Lecture 3: System Calls

Fall 2019
Jason Tang

Slides based upon Operating System Concept slides,
http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html

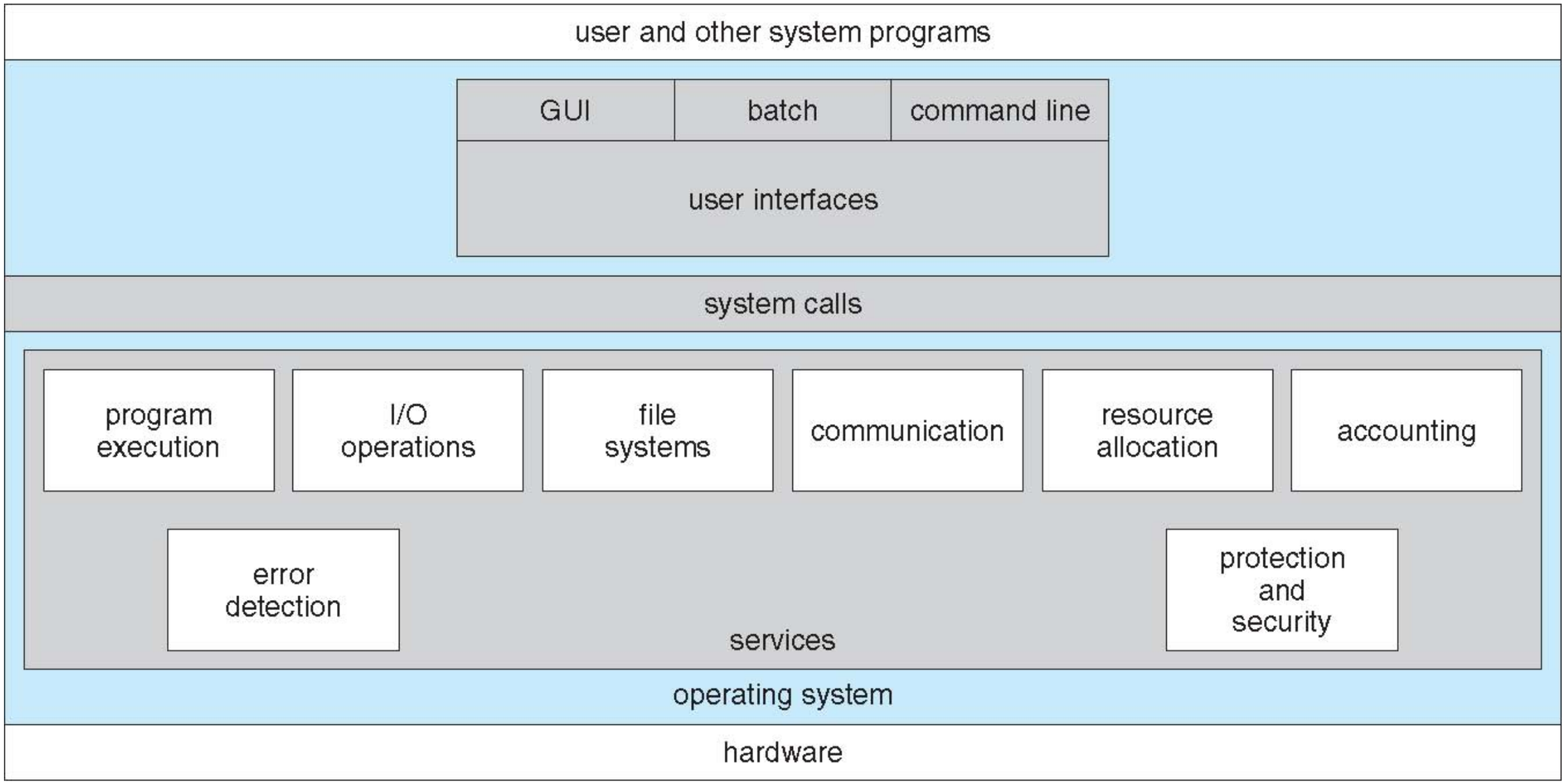
Topics

- System Calls
- Calling Conventions
- Kernel Designs

Operating System Services

- OS provides an environment to execute programs and provide services to programs and users
    - User interfaces: command line (CLI), graphical (GUI), batch
    - Program execution: loading program into memory, run program, end execution (normally or abnormally)
    - I/O operations: handle filesystem, networking, interprocess communication
    - Error detection: handle hardware failures, debugging
    - Resource allocation, accounting, protection, security

Operating System Services



System Calls

- On modern operating systems, processes do not talk to hardware directly, but **must** go through OS

- System Call: request from a process for OS to do some kind of work on its behalf

- Application Programming Interface (API): interface provided by OS, usually in a high-level language (C or C++), that is easier to work with than raw system calls

