Operating Systems (INFR09047)

2019/2020 Semester 2

Processes

abarbala@inf.ed.ac.uk

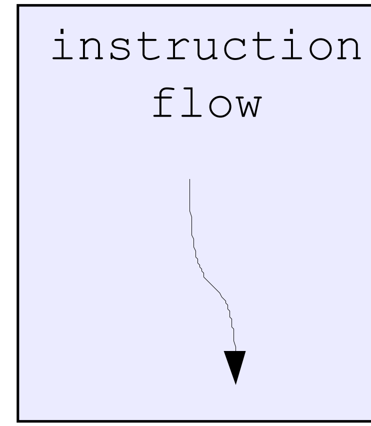
Chapter 3

Overview

- Process
- Process control block
- Process state
- Context switch
- Process creation and termination
- Interprocess communication

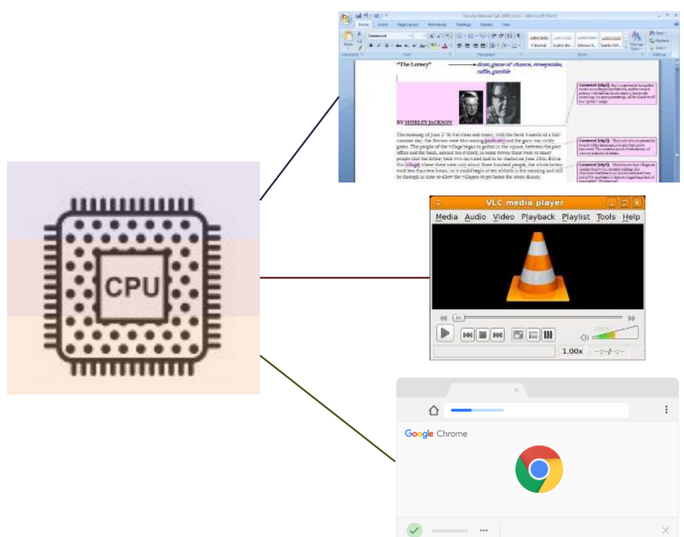
What is a "process"?

- **Process** is the OS's abstraction for **execution**
  - **Program** is the list of instructions, initialized data, etc.
  - A process is a **program in execution**
- **(Sequential) Process**
  - A **single flow/sequence** of instruction in execution
    - Process (an abstraction of the CPU)
  - An address space (an abstraction of memory)



address space

Only **one** process running on a processor core at any instant

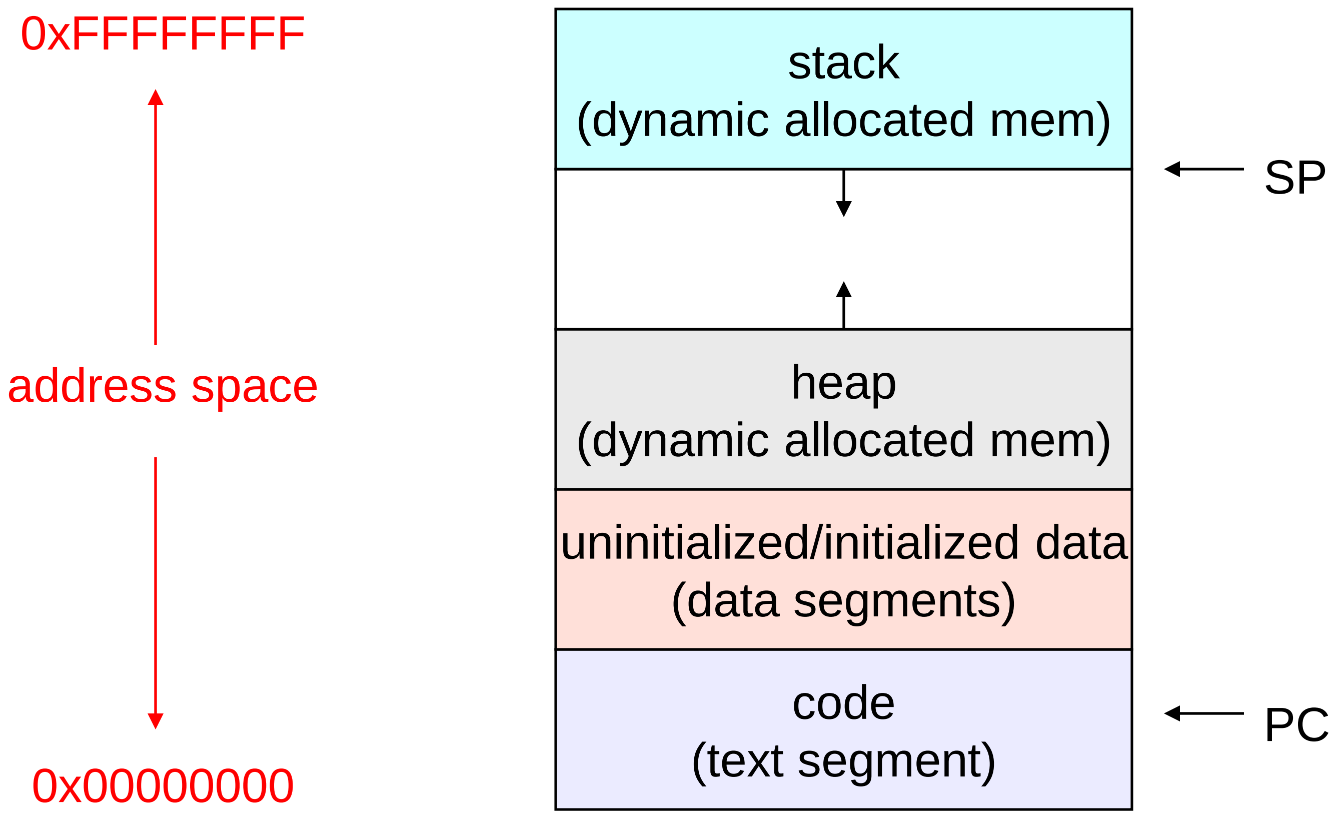


Different processes may run the **same** program

What’s “in” a process?

