# CMSC421: Principles of Operating Systems

**Nilanjan Banerjee**

*Assistant Professor, University of Maryland*
Baltimore County
nilanb@umbc.edu
http://www.csee.umbc.edu/~nilanb/teaching/421/

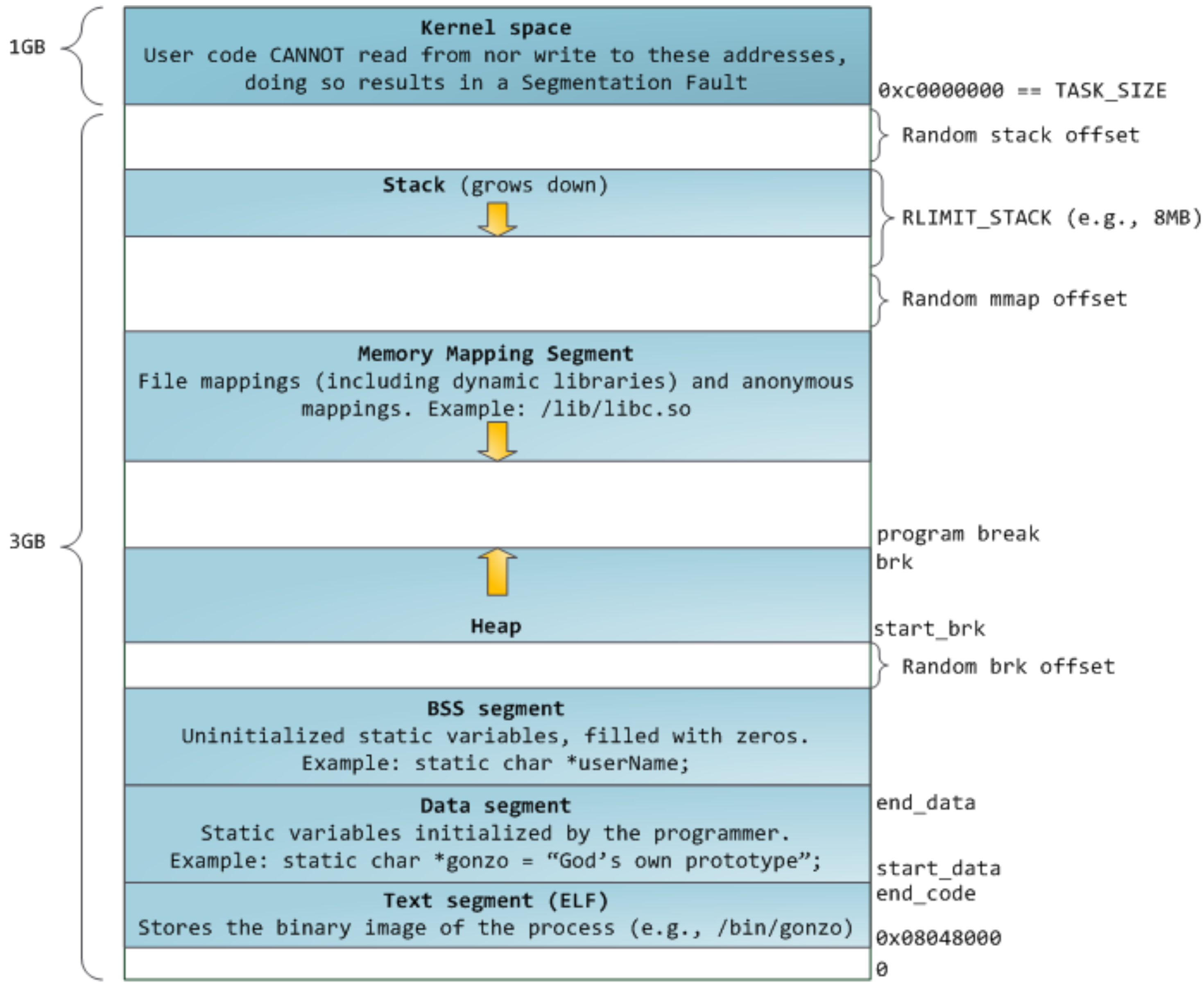
# Announcements

- Project 0 and Homework 1 are due this week
- Readings from Silberchatz [$2^{nd}$ chapter]

# Discussion 2

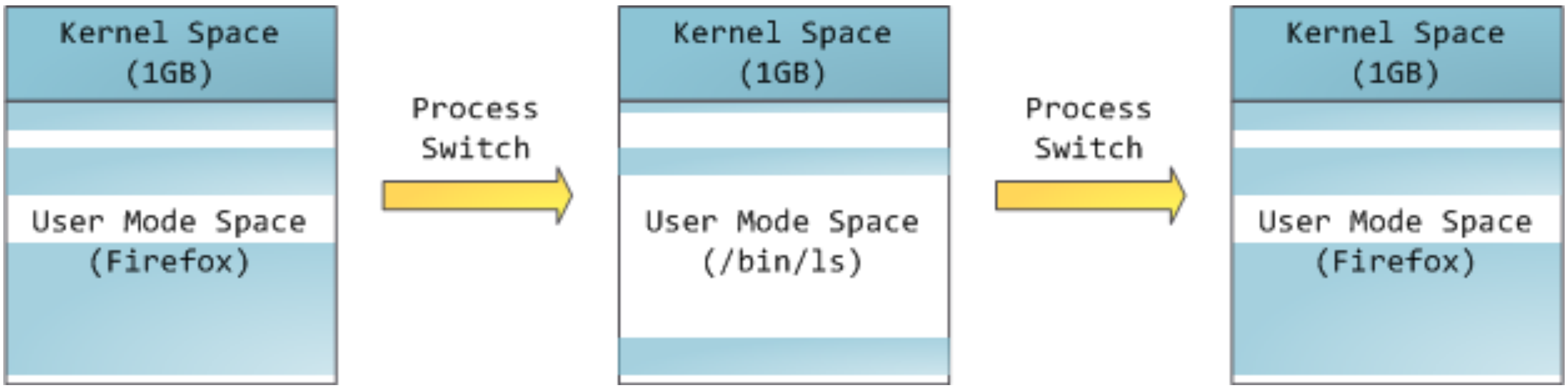
| | | |
|---|---|---|
| Powers of two | 1000 | 8 |
| (Powers of two - 1) | 0111 | 7 |
| N & (N-1) | 0000 | |

