some esoteric techniques being used
  - Nope…
    - Except that xk uses some data structures we're pretty sure you wouldn't choose…

The Process Control Block

- The PCB is a data structure with many, many fields:
  - process ID (PID)
  - parent process ID (PPID)
  - execution state
  - program counter, stack pointer, registers
  - address space info
  - UNIX user id (uid), group id (gid)
  - scheduling priority
  - accounting info
  - pointers for state queue
  - …

PCBs and CPU state

- When a process is running, its CPU state is on the CPU
  - PC, SP, 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

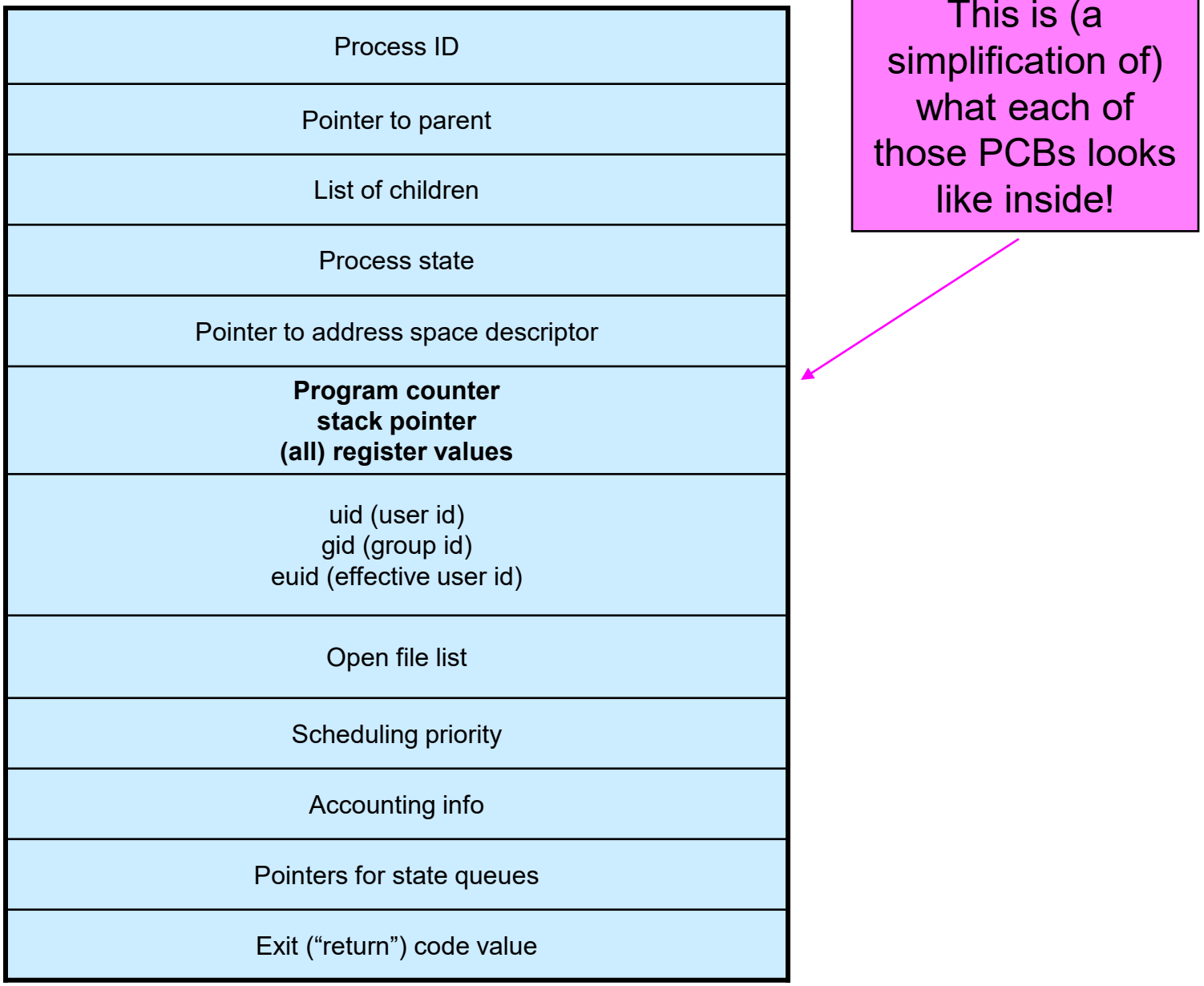
  the OS saves the CPU state of the running process in that process's PCB

  (In xk, the CPU register state is saved at the bottom of the kernel stack associated with the process)

- When the OS returns the process to the running state, it loads the hardware registers with values from that process's PCB – 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
    - *See programming exercise from the first week*

- Choosing which process to run next is called scheduling


Process ID
Pointer to parent
List of children
Process state
Pointer to address space descriptor
Program counter
stack pointer
(all) register values
uid (user id)
gid (group id)
euid (effective user id)
Open file list
Scheduling priority
Accounting info
Pointers for state queues
Exit (“return”) code value
This is (a simplification of) what each of those PCBs looks like inside!

Scope of OS Resources

- OS resources are things like open file tables, network connection points (sockets), pipes, shared memory regions, …
- Each requires its own descriptor block
- If the block is embedded in a PCB, the scope of that resource can be only that one process
- If the block is stored separately and the PCB contains a reference (pointer) to it, the resource can be shared
- What's shared and what isn't is part of OS API design

Allocation of OS Resources

- How should process control blocks be allocated?
  - Statically?
    - A fixed size block of memory is allocated by declaring an array of fixed size in the code
    - A region of memory is allocated during boot
    - Why would you much rather do the latter, if you're going to allocate statically?
  - Dynamically?
    - At the extreme, allocate space for a PCB each time a process is created
    - Free a PCB each time a process terminates
      - Possibly cache free PCBs to avoid allocation/deallocation overheads
  - Which should the OS use?