- A process consists of (at least)
  - An **address space**, containing
    - Code (instructions) for the running program
    - Data for the running program (static data, heap data, stack)
  - A **CPU state**, consisting of
    - Program counter (PC), indicating the next instruction
    - Stack pointer, current stack position
    - Other general-purpose register values
  - A set of **OS resources**
    - Open files, network connections, sound channels, …
- In other words, everything needed to run the program
  - or to re-start, if interrupted

A Process's Address Space (32bit)



The OS Process Namespace

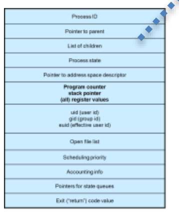
- Each process is identified by a process ID (PID)
  - An integer
- The PID namespace is global to the system
  - Only one process at a time has a specific PID
- Operations that create processes return a PID
  - E.g., **fork()**
- Operations on processes take PIDs as an argument
  - E.g., **kill(), wait(), nice()**
- May differ based on the specific operating system

Representation of Processes by the OS

- OS maintains a data structure to keep **track** of a process’s state
  - **Process control block** (**PCB**), or **process/task 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 state is transferred out of the hardware into the PCB
- … and when the process **is running?**
  - Its state is spread between the PCB and the CPU

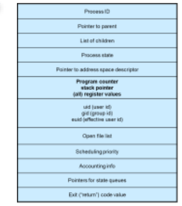
The PCB

- The PCB is a data structure with **many, many** fields
  - process ID (PID)
  - parent process ID
  - execution state
  - program counter, stack pointer, registers
  - address space info
  - UNIX user id, group id
  - scheduling priority
  - accounting info
  - pointers for state queues
- In Linux (stable 5.4.14)
  - defined in **`task_struct`** ([`include/linux/sched.h`](include/linux/sched.h))
  - more than **100 fields!**



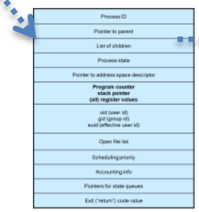
Process ID

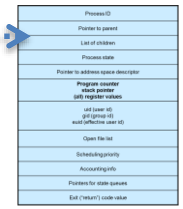
Pointer to parent



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 a PCBs looks like inside

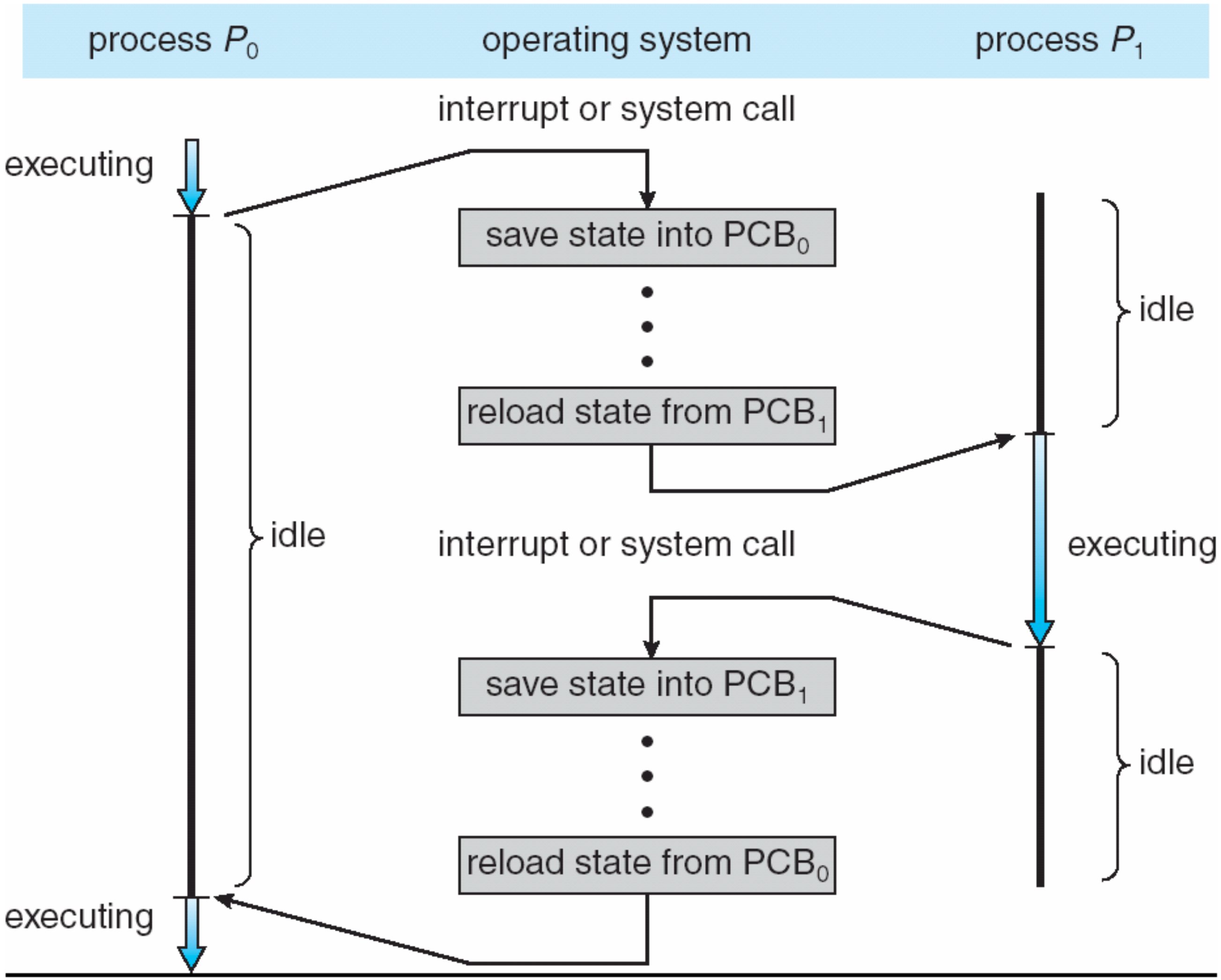
PCBs and CPU state #1

- When a process is running, its **CPU state is inside the CPU**
  - PC, SP, registers
  - CPU contains current values
- When the OS gets control because of a …
  - Syscall: 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

PCBs and CPU state #2

- 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/second
  - Takes a few microseconds on today's hardware
  - Still expensive relative to thread-based context switches***
- Choosing **which process to run next** is called **scheduling**