# Primer into kernel and user space memory



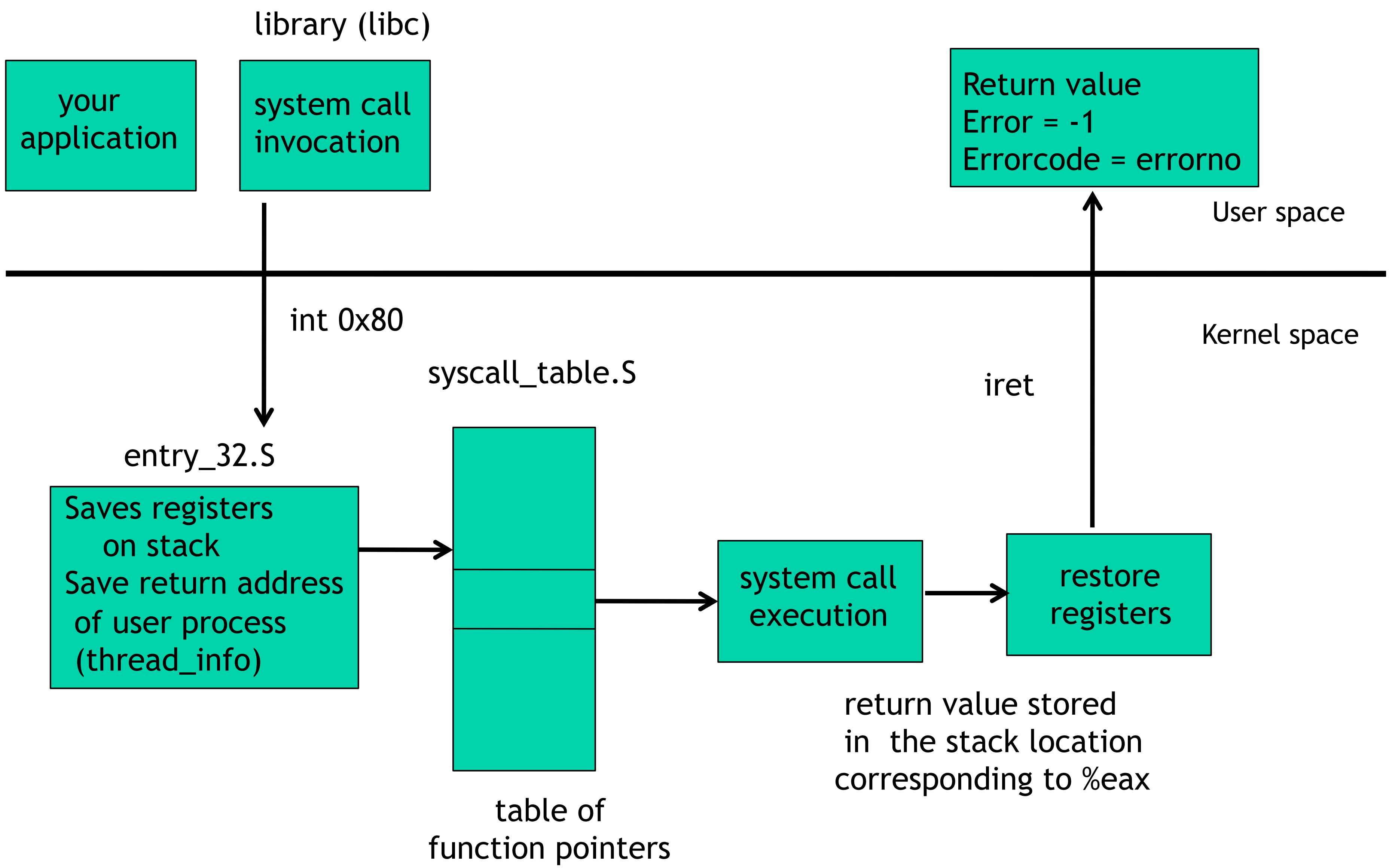

**Acknowledgement: http://duarts.org/gustavo/blog/category/internals**

# Primer into how context switching happens




Acknowledgement: http://duarts.org/gustavo/blog/category/internals

# Flow of control during a system call invocation



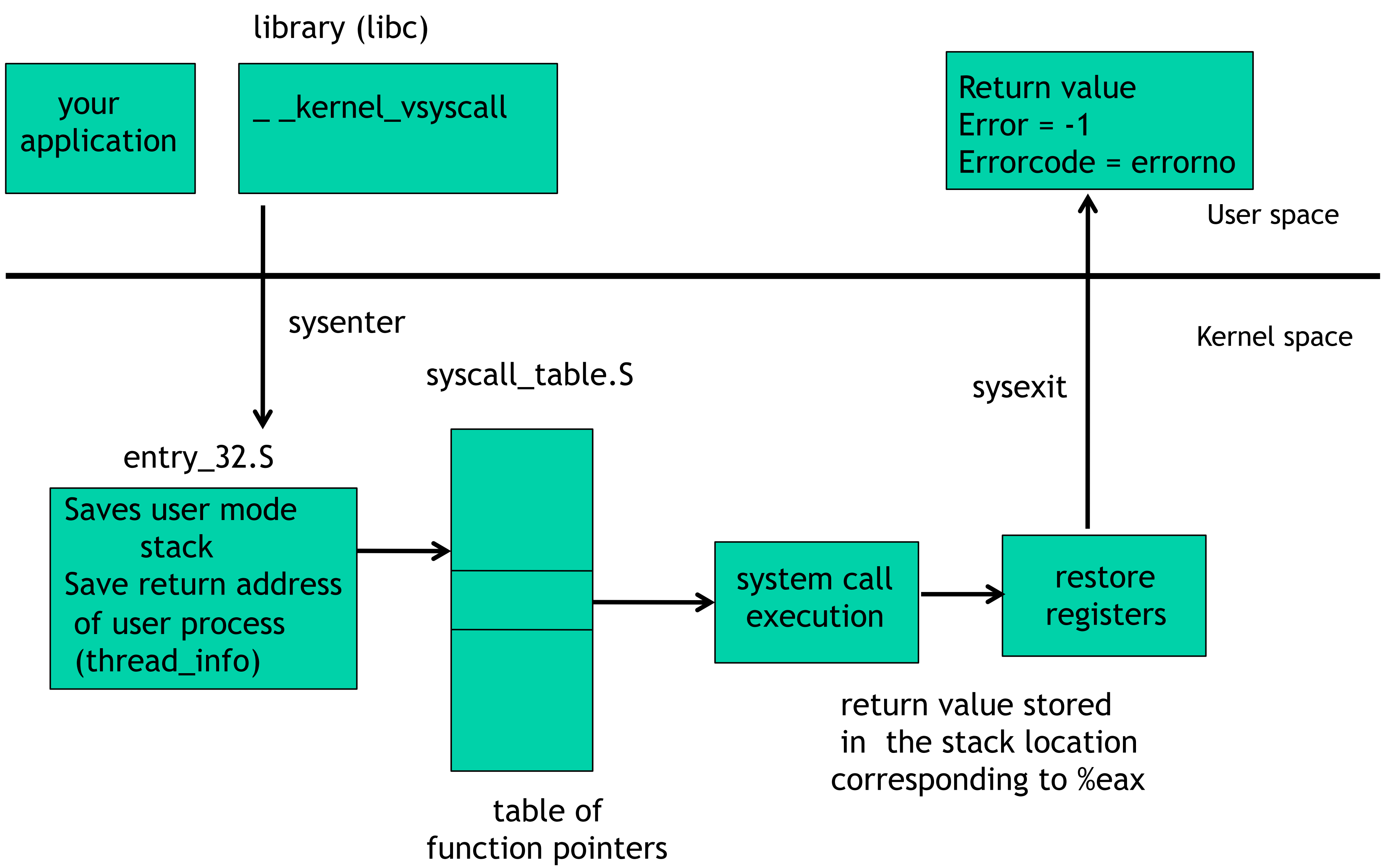
**Kernel dive.**

## Using sysenter/sysexit in Linux > 2.5

- Sysenter/sysexit is also called “Fast system Call”
  - Available in Pentium II +

- Sysenter is made of three registers
  - SYSENTER_CS_MSR -- selecting segment of the kernel code (figuring out which kernel code to run)
  - SYSENTER_EIP_MSR --- address of the kernel entry
  - SYSENTER_ESP_MSR --- kernel stack pointer

# Simplified view of sysenter/sysexit in Linux > 2.5



## Lets write a system call in the kernel (sys_strcpy)

int strcpy(char *src, char *dest, int len)

can return values of
size of at most long? Why?

↓

asmlinkage long sys_strcpy(char *src, char *dest, int len)

↑

compiler directive
params will be read from stack

## Important kernel files/ data structures for system calls

- implementation file for the sys call
  - kernel/sys.c (most of the system calls are implemented)
  - You can implement a system call anywhere

- include/asm-i386/unistd.h
  - Defines the *number* of a system call
  - Defined the total number of system calls.

