The Programming Interface

Instructor: Youngjin Kwon

Let's think about interface

- OS designer
  - Know internals of how OS works
  - Know internals of how hardware works
- Application developer
  - Do not want to know internals of OS and hardware
- OS provides a simplistic view to use and manage OS internal resources by **APIs**
  - System calls

System call

What to consider designing APIs?

- What functionality should be in kernel?
- What functionality should be in user library?

→ How should OS be organized?

Some thoughts

- Let's focus on building APIs for process creation

- We can put the functionality of managing processes and IO in user-level
  - *What is a problem?*

Security ..?

Some thoughts (Cont'd)

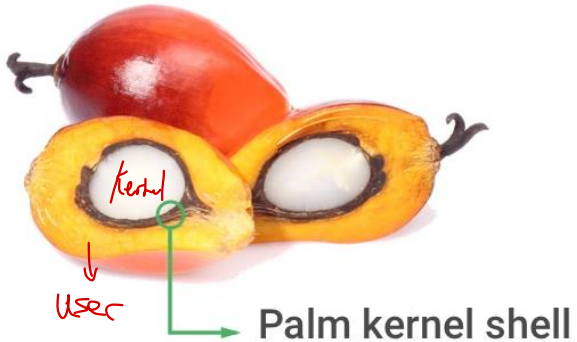
- So, the functionality must be in the kernel

- Then, how does the APIs look like?
  - Designing good APIs is a hard problem
  - Our approach: Let's look at use cases and build APIs to reflect the use cases

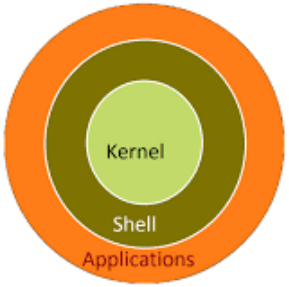
Main Points

- Creating and managing processes
  - fork, exec, wait
- Performing I/O
  - open, read, write, close
- Monolithic kernel & Microkernel

Use case: Shell



- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some tasks
  - Windows, MacOS, Linux all have shells



- Example: to compile a C program

  $ cc –c sourcefile1.c ➡ It requires create a process and write data to sourcefile1.o

  $ cc –c sourcefile2.c

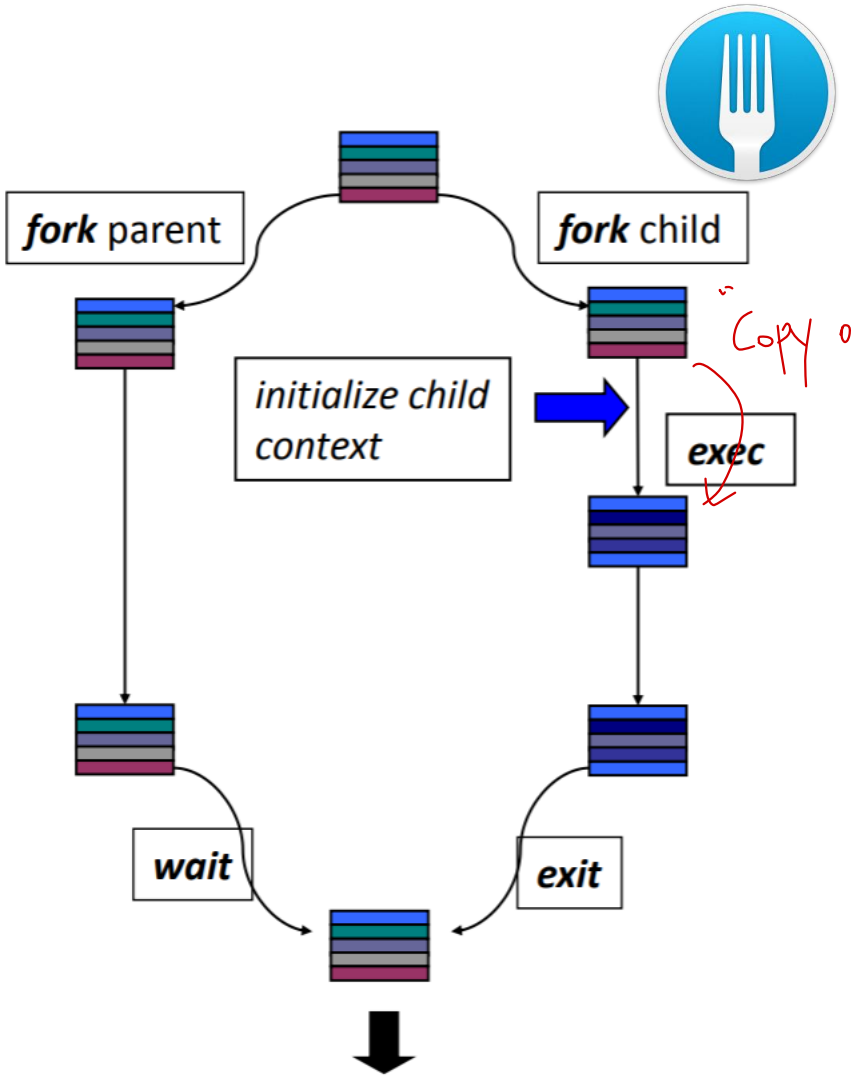
  $ ln –o program sourcefile1.o sourcefile2.o