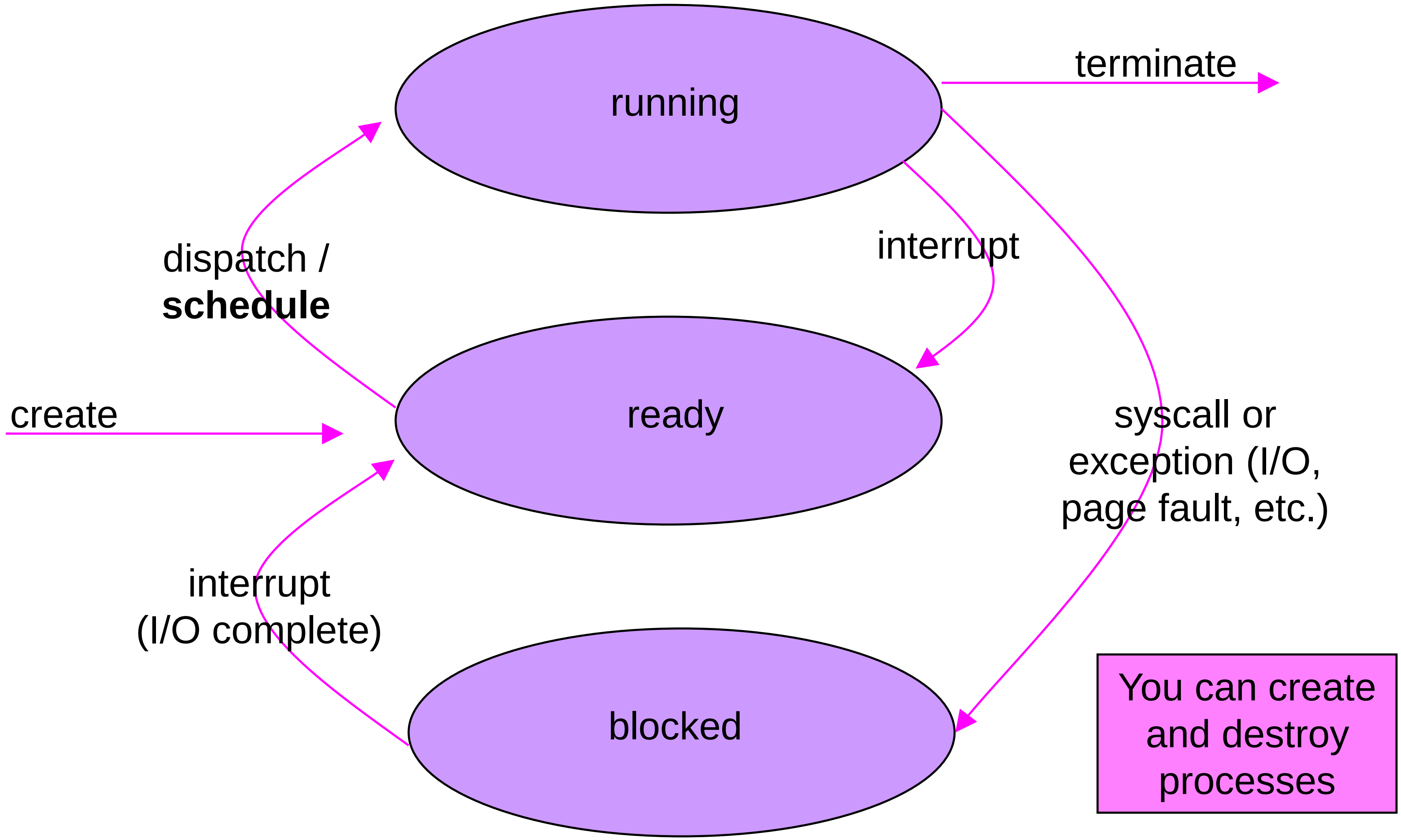
Process context switch



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 **`top`**, STAT column shows current state
  - UNIX: run **`ps`**
  - *Which state is a process in most of the time?*