Allocation of OS Resources

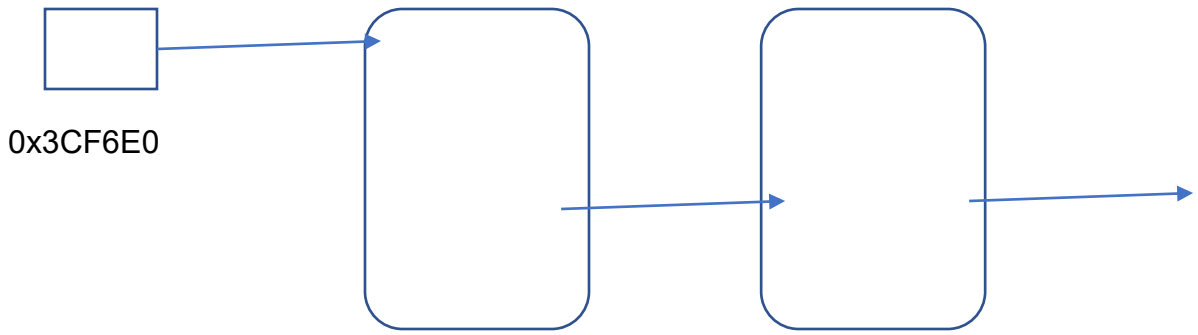
- How should process control blocks be allocated?
  - Statically?
    - A fixed size block of memory is allocated by declaring an array of fixed size in the code
    - A region of memory is allocated during boot
    - Why would you much rather do the latter, if you're going to allocate statically?
  - Dynamically?
    - At the extreme, allocate space for a PCB each time a process is created
    - Free a PCB each time a process terminates
      - Possibly cache free PCBs to avoid allocation/deallocation overheads
  - Which should the OS use?
- You have to worry about running out of memory
  - What to do then?
  - Can't start new process...
    - Means you can't bring up a tool that kills an old process
  - Programmer error can exhaust all system memory...

The OS kernel is not a process

- The x86 architecture supports switching “task” as part of the hardware action on entry to the OS
  - But xk (and most OS’s) basically work around that feature
- On entry to the OS, the CPU is still running in the context of the process that was running
  - Address space
  - Registers (to be saved)
  - Current PCB
- On exit back to user level, the context of dispatched process has been re-established
- There’s a brief moment in between when “it’s just code”
  - The CPU doesn’t know anything about processes…

PCB Chaining (except in xk)

- In “real systems” the PCB contains a pointer field that allows the PCB to be put on (a single, arbitrary) linked list
- For instance, there might be a linked list of processes in the ready state (i.e., waiting for the cpu)
- There might be a linked list of processes blocked on disk 2 (either waiting for an operation to complete or else to initiate one)



- Every blocked processes is on some list
- Every blocked process is on exactly one list
  - You can be blocked on only one thing (because when you block on the first thing you can’t execute code to block on the second)
  - As usual, such a simple constraint is too confining, and so there’s a way to work around it (select)
    - How can that work?

PCB Chaining in xk

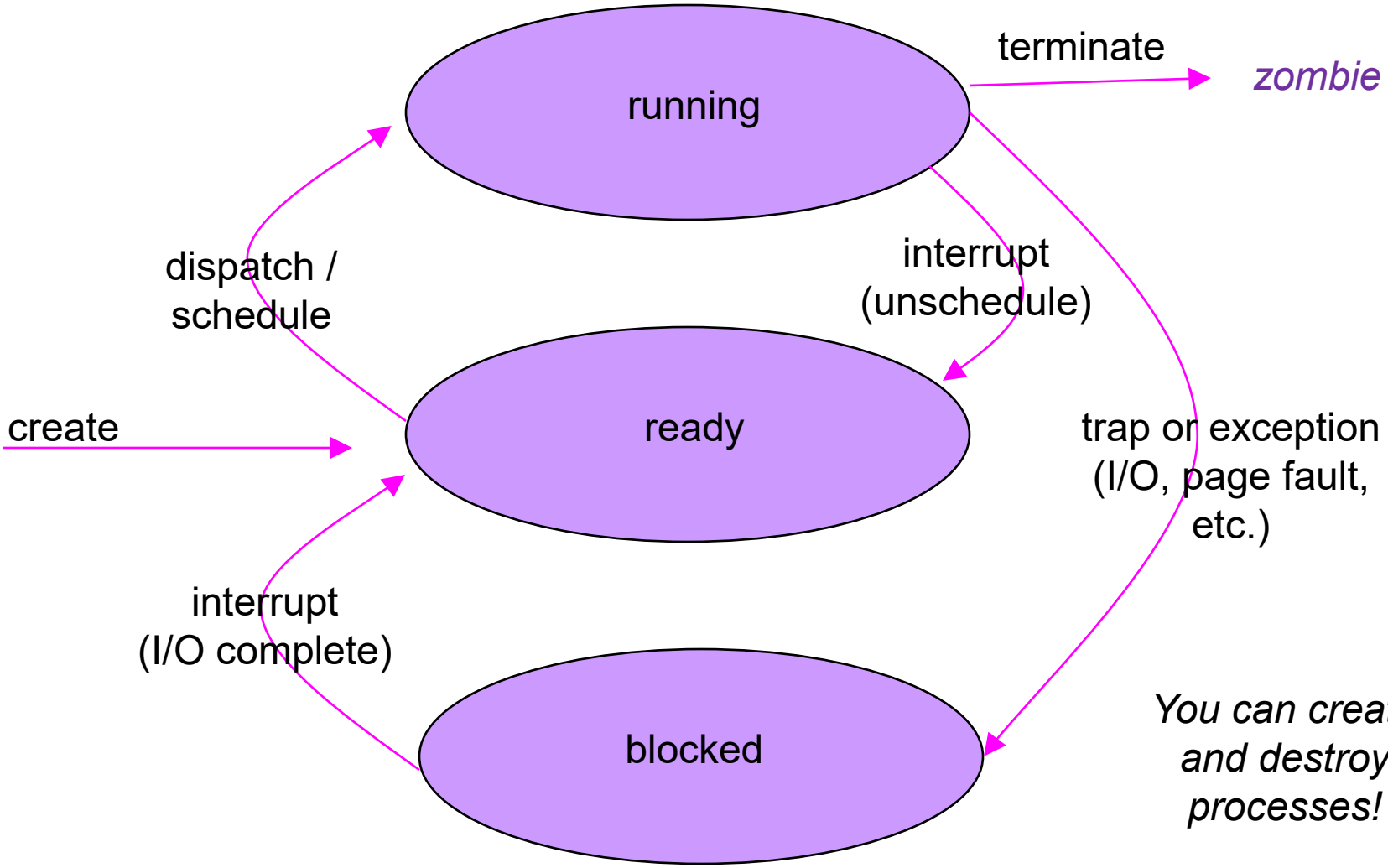
- xk uses a “simpler” scheme
  - A statically allocated array of PCBs (like always)
  - A field that can hold an address
  - To understand a “list” of PCBs, xk scans the entire array of PCBs looking for a particular value in the special address field

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?