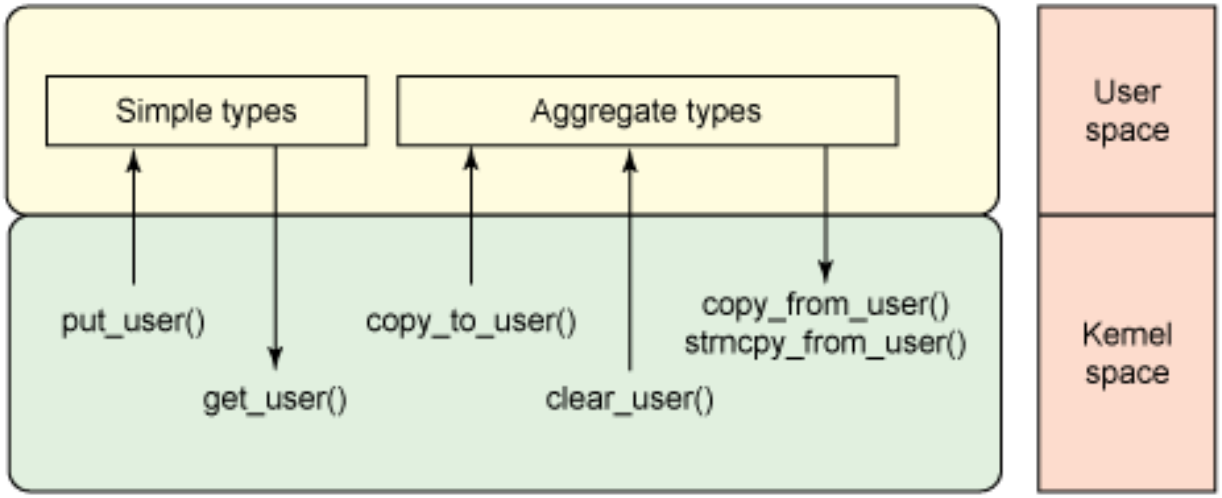
- arch/i386/kernel/syscall_table.S
  - Stores the system call table
  - Stores the function pointers to system call definition

# Issues to think about when writing system calls

- Moving data between the kernel and user process
  - Concerns: security and protection

- Synchronization and concurrency **(will revisit)**
  - Several (so called) kernel threads might be accessing the same data structure that you want to read/write
  - Simple solution (disable interrupts “cli”)
    - Usually not a good idea
  - Big problem in preemptive CPU (which is almost every CPU) and multi-processor systems
    - CONFIG_SMP or CONFIG_PREEMPT

# Useful kernel API functions for bidirectional data movement

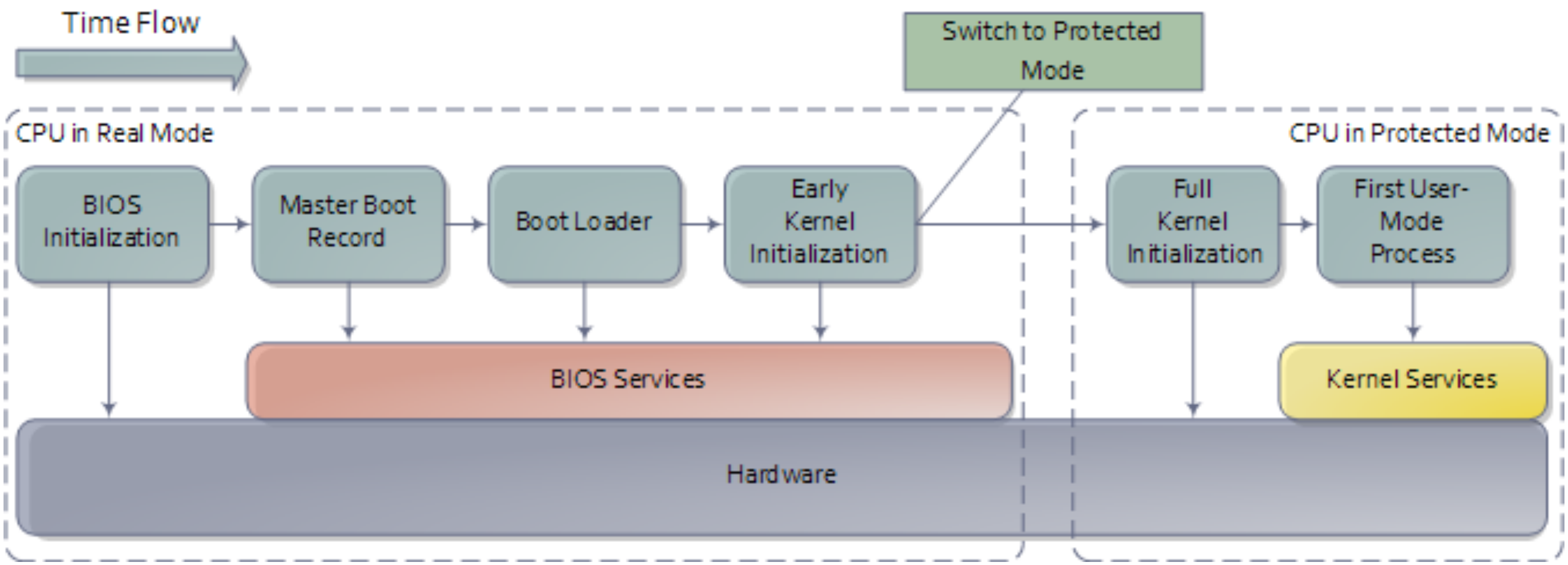



- *access_ok (type, addr, size)*: type (VERIFY_READ, VERIFY_WRITE)
- *get_user(x, ptr)* --- read a char or int from user-space
- *put_user(x, ptr)* --- write variable from kernel to user space
- *copy_to_user(to, from, n)* --- copy data from kernel to userspace
- *copy_from_user(to, from, n)* – copy data to kernel from userspace
- *strnlen_user(src, n)* – checks that the length of a buffer is n
- *strcpy_from_user(dest, src, n)* ---copies from kernel to user space

**Acknowledgement: http://www.ibm.com/developerworks/linux/library/l-kernel-memory-access/index.html**

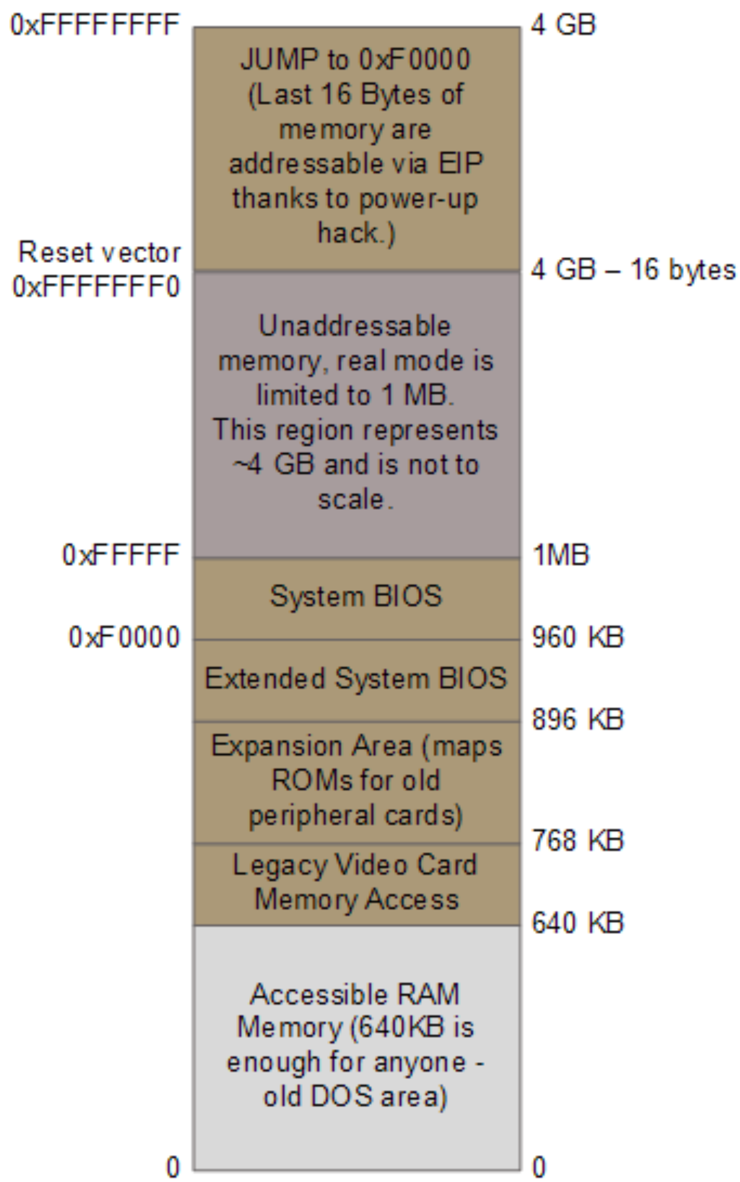
# Bootup Process

BIST (Built in Self Test) → CPU loads BIOS code → BIOS reads MBR and loads Bootloader → Bootloader Starts the kernel




