Week 8 Lecture 1

NWEN 241
Systems Programming

Alvin Valera

Alvin.valera@ecs.vuw.ac.nz

Content

- Socket programming (cont.)
- System calls
- Process management

Closing a socket

- Socket must be closed after its use

```
int shutdown(int sockfd, int how);
```

```
int close(int sockfd);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *how* can either be SHUT_RD (further receptions disallowed), SHUT_WR (further transmissions disallowed), or SHUT_RDWR (further receptions and transmissions disallowed)
  - Shutdown blocks communication without destroying the socket, close blocks the communication and destroys the socket.
  - If successful, returns 0, otherwise, returns -1

Some points to note

```
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

bind()assigns the address specified by addr to the socket referred to by the file descriptor sockfd.

```
if (bind(fd, (struct sockaddr *)&addr, sizeof(addr))<0) {
    printf("Error binding socket");
    exit(0);
}
```

A sockaddr is used to refer to any type of address.

The only purpose of this structure is to **cast** the structure pointer passed in addr

```
struct sockaddr {
   sa_family_t sa_family;
   char        sa_data[14];
            }
```

```
struct sockaddr_un {
   sa_family_t sun_family; /*AF_UNIX*/
   char        sun_path[108];
/* Pathname */
            };
```

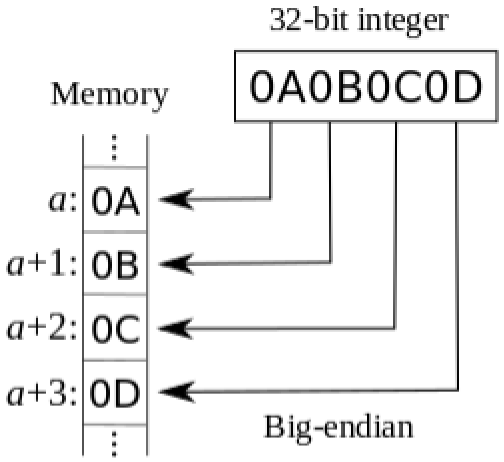
```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
};

struct in_addr {
    unsigned long s_addr;
};
```

The rules used in name binding vary between address families. The actual structure passed for the addr argument depends on the address family.

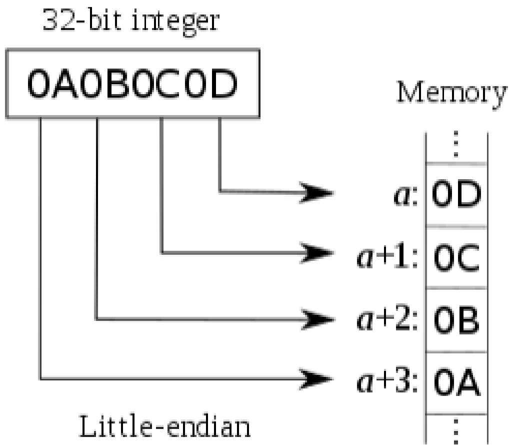
Some points to note

- Little-endian and big-endian issue: Some computers write data "left-to-right" and others "right-to-left".

- A machine can read its own data just fine - problems happen when one computer stores data and a different type tries to read it.



big-end first



little-end first

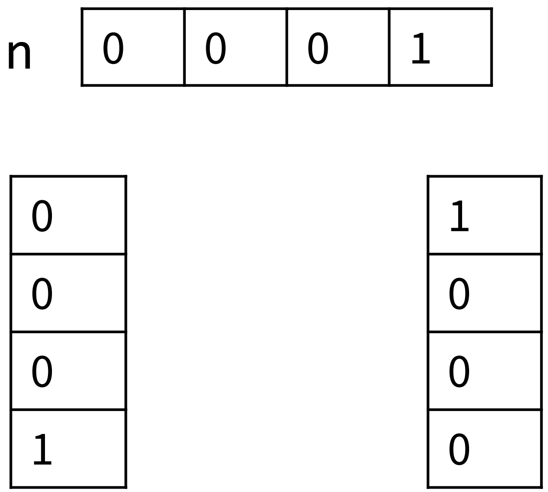
```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

How can we check endianness

- Use command-line utility: `lscpu (linux)`

- Write your own program in C:

