Server: step 5

- **Send** and receive data

ssize_t send(int *sockfd*, const void **buf*, size_t *len*, int *flags*);

  - *sockfd* is the socket file descriptor (returned by accept())
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options

- Used in connection-oriented sockets (TCP)
- If successful, returns number of characters sent, otherwise, returns -1
- send(sockfd, buf, len, 0); is equivalent to write(sockfd, buf, len);

Server: step 5

- **Send** and receive data

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

  - *sockfd* is the socket file descriptor (returned by socket())
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options
  - *dest_addr* is a pointer to the structure struct sockaddr which will contain the details of the peer socket
  - *addrlen* is a pointer to the length of what dest_addr points to
- Used in non-connection-oriented sockets (UDP)
- If successful, returns number of characters sent, otherwise, returns -1

Server: step 5 example using send()

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
...
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                        (socklen_t*)&addrlen);
...

char msg[] = "hello, world";
ssize_t r = send(client_fd, msg, strlen(msg), 0);
if(r < 0) {
    printf("Error sending message");
    close(client_fd);
    exit(0);
}
```

Server: step 5

- Send and **receive** data

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

  - *sockfd* is the socket file descriptor (returned by accept())
  - *buf* is a pointer to buffer to be received
  - *len* is the length of buffer to be received
  - *flags* is bitwise OR of zero or more options

- Used in connection-oriented sockets (TCP)
- If successful, returns number of characters received, otherwise, returns -1
- If peer socket is shutdown/closed, will return 0
- recv(sockfd, buf, len, 0); is equivalent to read(sockfd, buf, len);

Server: step 5

- Send and **receive** data

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

- *sockfd* is the socket file descriptor (returned by socket())
- *buf* is a pointer to buffer to be received
- *len* is the length of buffer to be received
- *flags* is bitwise OR of zero or more options
- *src_addr* is a pointer to the structure struct sockaddr which will contain the details of the peer socket
- *addrlen* is a pointer to the length of what src_addr points to

- Used in non-connection-oriented sockets (UDP)
- If successful, returns number of characters received, otherwise, returns -1
- If peer socket is shutdown/closed, will return 0

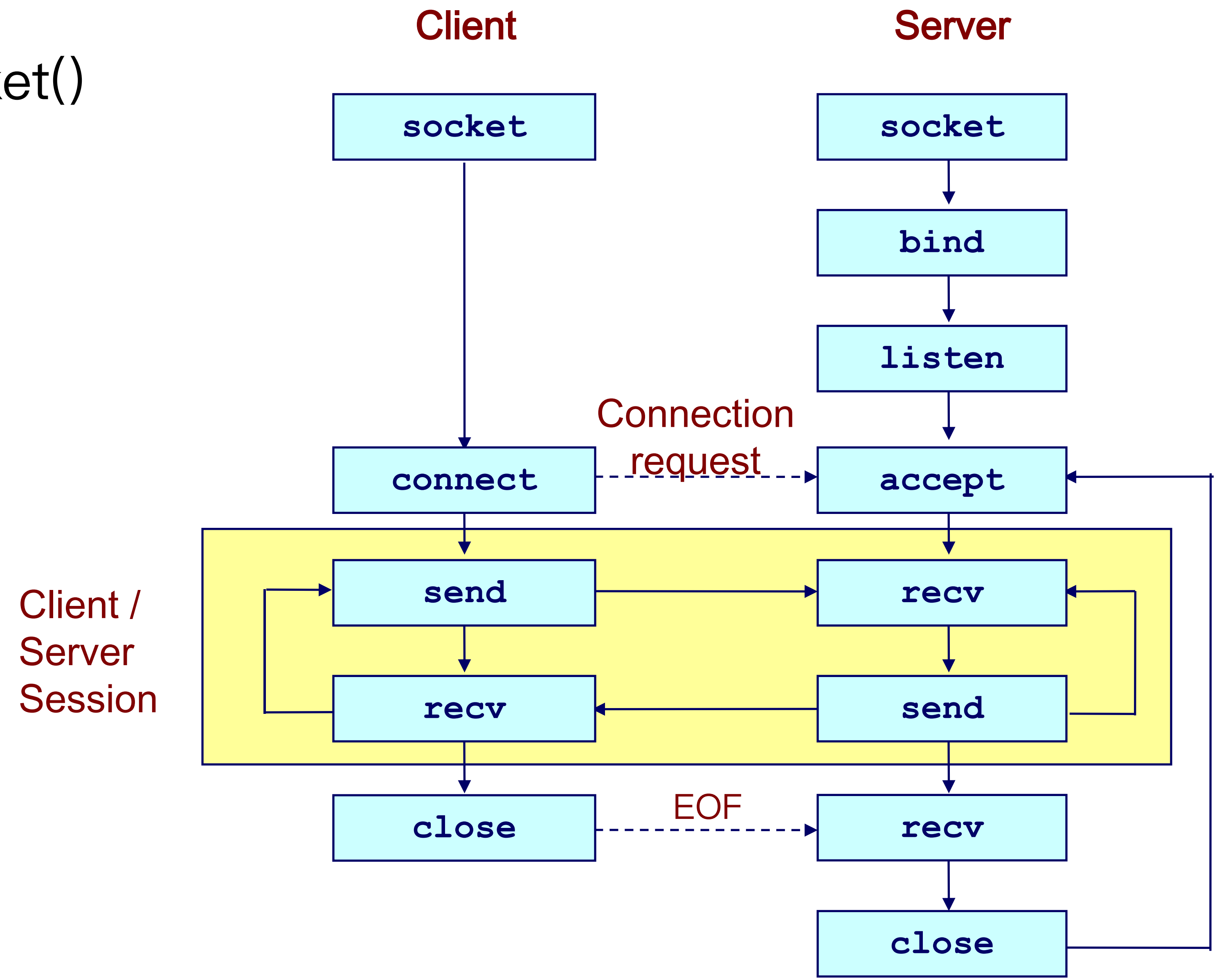
Server: step 5 example using recv()

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
…
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                       (socklen_t*)&addrlen);
…

char incoming[100];
ssize_t r = recv(client_fd, incoming, 100, 0);
if(r <= 0) {
    printf("Error receiving message");
    close(client_fd);
    exit(0);
}
// Do something with receiving message
printf("Received message: %s", incoming);
```