Acknowledgement:http://duartes.org/gustavo/blog/post/how-computers-boot-up

# Memory Organization during bootup

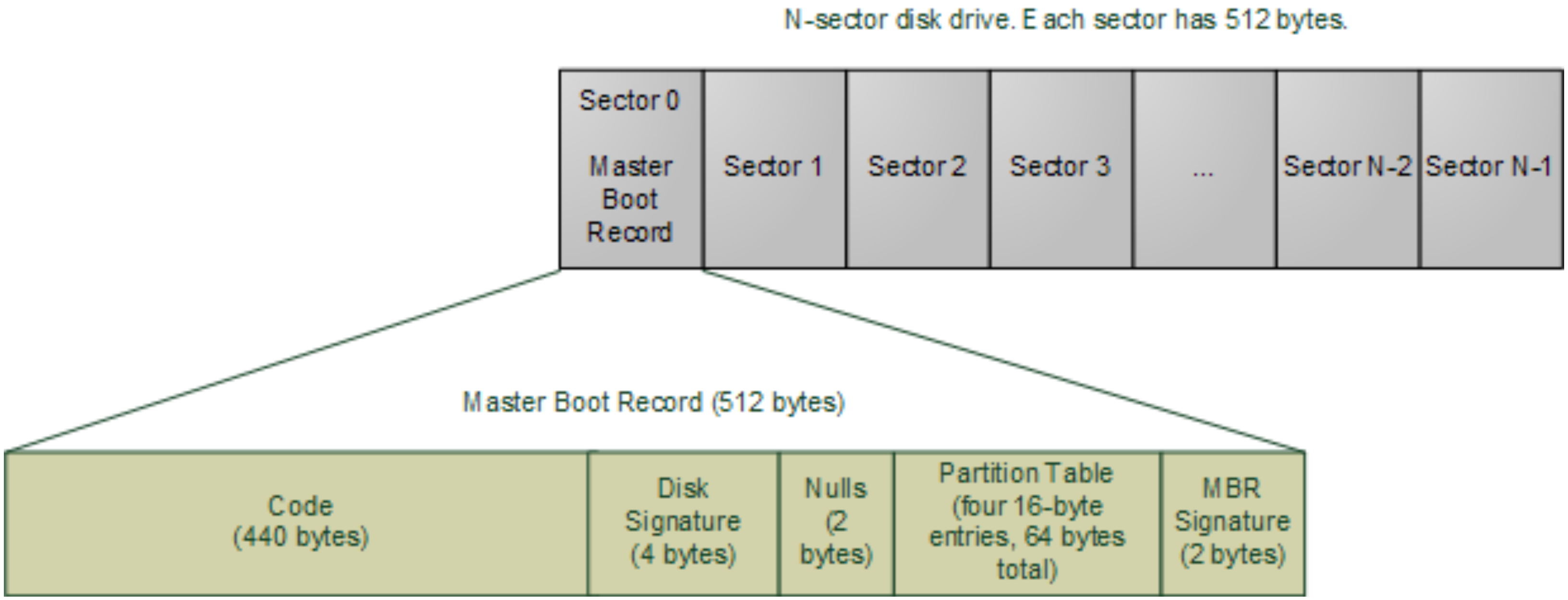



**BIOS load up**

**BIOS load up**

**Acknowledgement:http://duartes.org/gustavo/blog/post/how-computers-boot-up**

# Reading the first disk sector




**Boot loader Stage 1**
**(loads Stage 2)**

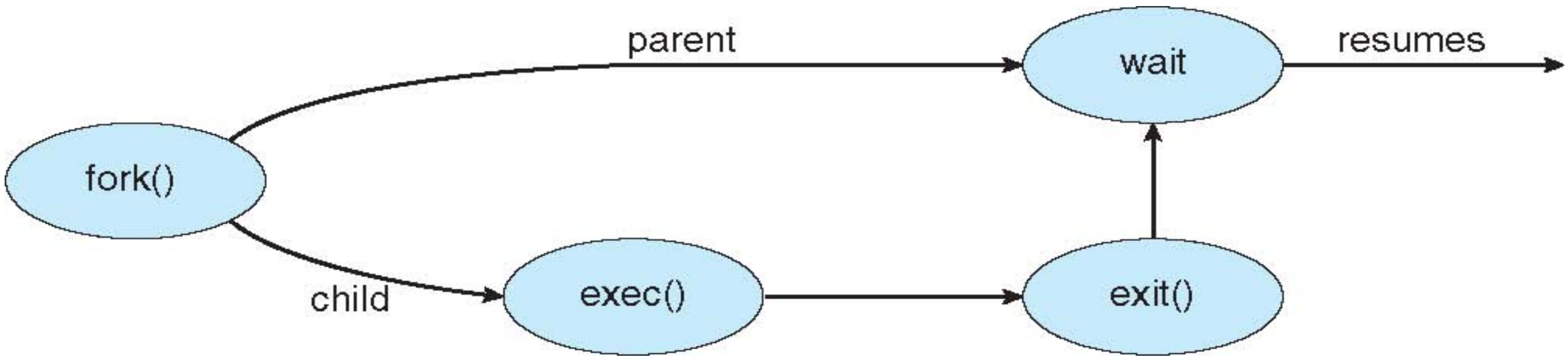
**Boot loader Stage 2**
**(presents users with OS options)**

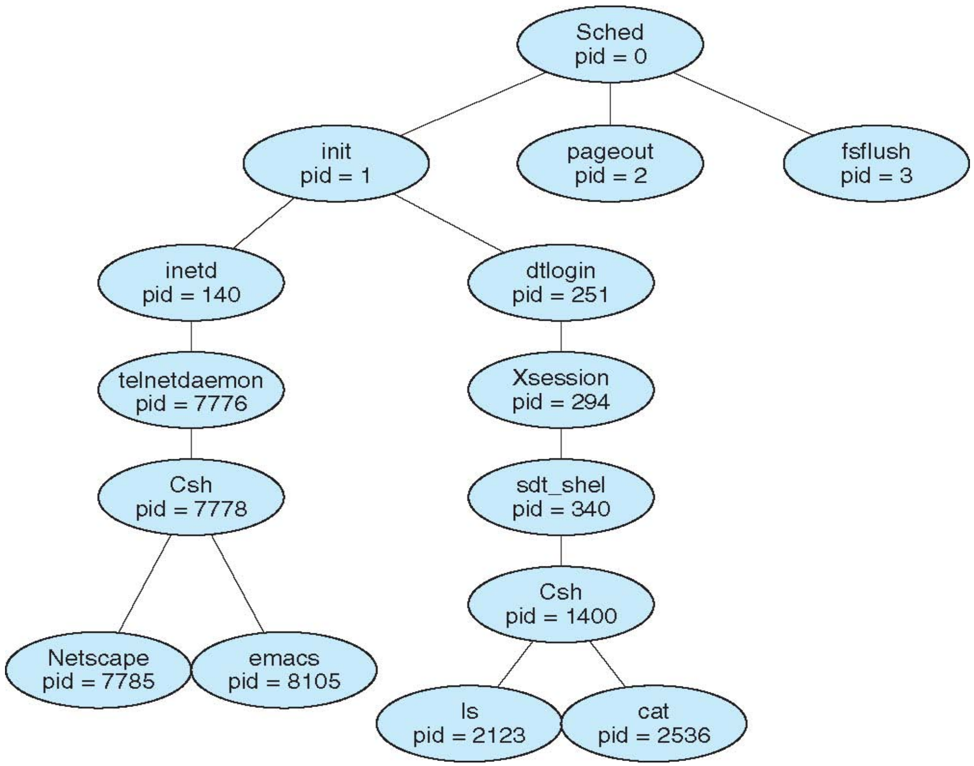
**Boot loader Stage 3**
**(loads the OS)**