```
#include<stdio.h>
void main(){
int n = 1;
// little endian if true
if(*(char *)&n == 1)
  printf("Little endian");
else
    printf("Big endian");
}
```



big-end first

little-end first

System Calls

How to know which system calls are invoked?

Two commands:

a) **ltrace** – traces call to library functions

b) **strace** -traces system calls

See details in Linux manual pages

Usage :

ltrace ./<program executable file>

ltrace –S ./<program executable file> (also display Kernel system calls)

How to know which system calls are invoked?

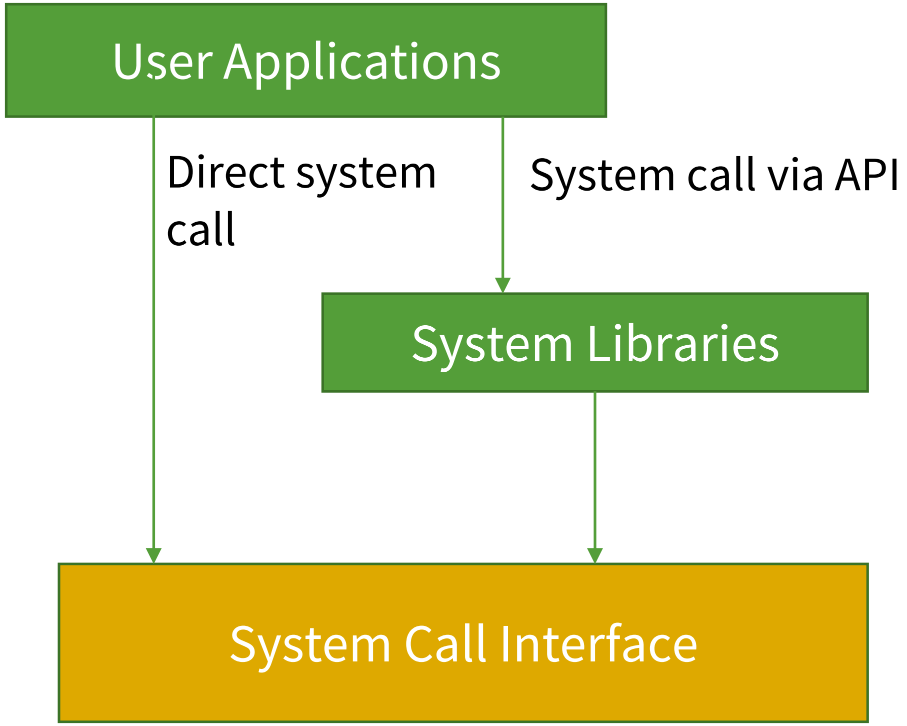
ltrace –S output

```
SYS_brk(0)                                              = 0x7fffba3a9000
SYS_access("/etc/ld.so.nohwcap", 00)                    = -2
SYS_access("/etc/ld.so.preload", 04)                    = -2
SYS_openat(0xffffff9c, 0x7f9e8d421428, 0x80000, 0)      = 3
SYS_fstat(3, 0x7fffc27475d0)                            = 0
SYS_mmap(0, 0x8148, 1, 2)                               = 0x7f9e8d756000
SYS_close(3)                                            = 0
SYS_access("/etc/ld.so.nohwcap", 00)                    = -2
SYS_openat(0xffffff9c, 0x7f9e8d629dd0, 0x80000, 0)      = 3
SYS_read(3, "\177ELF\002\001\001\003", 832)             = 832
SYS_fstat(3, 0x7fffc2747630)                            = 0
SYS_mmap(0, 8192, 3, 34)                                = 0x7f9e8d750000
SYS_mmap(0, 0x3f0ae0, 5, 2050)                          = 0x7f9e8d000000
SYS_mprotect(0x7f9e8d1e7000, 2097152, 0)                = 0
SYS_mmap(0x7f9e8d3e7000, 0x6000, 3, 2066)               = 0x7f9e8d3e7000
SYS_mmap(0x7f9e8d3ed000, 0x3ae0, 3, 50)                 = 0x7f9e8d3ed000
SYS_close(3)                                            = 0
SYS_arch_prctl(4098, 0x7f9e8d7514c0, 0x7f9e8d751e10, 0x7f9e8d750998) = 0
SYS_mprotect(0x7f9e8d3e7000, 16384, 1)                  = 0
SYS_mprotect(0x7f9e8da00000, 4096, 1)                   = 0
SYS_mprotect(0x7f9e8d627000, 4096, 1)                   = 0
SYS_munmap(0x7f9e8d756000, 33096)                       = 0
printf("Little endian" <unfinished ...>
SYS_fstat(1, 0x7fffc27477f0)                            = 0
SYS_ioctl(1, 0x5401, 0x7fffc2747750, 2)                 = 0
SYS_brk(0)                                              = 0x7fffba3a9000
SYS_brk(0x7fffba3ca000)                                 = 0x7fffba3ca000
<... printf resumed> )                                  = 13
SYS_write(1, "Little endian", 13Little endian)                = 13
SYS_exit_group(0 <no return ...>
```

Invoking System calls

There are two different methods by which a program can invoke system calls:

- **Directly**: by making a system call to a function (i.e., entry point) built directly into the kernel, or
- **Indirectly**: by calling a high-level Application Programming Interface (API) (provided by Linux system library and language library) that invokes the system call.
- Mostly accessed by via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)



System call implementation

- Typically, a number is associated with each system call
  - System call interface maintains a table indexed according to these numbers

- System call interface invokes intended system call in kernel and returns status of the system call and any return values

- Caller need not know about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API

Linux system call table

- First few lines of the table
- For more information: https://github.com/torvalds/linux/blob/v3.13/arch/x86/syscalls/syscall_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
```

Directly Invoking System calls

```
 .global _start

  .text
_start:
  # write(1, message, 13)
  mov     $1, %rax                # system call 1 is write
  mov     $1, %rdi                # file handle 1 is stdout
  mov     $message, %rsi          # address of string to output
  mov     $13, %rdx               # number of bytes
  syscall                         # invoke operating system to do the write

# exit(0)
  mov     $60, %rax               # system call 60 is exit
  xor %rdi, %rdi                  # we want return code 0
  syscall                         # invoke operating system to exit

  .data
message:
  .ascii "Hello, world\n"
```

To make a **direct system call we need low-level programming**, generally in assembler.

User need to know **target architecture,** cannot create CPU independent code.