Client: step 1

- Create a socket with the socket() system call
- Same as server step 1



Client: step 2

- Connect the socket to the address of the server using the connect() system call
  - This step is only required for connection-oriented sockets (TCP)

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

- *sockfd* is the socket file descriptor (returned by socket())
- *addr* is a pointer to the structure struct sockaddr which will contain the details of the server socket
- *addrlen* is a pointer to the length of what addr points to

- If successful, returns 0, otherwise, returns -1

Client: step 3

- Send and receive data
- Same as server step 5

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)

  - If successful, returns 0, otherwise, returns -1

Week 12 Lecture 2

XMUT-NWEN 241 - 2024 T2

Systems Programming

Felix Yan

School of Engineering and Computer Science

Victoria University of Wellington

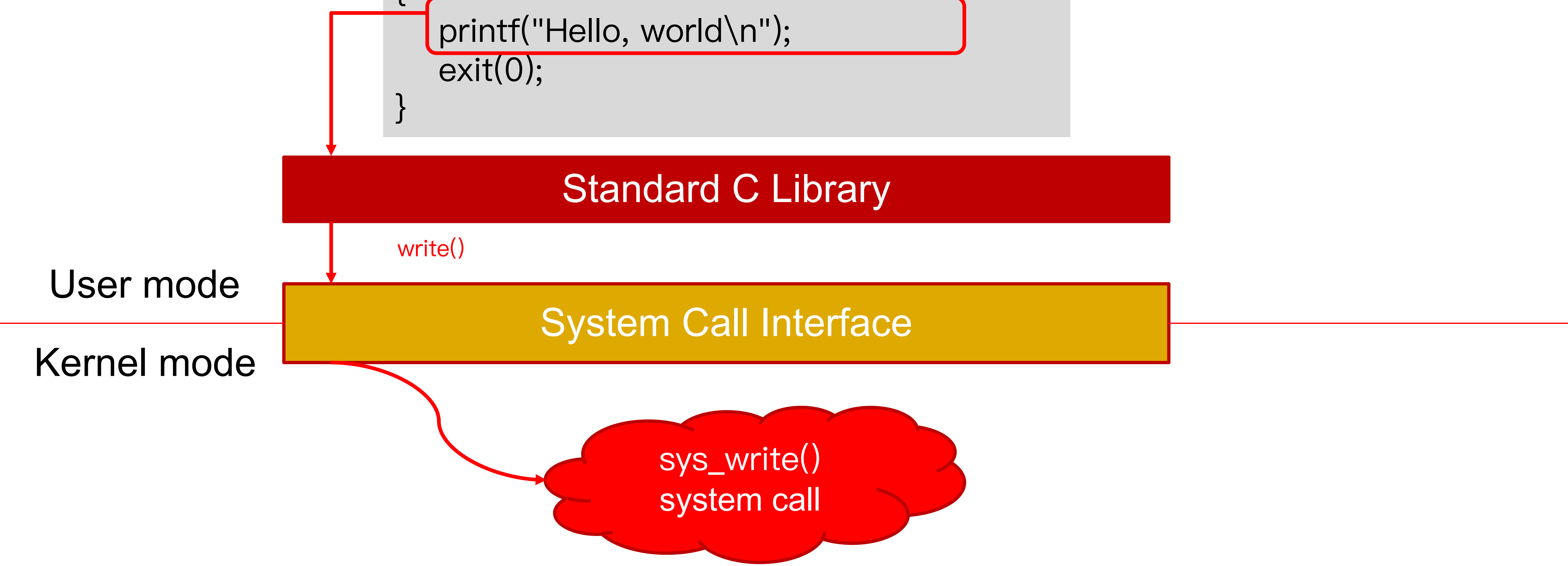
Content

- System calls (in a bit more detail)

- **Categories** of System Calls

Recap: System call invocation –*Example*

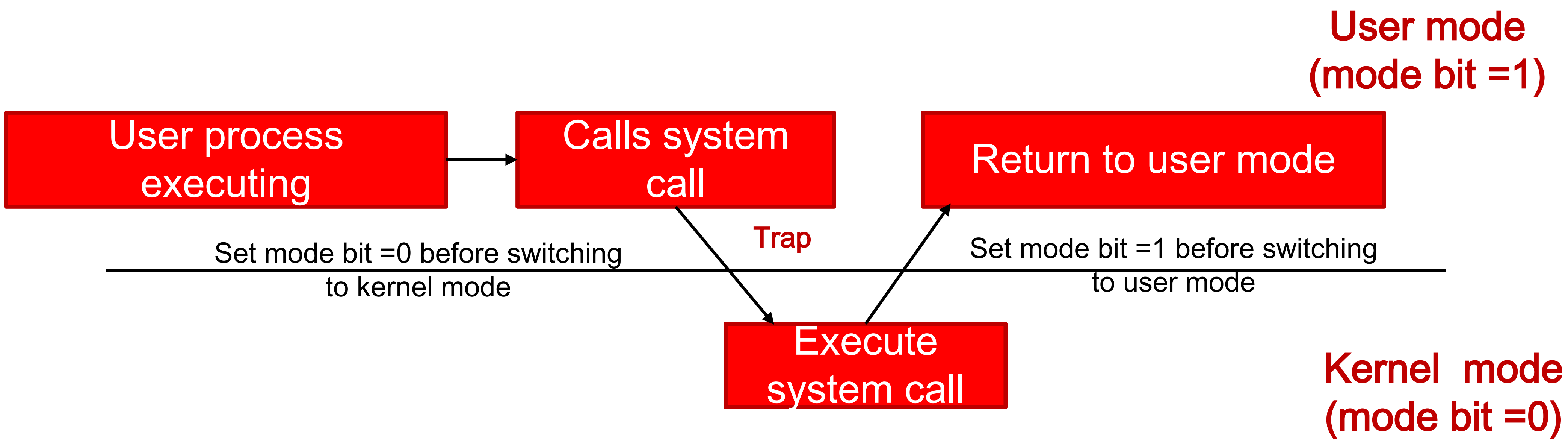
```
#include <stdio.h>
void main(void)
{
    printf("Hello, world\n");
    exit(0);
}
```



The Complete picture

- A system call is a **call to a function that is a part of the kernel** to request service from the operating system.

