Week 10 Lecture 1

NWEN 241
Systems Programming

Alvin C. Valera

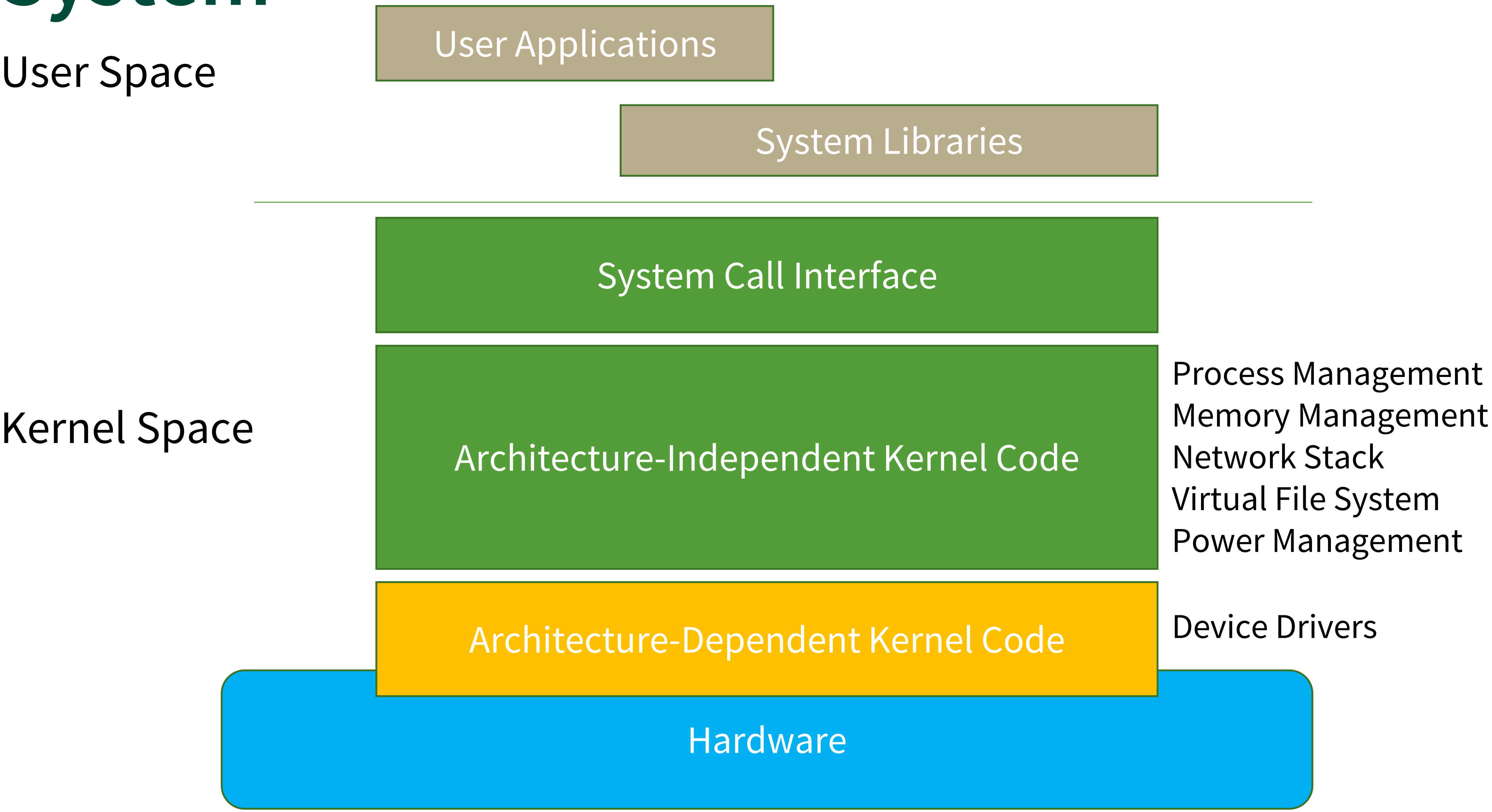
alvin.valera@ecs.vuw.ac.nz

Content

- Overview of system calls

- Process vs program

Recall from Week 1: Linux Operating System



User Space

Kernel Space

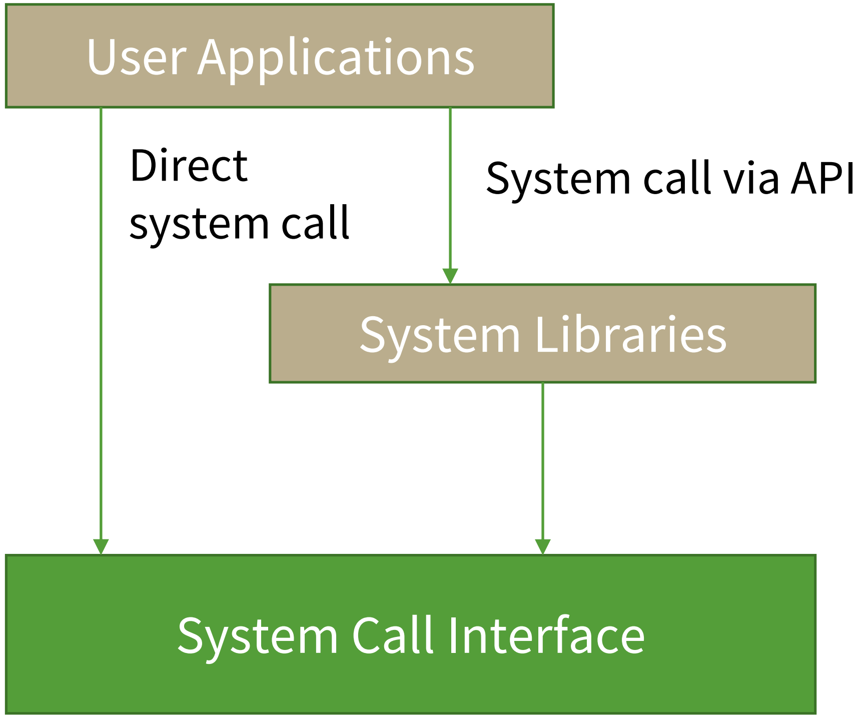
User Applications
System Libraries
System Call Interface
Architecture-Independent Kernel Code
Architecture-Dependent Kernel Code
Hardware

Process Management
Memory Management
Network Stack
Virtual File System
Power Management

Device Drivers

System calls

- Mechanism used a program to request service from the operating system
- Mostly accessed by via a high-level **Application Programming Interface** (API) rather than direct system call use
  - APIs are provided by the system libraries
- Three most common APIs:
  - **Win32 API** for Windows
  - **POSIX API** for POSIX-based systems (including virtually all versions of 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
```

Direct system call example