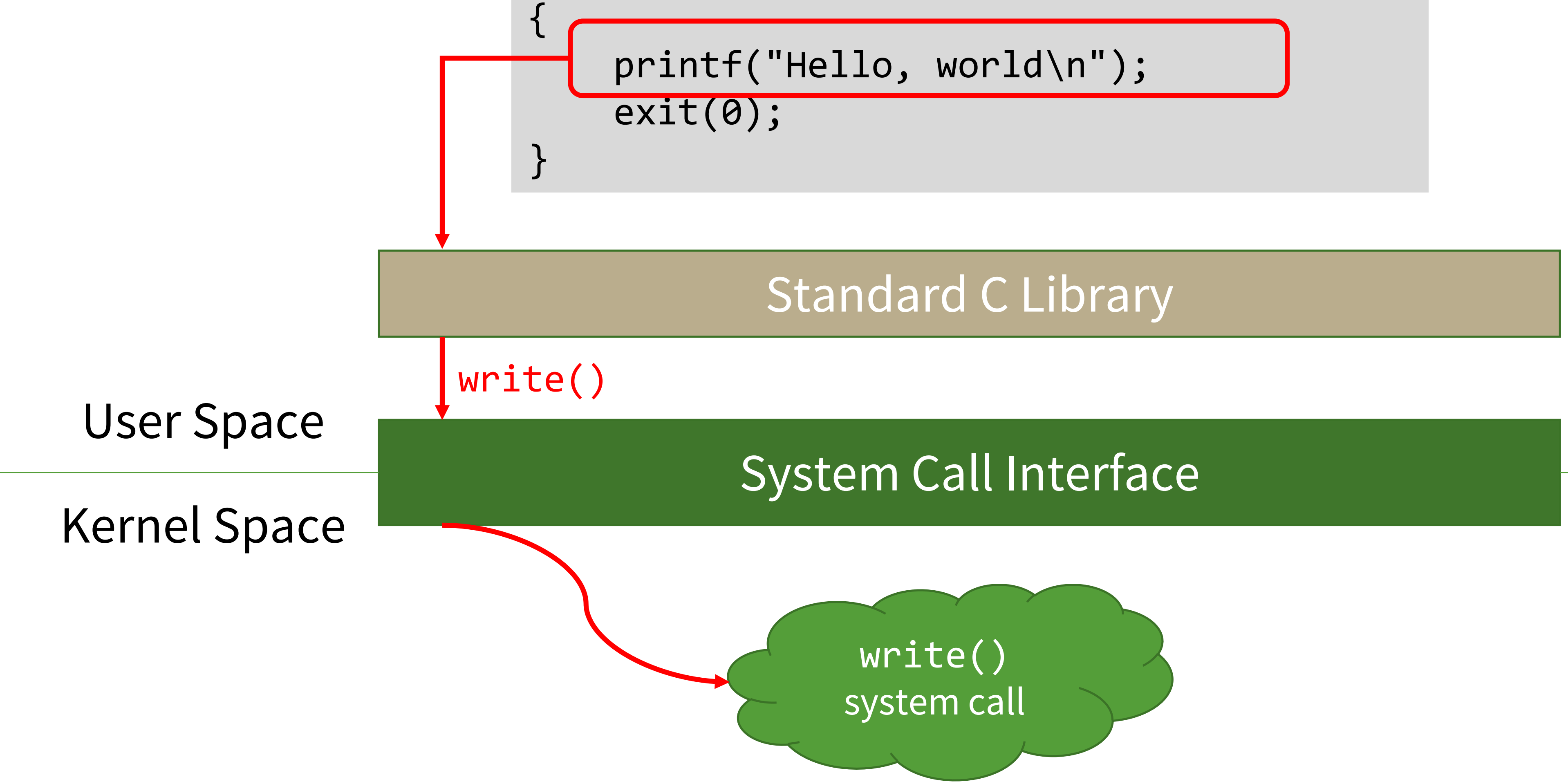
Simpler version

```
#include <stdio.h>

void main(void)
{
    printf("Hello, world\n");
    exit(0);
}
```

Will invoke `write()` system call via API (standard C library)

Simpler version

```
#include <stdio.h>
void main(void)
{
    printf("Hello, world\n");
    exit(0);
}
```



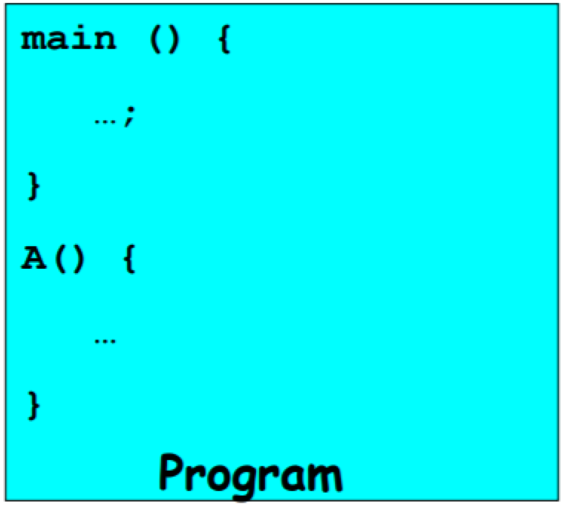
Categories and examples of system calls

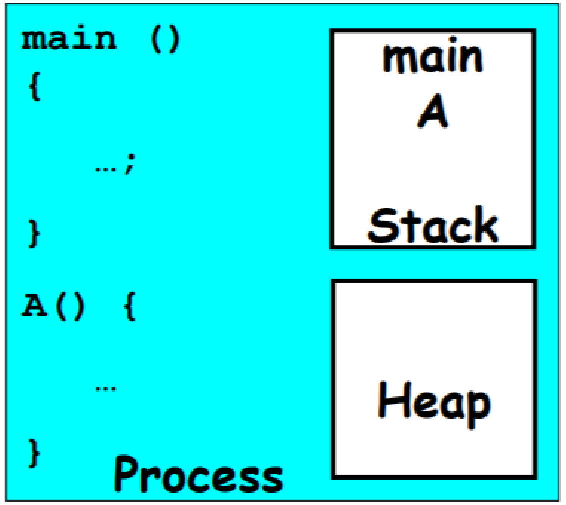
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