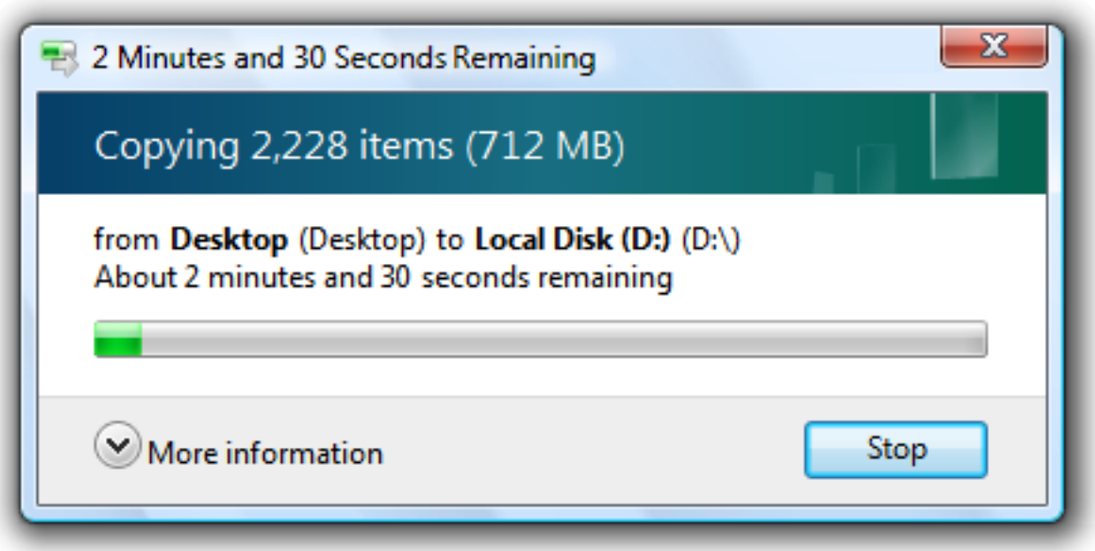
  - Win32 API for Windows, accessible through kernel32.dll

  - POSIX for macOS, Linux, and other Unix-like, accessible through libc.so

Example of System Calls

- Example: Sequence of system calls to copy contents of one file to another

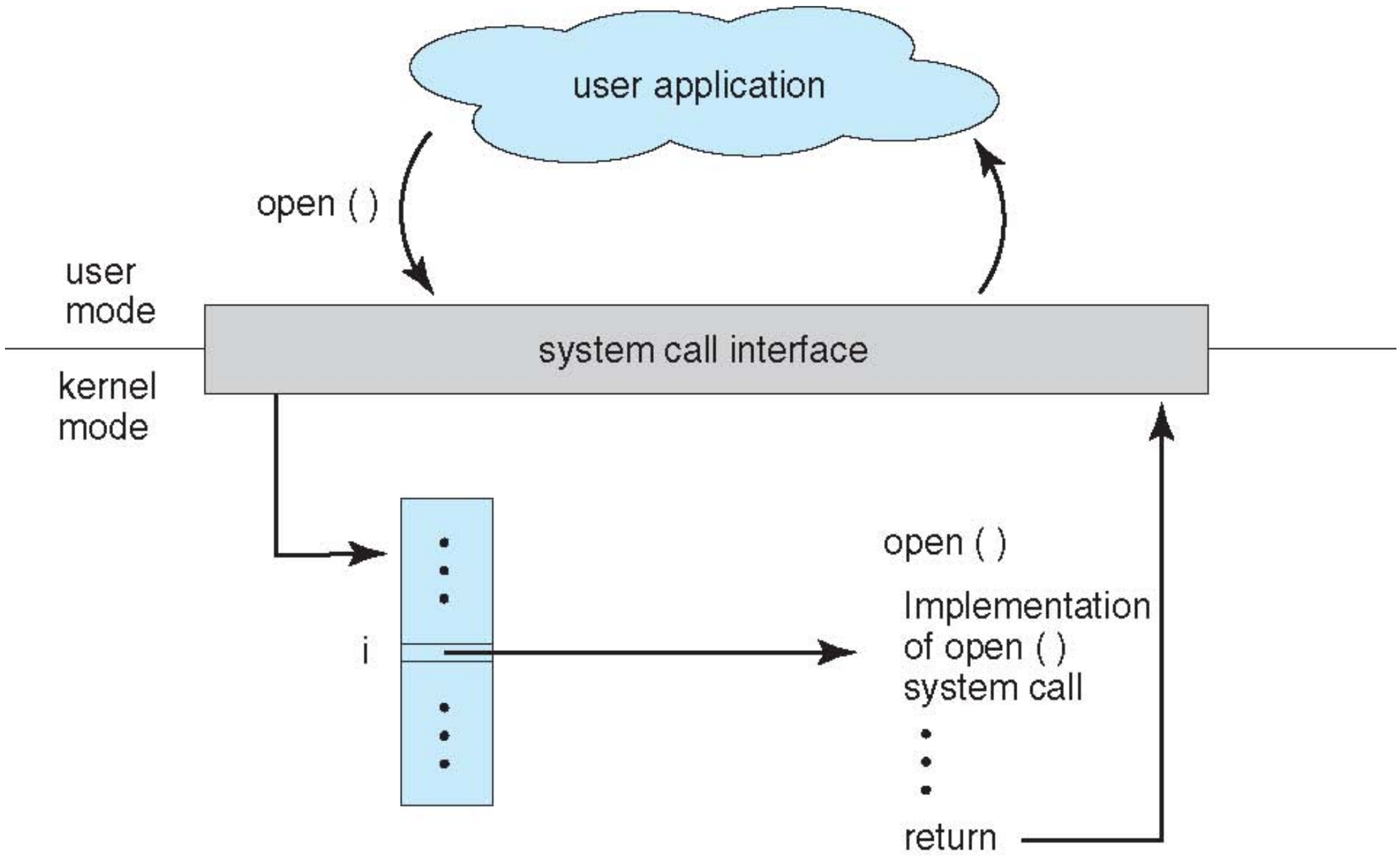
1. Display dialog to choose source file
2. Display dialog to choose destination folder
3. Display progress dialog
4. Open source file for reading
5. Open destination file for creating and writing
6. Loop while more source bytes remaining:
    1. Read *n* bytes from source file
    2. Write *n* bytes to destination file
    3. Update progress dialog
7. Close source file
8. Close destination file
9. Close progress dialog
10. Terminate normally