Design questions

- If the shell runs at user-level, what system calls does it make to run each of the programs?
  - Ex: gcc, ln

- Why does process creation need system calls?
  - Why kernel-level? → for protection
  - Who has to manage process states? A parent process or kernel? ?

UNIX Process Management APIs



int pid = fork();

Create a new process that is a clone of its parent.

"Copy on write" → lazy copy when write happens at child

exec*("program" *[, argvp, envp]*);

Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);

Exit with status, destroying the process.

int pid = wait*(&status);

Wait for exit (or other status change) of a child.

Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {          // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                       // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Questions

- Can UNIX fork() return an error? Why?

ex. Too many processes, No memory to allocate

- Can UNIX exec() return an error? Why?

ex. No binary file to execute

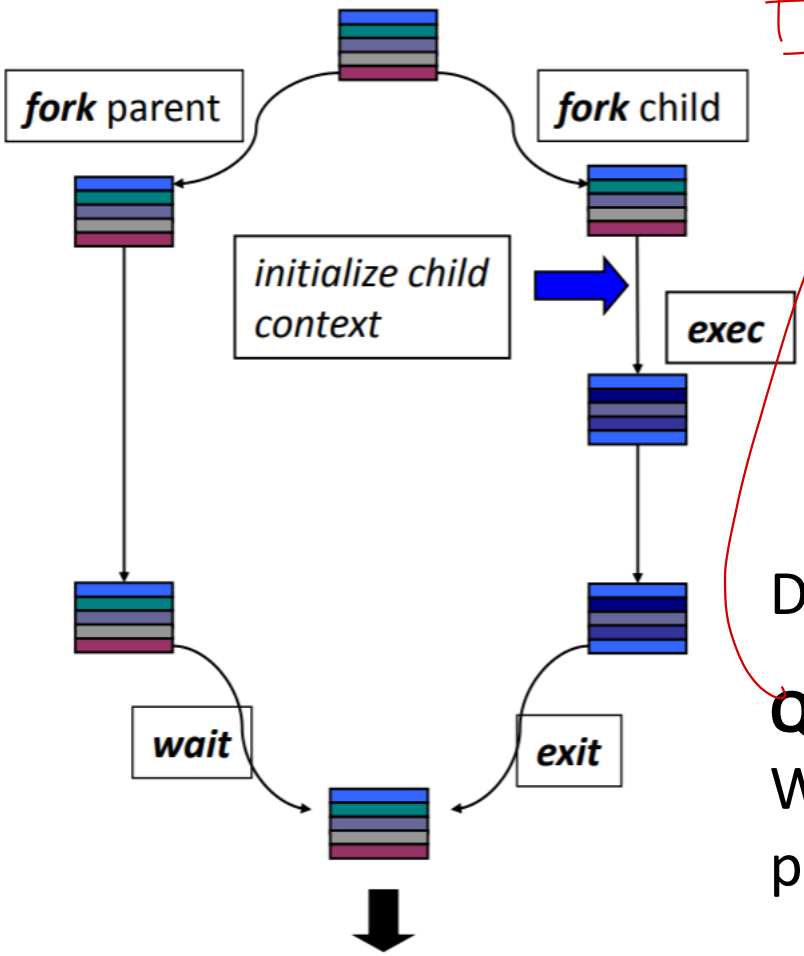
- Can UNIX wait() ever return immediately? Why?