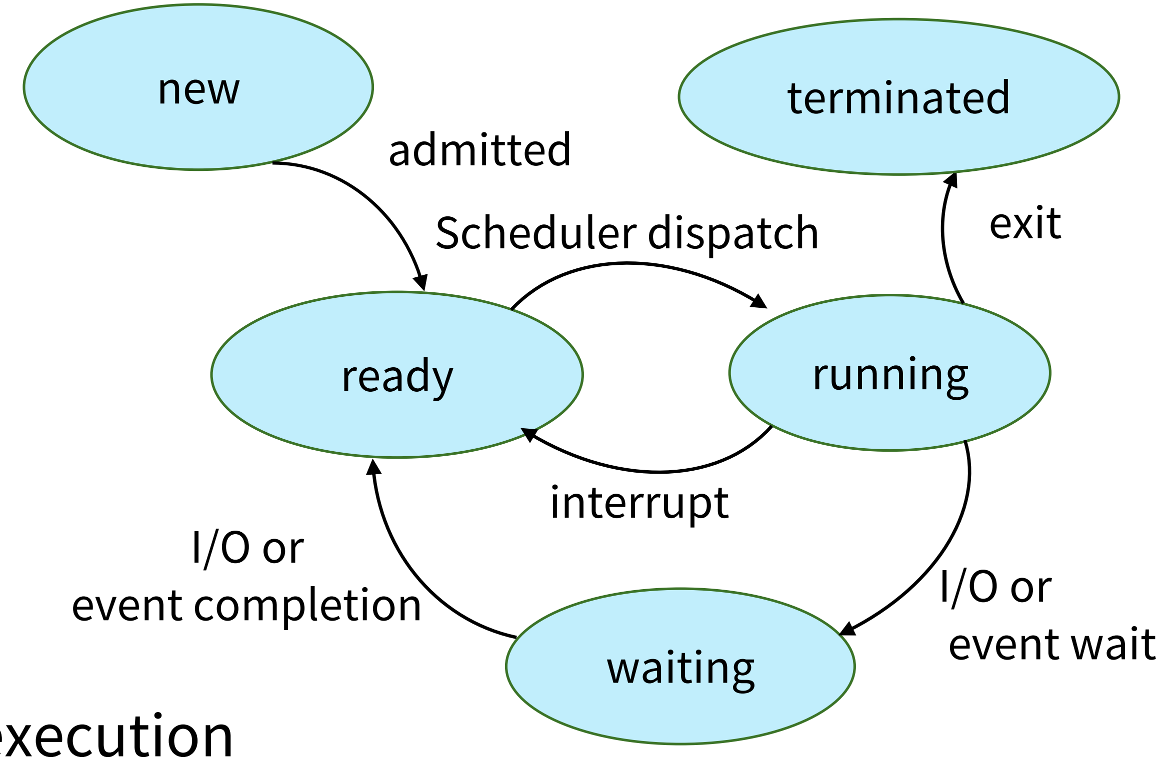
- Unix and Linux both conform to POSIX standard (GNU C Library - glibc)
- POSIX: Portable Operating System Interface

Process Management

Process Vs Program

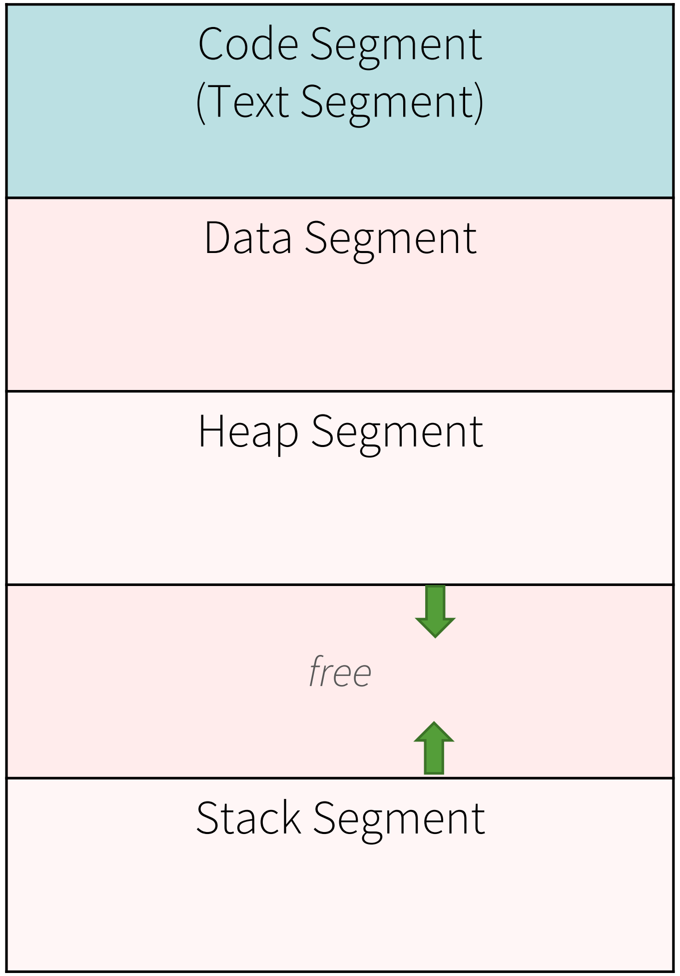






- Program is static, with the potential for execution
- Process is a program in execution and have a state
- One program can be executed several times and thus has several processes