Typical System Call Implementation

- Each system call has a unique numeric identifier
    - OS has a system call table that maps numbers to functionality requested
- When invoking a system call, user places system call number and associated parameters in an “agreed upon” location, then executes the trap instruction
- OS retrieves system call number and parameters, then performs action
- OS writes output data and return value to an “agreed upon” location, then resumes user process

System Call Interface



Examples of System Calls

- Process control: create process, terminate, load program, get process attributes, wait for time, wait for event, allocate memory, obtain locks, debugging support

- I/O: create file, open file, read file, get file attributes

- Device management: request device, release device, read data

- Information maintenance: get system time, set system time

- Communications: send message, receive message, share data with another process

- Protection: get permissions, allow access, deny access

Parameter Passing

- OS writer and user programs rely upon convention when choosing where to store parameters and return values:
    - Simplest: put all values in registers (hopefully there are enough!)
    - Memory region: write to memory, then store starting memory address in a register
    - Push values onto stack; OS will pop values off the stack (based upon stack register)
- Usually, hardware constraints dictate which system call convention used

Linux x86-64 System Call Convention

1. User-level applications use as integer registers for passing the sequence %rdi, %rsi, %rdx, %rcx, %r8 and %r9. The kernel interface uses %rdi, %rsi, %rdx, %r10, %r8 and %r9.