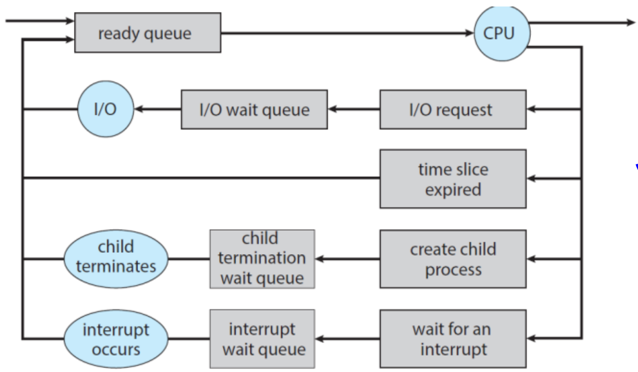
Process States and State Transitions

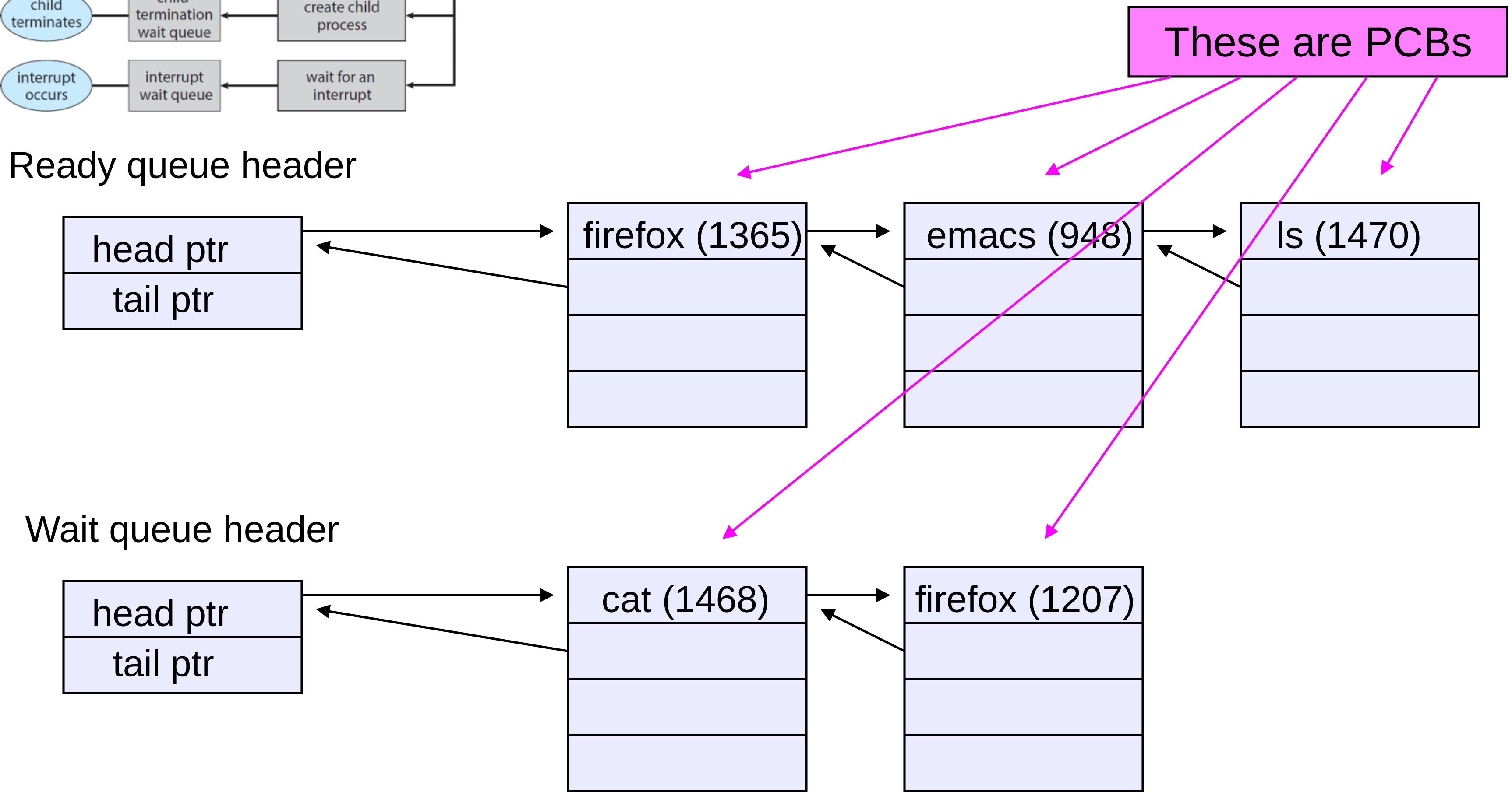


State Queues #1

- 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, …
    - But there maybe multiple waiting queues
  - 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
- The PCBs are moved between queues, which are implemented as linked lists

State Queues #2





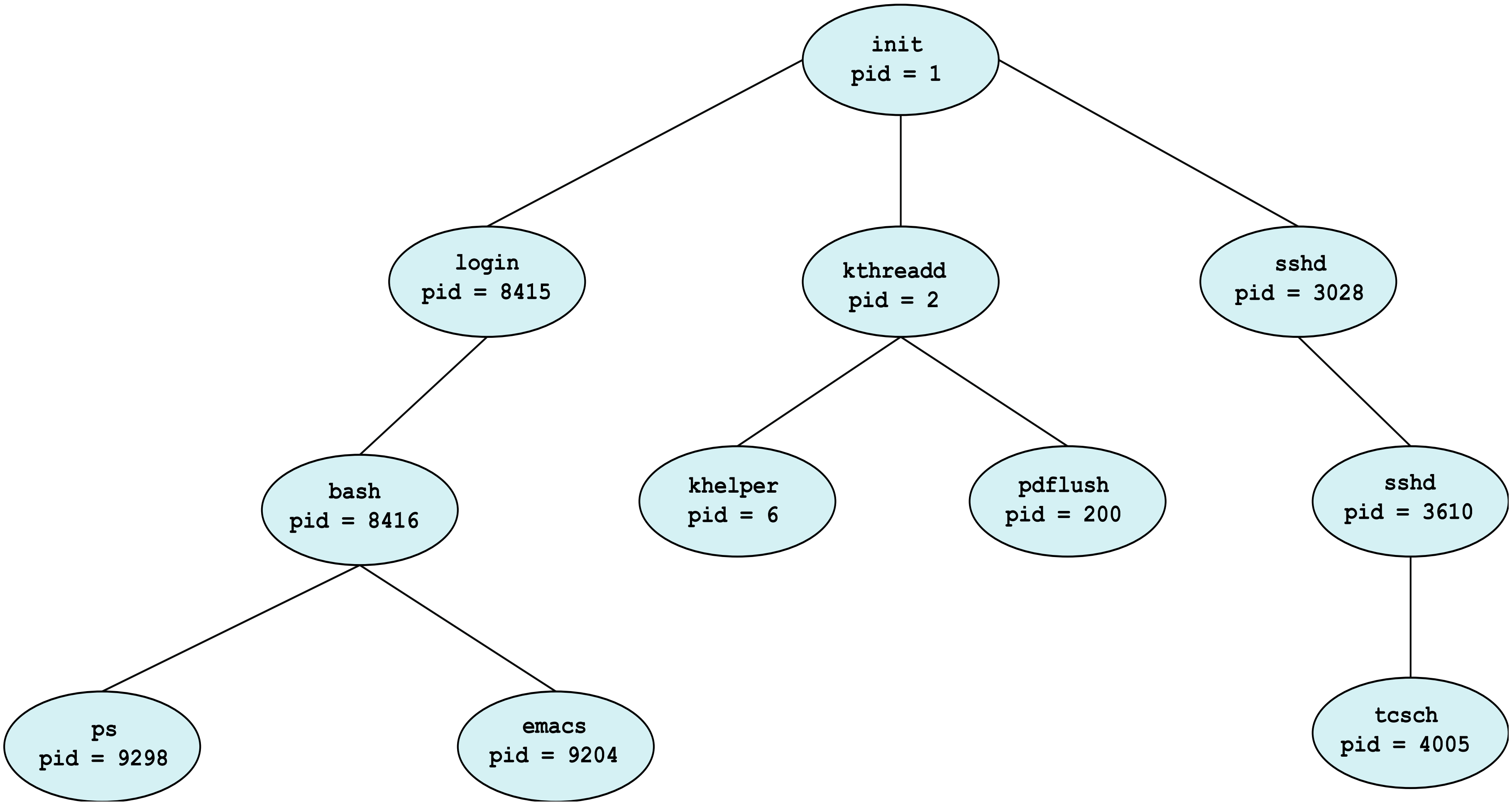
- There may be **many wait queues**, one for each type of wait (specific device, timer, message, …)

PCBs and State Queues

- PCBs are **data structures**
  - Dynamically allocated 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 -ef`**, look for PPID field
  - *What creates the first process, and when?*

