

Operating Systems Concepts

System Calls and Context Switches

CS 4375, Fall 2025

Instructor: MD Armanuzzaman (*Arman*)

marmanuzzaman@utep.edu

September 15, 2025

Summery

- More xv6 system calls
- Inter process communication
 - Pipes
- In class activity
 - Implement IPC through pipes

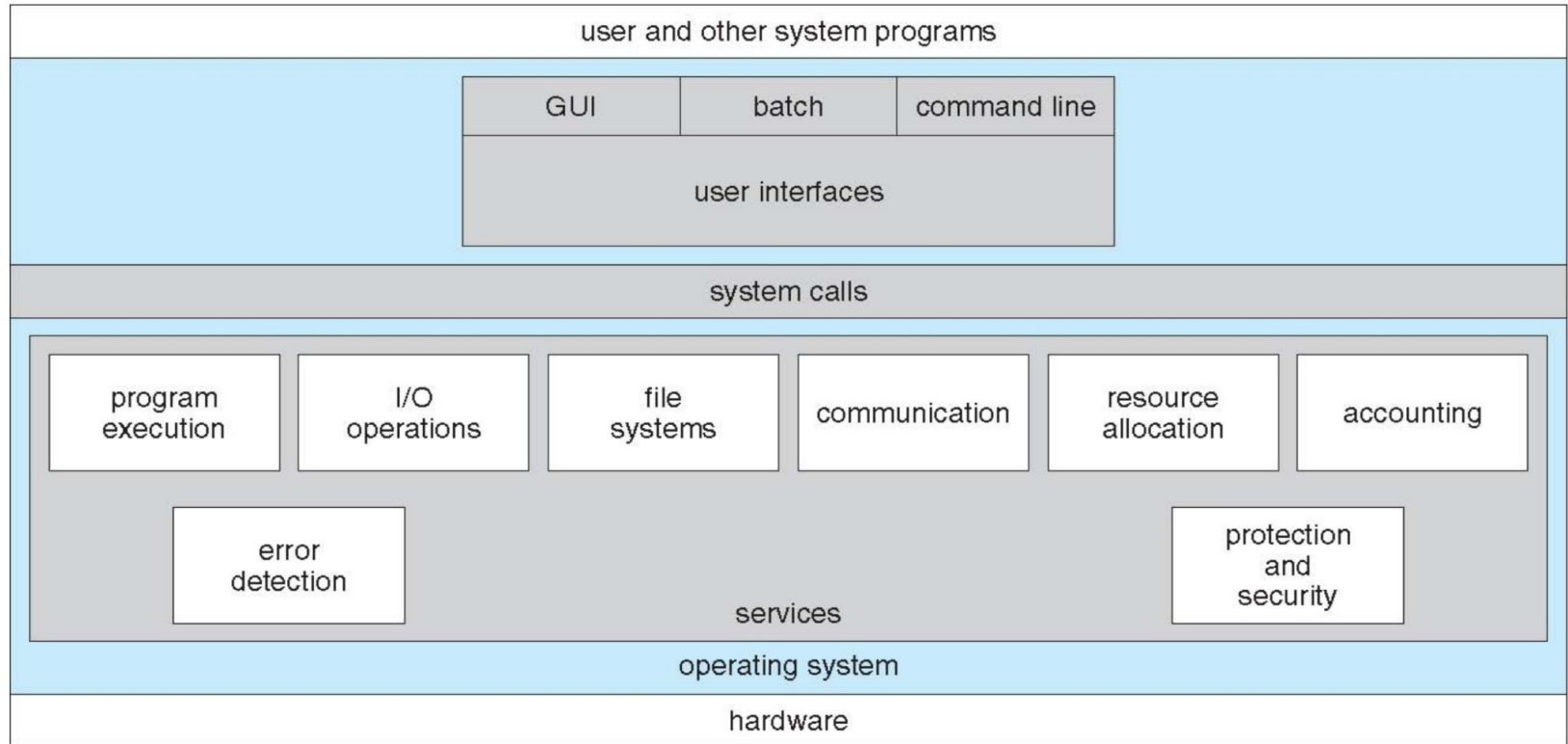
Agenda

- Operating system Services
- Operating system design choices
 - Direct execution protocol
 - Limited direct execution protocol
- System calls
 - Implementation of system calls
- Context switch between processes

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



Operating System Services

- Virtualize the CPU
 - Performance
 - Avoid overhead
 - Maximize CPU usage
 - Control
 - Provide performance while retaining control
 - CPU time, I/O resources, etc.