2. A system-call is done via the syscall instruction. The kernel destroys registers %rcx and %r11.

3. The number of the syscall has to be passed in register %rax.

4. System-calls are limited to six arguments, no argument is passed directly on the stack.

5. Returning from the syscall, register %rax contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is -errno.

6. Only values of class INTEGER or class MEMORY are passed to the kernel.

https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI

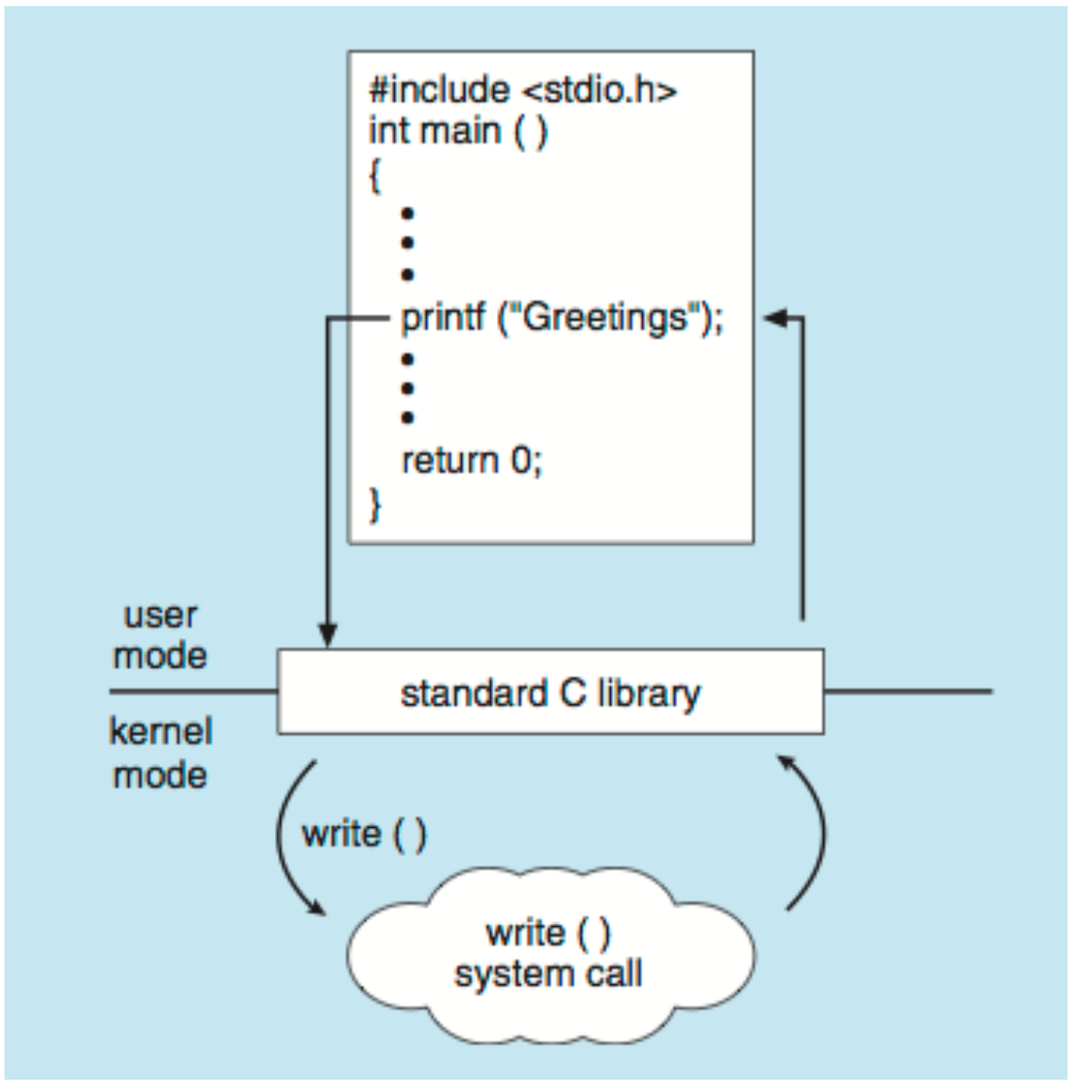
Linux x86-64 System Call Numbers

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0  common   read            __x64_sys_read
1  common   write           __x64_sys_write
2  common   open            __x64_sys_open
3  common   close           __x64_sys_close