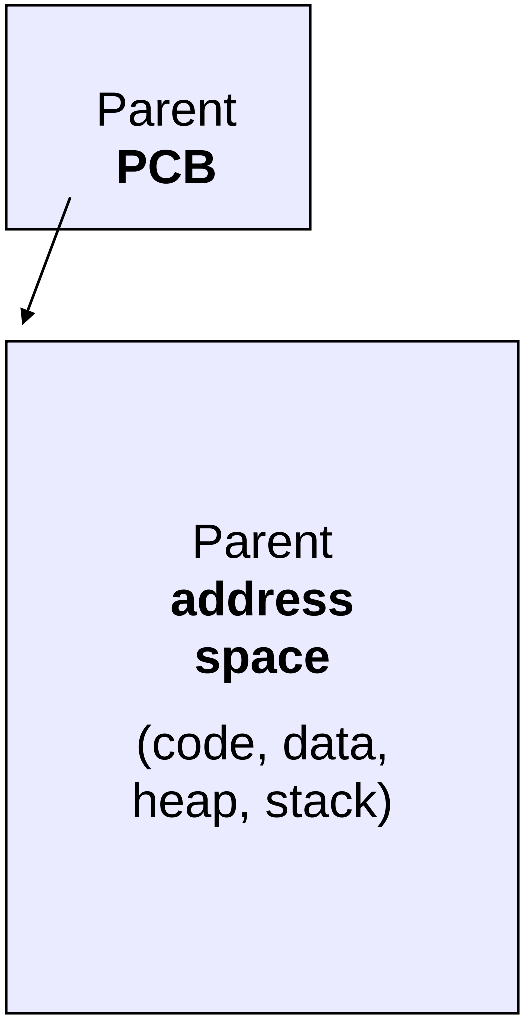

init
pid = 1
login
pid = 8415
kthreadd
pid = 2
sshd
pid = 3028
bash
pid = 8416
khelper
pid = 6
pdflush
pid = 200
sshd
pid = 3610
ps
pid = 9298
emacs
pid = 9204
tcsch
pid = 4005

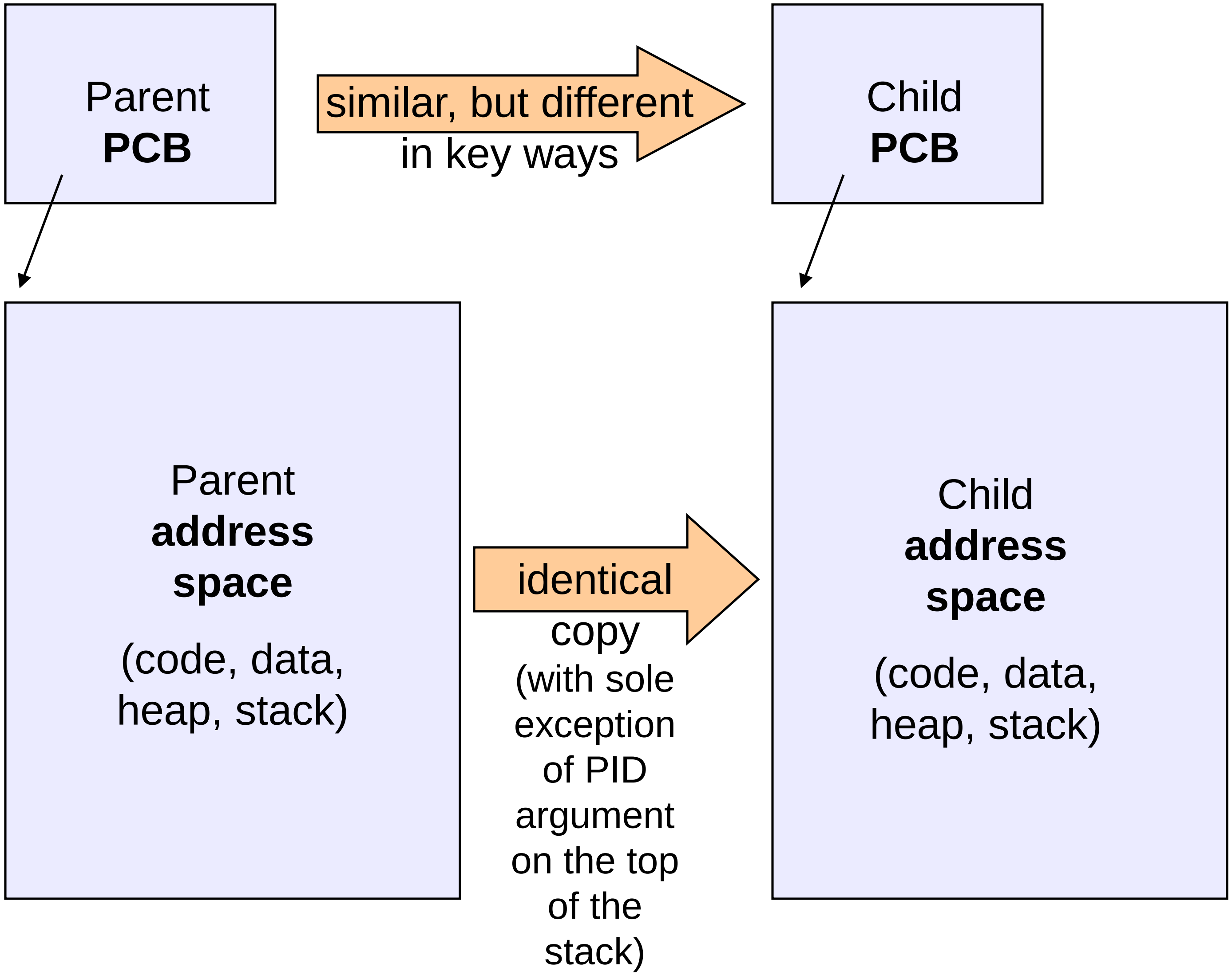
Process Creation Semantics

- (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
    - returns the child's PID to the parent
    - returns 0 to the child
- **`fork()`** = "clone me"