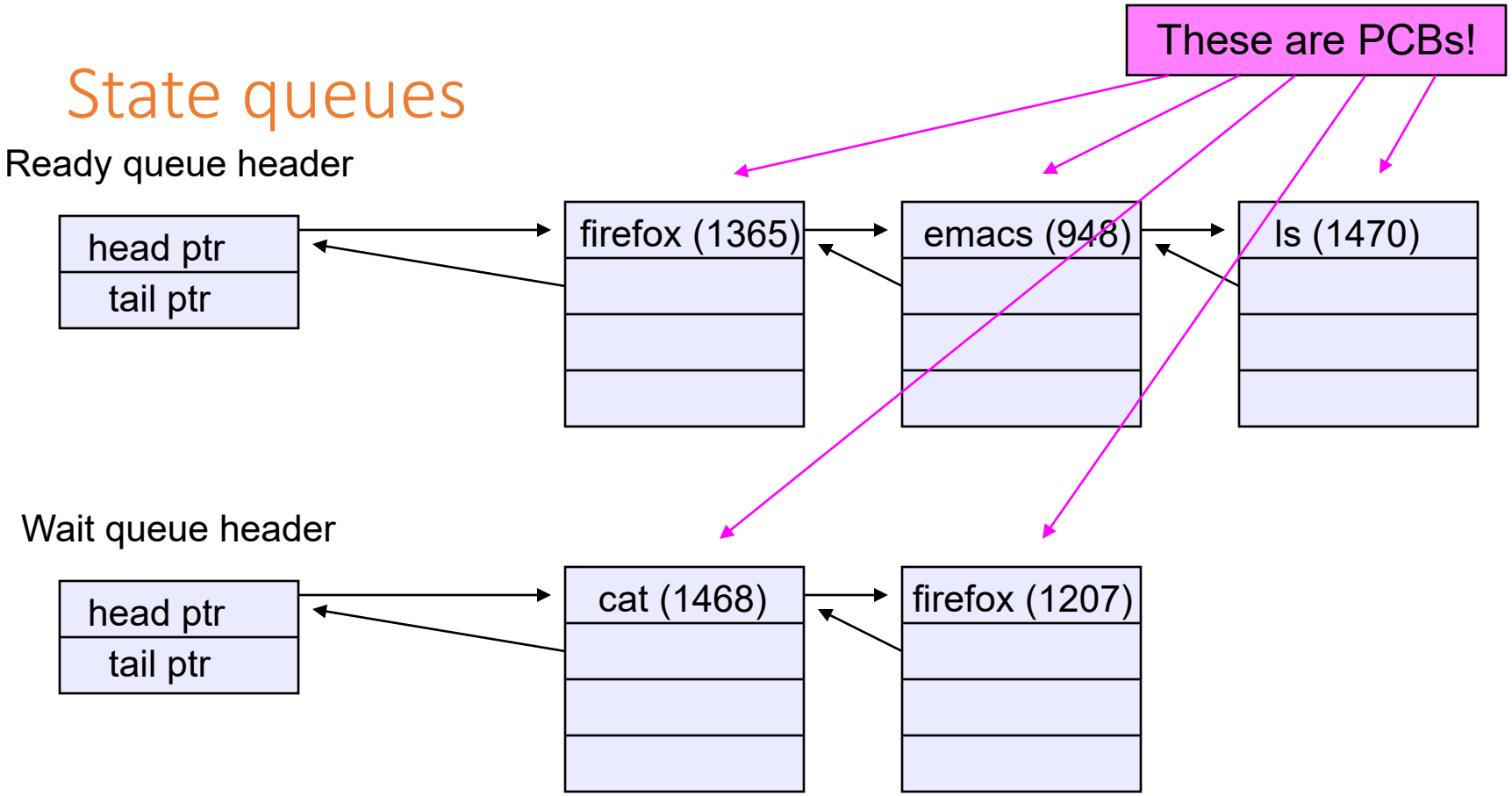
Process states and state transitions



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 PCB is queued onto a state queue according to the current state of the process it represents
  - 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 (particular device, timer, message, …)

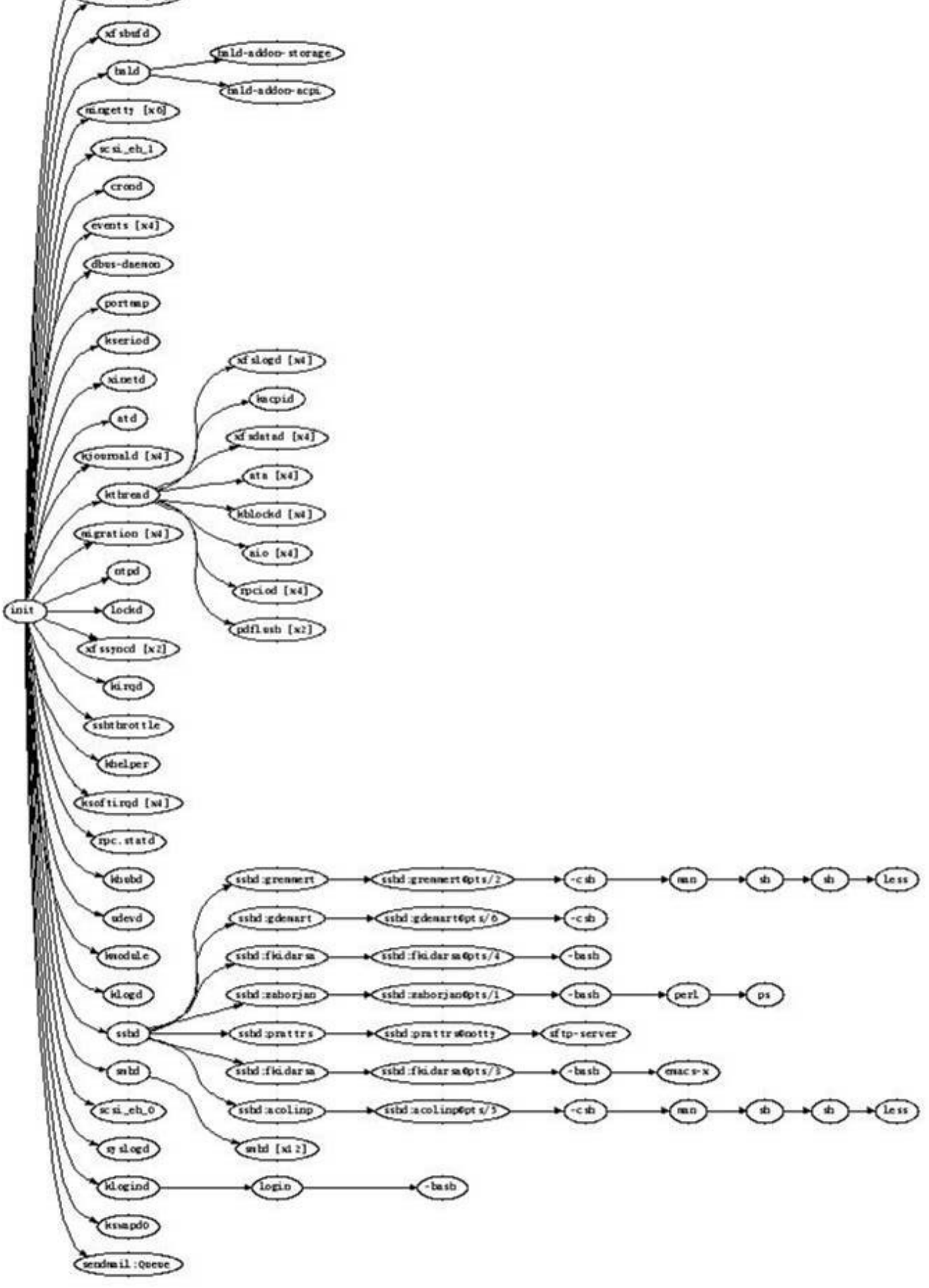
PCBs and state queues

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

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?