**Acknowledgement:http://duartes.org/gustavo/blog/post/how-computers-boot-up**

**Lets take a look at some code**
**(Coreboot, GRUB, Kernel)**

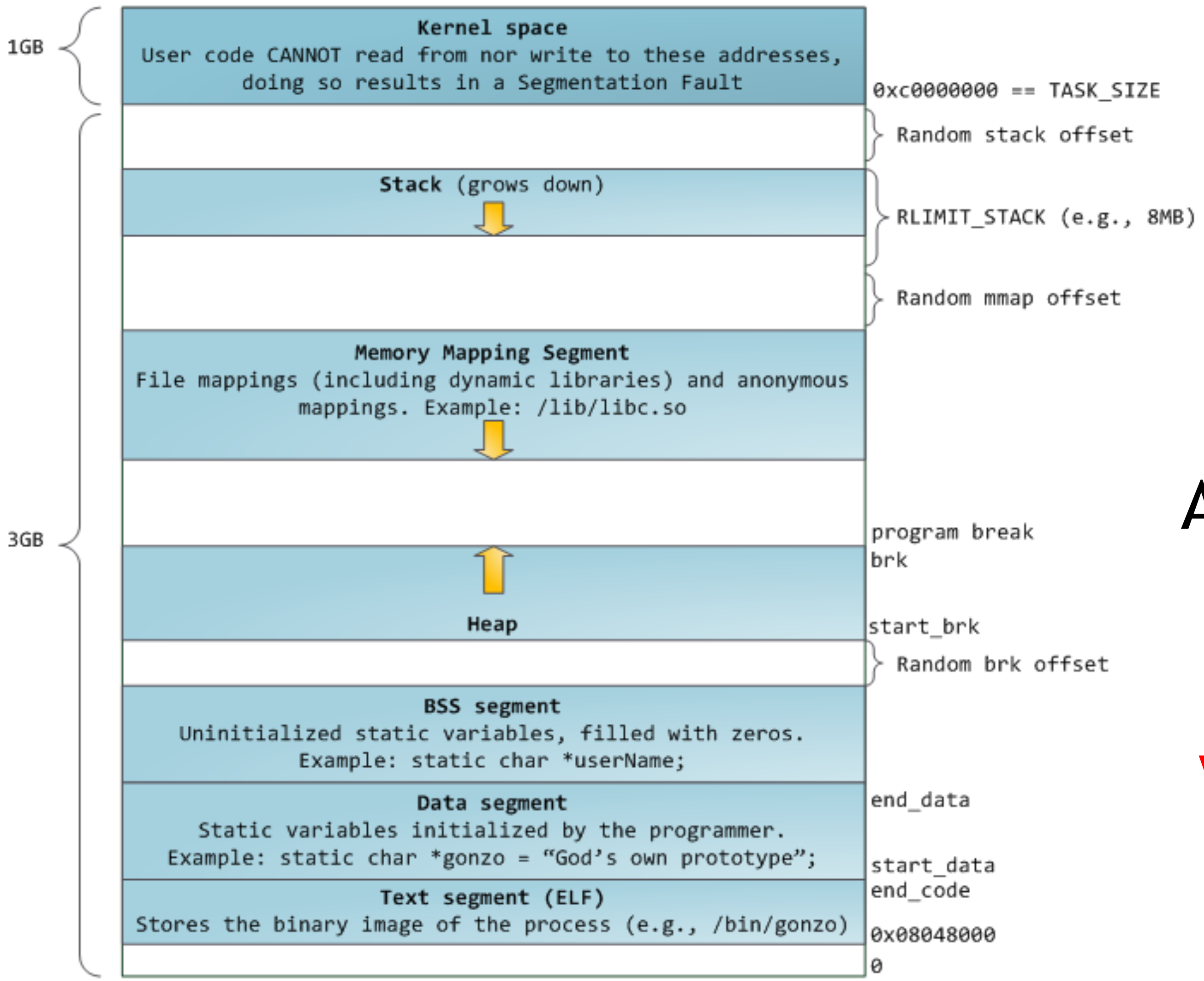
# Creating Processes (fork())




system process tree

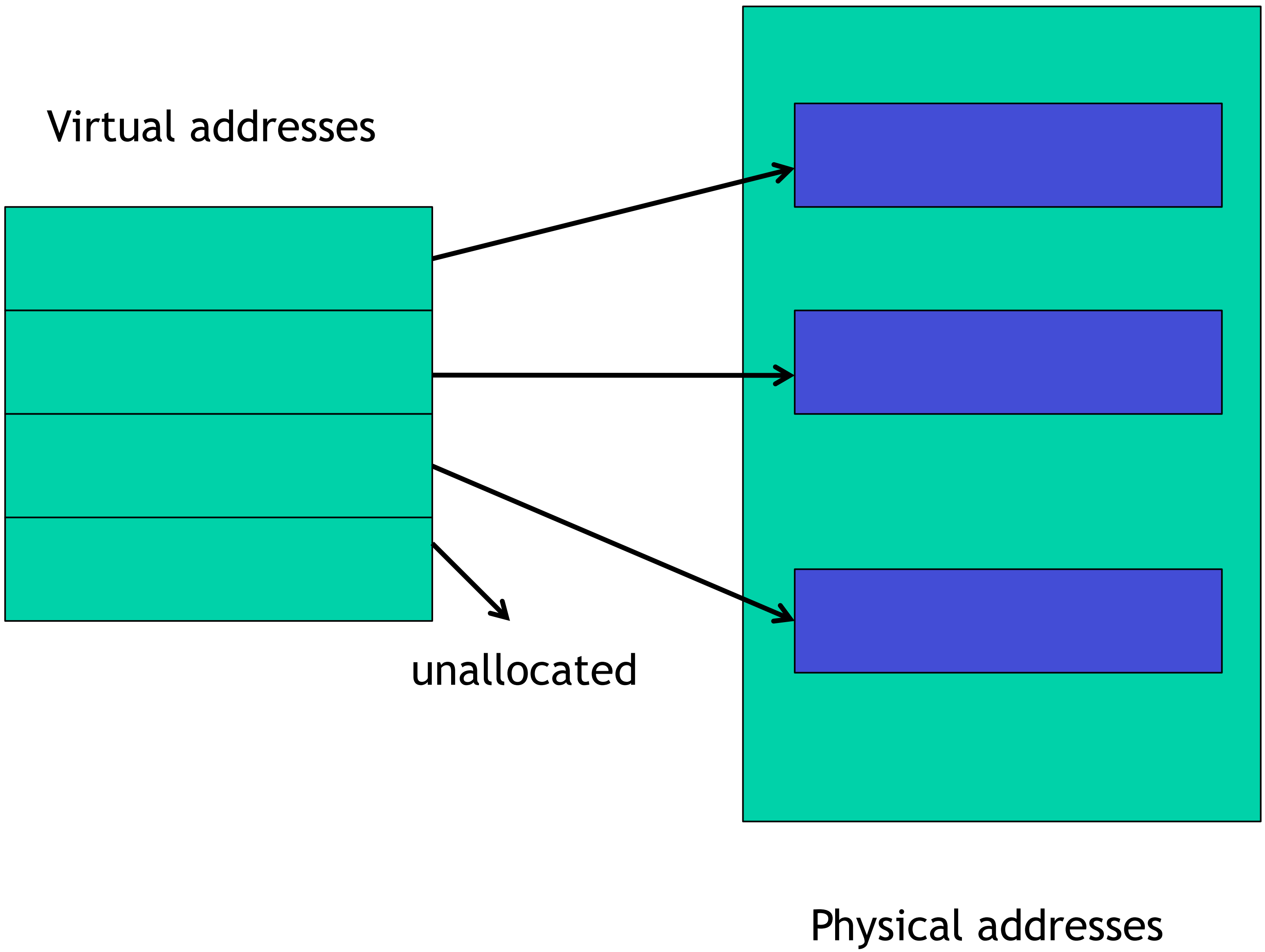
# Fork() primer into virtual memory management