4  common   stat            __x64_sys_newstat
5  common   fstat           __x64_sys_newfstat
6  common   lstat           __x64_sys_newlstat
7  common   poll            __x64_sys_poll
8  common   lseek           __x64_sys_lseek
9  common   mmap            __x64_sys_mmap
```

https://elixir.bootlin.com/linux/latest/source/arch/x86/entry/syscalls/syscall_64.tbl

Standard C Library Example

- Many standard C functions invoke underlying system calls
- Example: on Linux x86-64, `printf()` internally invokes `write()` (syscall number 1)



Operating System Designs

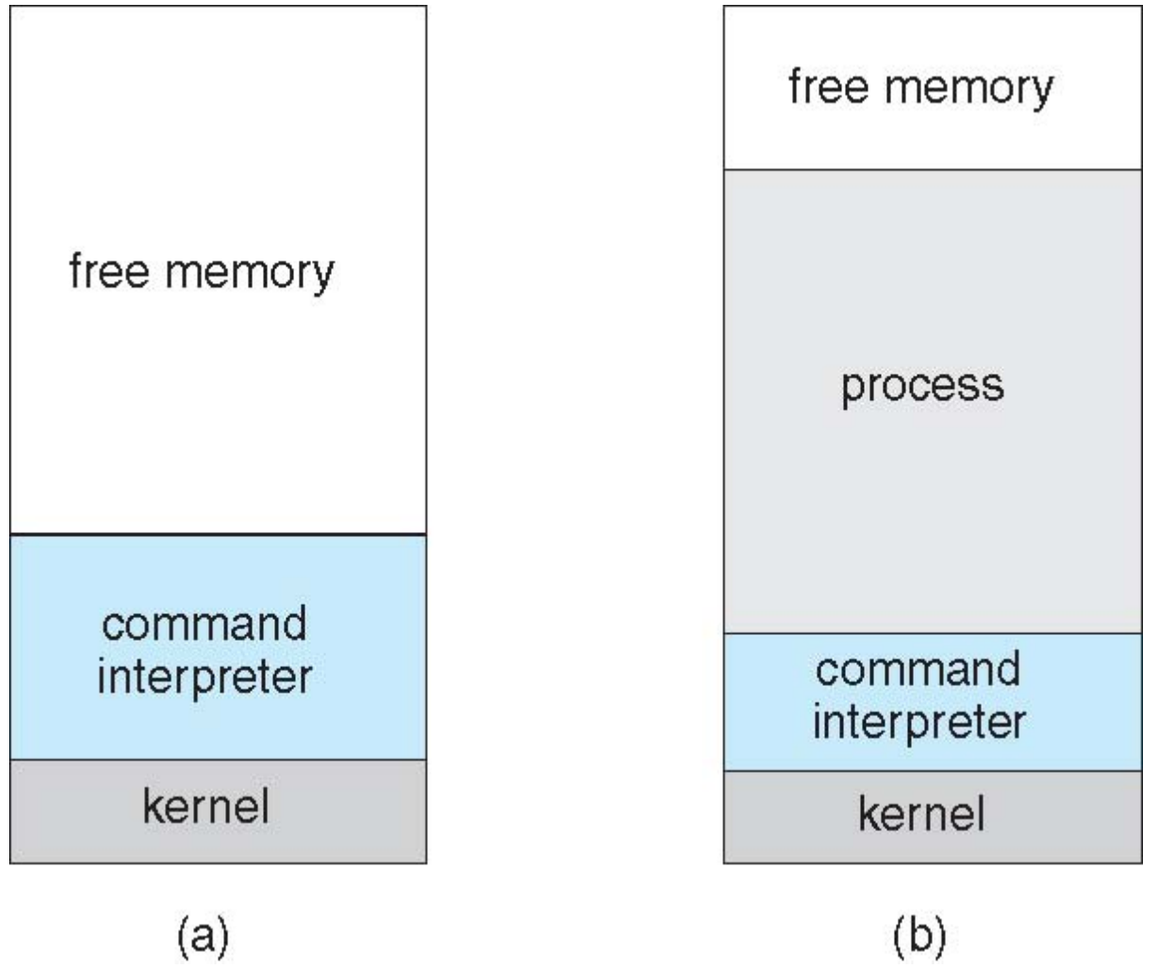
- Every OS has its purpose and internal design, though some designs are more successful than others

- Early OSes written entirely in assembly language; modern ones are written in C or C++

- Categories of OS design structures:
  - simple - MS-DOS
  - monolithic kernel - Unix
  - microkernel - Mach

Example: MS-DOS

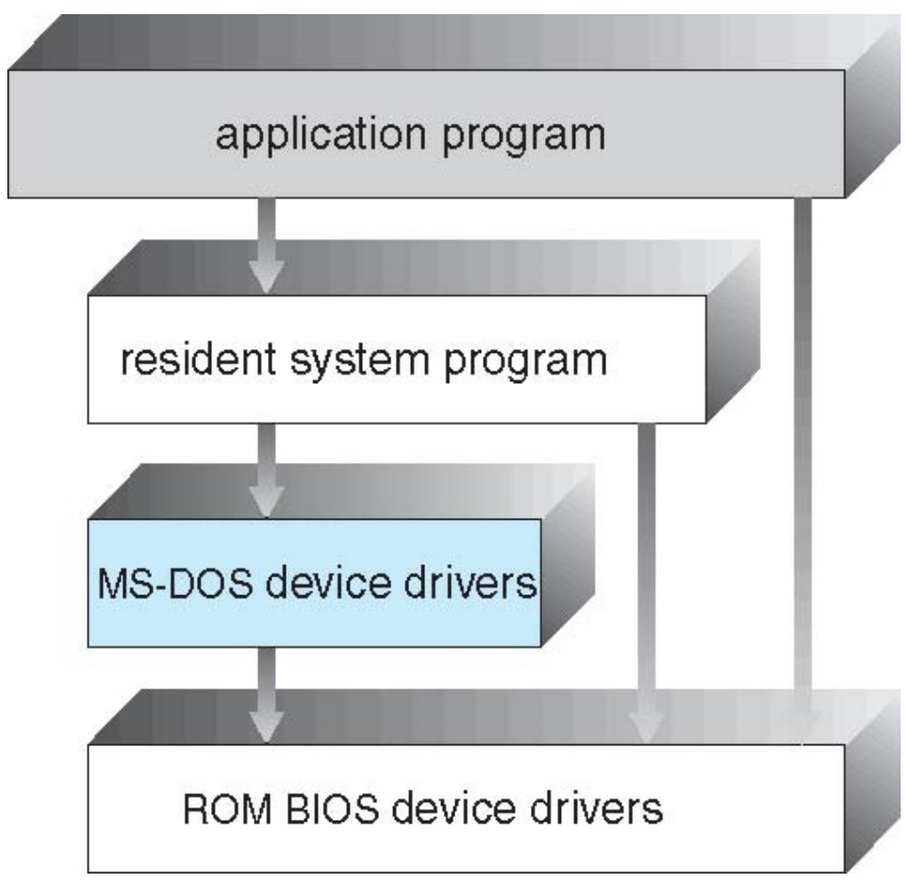
- Single-tasking, no protected-mode, single memory space
- Shell invoked when system booted
- Loads program into memory, overwriting all but the kernel
- Upon program exit, reload shell



At system startup While running program

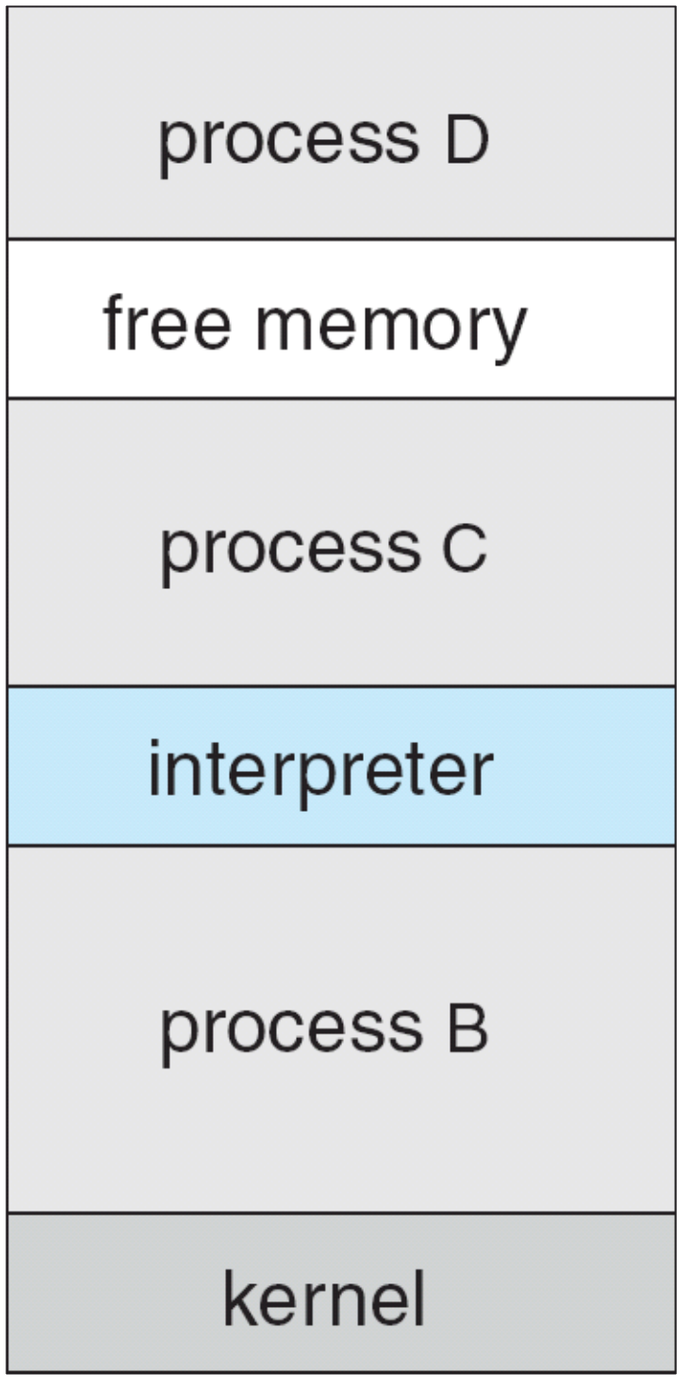
Simple Structure: MS-DOS

- Written to provide most functionality in least space
- Not divided into modules
- Interfaces and levels of functionality not well separated



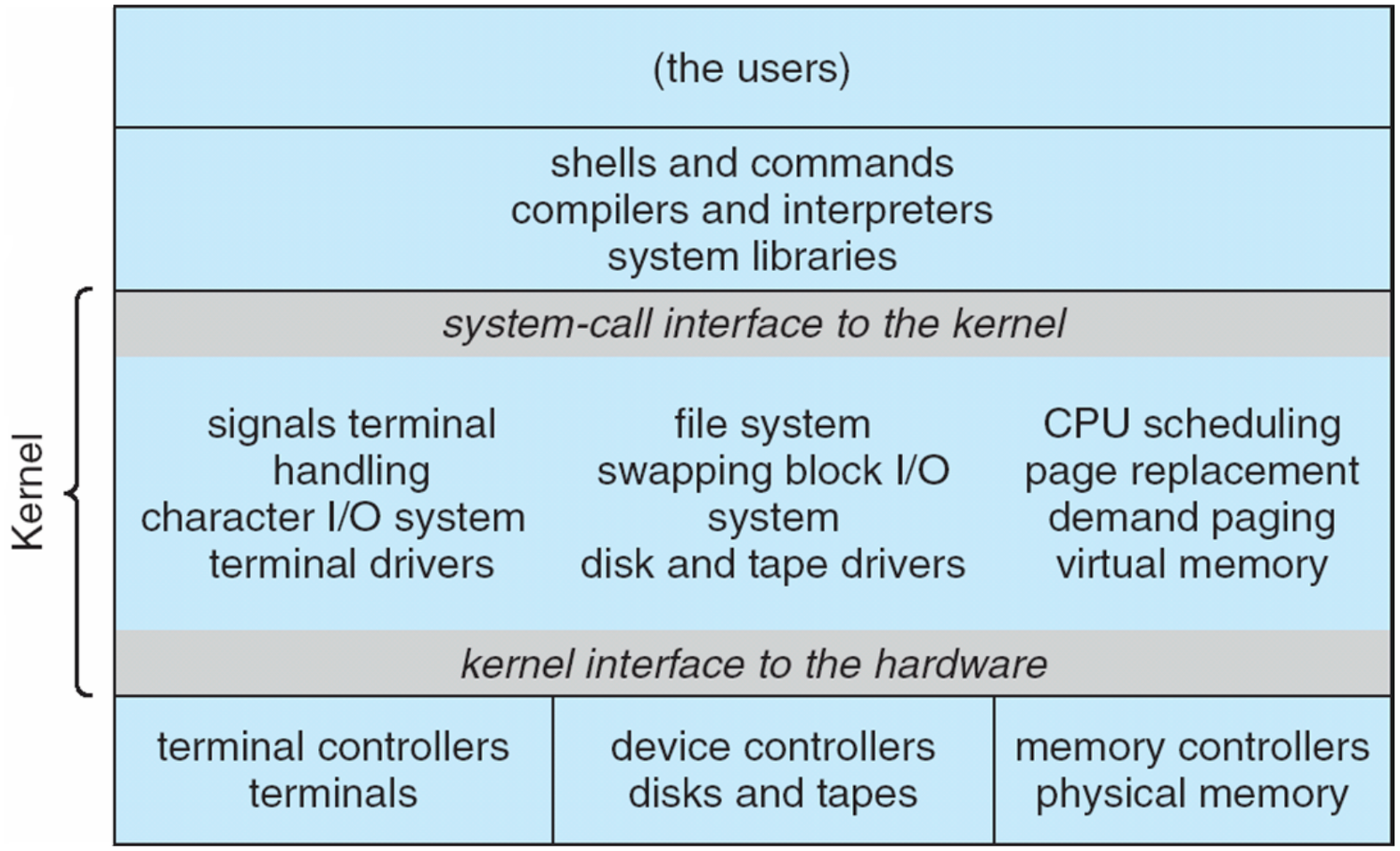
Example: FreeBSD

- Unix-like, multitasking, protected-mode
- Upon login, OS executes user's shell
- Shell executes `fork()` syscall to create new process