- $ ps –ejH
  - prints process tree


xfsbufd
hald
hald-addon-storage
hald-addon-acpi
mingetty [x6]
scsi_eh_1
crond
events [x4]
dbus-daemon
portmap
kseriod
xinetd
atd
kjournald [x4]
kthread
migration [x4]
ntpd
init
lockd
xfssyncd [x2]
khubd
sshdthrottle
khelper
ksoftirqd [x4]
rpc.statd
khubd
udevd
kmodule
klogd
sshd
sshd
scsi_eh_0
syslogd
klogind
kswapd0
sendmail: Queue
xfslogd [x4]
kacpid
xfsdatad [x4]
ata [x4]
kblockd [x4]
aio [x4]
rpciod [x4]
pdflush [x2]
sshd:grennert
sshd:grennert@pts/2
-csh
man
sh
sh
less
sshd:gdenart
sshd:gdenart@pts/6
-csh
sshd:fki.darsa
sshd:fki.darsa@pts/4
-bash
sshd:zaborjan
sshd:zaborjan@pts/1
-bash
perl
ps
sshd:prattrs
sshd:prattrs@notty
sftp-server
sshd:fki.darsa
sshd:fki.darsa@pts/3
-bash
emacs-x
sshd:acolinp
sshd:acolinp@pts/5
-csh
man
sh
sh
less
sshd [x12]
login
-bash

Process creation semantics

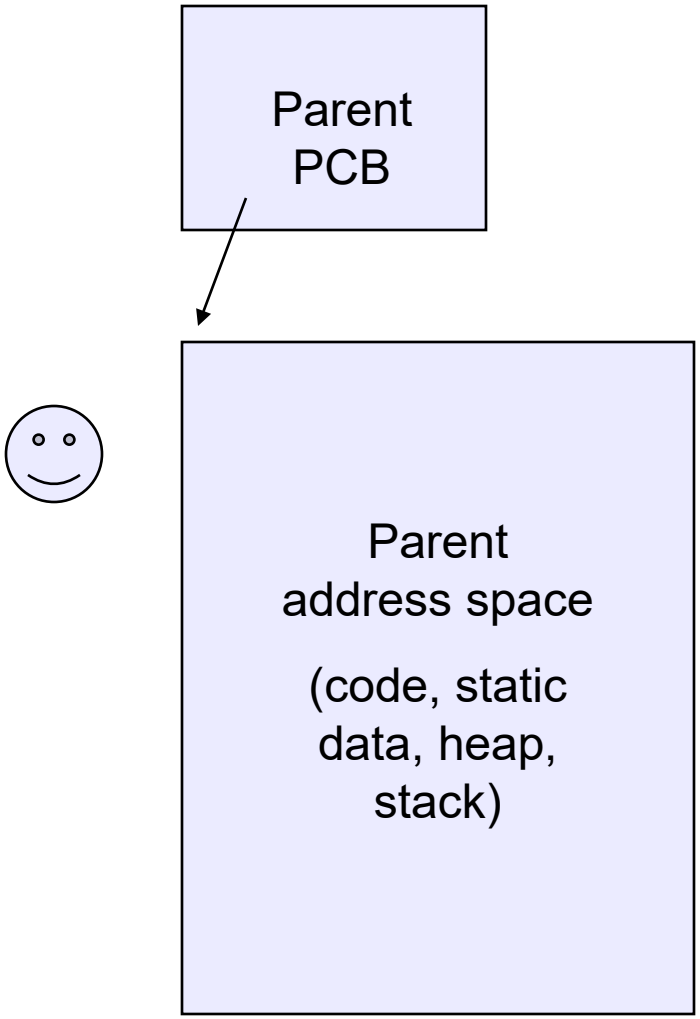
- (Depending on the OS) child processes inherit certain attributes of the parent
  - Examples:
    - Open file table: implies stdin/stdout/stderr
    - On some systems, resource allocation to parent may be divided among children
- (In Unix) when a child is created, the parent may either wait for the child to finish, or continue in parallel