```
 .global _start

  .text
_start:
  # write(1, message, 13)
  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"
```

Requires knowledge of assembly language!

Tedious!

That’s why we use C/C++ APIs for system calls

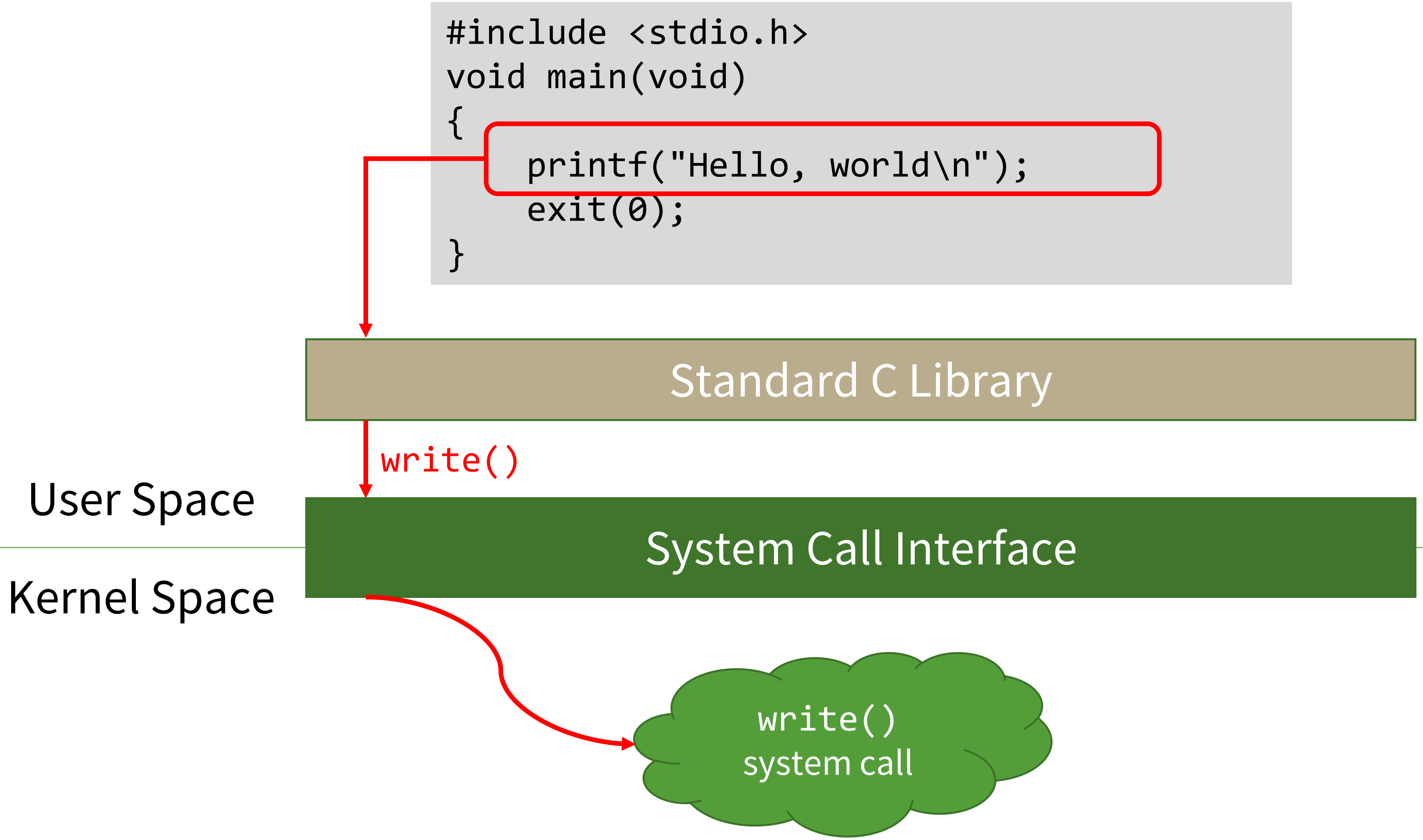
Simpler version