Process in memory

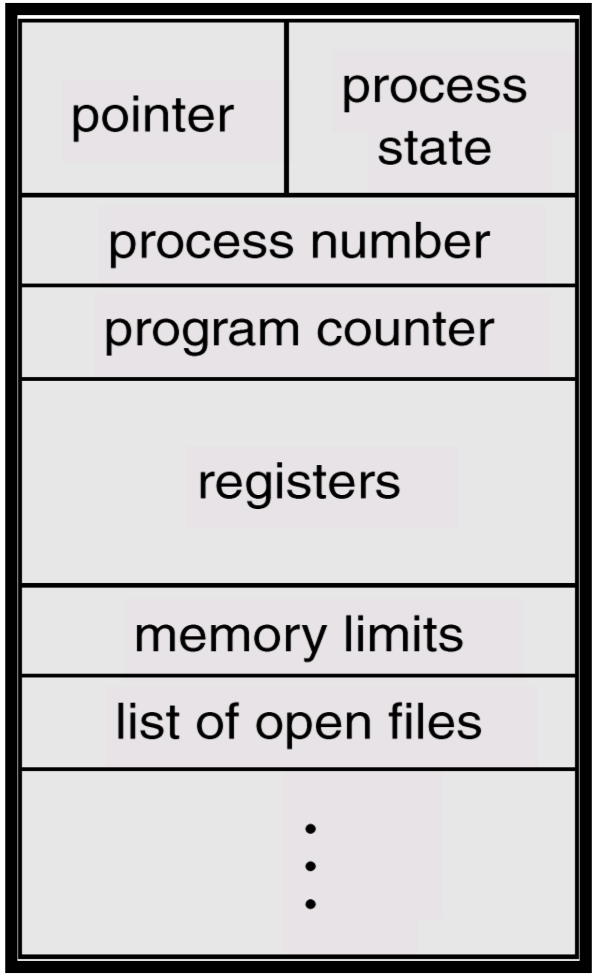


- **Text / Code Segment**
  - Contains program's machine code
- **Segments for Data**

  spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs

Process control block

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
- A process is named using its process ID (PID) or process #
- Stored in a process control block (PCB)



Process representation in Linux

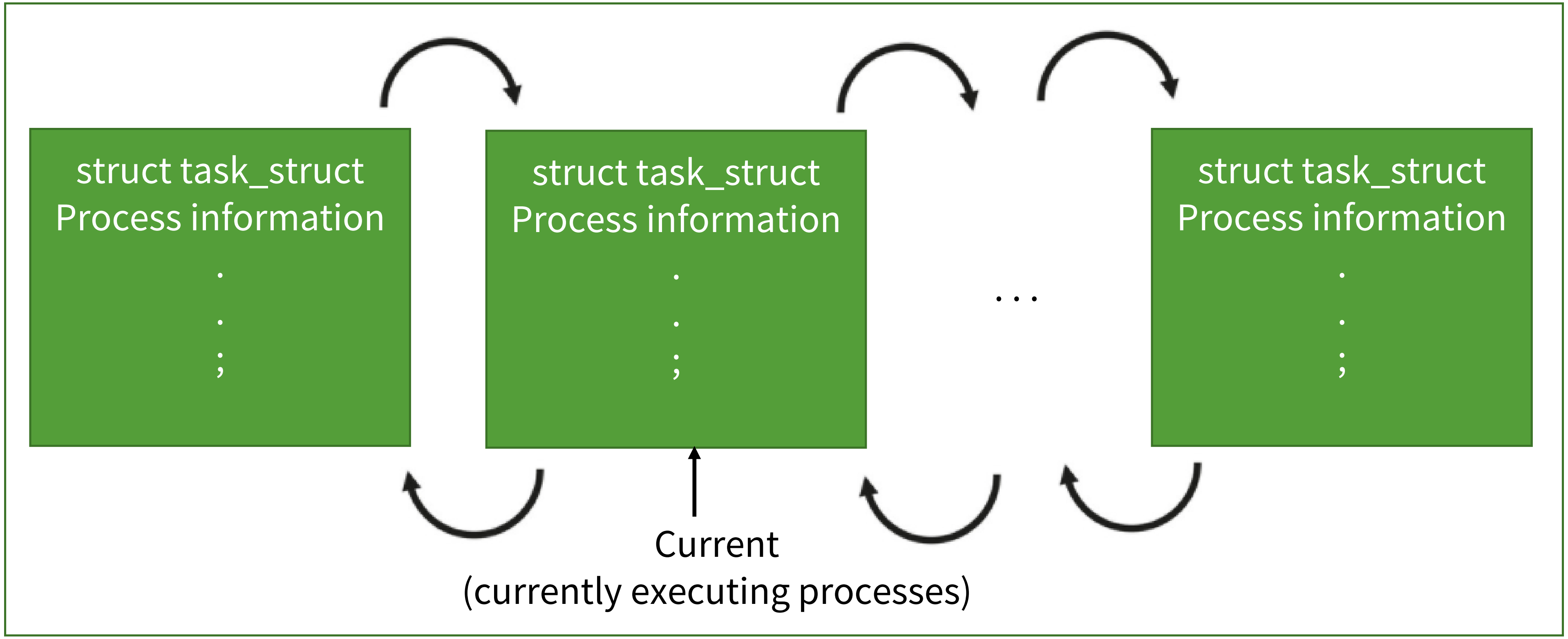
- Represented by structure task_struct
  - See https://github.com/torvalds/linux/blob/master/include/linux/sched.h for more information