Child is already terminated

Error → Reported to user applications

"errorno"

UNIX Process Management APIs



Compile, and then memory, but why fork?

$ cc –c sourcefile1.c

parent

forks

Bash ($) → Bash (child)

→ Run compiler (cc) ← exec

→ [compile] ↓

→ exit

→ return to Bash ($) ← parent wait

Does execution path above match APIs?

Question: Why fork?

What if just run compile in the same process as bash ($)?

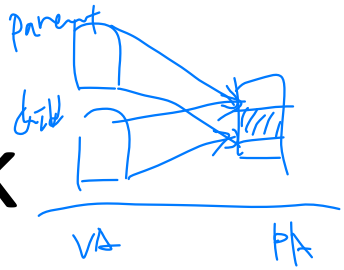
bash ⟷ compile

both don't believe each other

violate protection between bash & child

about person

Implementing UNIX fork



Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- *Initialize the address space with a copy of the entire contents of the address space of the parent*
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

Copy on write

Not exactly copy, optimization

to protect between parent & child

Implementing UNIX exec

- Steps to implement UNIX exec
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''

Application executable file

+ELF binary format

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // create a child process
    if (child_pid == 0) {
       exec(prog, args);       // I'm the child process.  Run program
      // NOT REACHED
    } else {
       wait(child_pid);       // I'm the parent, wait for child
       return 0;
    }
}
```

What to consider designing APIs?

- What functionality should be in kernel?

→ Protection

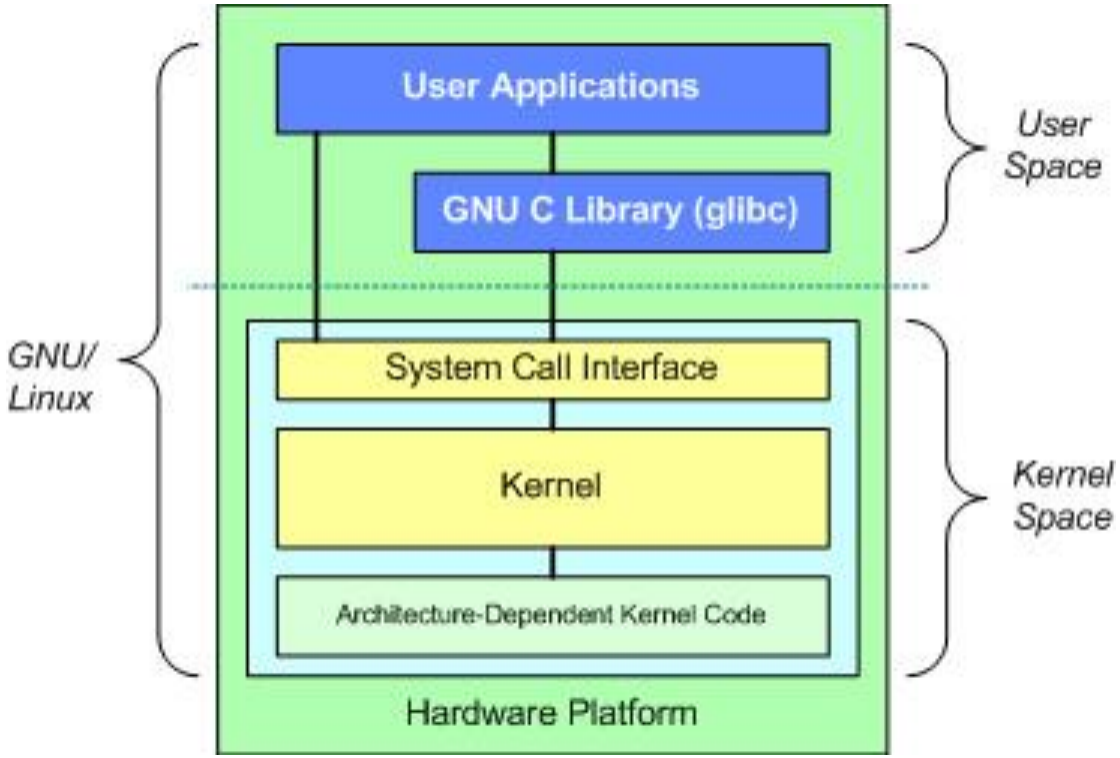
- **What functionality should be in user library?**

standard libc

→ Illusion, Abstraction

→ How should OS be organized?

Standard C library



Why is the standard library required?

Which goes which registers? → Protocol : Application Binary Interface (ABI)