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


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

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) { /* error */
    printf("Error\n");
    return 1;
  } else if (pid > 0) {/* parent */
    printf("Child of %s is %d\n", name, pid);
    return 0;
  } else {/* child */
    printf("My child is %d\n", pid);
    return 0;
  }
}
```

testparent output

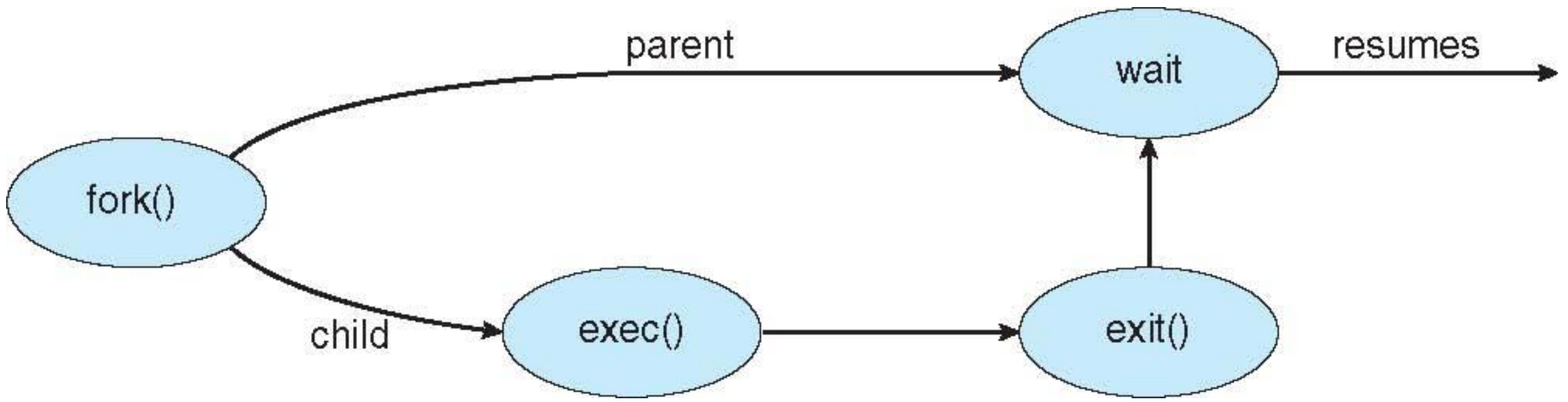
```
spinlock% gcc -o testparent testparent.c
spinlock% ./testparent
My child is 486
Child of testparent is 0
spinlock% ./testparent
Child of testparent is 0
My child is 571
```

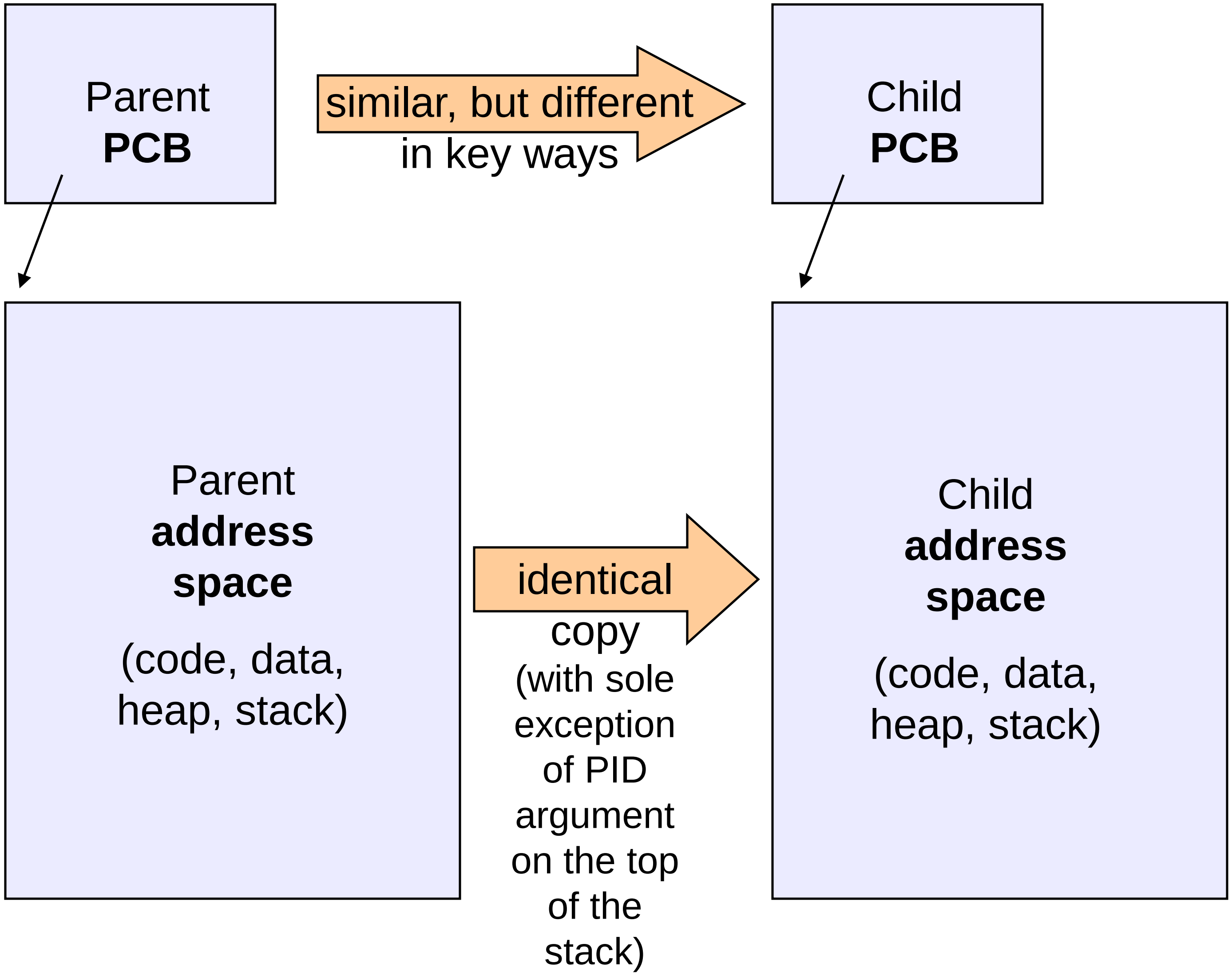
exec() vs. fork()

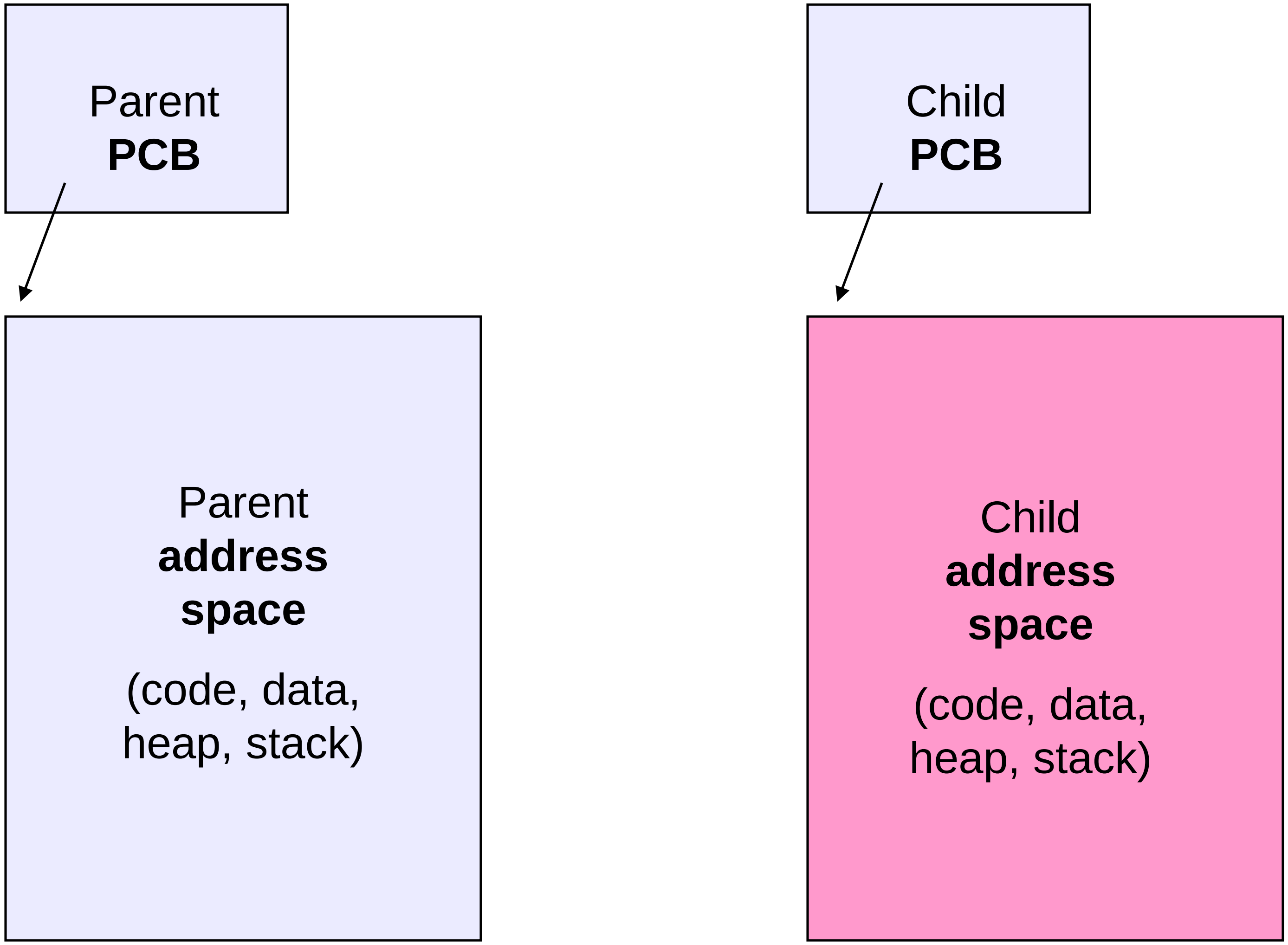
- **Q**: So how do we start a new program, instead of just forking the old program?
- **A**: First **fork**, then **exec**
  - **`int exec(char * prog, char * argv[])`**

- **`exec()`**
  - 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 PCB onto ready queue
  - Note: <u>**does not create a new process!**</u>

exec() and fork()




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


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