- Some of the structure members

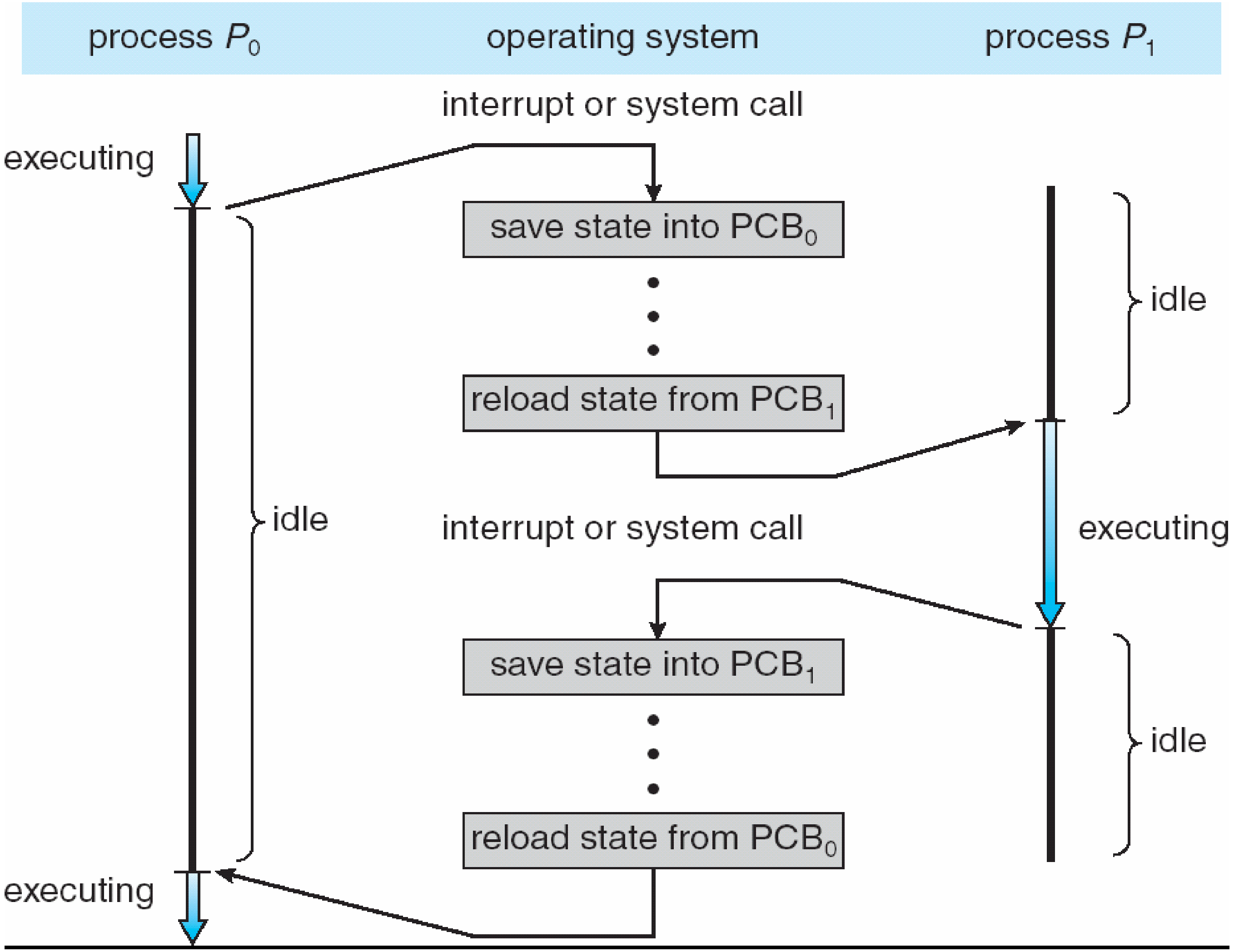
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process’s parent */
struct list_head children; /* this process’s children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

Process representation in Linux

- Represented by structure task_struct
  - See https://github.com/torvalds/linux/blob/master/include/linux/sched.h for more information