A system call allows a program to request a service—for example, open a file or create a new process—from the kernel. At the assembler level, making a system call requires the caller to assign the unique system call number and the argument values to particular registers, and then execute a special instruction (e.g., SYSENTER on modern x86 architectures) that switches the processor to kernel mode to execute the system-call handling code. Upon return, the kernel places the system call's result status into a particular register and executes a special instruction (e.g., SYSEXIT on x86) that returns the processor to user mode. The usual convention for the result status is that a non-negative value means success, while a negative value means failure. A negative result status is the negated error number (`errno`) that indicates the cause of the failure.

All of the details of making a system call are normally hidden from the user by the C library, which provides a corresponding wrapper function and header file definitions for most system calls. The wrapper function accepts the system call arguments as function

write(fd, buf, ...) convenient

libc → ABI → kernel

register ..... (unconvenient)

https://lwn.net/Articles/534682/

What to consider designing APIs?

- What functionality should be in kernel?
- What functionality should be in user library?

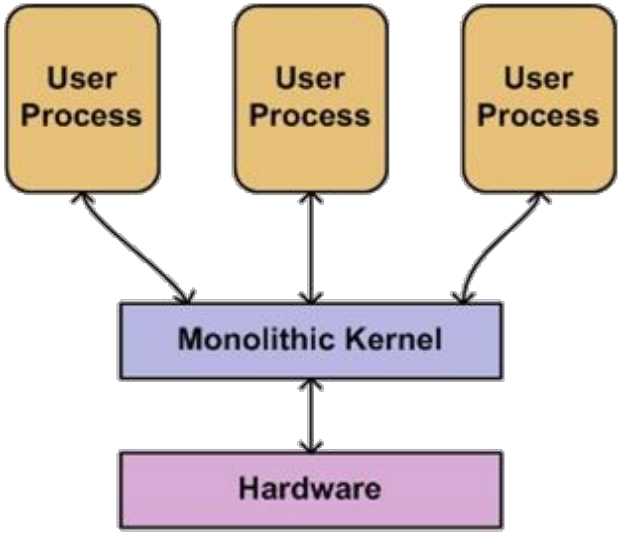
→ **How should OS be organized?**

option 1: All functionalities in a single kernel

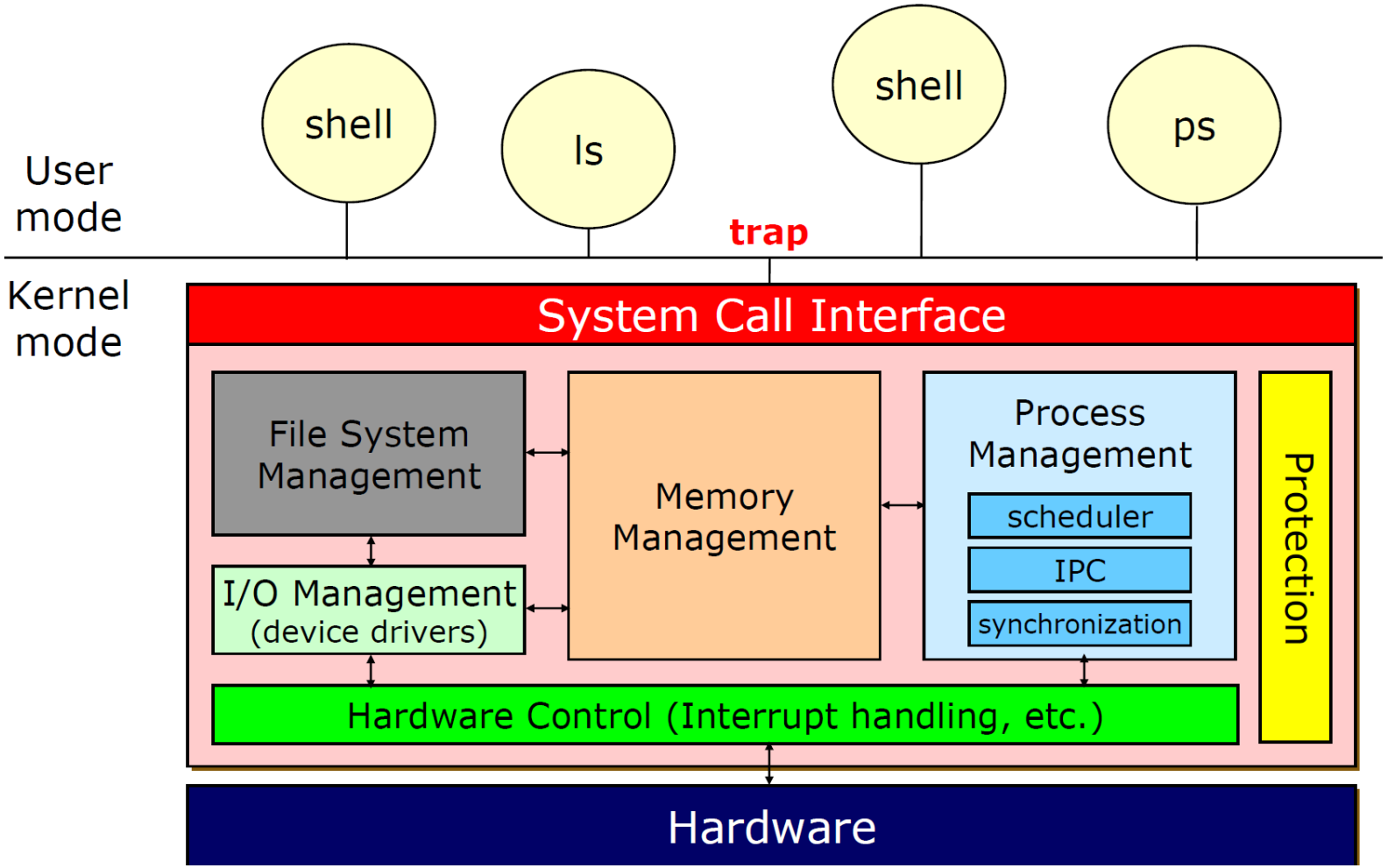
option 2: Functionalities are partitioned to kernel and several services

Monolithic Kernel

- All OS functionality is included in the kernel
  - Applications do not have control over resources
- Advantages
  - well-understood, **good performance**
- Disadvantages