UNIX process creation details

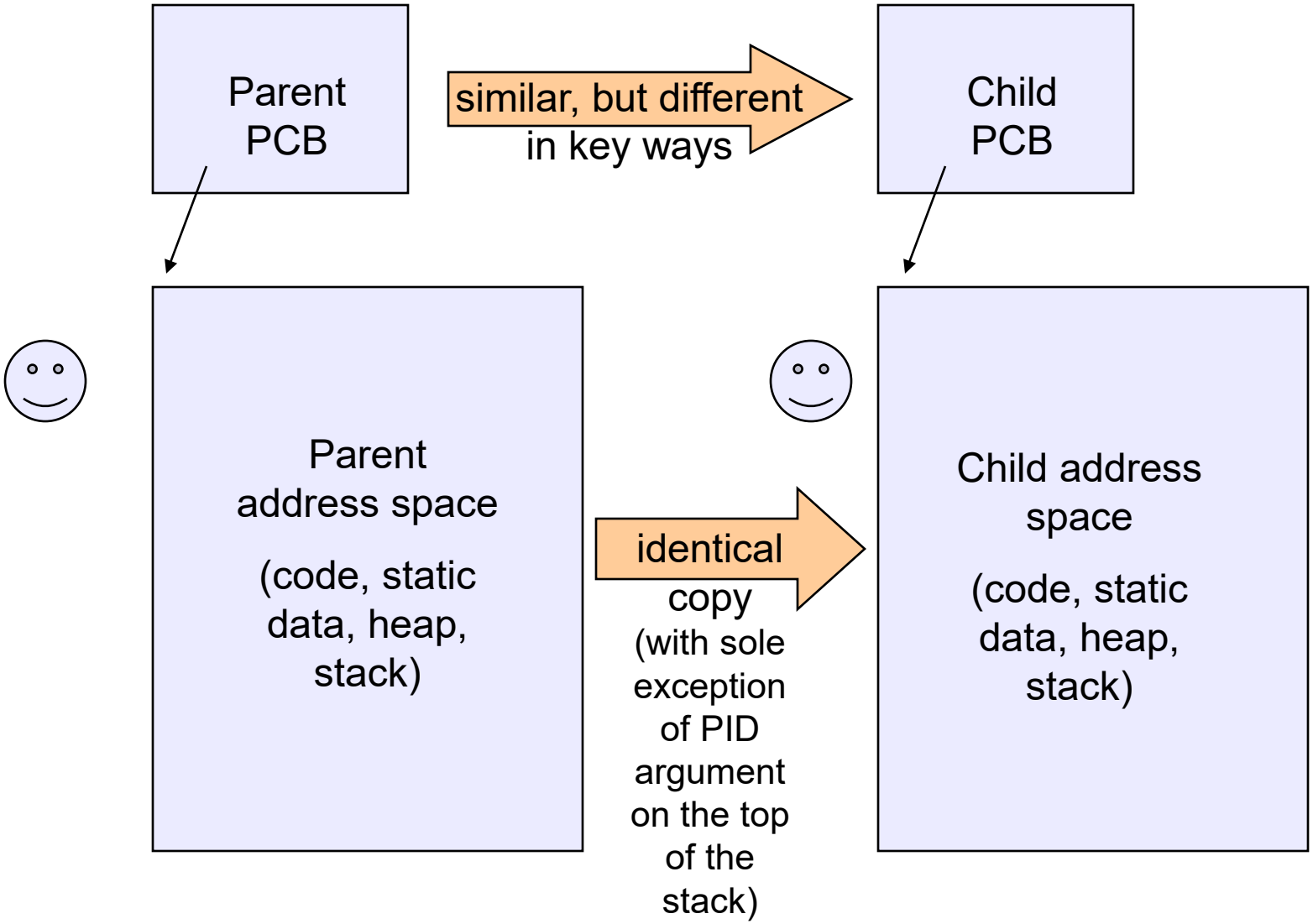
- UNIX process creation through `fork()` system call
  - creates and initializes a new PCB
    - initializes kernel resources of new process with resources of parent (e.g., open files)
    - initializes PC, SP to be same as parent
  - creates a new address space
    - initializes new address space with a copy of the entire contents of the address space of the parent
  - places new PCB on the ready queue

- the `fork()` system call "returns twice"
  - once into the parent, and once into the child
  - value returned from call depends...
    - returns the child's PID to the parent
    - returns 0 to the child

- `fork()` = "make a copy of me in my current state"


Parent
PCB
Parent
address space
(code, static
data, heap,
stack)

$ ./myProgram


Parent PCB
similar, but different in key ways
Child PCB
Parent address space
(code, static data, heap, stack)
identical copy
(with sole exception of PID argument on the top of the stack)
Child address space
(code, static data, heap, stack)

$ ./myProgram

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

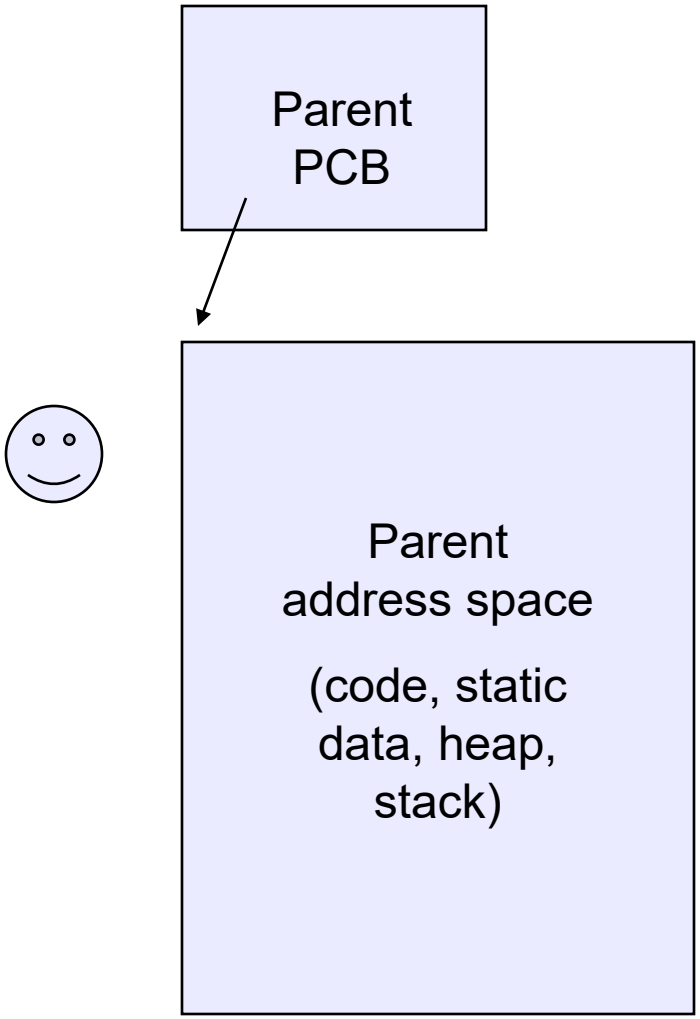
```
spinlock% gcc -o testparent testparent.c
spinlock% ./testparent
My child is 486
Child of testparent is 0
spinlock% ./testparent
Child of testparent is 0
My child is 571
```

exec() vs. fork()