```
#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);
}
```

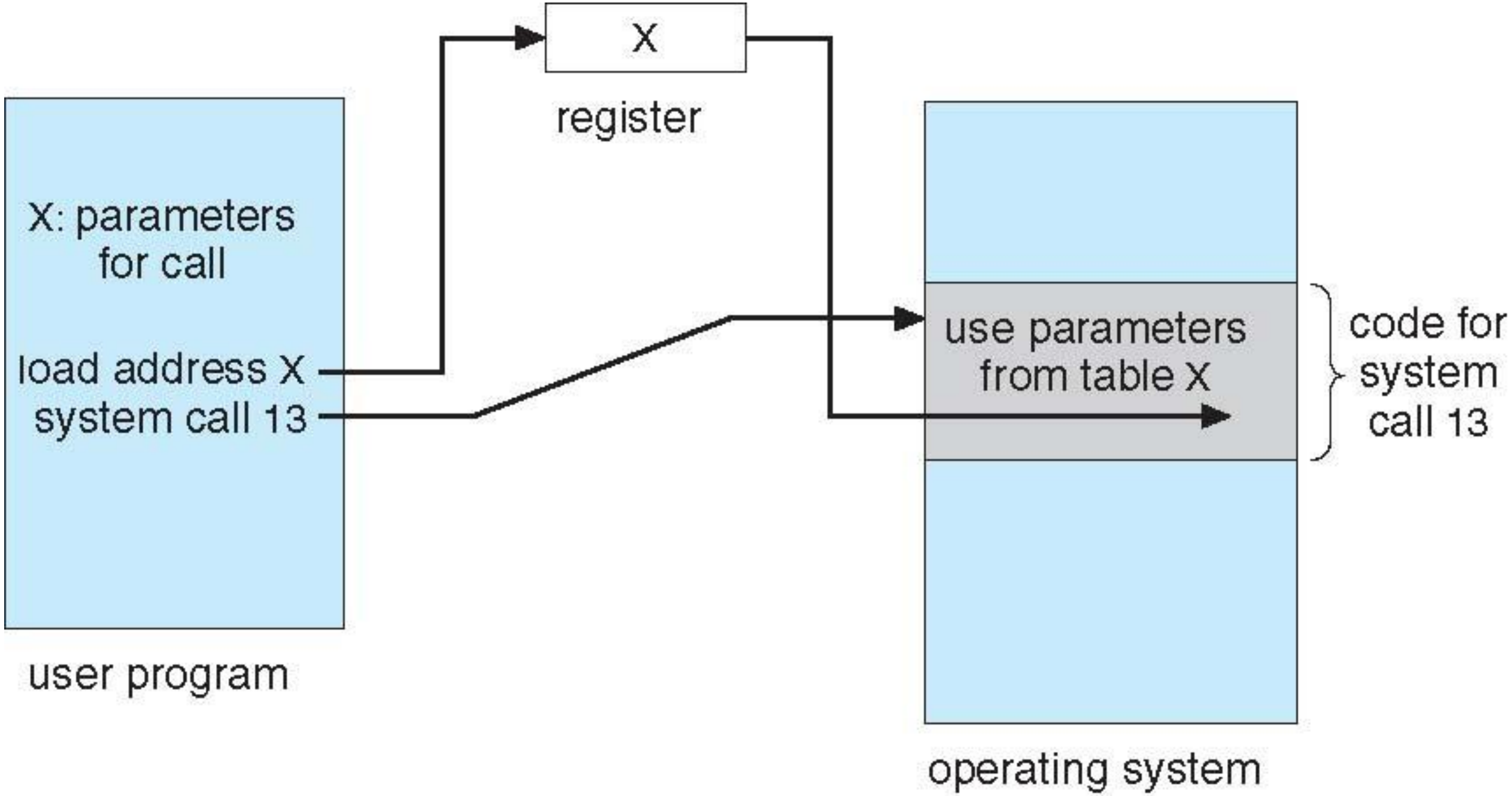
Parameter passing

- Often, more information is required than just identity of system call
  - Exact type and amount of information vary according to OS and call

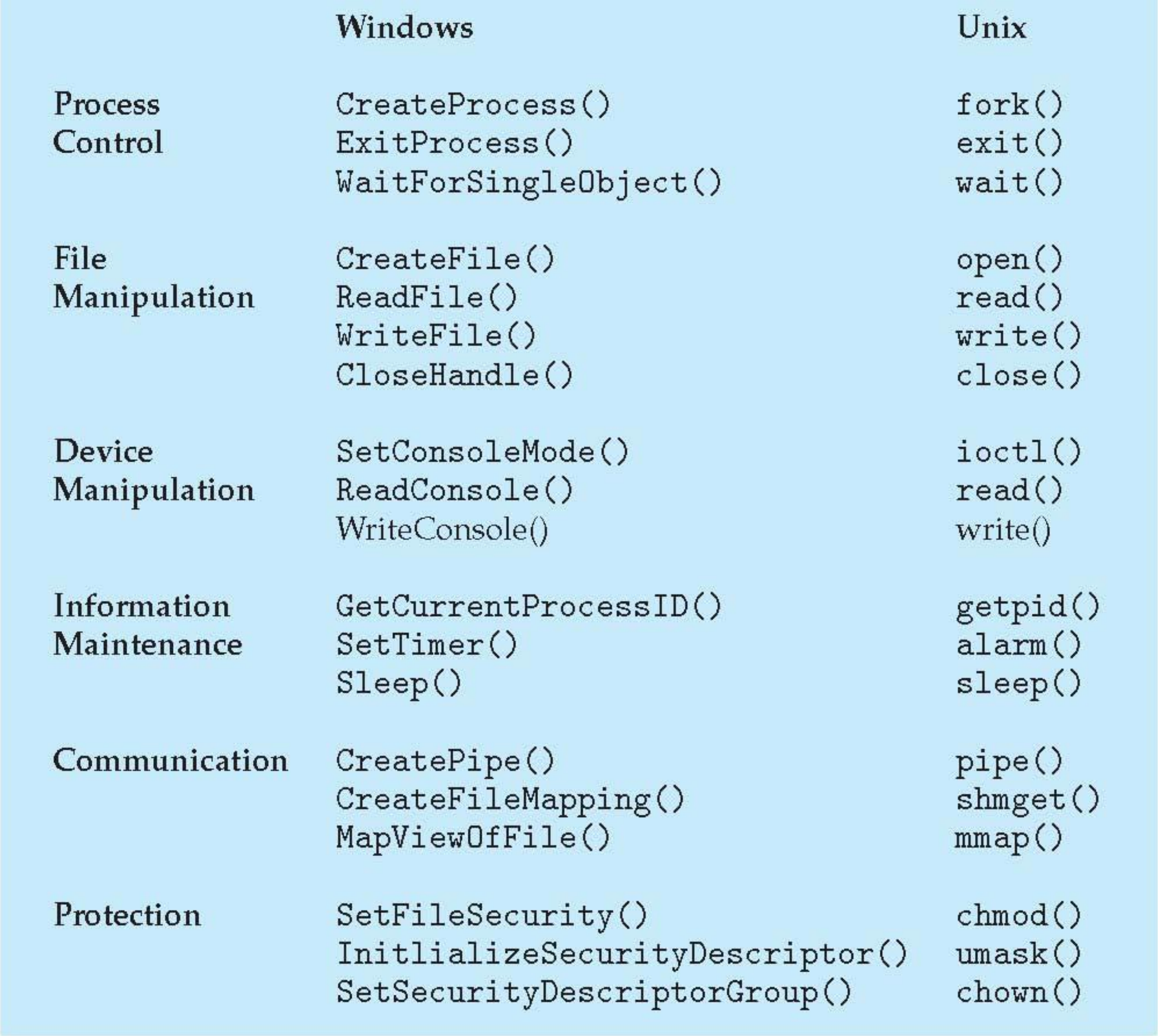