Basic direct execution protocol

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute call main()	Run main()
	Execute return from main
Free memory of process	
Remove from process list	

Analysis

- Pros
 - Fast: runs natively on the hardware CPU
- Cons
 - What about restricted operations?
 - I/O request to a disk, or gaining access to more system resources such as CPU or memory
 - Giving full access to a user process
 - A process can do whatever it wants

Analysis

- Pros

- Fast: runs natively on the hardware CPU

- Cons

OS and the hardware work together to provide limited direct execution

- Giving full access to a user process

- A process can do whatever it wants

Solution

- Hardware
 - Different modes of execution
 - **User mode**
 - Does not have full access to resource
 - **Kernel mode**
 - OS have full access to the hardware resources
 - Provides special instructions to **trap** into the kernel and **return-from-trap** back to user-mode programs, and instructions for the OS to tell the hardware where the **trap table** resides in memory.

Solution

- OS
 - System calls
 - Allows the user program to request privileged operation
 - Flow of system calls
 - Trap into the kernel (special instruction)
 - Raises privilege level
 - Kernel execution the privilege operation
 - Return-from-trap to the user program (special instruction)
 - Reduced privilege

Solution

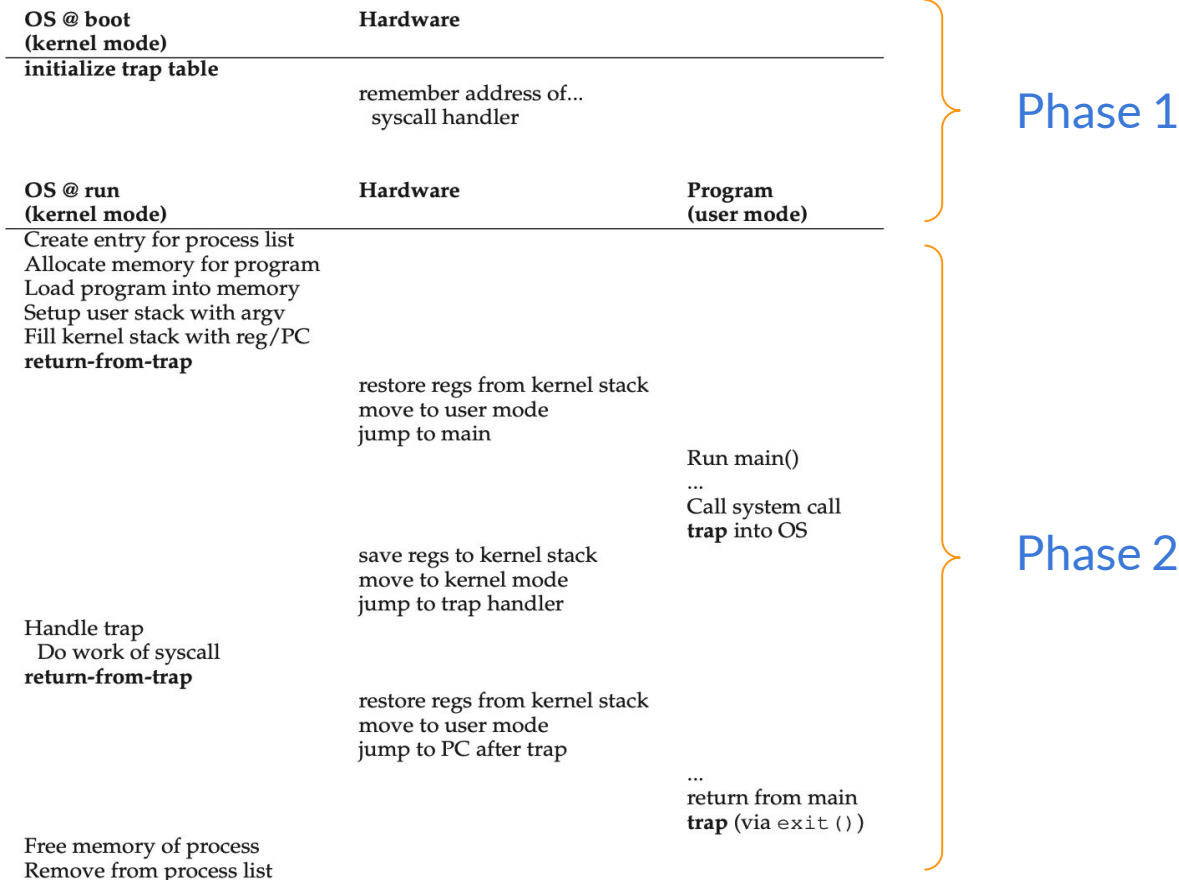
- OS
 - **System calls**
 - Allows the user program to request privileged operation
 - **Accurate return to the caller**
 - Needs to store important register
 - In x86 (architecture specific)
 - Per-process **kernel stack**
 - The hardware does the store or restore