Making process creation **faster**

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

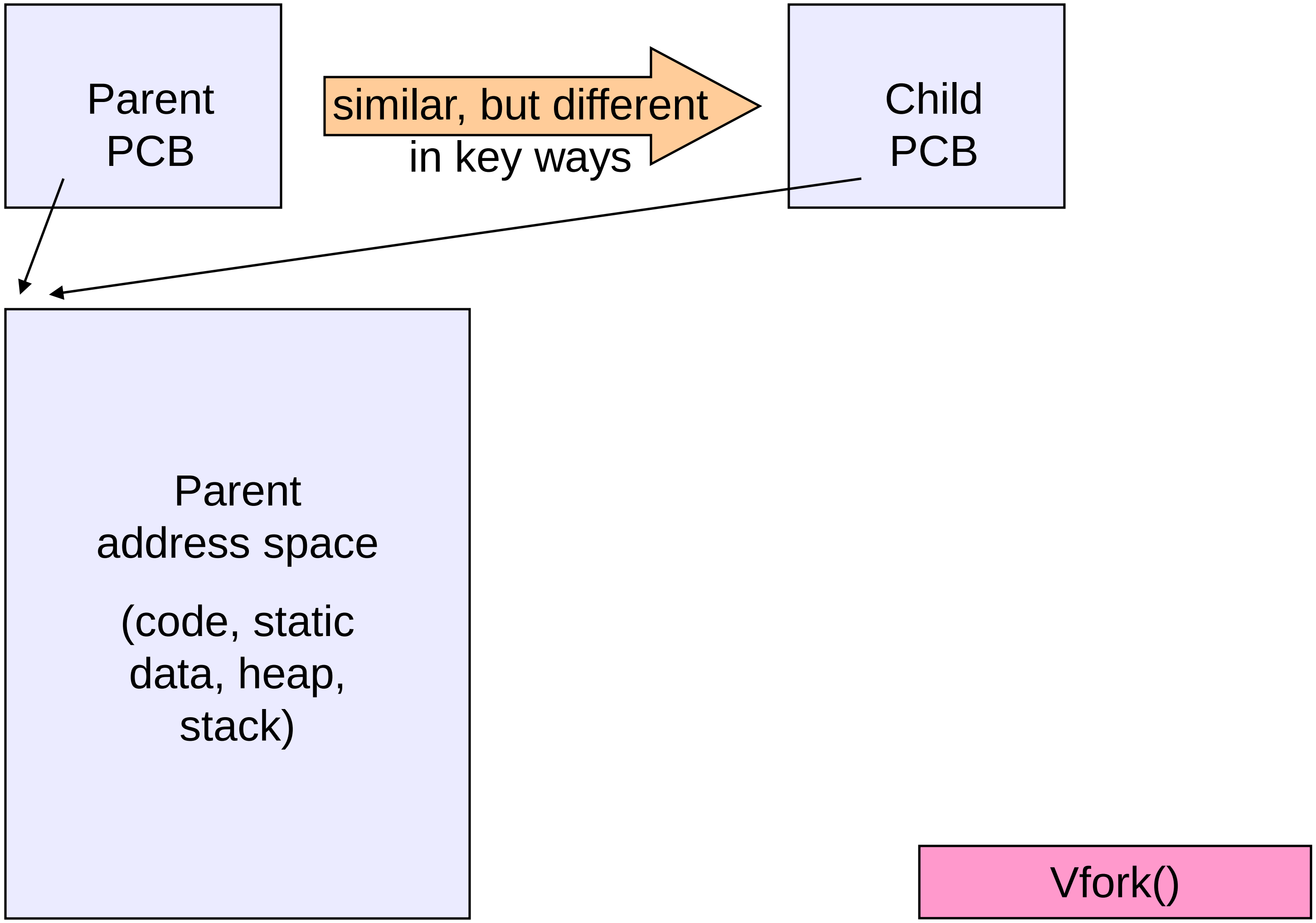
- Implementing fork() that way **is slow**
  - Have to **allocate** physical memory for the new address space
  - Have to **set up** child's page tables to map new address space
  - Have to **copy** parent's address space contents into child's address space
    - Which you are likely to destroy with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

Method 1: **vfork()**

- **vfork()** is an older way (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

Book @ Chapter 10.3



Book @ Chapter 10.3

Method 2: **Copy-On-Write (COW)**

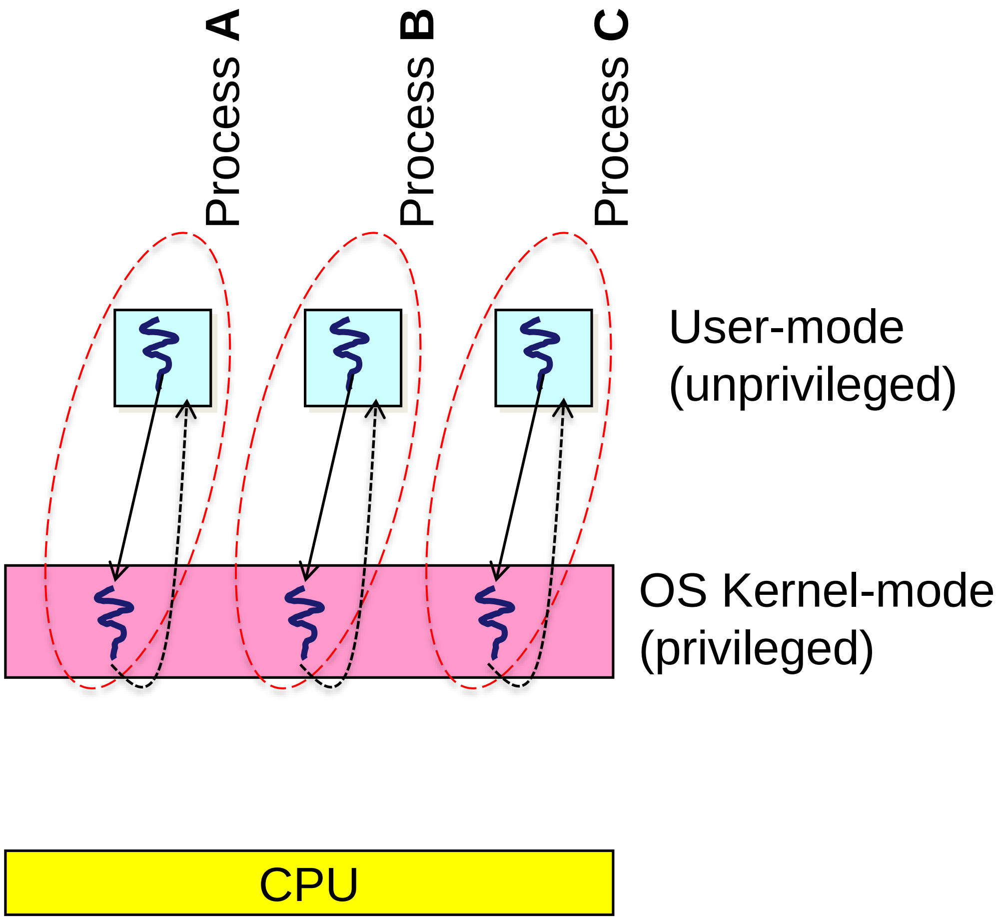
- Retains the original semantics, but **copies "only what is necessary"** rather than the entire address space
- On **fork()**
  - Create a new address space
  - Initialize page tables with same mappings as the parent's (i.e., they both point to the same physical memory)
    - No copying of address space 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.

Book @ Chapter 10.3

Minimal 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);
    }
  }
}
```