    - Child process executes `exec()` syscall to load and run program



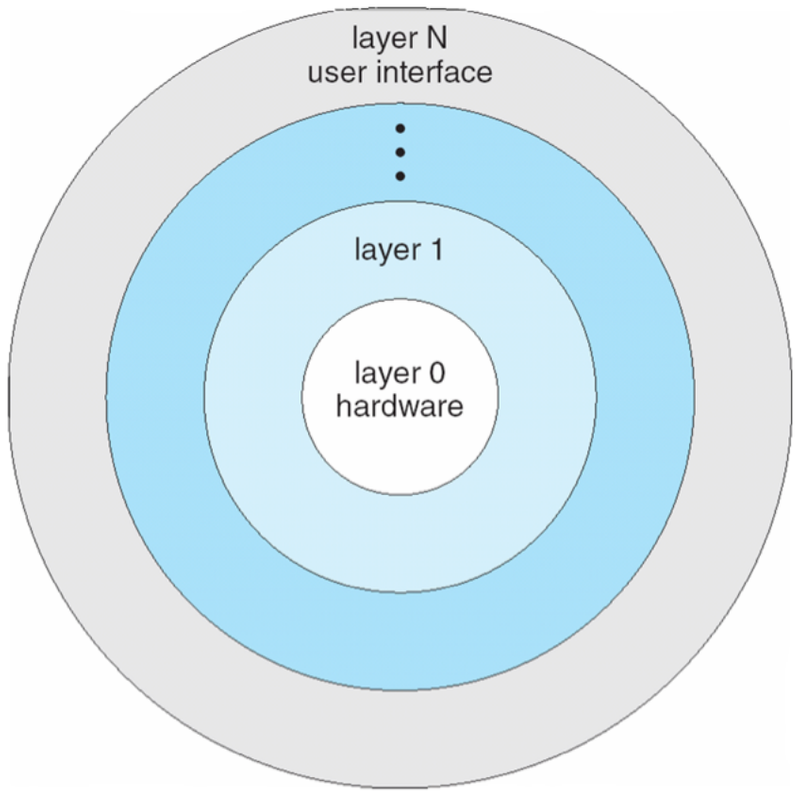
Monolithic Structure: Unix

- Kernel consists of everything between syscall interface and physical hardware
- Effectively a single process that handles everything



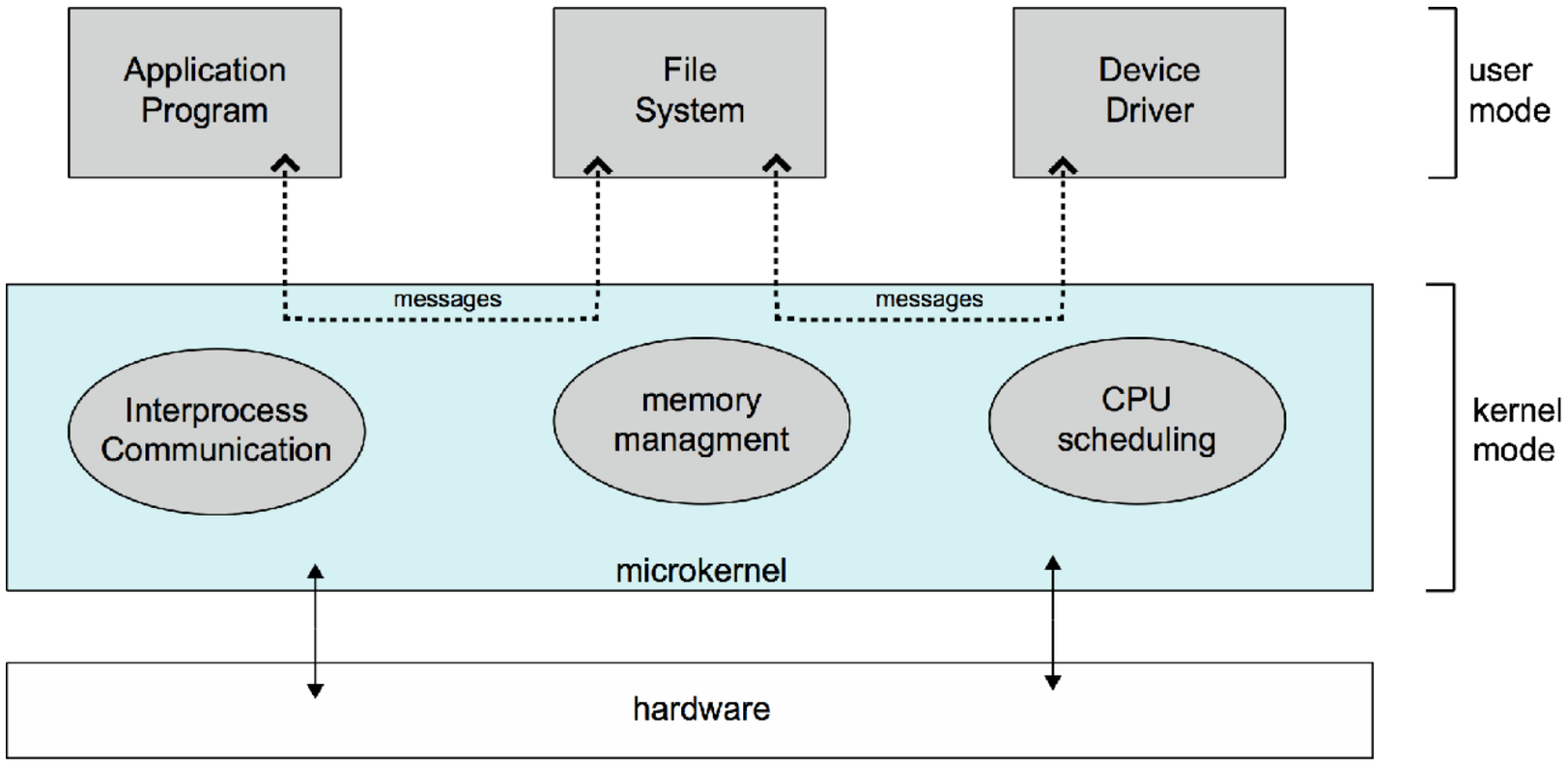
Layered Approach

- In practice, a monolithic kernel is divided into layers
- Layer 0 is hardware, layer 1 handles core functionality, layer *n* relies upon layer *n - 1*
- **Everything** but outermost user layer runs in kernel space



Microkernel Structure: Mach

- Microkernel handles memory allocation, process scheduling, and interprocess communication
- **Everything** else moved into a user process
- Filesystem accesses, networking, user interfaces, …



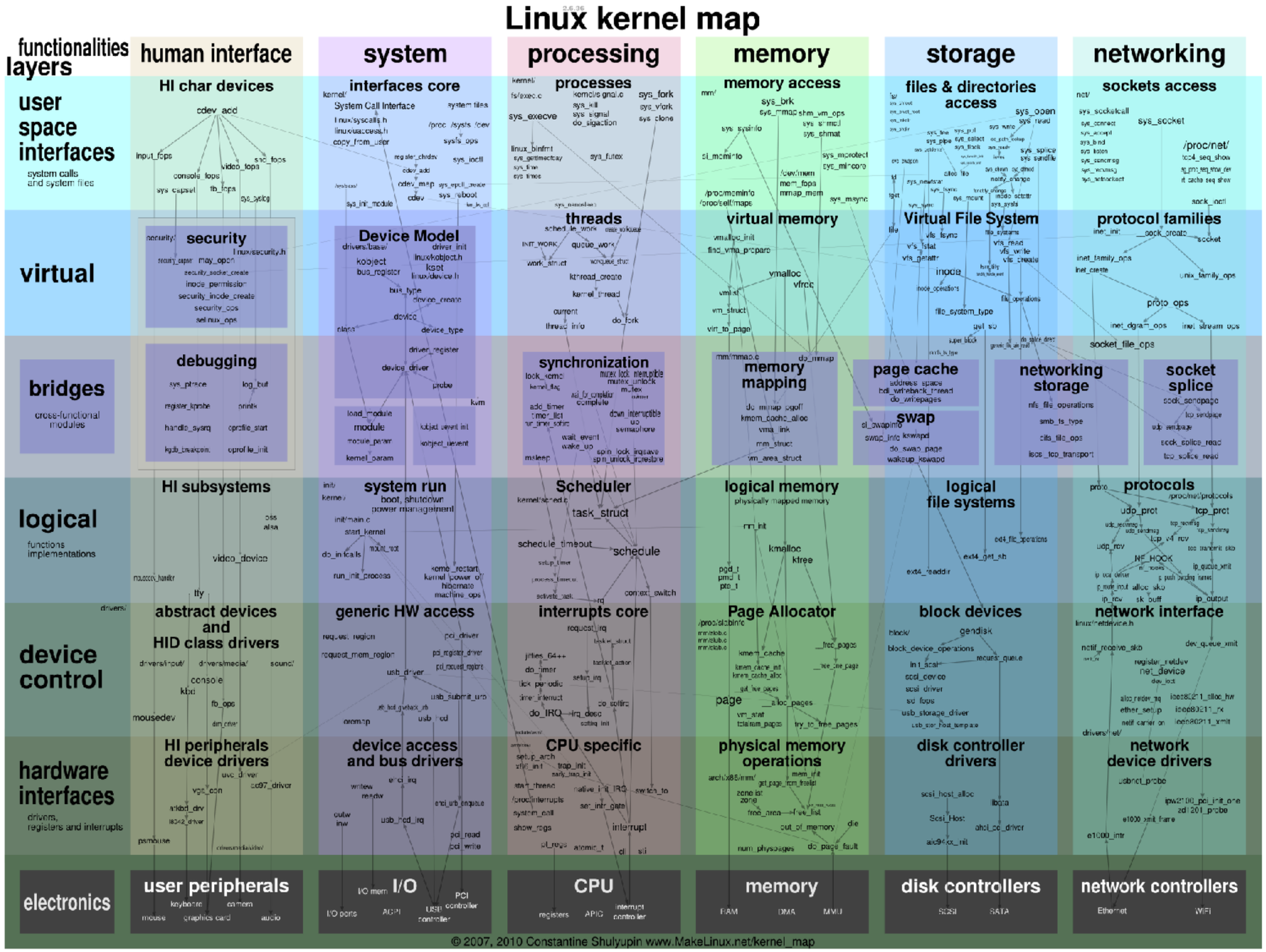
Monolithic versus Microkernel

	Monolithic	Microkernel
Reliability	If one driver crashes, entire kernel fails	Kernel can restart system processes as needed
Ease of Development	Must get entire kernel to work	Able to test just one part without affecting rest
Speed	No extraneous context switching, thus faster	Slower due to message passing
Memory	Relatively modest in memory usage	Memory footprint much larger

Hybrid Kernels

- In practice, modern OSes are hybrid, influenced by both monolithic and microkernel designs
  - Linux is more monolithic, Windows and macOS are more microkernel
- Loadable kernel modules: bit of code that kernel can load (and usually unload) while system is running, to extend functionality
  - Examples: Linux kmod, Windows device driver, macOS extension

Linux Kernel Design



http://www.makelinux.net/kernel_map/