Solution

- OS
 - System calls
 - Allows the user program to request privileged operation
 - Trap table
 - Kernel boots first (privileged)
 - Sets up the **trap table**
 - Informs the hardware about **trap handlers** (Privileged instructions)
 - Hardware remembers until next reboot

Allowing user processes to jump to anywhere in the kernel is dangerous!!

Limited Direct Execution Protocol



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

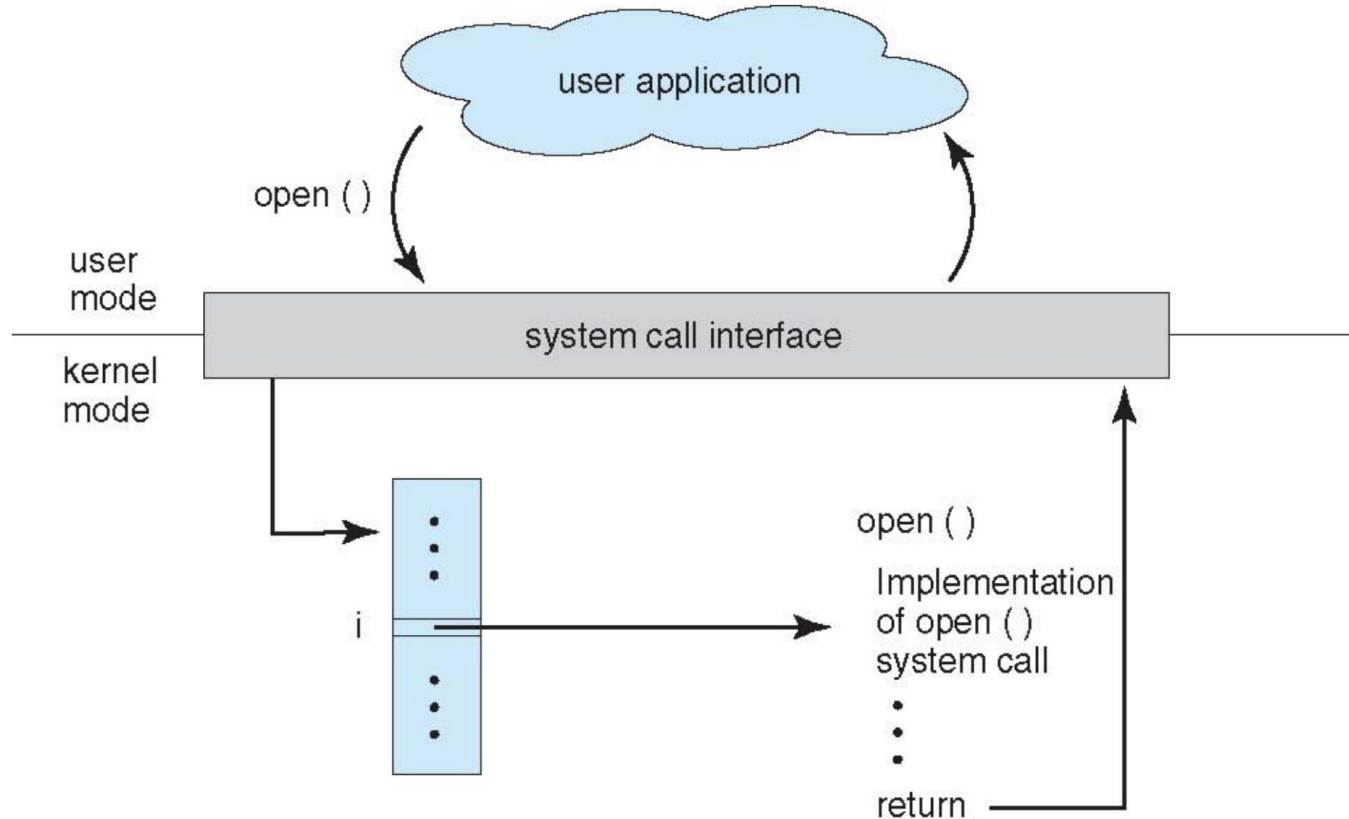
Typical System Call Implementation

- 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

Why system calls look like function calls?