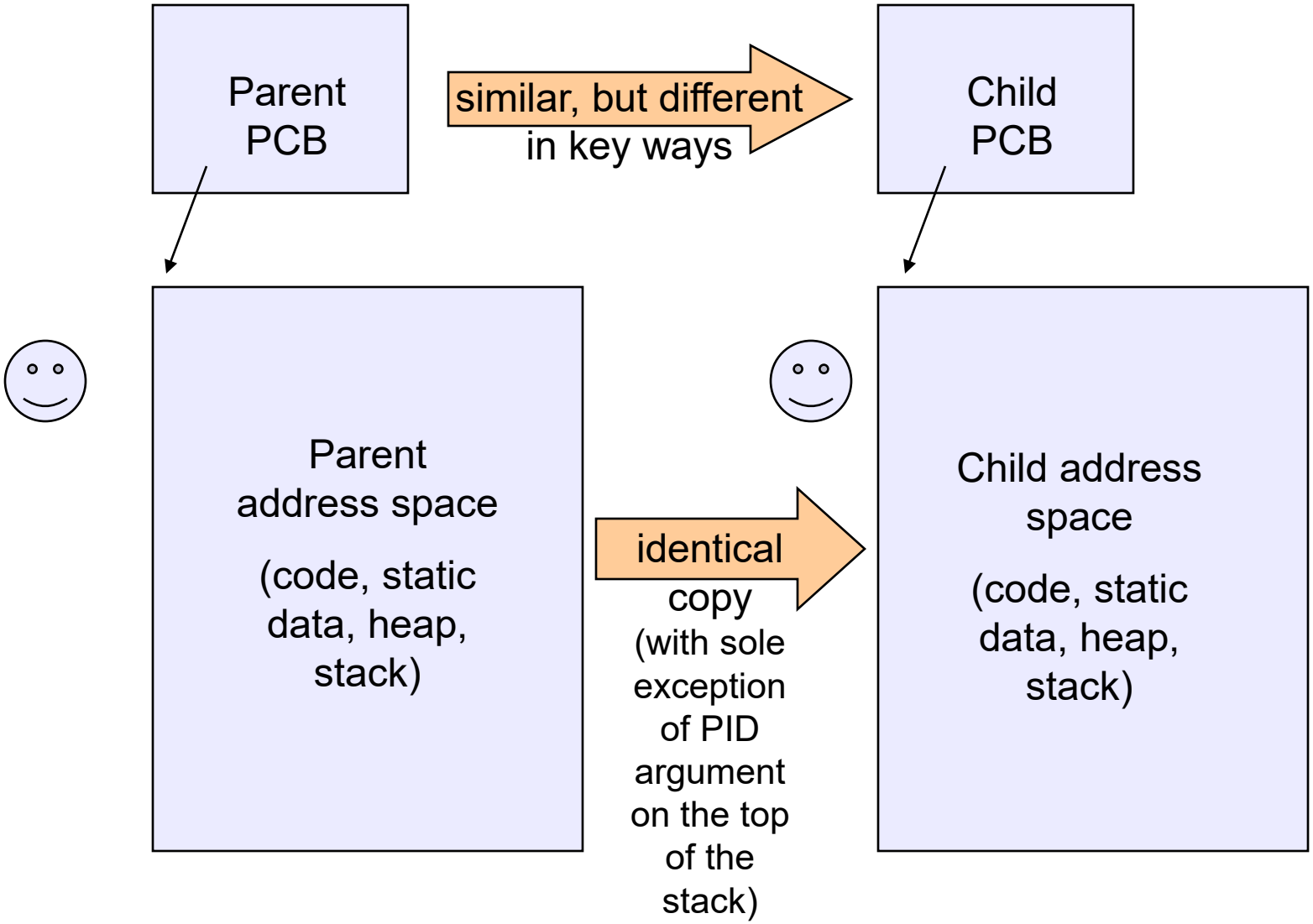
- Q: So how do we start a new program, instead of just forking the old program?
- A: First fork, then exec
  - `int exec(char * prog, char * argv[])`

- **`exec()`**
  - note: *does not create a new process!*
    - use fork() for that
  - the current process is blocked
    - made non-runnable
    - not on any list (but is "remembered" as part of OS code path functioning)
  - program 'prog' is loaded into the already existing address space
    - i.e., over-writes the existing process image
  - initializes hardware context, args for new program
    - sets PC to entry point, sets SP to bottom of empty stack
    - address space info remains unchanged
  - places existing PCB onto ready queue

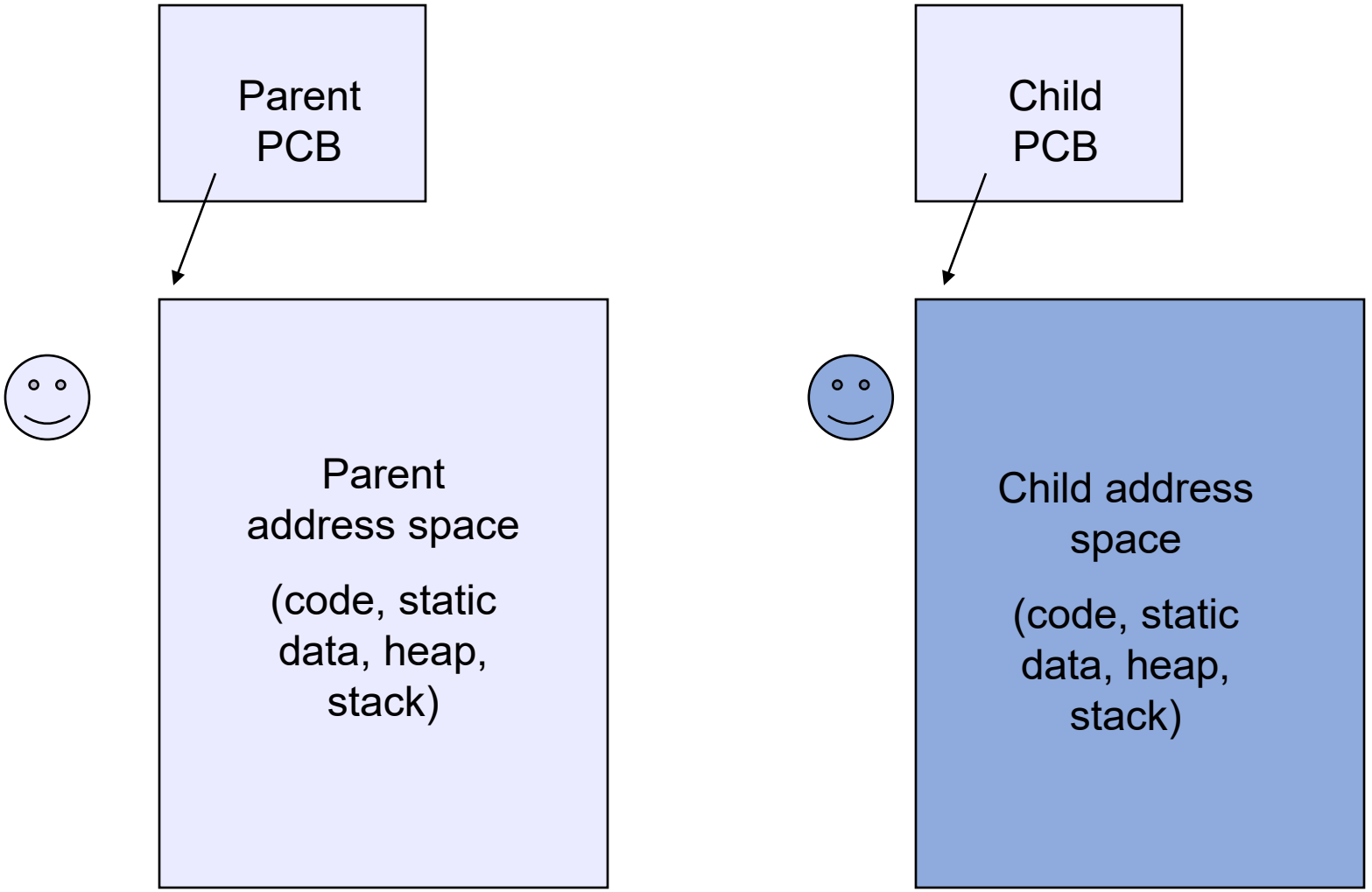
- So, to run a new program:
  - fork()
  - Child process does an exec()
  - Parent either waits for the child to complete, or not