Process switching

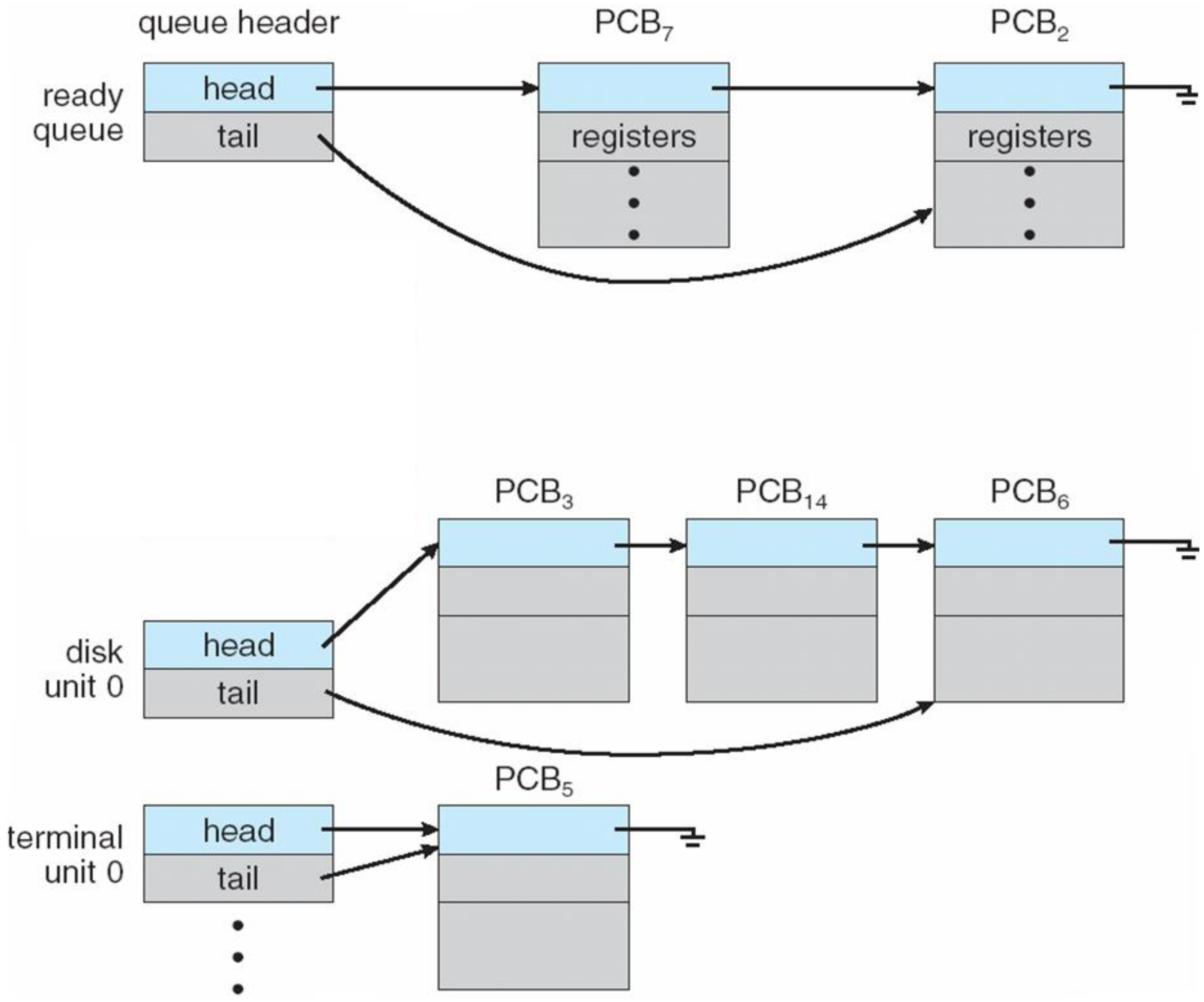


Process scheduling

- **Process scheduler** selects among ready processes for next execution on CPU

- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

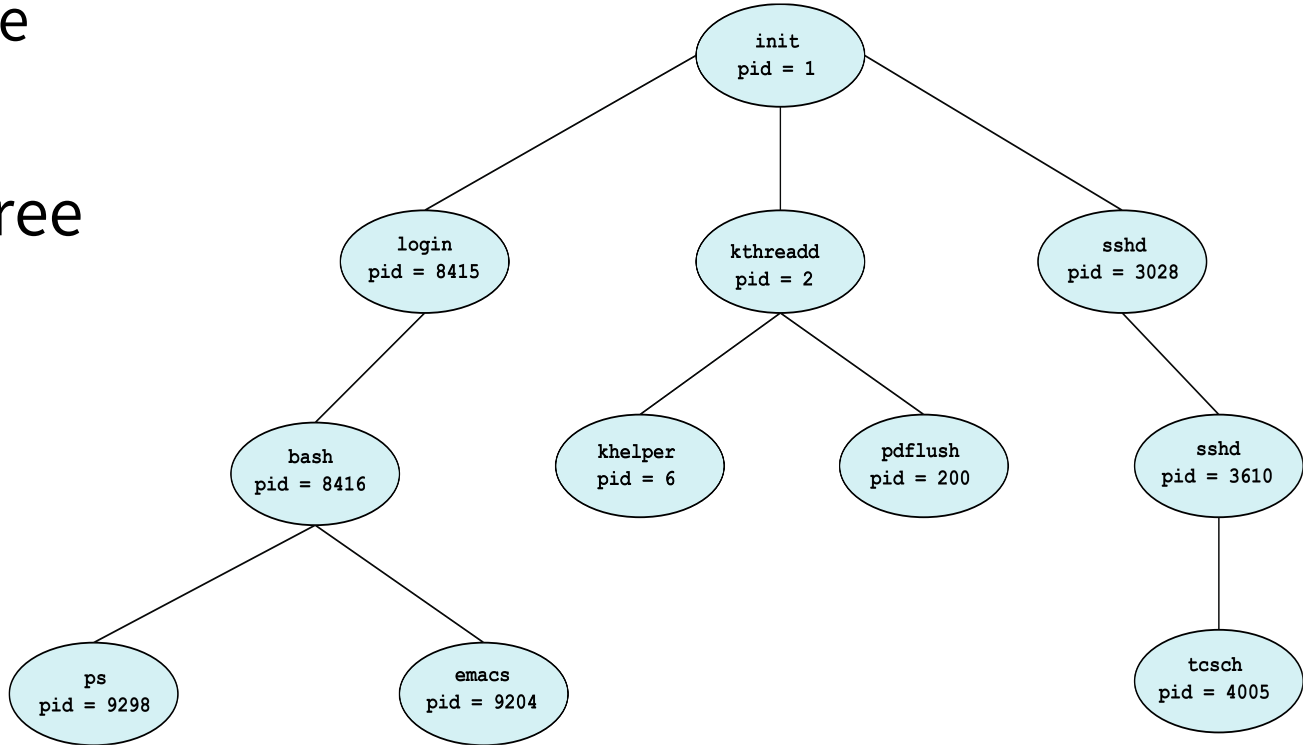
Ready queue and various I/O device queues