- Syntax
 - `fork()`, `exit ()`, `open()` or `read()`
- Because it's a function call
 - Calls systems calls **internally**
 - Uses special **trap** instruction
 - Sets up values in **specific registers**

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

Parameter Passing

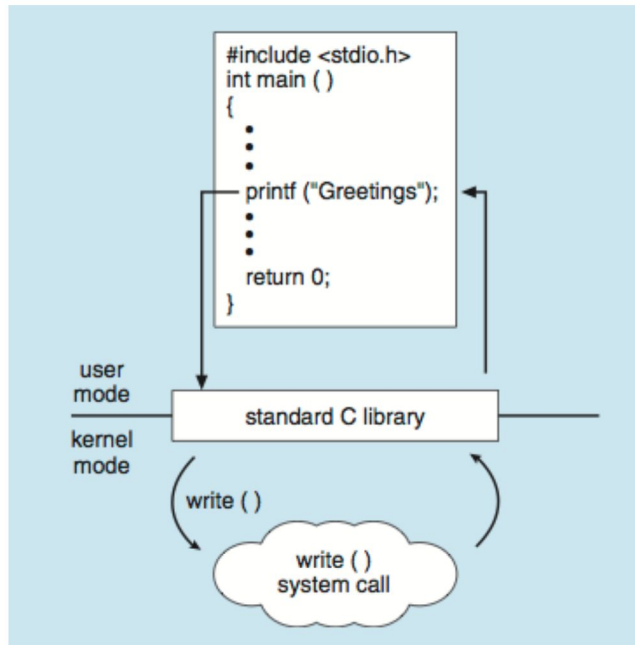
- OS writer and user programs rely upon convention when choosing where to store parameters and return values:
 - Simplest: put all values in **registers** (hopefully there are enough!)
 - **Memory region**: write to memory, then store starting memory address in a register
 - Push values onto **stack**; 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.

Standard C Library Example

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



Switch between processes

- No one wants to use a OS that can run one application at a time
- How can the operating system regain control of the CPU so that it can switch between processes?
- Cooperative Approach: Wait For System Calls
 - Common for most procedures
 - When kernel regains control, serve a different process
 - What is a process never makes a system call?
 - Malicious or by mistake

Switch between processes

- No one wants to use a OS that can run one application at a time
- How can the operating system regain control of the CPU so that it can switch between processes?
- Non-Cooperative Approach: The OS Takes Control
 - Timer interrupt
 - Raise an interrupt every so many milliseconds
 - Pre-configured interrupt handler in the OS runs

LDE Protocol (Timer Interrupt)

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember addresses of... syscall handler timer handler	
start interrupt timer	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call <code>switch()</code> routine save regs(A) to <code>proc-struct(A)</code> restore regs(B) from <code>proc-struct(B)</code> switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B
		...

Context Switch in xv6 (swtch.S)

```
# void swtch(struct context *old, struct context *new);  
#  
# Save current registers in old. Load from new.
```

```
.globl swtch  
swtch:  
    sd ra, 0(a0)  
    sd sp, 8(a0)  
    sd s0, 16(a0)  
    sd s1, 24(a0)  
    sd s2, 32(a0)  
    sd s3, 40(a0)  
    sd s4, 48(a0)  
    sd s5, 56(a0)  
    sd s6, 64(a0)  
    sd s7, 72(a0)  
    sd s8, 80(a0)  
    sd s9, 88(a0)  
    sd s10, 96(a0)  
    sd s11, 104(a0)  
  
    ld ra, 0(a1)  
    ld sp, 8(a1)  
    ld s0, 16(a1)  
    ld s1, 24(a1)  
    ld s2, 32(a1)  
    ld s3, 40(a1)  
    ld s4, 48(a1)  
    ld s5, 56(a1)  
    ld s6, 64(a1)  
    ld s7, 72(a1)  
    ld s8, 80(a1)  
    ld s9, 88(a1)  
    ld s10, 96(a1)  
    ld s11, 104(a1)  
  
    ret
```

Announcement

- Homework 1
 - Due Monday September 22nd
 - **DON'T FORGET TO GIVE GITHUB REPO ACCESS PERMISSION TO THE TA**
 - danielmarin350@gmail.com
- **THERE WILL BE CHECKS ON HOMEWORK SUBMISSION**
 - Randomly picked 20 student per assignment