- When a program needs to access system resources, it makes a system call and a **context-switch** between the user program and the kernel is performed.



How to know which system calls are invoked?

```
#include<stdio.h>

int main()
{
  printf("Hello World") ;
  return 0;
}
```

hello.c

Compile

```
$ gcc -o hello hello.c
```

Run

```
$ ./hello
```

Output

```
Hello World
```

ltrace

```
ltrace ./hello
```

ltrace output

```
printf("Hello World")
Hello World+++ exited (status 0) +++
```

How to know which system calls are invoked?

strace output

```
execve("./hello", ["./hello"], 0x7fffc8e68920 /* 21 vars */) = 0
brk(NULL)                               = 0x7ffff38a6000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=30022, ...}) = 0
mmap(NULL, 30022, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff4e26e1000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4e26d0000
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff4e2000000
mprotect(0x7ff4e21e7000, 2097152, PROT_NONE) = 0
mmap(0x7ff4e23e7000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7ff4e23e7000
mmap(0x7ff4e23ed000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff4e23ed000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7ff4e26d14c0) = 0
mprotect(0x7ff4e23e7000, 16384, PROT_READ) = 0
mprotect(0x7ff4e2a00000, 4096, PROT_READ) = 0
mprotect(0x7ff4e2627000, 4096, PROT_READ) = 0
munmap(0x7ff4e26e1000, 30022)           = 0
fstat(1, {st_mode=S_IFCHR|0660, st_rdev=makedev(4, 1), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x7ffff38a6000
brk(0x7ffff38c7000)                     = 0x7ffff38c7000
write(1, "Hello World", 11Hello World)           = 11
exit_group(0)                           = ?
+++ exited with 0 +++
```

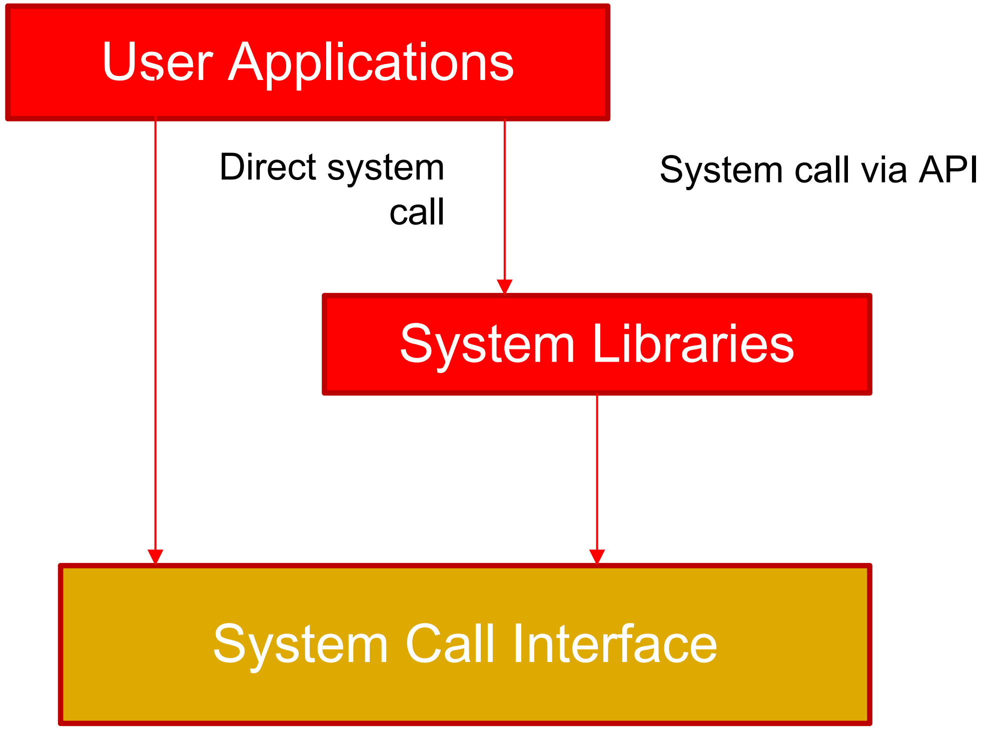
Invoking System calls

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 higher-level library routine (provided by Linux system library and language library) that invokes the system call.
- The system calls and system libraries together constitute the system call **application programming interface (API).**