Parent
PCB
Parent
address space
(code, static
data, heap,
stack)

$ ./myProgram


Parent PCB
similar, but different in key ways
Child PCB
Parent address space
(code, static data, heap, stack)
identical copy
(with sole exception of PID argument on the top of the stack)
Child address space
(code, static data, heap, stack)

$ ./myProgram



After *exec* of
./myProgram

$ ./myProgram

Making process creation faster

- The semantics of fork() say the child's address space is a copy of the parent's

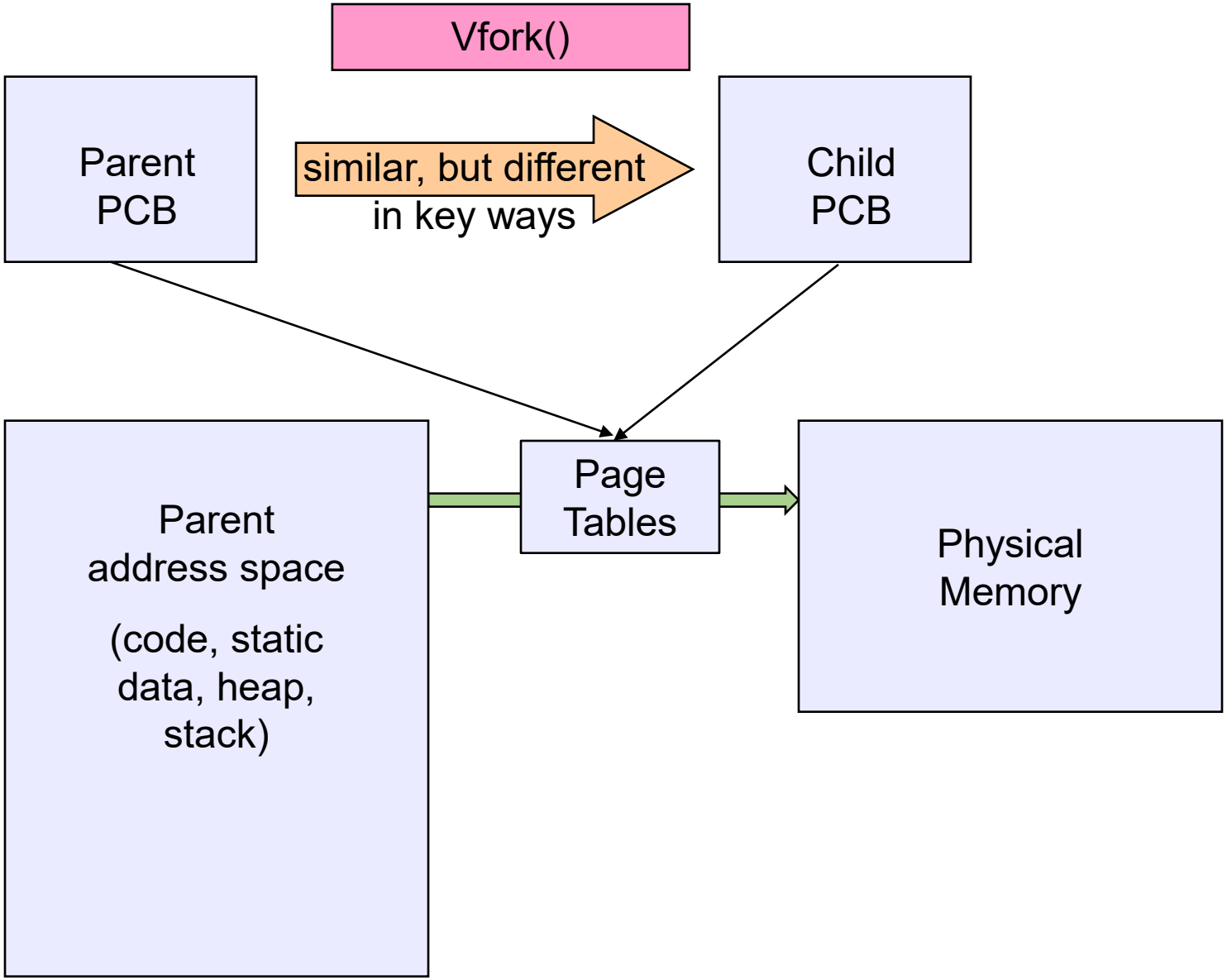
- Implementing fork() that way is slow
  - Have to allocate physical memory for the new address space
  - Have to set up child's page tables to map new address space
  - Have to copy parent's address space contents into child's address space
    - Which you are likely to immediately blow away with an exec()

Method 1: vfork()