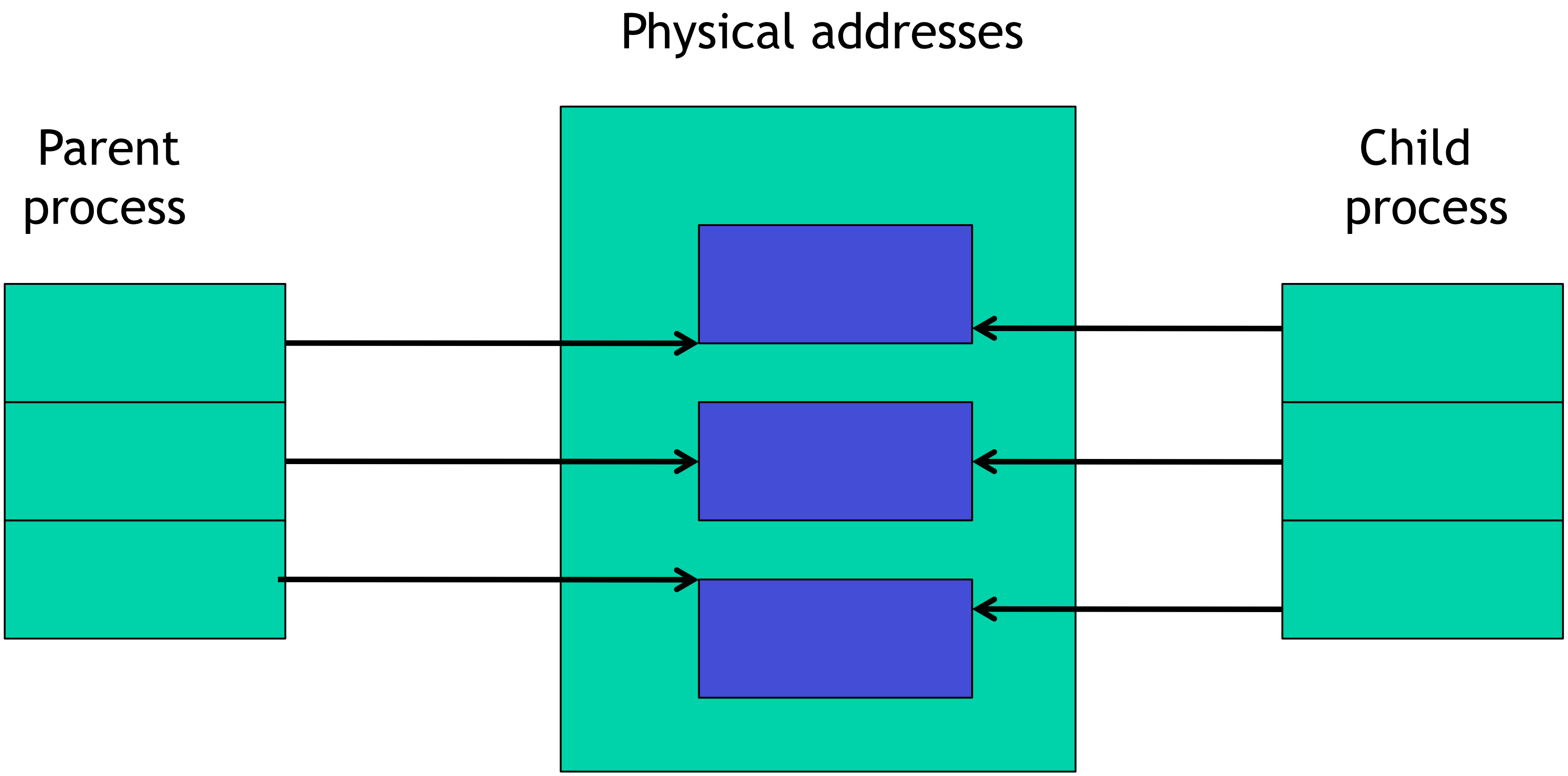

virtual
Address space

why virtual?

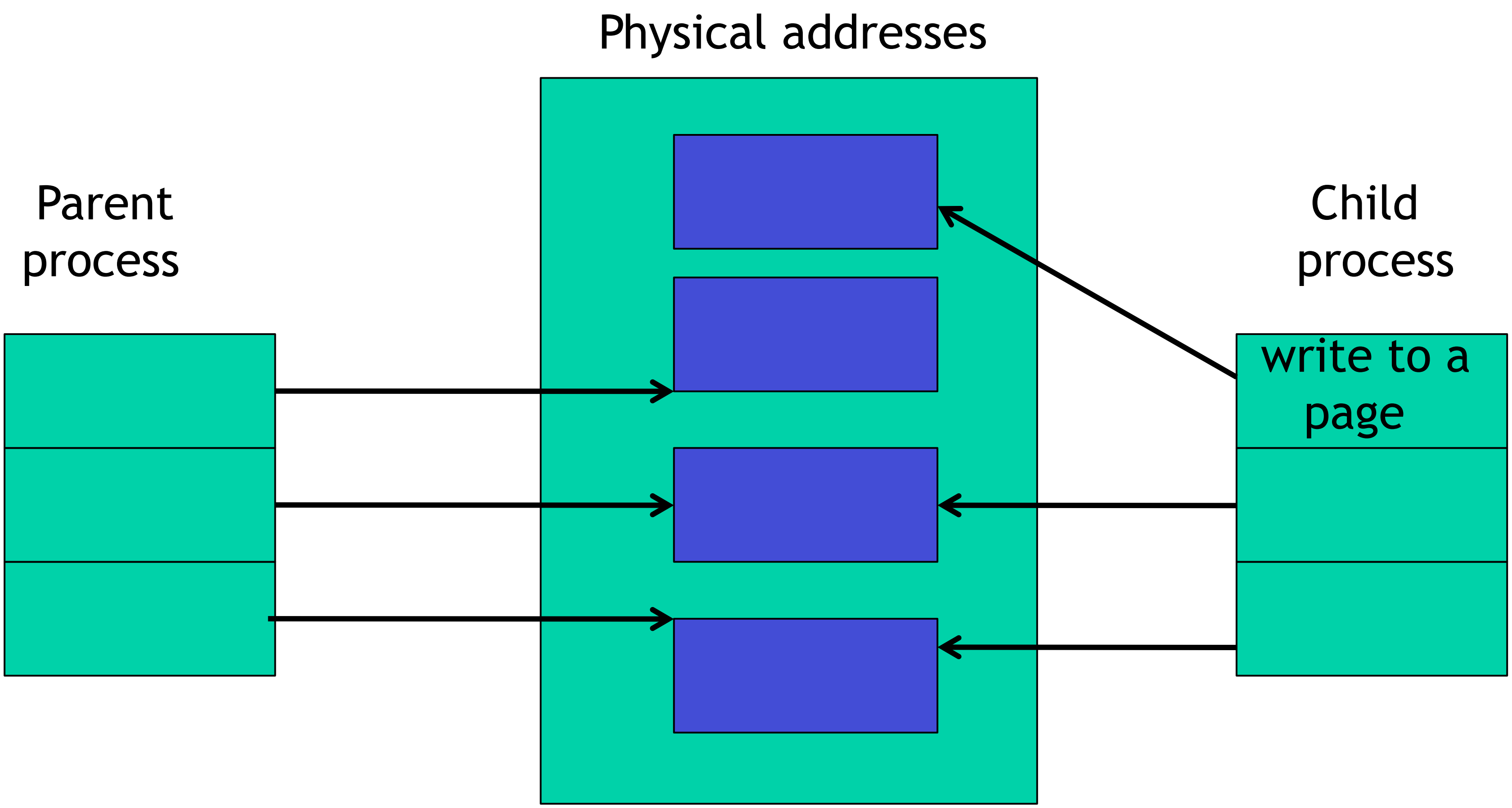
# Fork() primer into virtual memory management