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

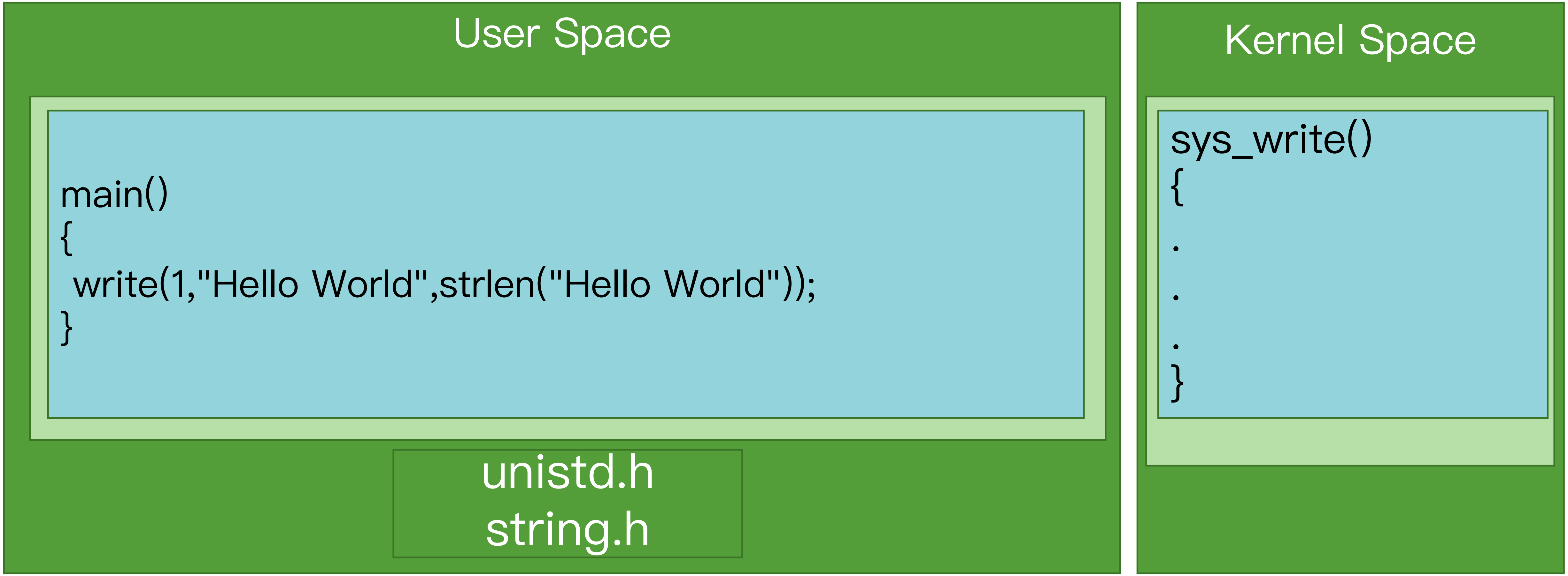
- To make a **direct system call we need low-level programming**, generally in assembler. User need to know **target architecture,** cannot create CPU independent code.

```
 .global _start

  .text
_start:
  # write(1, message, 13)
  mov     $1, %rax                # system call 1 is to 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
  .data
message:
  .ascii "Hello, world\n"
```

Tedious and machine dependent

Invoking System calls through library routines



Categories of System calls

Categories and examples of system calls

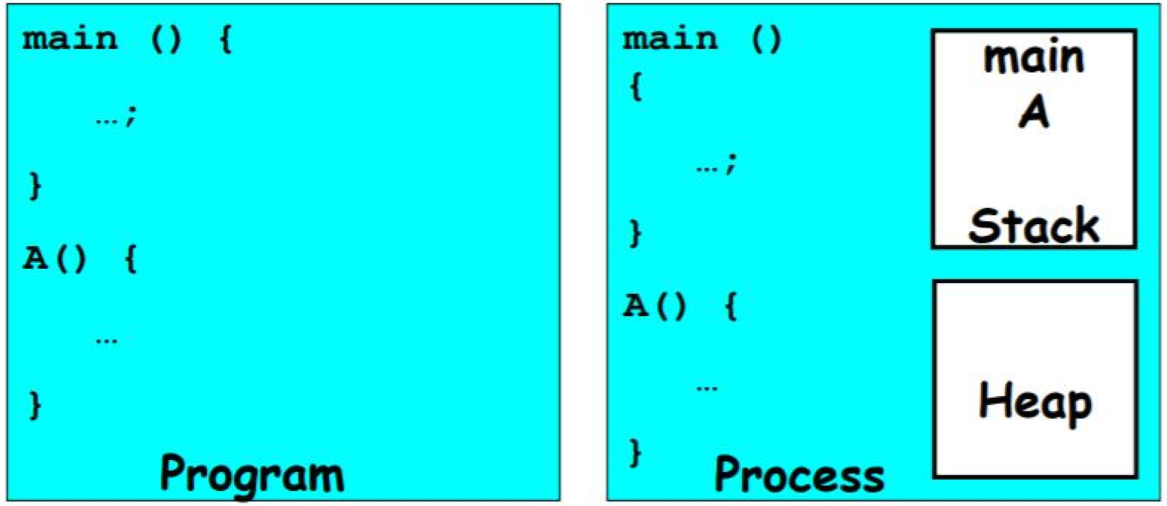
	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information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

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

Categories of System Calls

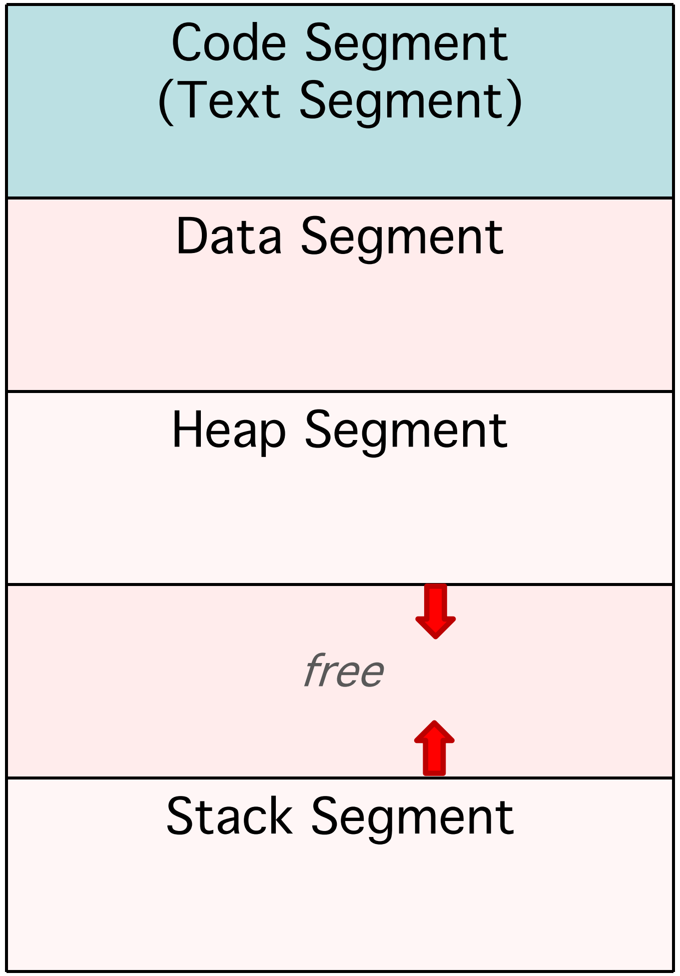
- File manipulation – (create, delete, open, close)
- Process Control – (create, terminate)
- Device Management – (request, release)
- Information Maintenance – (time, date, get / set system date)
- Communications – (create , delete connection, receive, send message)
- Protection – (create , delete connection, receive, send message)