Three general methods used to pass parameters to the OS

- Simplest: pass the parameters in registers
  - In some cases, may be more parameters than registers
- Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - This approach taken by Linux
- Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed

Linux system call: passing parameters via table



Types and examples of system calls

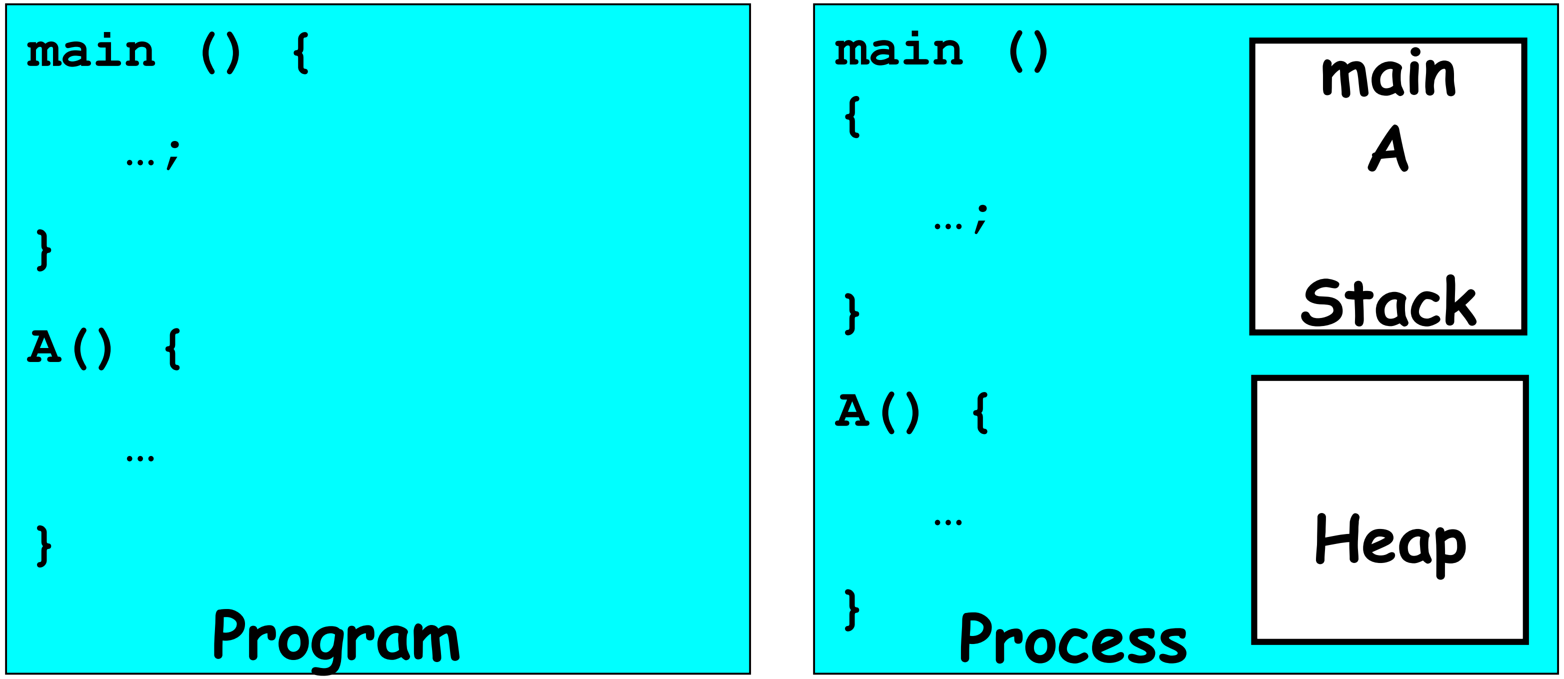


	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information 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 use POSIX standard

- POSIX: Portable Operating System Interface

Process vs program