- vfork() is the older (now uncommon) of the two approaches we'll discuss

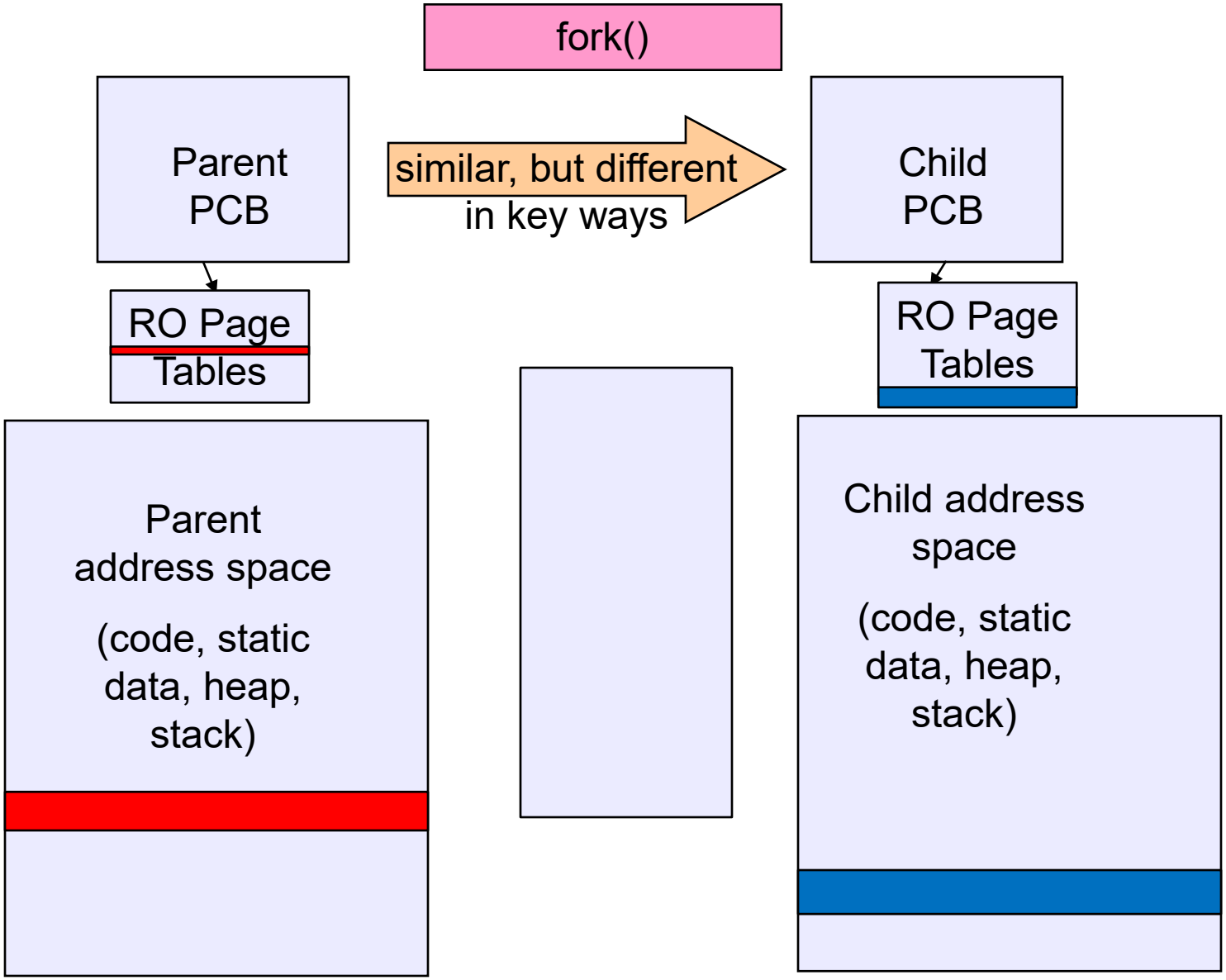
- Instead of "child's address space is a copy of the parent's," the semantics are "child's address space *is* the parent's"
  - Only sometimes works
  - The sometimes is a pretty common case

- Forking process has to "promise" that its code won't won't modify the address space in the child before doing before doing an exec()
    - Unenforced! You use vfork() at your own peril
- When exec() is called, a new address space is created and it's loaded with the new executable
  - Saves wasted effort of duplicating parent's address space, just to immediately overwrite it in child
  - Parent is blocked until execve() is executed by child


Vfork()
Parent
PCB
similar, but different
in key ways
Child
PCB
Parent
address space
(code, static
data, heap,
stack)
Page
Tables
Physical
Memory

Method 2: copy-on-write (COW)

- 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.


fork()
Parent PCB
similar, but different in key ways
Child PCB
RO Page Tables
RO Page Tables
Parent address space
(code, static data, heap, stack)
Child address space
(code, static data, heap, stack)

What If Processes Want to Cooperate

- Processes provide isolation (protection) – great!
- But sometimes you want processes to communicate / cooperate
- How can one process “provide input” to another?
- Inter-process communication (IPC)

What If Processes Want to Cooperate

- Processes provide isolation (protection) – great!
- But sometimes you want processes to communicate / cooperate
- How can one process "provide input" to another?

1. command line arguments (argv values)
   - available only to parent process
2. communicate through files
   - one writes and the other reads
3. optimize that: pipes
   - use memory buffers, not files
   - We'll see that this works only if the processes are related (usually siblings)
4. environment variables
   - Why?

IPC (cont).

- Additional mechanisms:
  5. named pipes
     - like pipes, except that unrelated processes can use them
       - need a namespace
         - use file system names
     - man 3 mkfifo
  6. named shared memory regions
     - shm_open() followed by mmap()
     - “cut out the middle man”
  7. **sockets / Internet protocols**
     - robust – prepared to communicate using a heavyweight middle man!
     - optimized when endpoints are on the same machine

IPC: “exception handling”

- Processes can register event handlers
  - Feels a lot like event handlers in Java, which ..
  - Feel sort of like catch blocks in Java programs
  - sigaction()
- When the event occurs, OS causes process to jump to event handler routine
- This is very similar to the exception mechanism used by the OS
  - For the OS, the hardware is the agent that causes the asynchronous jump to the OS’s event handler
  - For the application, the OS is the agent
- Policy/mechanism separation
  - event detection by the OS
  - lets the application do something in response other than the default built into the OS
    - Why is there a default?

IPC: signals

- The “exception mechanism” at this level is usually called “signal handling”
- A “signal is delivered to the application” to when an event occurs
- Signals can be generated by code, including code in other processes
- So, signals are an elementary form of IPC
  - signal is generated by another process
  - send signal using kill (man 2 kill)
  - Only argument of the communication is a single int, the signal number

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

Unix Shells

- Shells are just user-level programs
  - What's that mean?
- They're mainly oriented towards launching other programs
  - Using fork() / exec()
- They typically have few "built-in" commands
  - ls, cat, etc. are executables, opaque to the shell
  - (What must be built in?)
- Shells usually offer ways to build "shell scripts"
  - E.g., some looping construct
  - You can view everything you type into a shell as a program that is being simultaneously created and executed
    - This is why programmers have traditional liked shells and users have traditionally liked clicking

UNIX shells – basic operation

```
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);
    }
  }
}
```

Unix Shells: Jobs / Redirection

- Shells usually offer ways to make “jobs” – assemblages of executions
  - ls | grep *.c | less
  - pushd sub && make && popd
  - pushd sub; make; popd
- One way the shell helps you compose jobs is by input-output redirection
  - You can make the output of one program the input of another, without ever writing to a file

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

Linux Job Control

- A “job” is an assemblage of processes
  - $ cat main.c
  - $ cat main.c | grep –w total | less

- Key concepts:
  - Controlling terminal
    - Follow parent PIDs up to “the top”
    - What is that processes stdin/stdout/stderr connected to?
    - Why does it matter?

  - Session
    - A way to group things that should be terminated if the controlling terminal goes away
    - “Session leader” – process that created the session
    - Sessions are named by integers
      - Use PID of the session leader
    - A forked process inherits the session of its parent
    - A process can set its own session id (setsid)
      - Unless it’s a “process group leader”

Linux Job Control

- Process Group
  - $ myprog | myotherprog | grep B
  - A process inherits the process group of its parent
  - A process can set its process group (setpgid)
  - The “process group leader” is the process that created the group
  - The process group’s name is an integer
    - The PID of the creating process
- Why have process groups?
  - Ctrl-C sends a SIGINT
  - A signal can be sent to a process group
    - Sent to each process in the group