Limited Direct Execution (ch. 6)

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Virtualizing the CPU

- Virtualizing the CPU by **time sharing**
- Two challenges:
  - **Performance**: avoid adding excessive overhead
  - **Control**: retain control over the CPU

  - Without control, a process could:
    - Run forever and take over the machine
    - Access information it should not be allowed to access

Direct Execution

- **Direct execution**: run the program directly on the CPU

Direct Execution

- **Direct execution**: run the program directly on the CPU

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	

Direct Execution

This approach is problematic:

1. Full access to hardware
   - A bug could corrupt the hardware, freeze the machine, etc.
2. How does the OS stop a running process?
   - Implementing **time sharing**
   - What if a process goes into an infinite loop?

Direct Execution

This approach is problematic:

1. Full access to hardware
   - A bug could corrupt the hardware, freeze the machine, etc.
2. How does the OS stop a running process?
   - Implementing **time sharing**
   - What if a process goes into an infinite loop?

> Without limits on processes, the OS is not in control of anything

Direct Execution

This approach is problematic:

1. Full access to hardware
   - A bug could corrupt the hardware, freeze the machine, etc.
2. How does the OS stop a running process?
   - Implementing **time sharing**
   - What if a process goes into an infinite loop?

> Without limits on processes, the OS is not in control of anything

The solution: **limited** direct execution

Restricted Operations

- New processor mode: **user mode**
  - Code in user mode is restricted
  - e.g., no I/O requests
- OS runs in **kernel mode**
  - Code in kernel mode can do what it likes

Restricted Operations

- New processor mode: **user mode**
  - Code in user mode is restricted
  - e.g., no I/O requests
- OS runs in **kernel mode**
  - Code in kernel mode can do what it likes

- What if a process wishes to perform a privileged operation?

System Calls

- Carefully expose functionality to user programs
- API for sensitive kernel operations
- Each system call has a known number

System Calls

- Classically, special **trap** instruction (Some modern processors offer less heavy methods)
  - Raises privilege level to **kernel mode**
  - Jumps into the kernel code
  - The **return-from-trap** instruction returns to program and reduces privilege level to **user mode**

System Calls

- Classically, special **trap** instruction (Some modern processors offer less heavy methods)
  - Raises privilege level to **kernel mode**
  - Jumps into the kernel code
  - The **return-from-trap** instruction returns to program and reduces privilege level to **user mode**
- The same dispatch mechanism is used on:
  - System call (process needs OS service)
  - Program fault (illegal memory access, division by zero, etc.)
  - Interrupt (event from an external device)

System Calls

- CPU checks **mode bit** before executing protected instructions
- OS sets up a **trap table** on initialization
  - List of **trap handlers**
  - Tells the hardware what code should run for various events (e.g., **system call**)

System Calls

- The Decode stage (Fetch-Decode-Execute) changes:
    - Protected instruction and not kernel mode? raise a **trap**
    - Usually the trap handler will terminate the process

- There are multiple modes: **protection rings**
    - Kernel, executive, superviso, user, etc...
    - Linux uses kernel and user modes only.

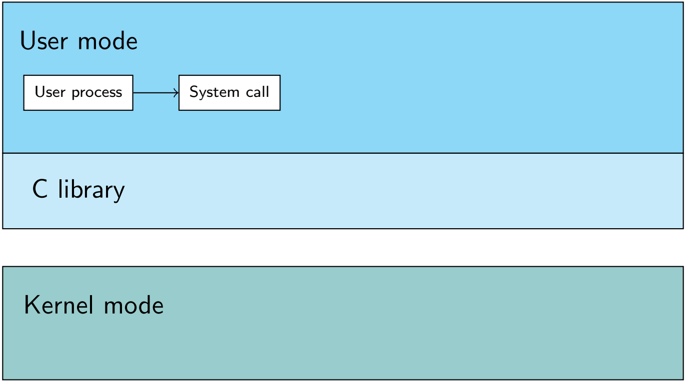
System Calls

- But system calls look like function calls!
  - It is a function call! into the C library
  - But inside is the **trap** instruction
  - Put arguments and system-call number in well-known locations, then execute the trap
  - Unpack return values, return control to the program