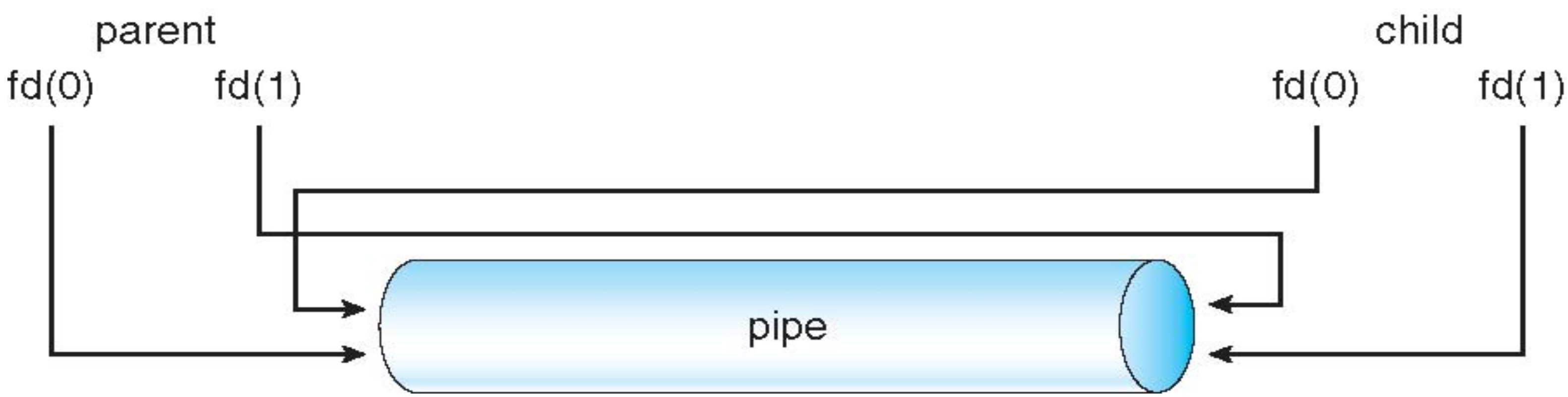
# Fork() Copy-on-write policy



# Fork() Copy-on-write policy



## Unnamed Pipes+dup2 : communication child/parent process

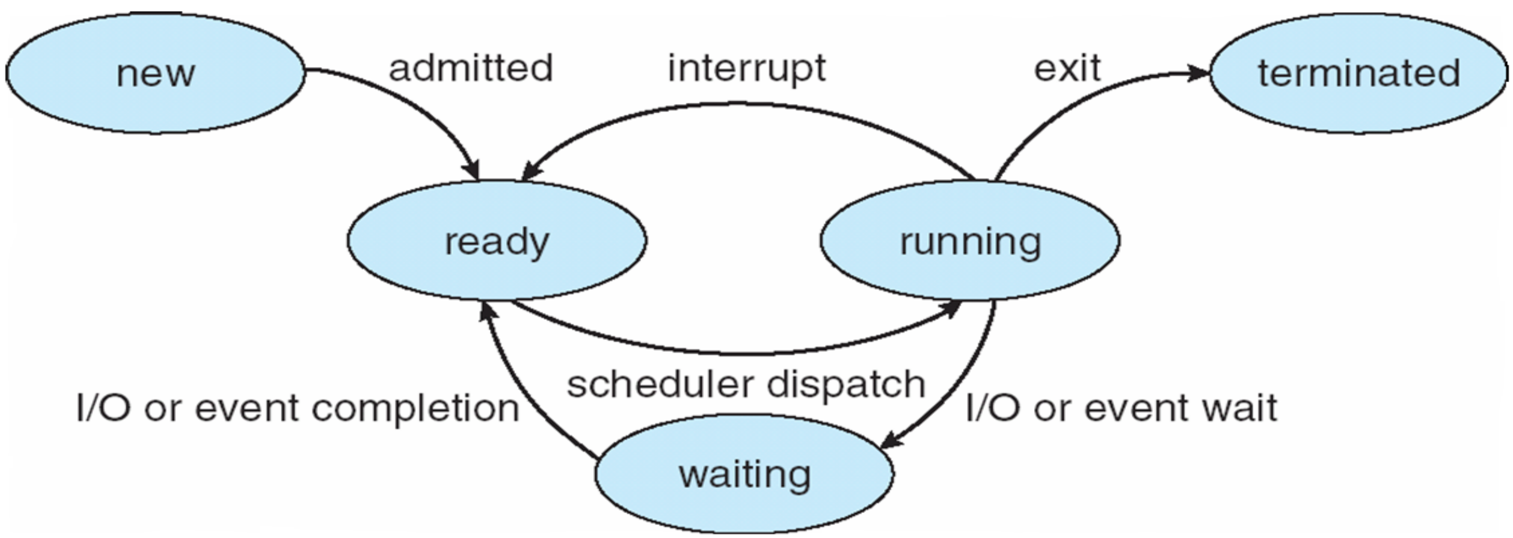



```
Pipe(fid); // where int fid[2] fid[0] is the read
    from the pipe and fid[1] is write to the pipe

dup2(oldfid, newfid) //creates an alias to oldfid
//very handy when you do not want to use file
    descriptors for
```