  - **No protection btw kernel components**
  - Not easily extended/modified
- Example:
  - Windows, BSD, Linux



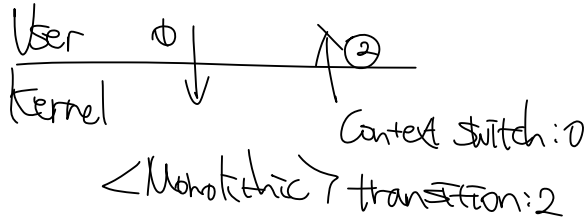
Operating System Structure

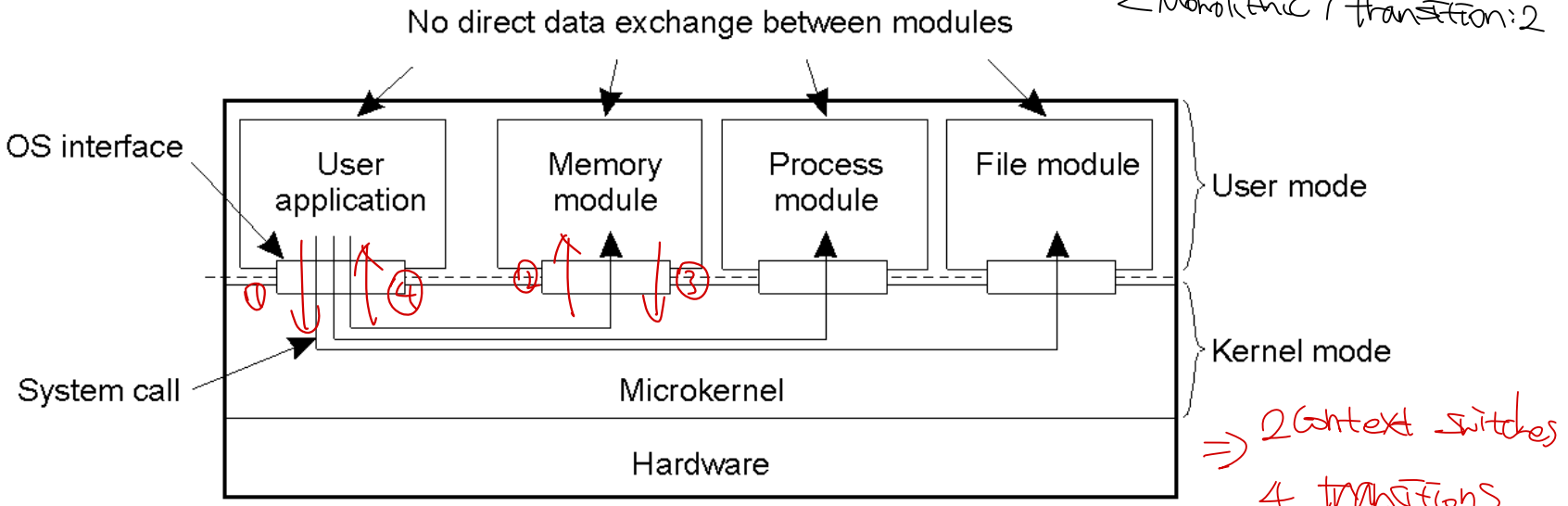


Microkernel

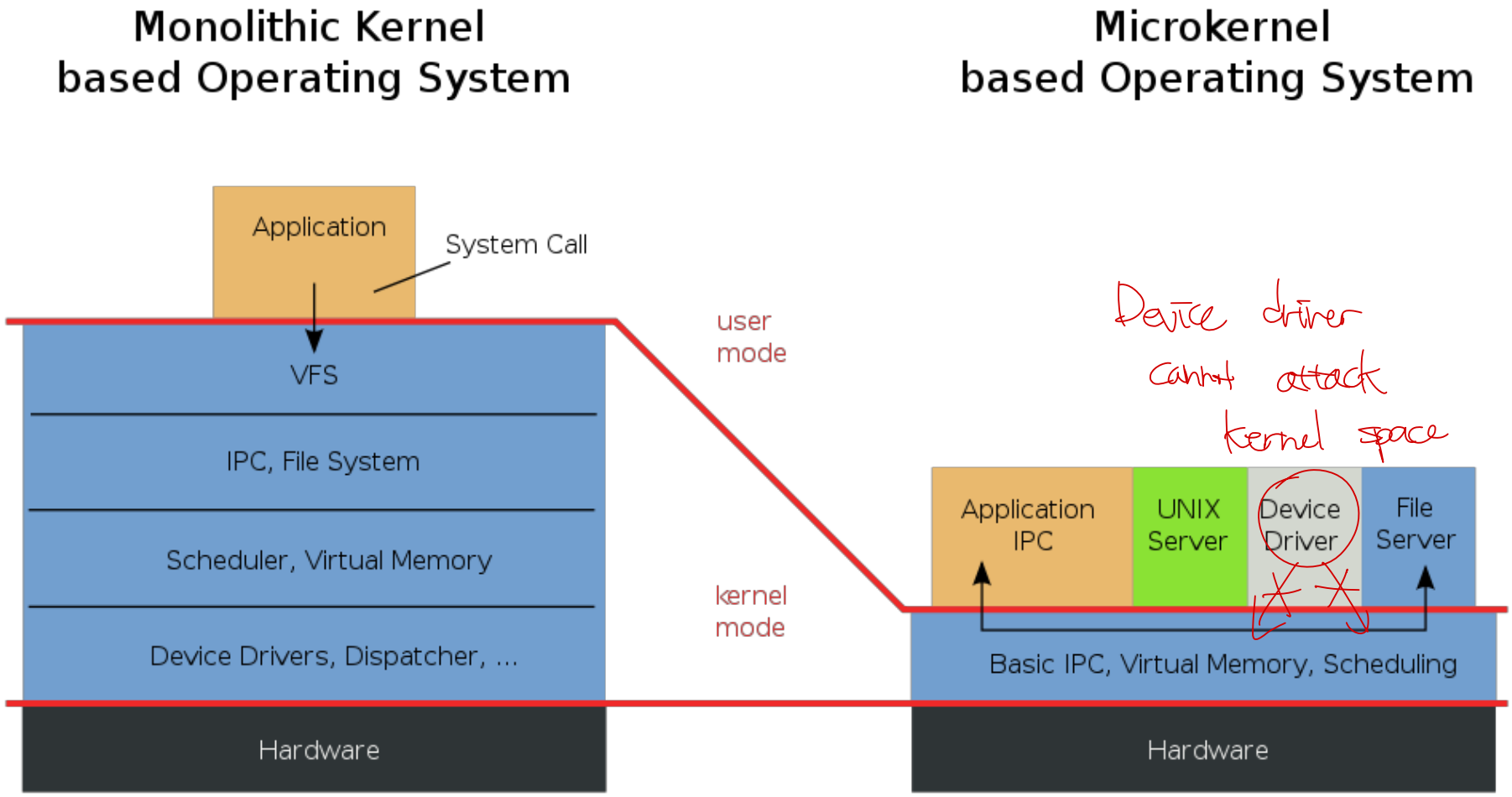
Slow: System call requires context switch

- OS kernel is very small – minimal functionality
  - The microkernel layer provides a set of minimal core services and is the interface to the hardware layer
- Other services (drivers, memory managers, etc.) are implemented as separate modules at user level
- Examples: Mach, L4





Monolithic Kernels VS Microkernels



Microkernel - Advantages

- Modularity → Modules in user-level
- Flexibility and extensibility
  - Easier to replace modules – fewer dependencies
  - Different servers can implement the same service in different ways — Modules
- **Safety (protection)**