Recap: 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

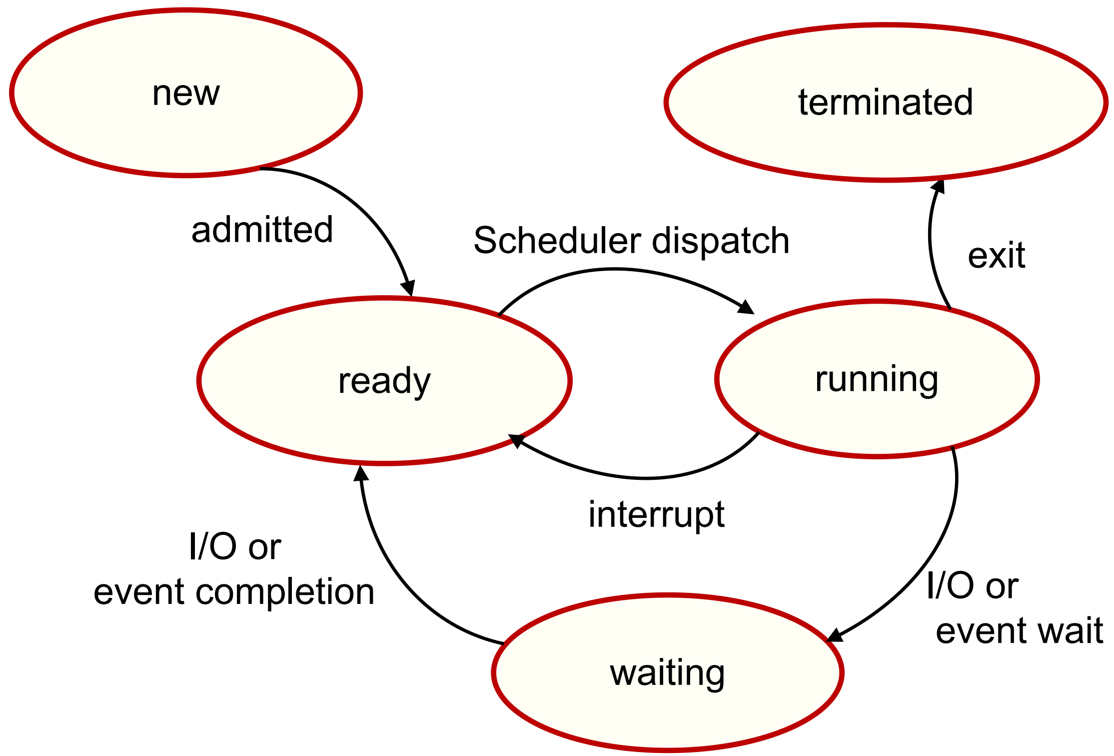
Process in memory



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

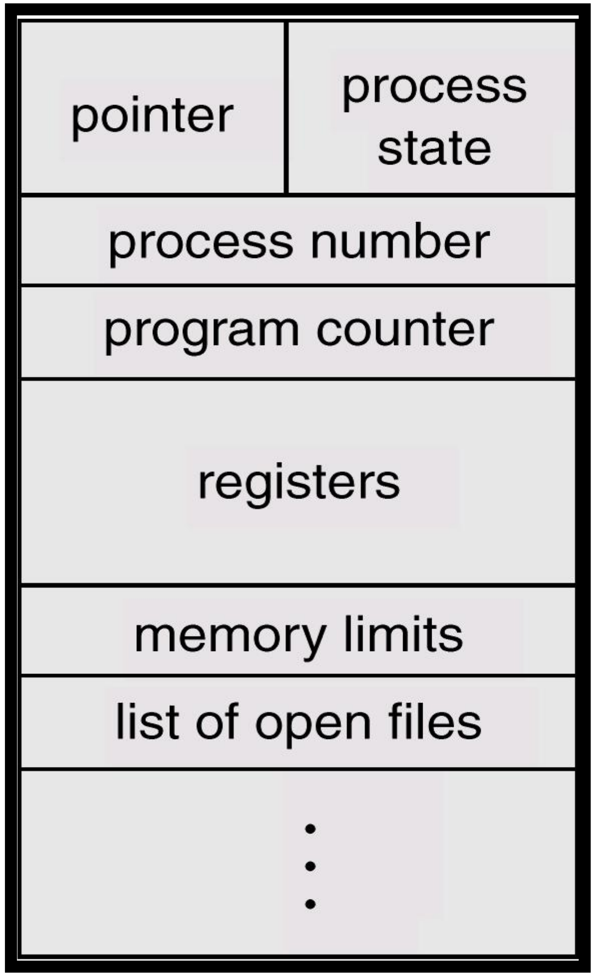
  spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs

Recap: Process lifecycle



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 #
- Data is stored in a process control block (PCB)



Process representation in Linux

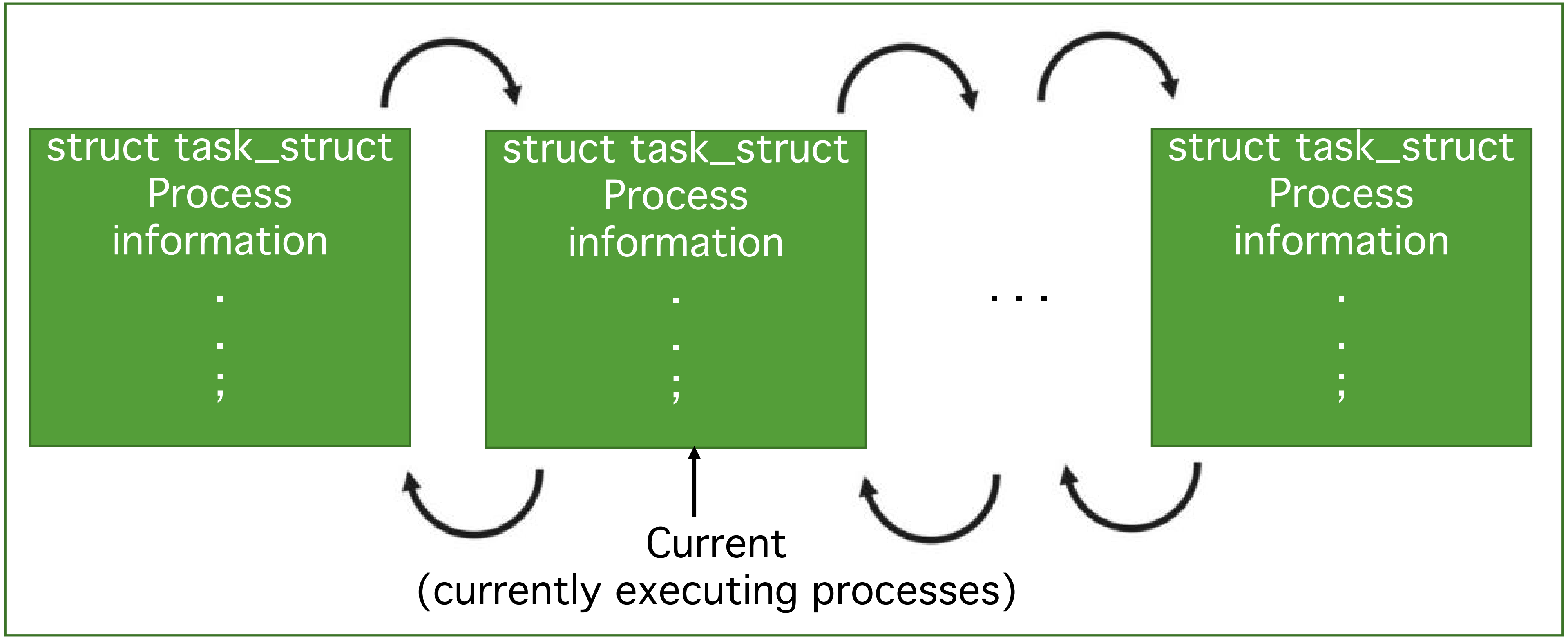
- Represented by structure task_struct
  - See https://github.com/torvalds/linux/blob/master/include/linux/sched.h for more information

- Some of the structure members

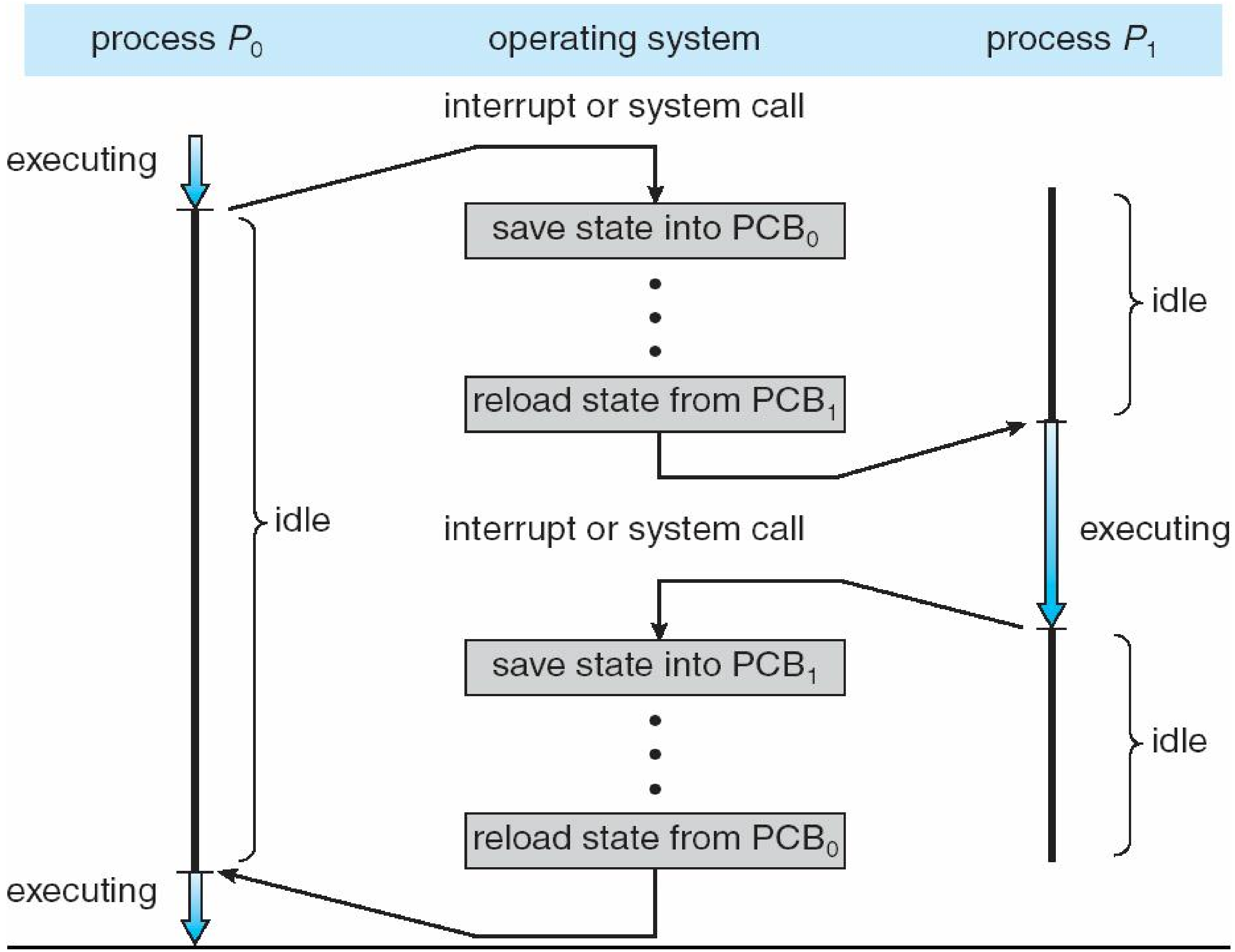
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct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