## Process States

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

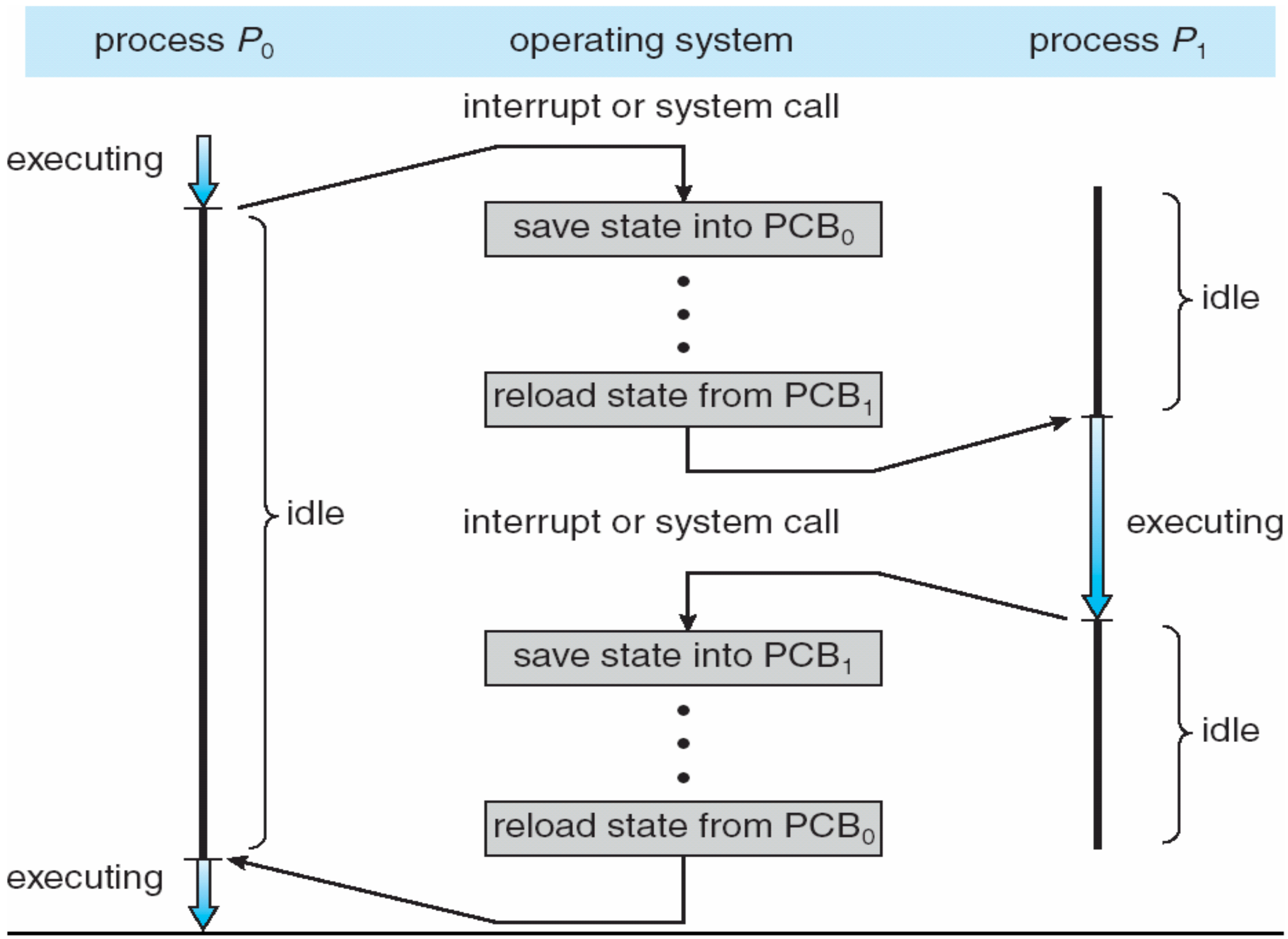


# Kernel data structure for processes (PCB)

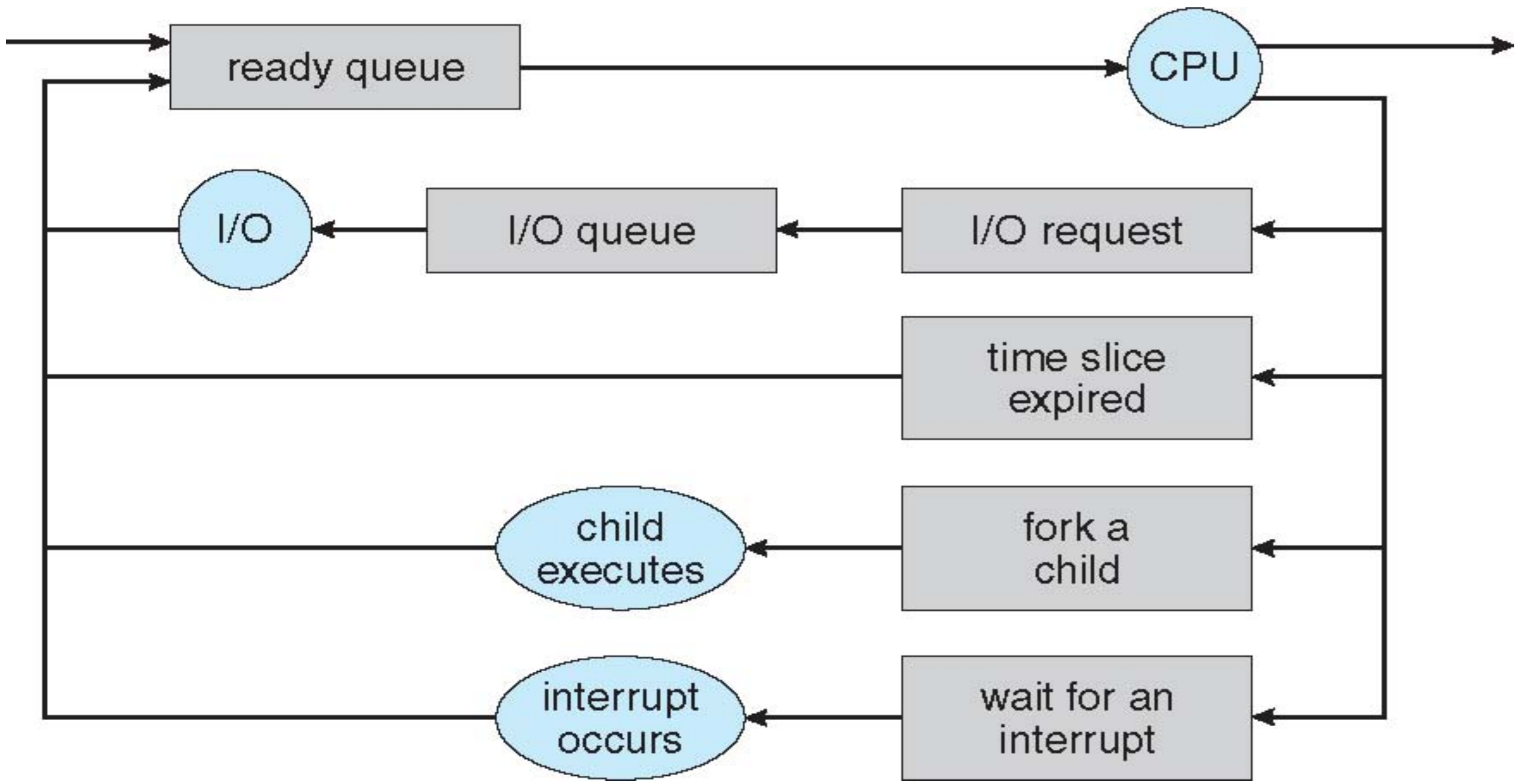
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

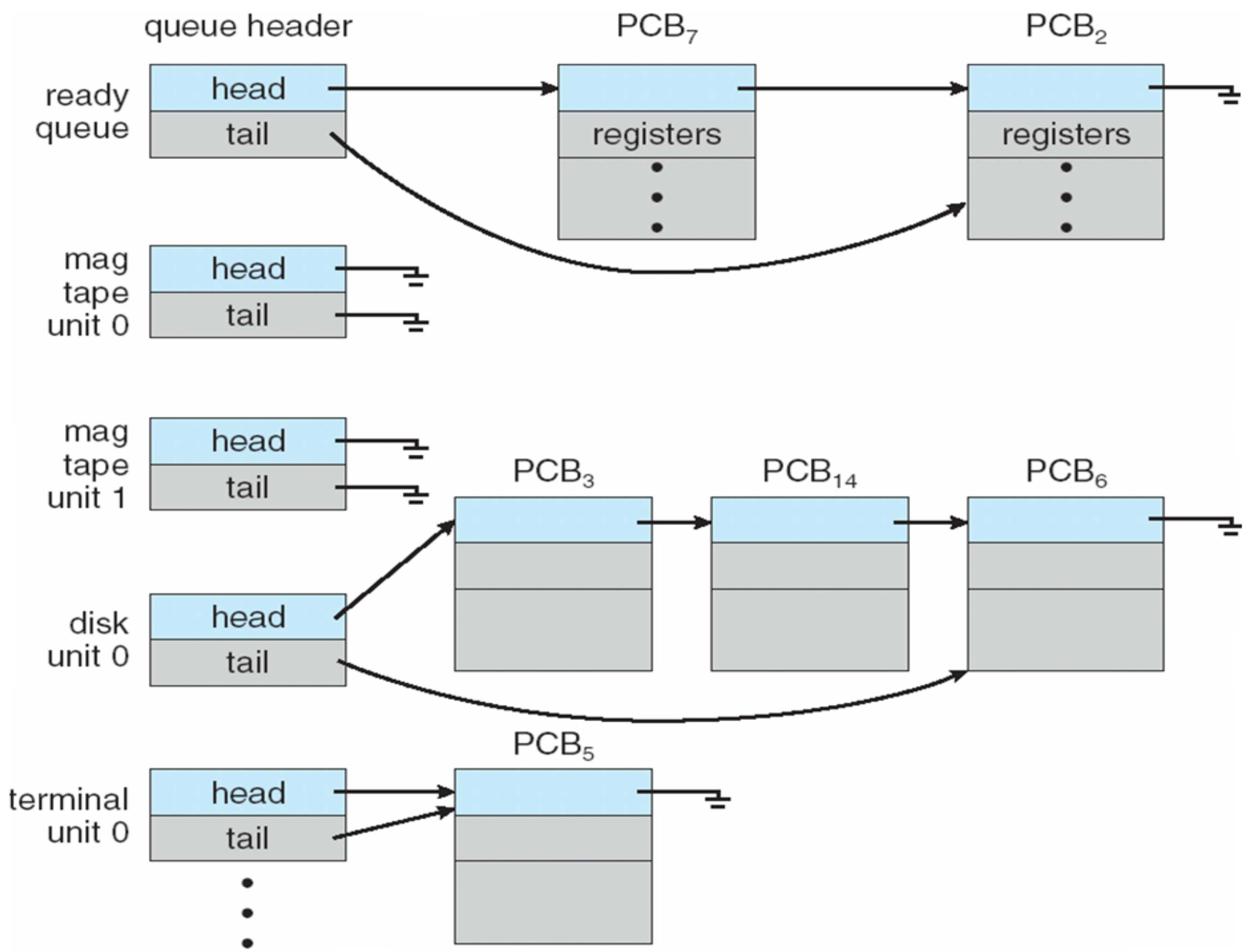
# Process Context Switch



# Process Scheduling



# Process Queues



**Lets take a kernel drive to study**
**the process data structure and fork() system call**

## Next class

- Process management
  - Inter-process communication (Named pipes, shared memory (shmget, mmap), message passing)
  - Intro to threads

**An in-class discussion**
**(a bit-hack)**