  - Each server is protected by the OS from other servers
  - Servers are isolated; errors in one don't affect others
- Correctness
  - Easier to verify a small kernel

Microkernels- Disadvantages

- Performance issue
  - Slow – due to "cross-domain" information transfers?
    - Server-to-OS, OS-to-server IPC is thought to be a major source of inefficiency
  - Much faster to communicate between two modules that are both in OS

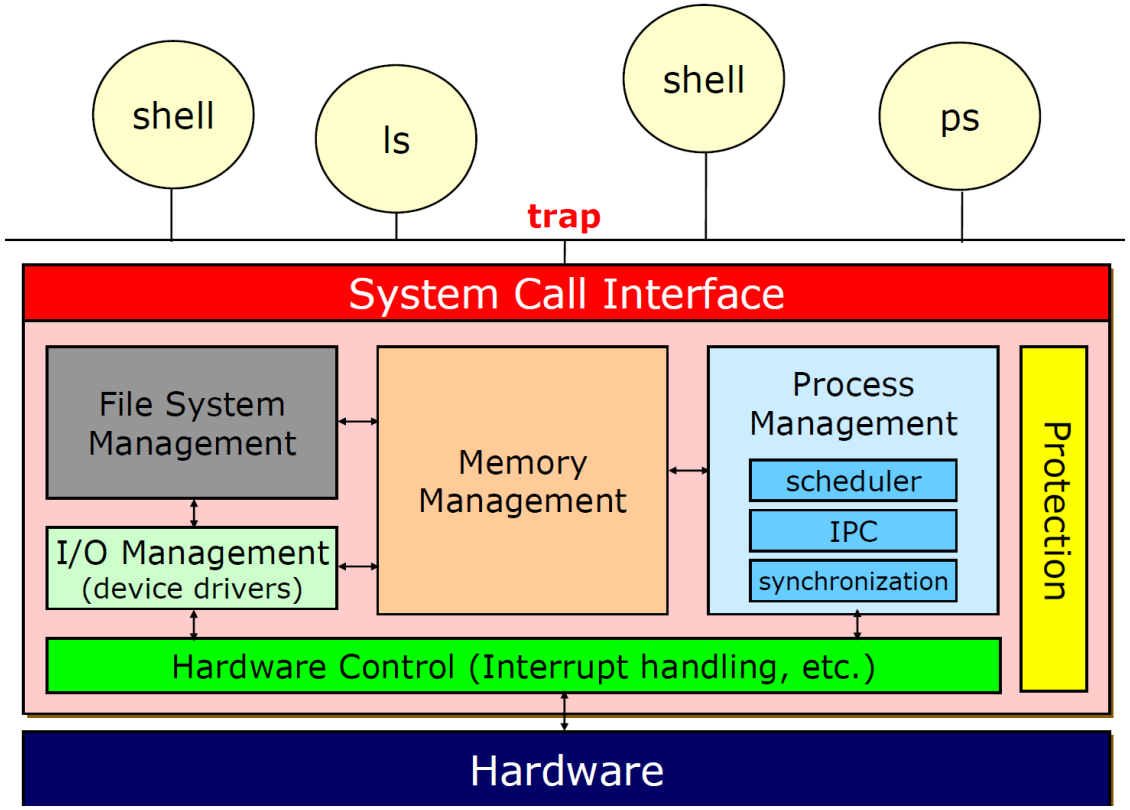
Inter Process Communication

Monolithic ?!

Hierarchy of Communication Mechanisms

- Fastest to Slowest
  - Function calls within same process
  - System calls (mode switch)
  - Context switch (process switch)
  - IPC between processes (message passing, on the same or different machines)

Big picture: where are we?



We've discussed:

- Protection
- Hardware control
- Scheduling
- APIs for process management
- APIs for IO