```
main () {
    …;
}
A() {
    …
}
```

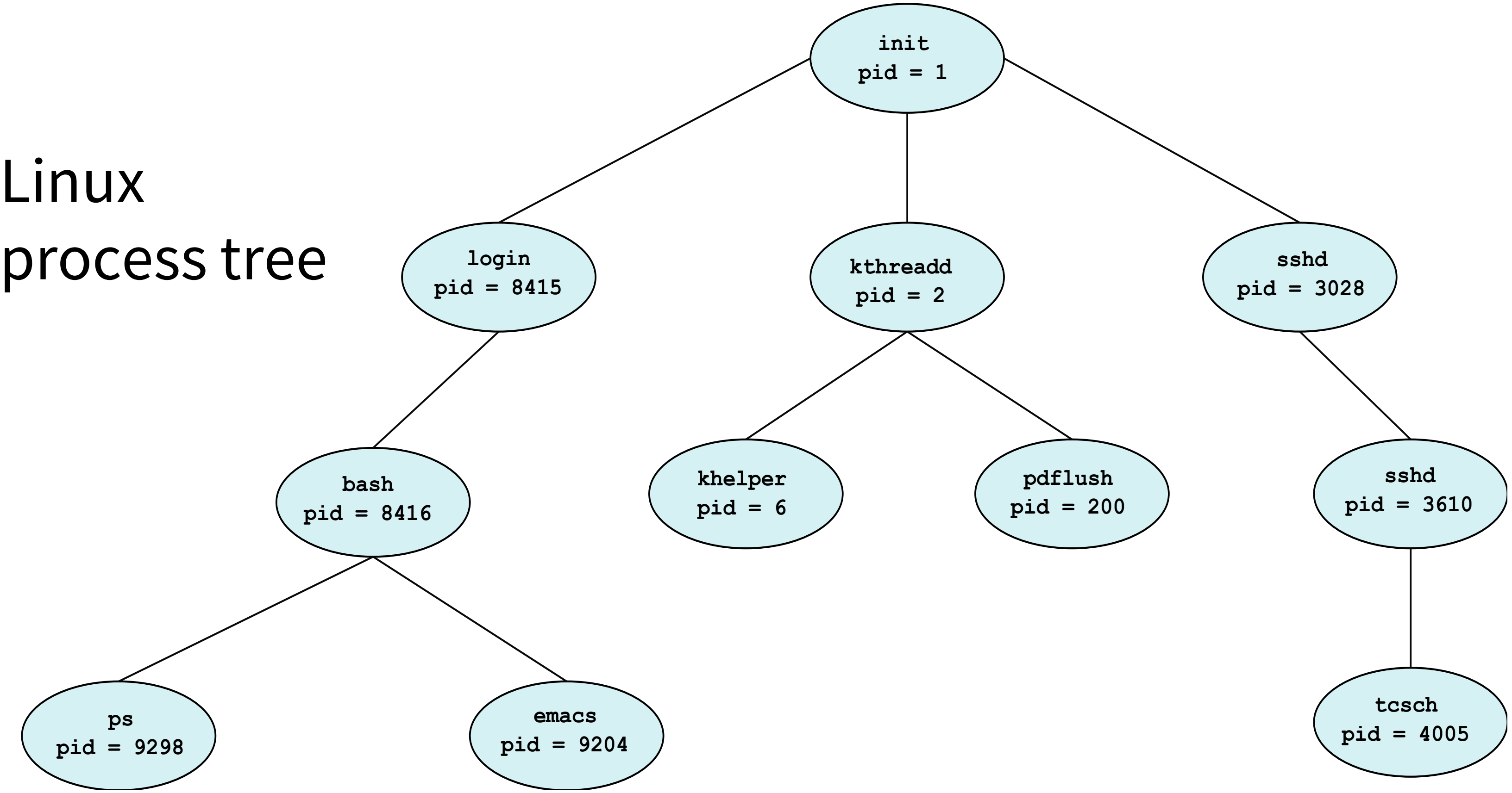
```
main ()
{
    …;
}
A() {
    …
}
```

- 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 management

- A process is created by another process, which, in turn create other processes → process tree

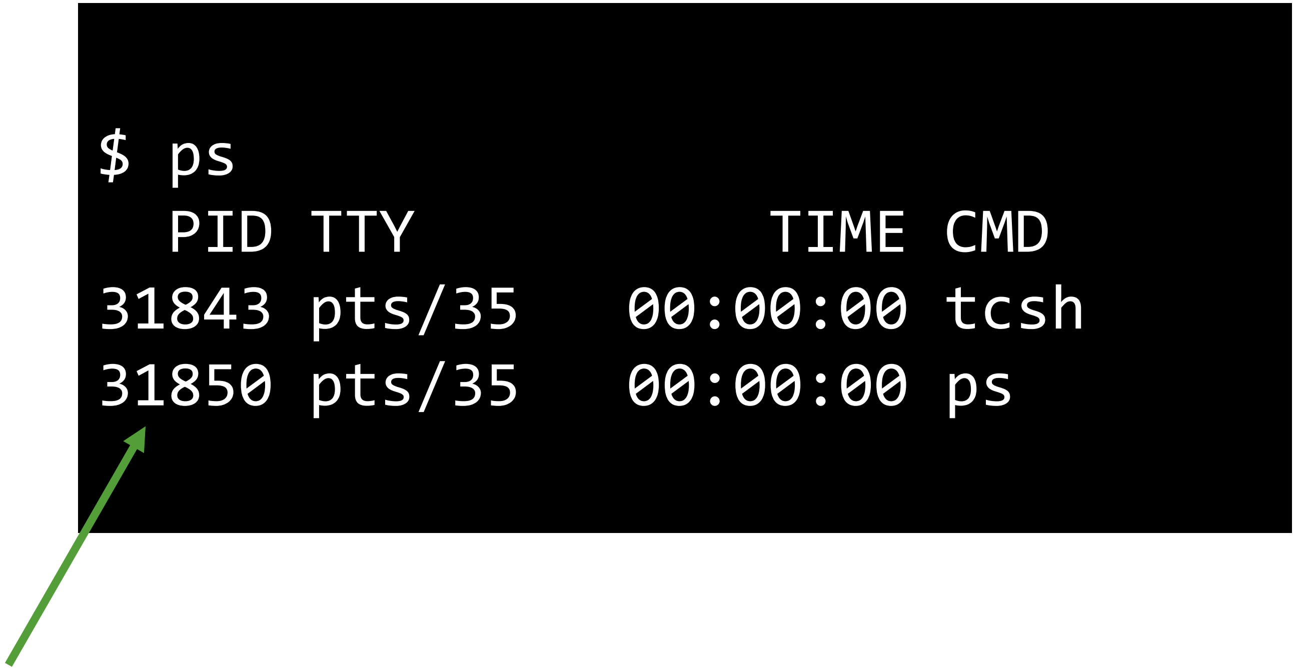
Linux process tree



Every process has a process ID (PID)

Linux ps command

- Used to obtain information about processes that are currently running



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