Process Initialization on Linux

- The init process (**Init** is the parent of all **processes**, executed by the kernel during the booting of a system**).**
- A process is created by another process, which, in turn create other processes → process tree

Linux process tree



Every process has a process ID (PID)

Linux ps command

- Used to obtain information about processes that are running in the current shell

```
$ ps
  PID  TTY           TIME    CMD
31843 pts/35     00:00:00   bash
31850 pts/35     00:00:00   ps
```

Process ID
Every process is assigned a PID by the kernel

Linux ps command

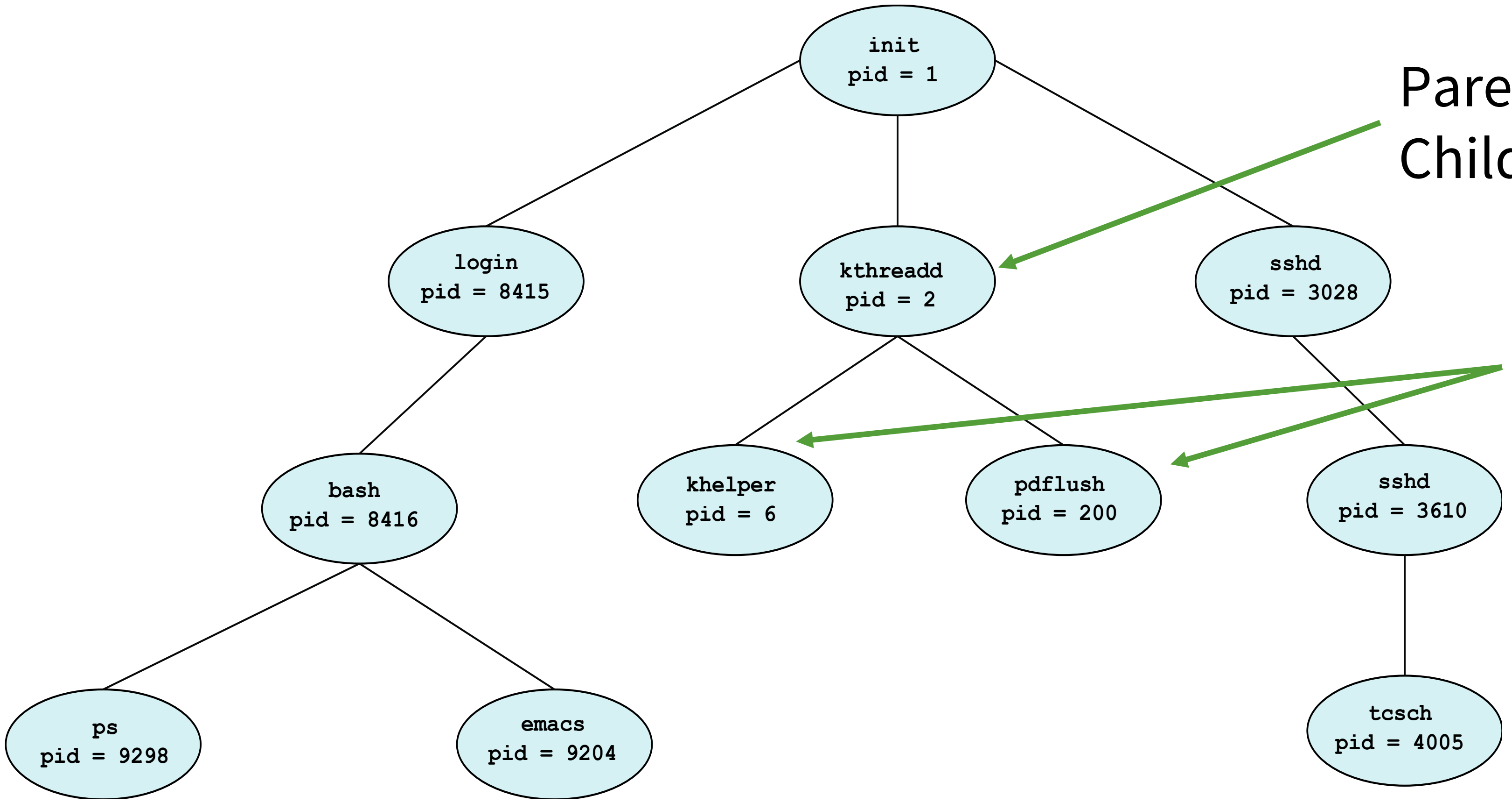
```
$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
sahnijy    31843 31835  0 12:37 pts/35   00:00:00 -bash
sahnijy    32100 31843  0 12:43 pts/35   00:00:00 ps -f
```

Parent Process ID

PID of the process that started the process

Parent and child

When liux starts it runs a single program, **init** with process id **1**



Parent of processes 6 and 200, Child of process 1

Children of process 2

Next Lecture

- System calls for **Process Management**