Processes and OS kernel

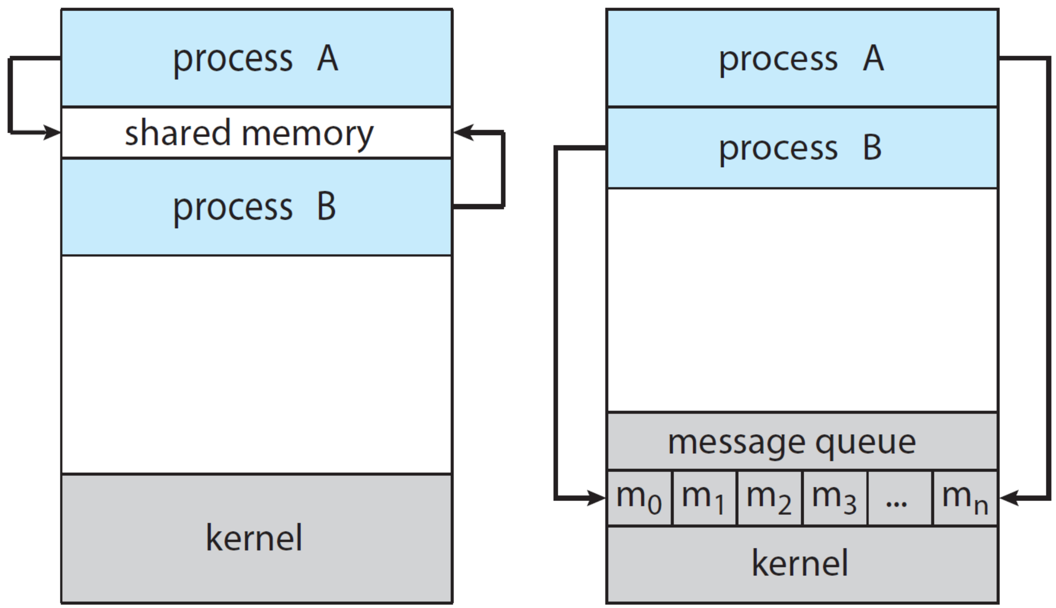


Interprocess Communication

- **Independent** processes

- **Cooperating** processes
  - Information **sharing**
    - More applications interested in the same information
  - Computation **speedup**
    - To exploit parallel hardware
  - **Modularity**
    - Reusability of components
- Require **interprocess communication** (IPC) mechanism to send and receive data
  - Shared memory
  - Message passing

Interprocess Communication



Shared memory

Message passing

Summary

- Process
- Process control block
- Process state
- Context switch
- Process creation and termination
- Interprocess communication