Process ID
Every process is assigned a PID by the kernel

Linux ps command

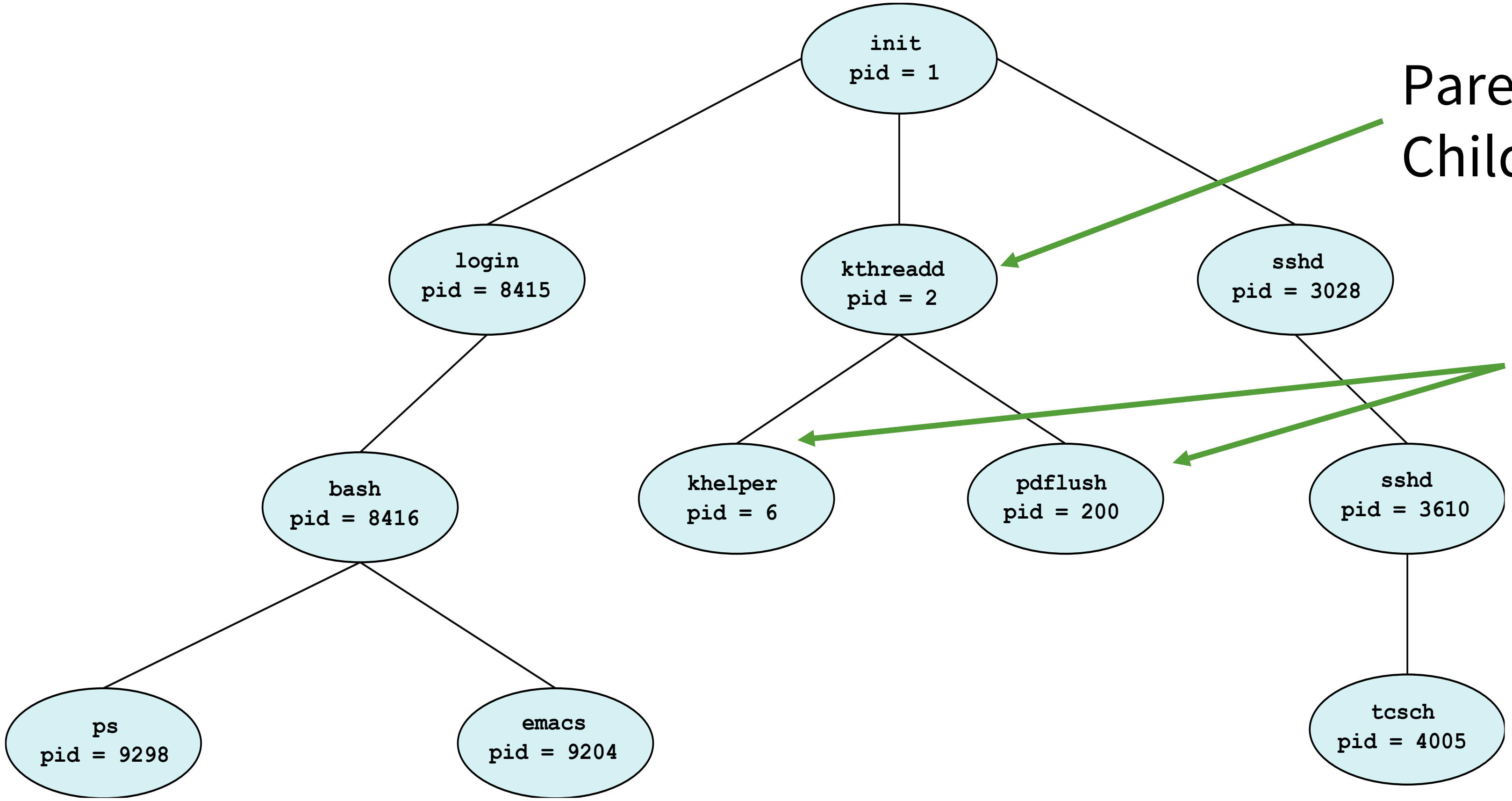
```
$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
alvin      31843 31835  0 12:37 pts/35   00:00:00 -tcsh
alvin      32100 31843  0 12:43 pts/35   00:00:00 ps -f
```

Parent Process ID

PID of the process that started the process

- In Linux, first process is called `init` and has PID of 1

Parent and child



Parent of processes 6 and 200,
Child of process 1

Children of process 2

What happens the parent of a process exits?

Parent and child

- After creating a child, the parent may either wait for it to finish or continue concurrently

- Daemon: a special type of process in Linux (and other Unix-like operating systems)
  - Created by a parent process that exits after giving birth to the child process

- Zombie: a process that has already exited but still has record in the kernel process table because the parent hasn't read the exit status yet

Next lecture

- Process management system calls