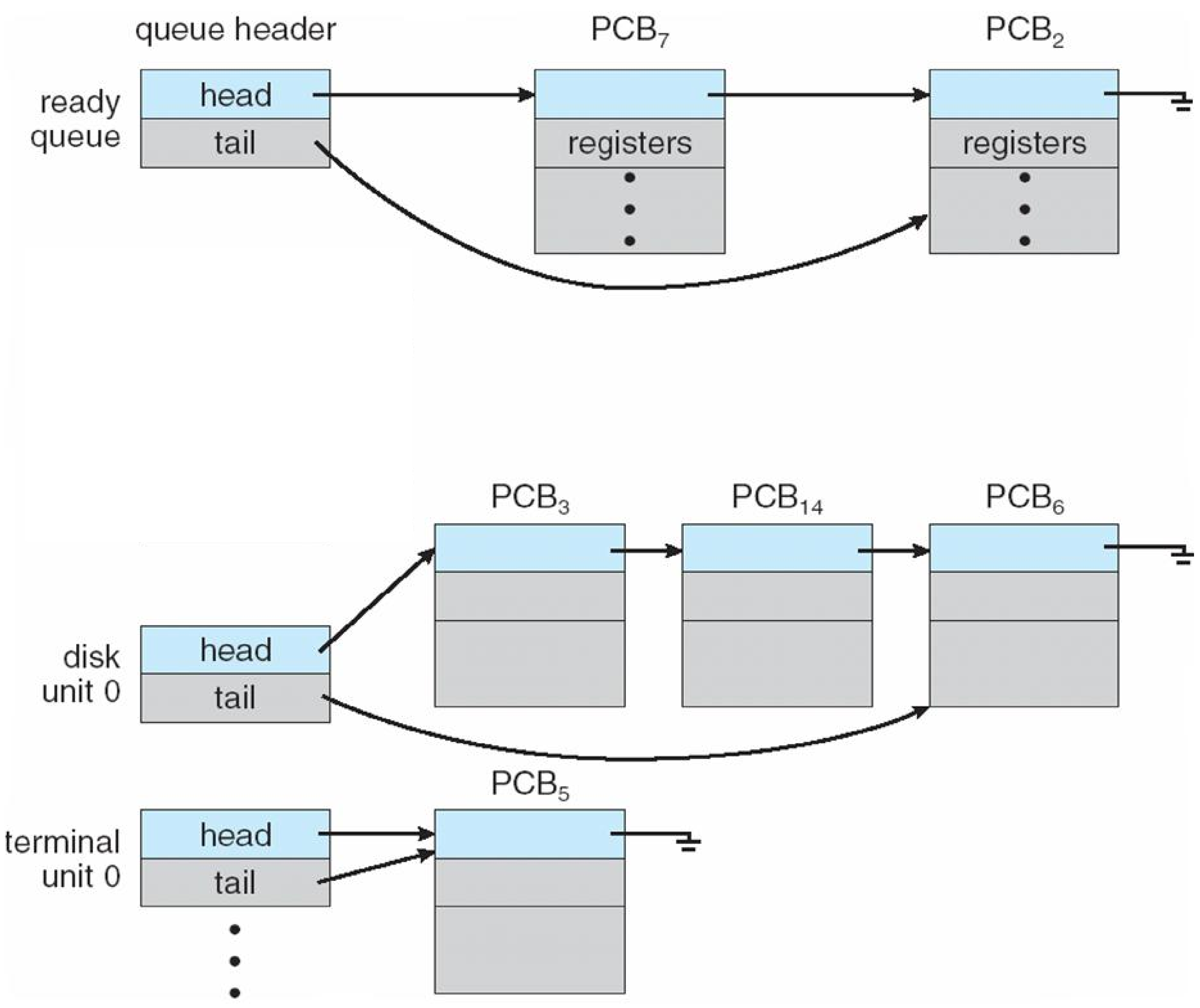


Process scheduling

- **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

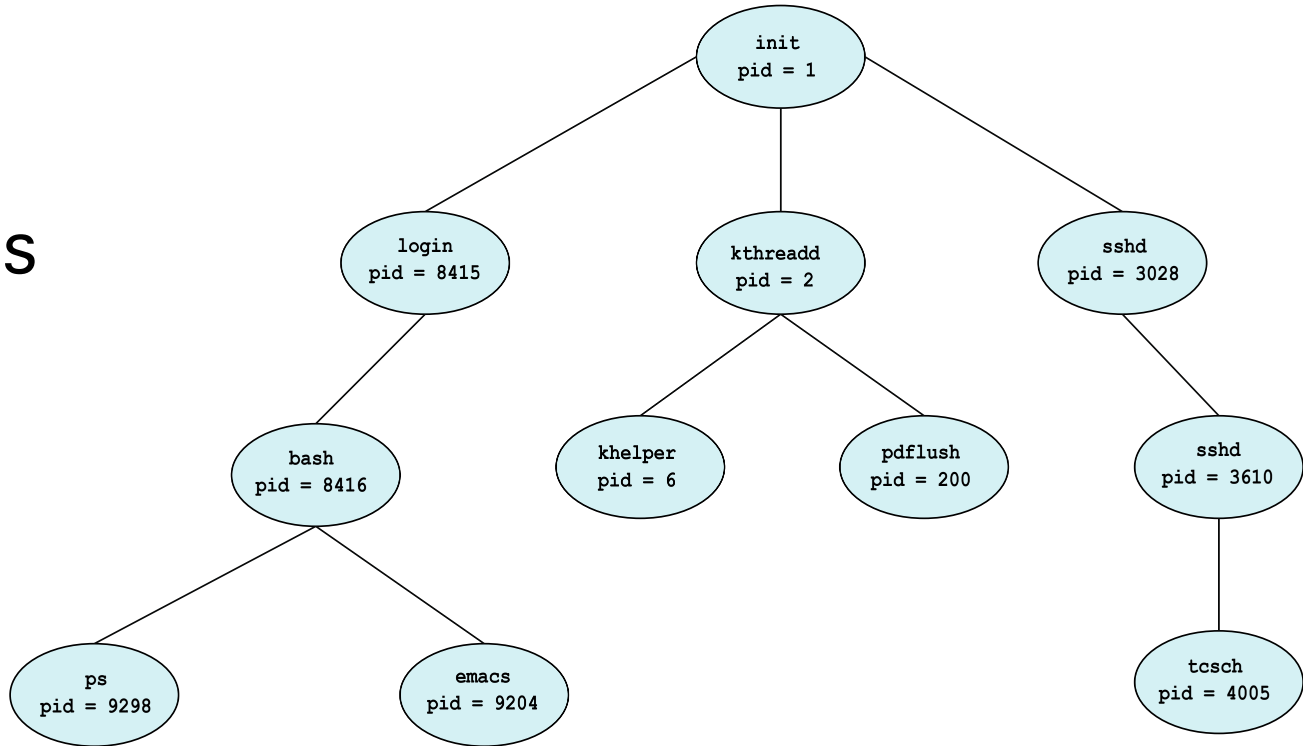
Ready queue and various I/O device queues



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

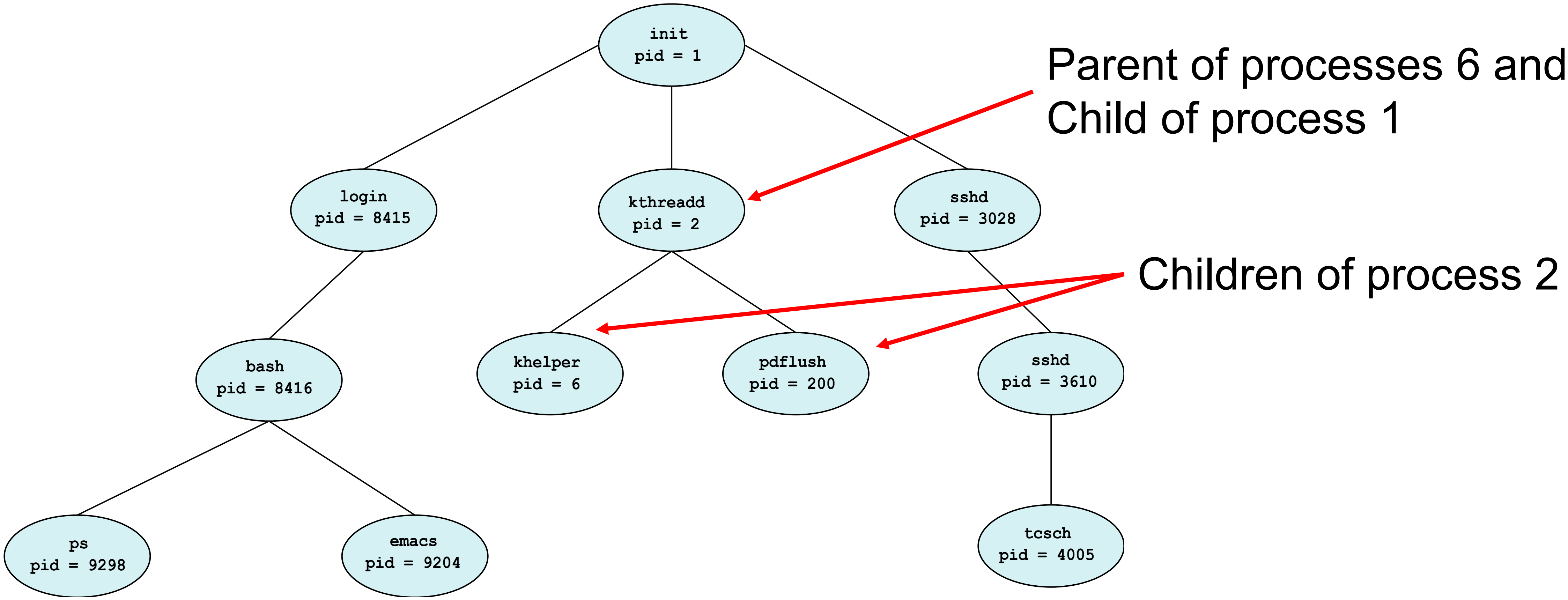
Linux ps command

```
$ 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**



Next lecture

- System calls for **Process Management**