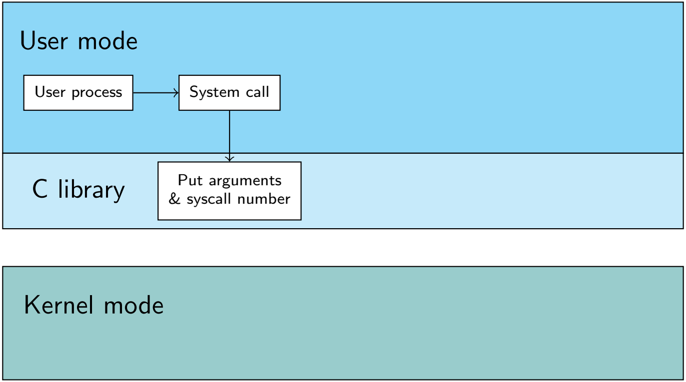
System Calls

System Calls

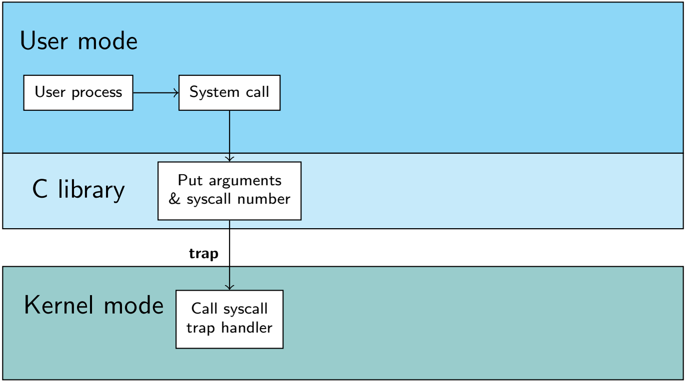
System Calls



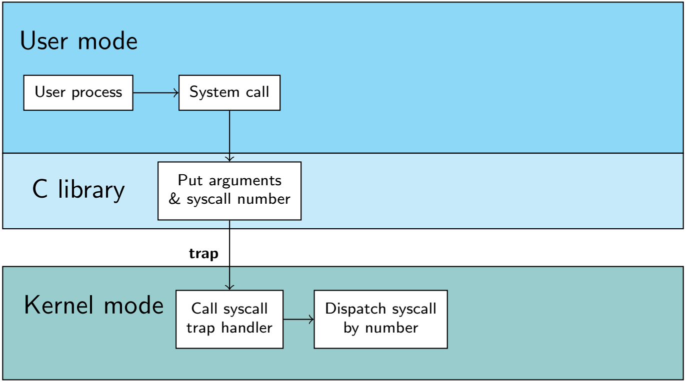
System Calls



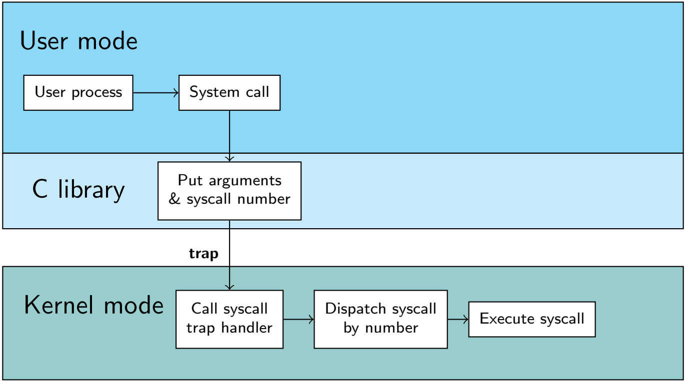
System Calls



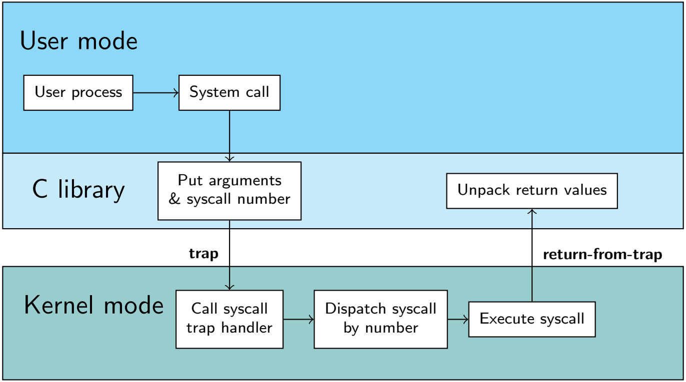
System Calls



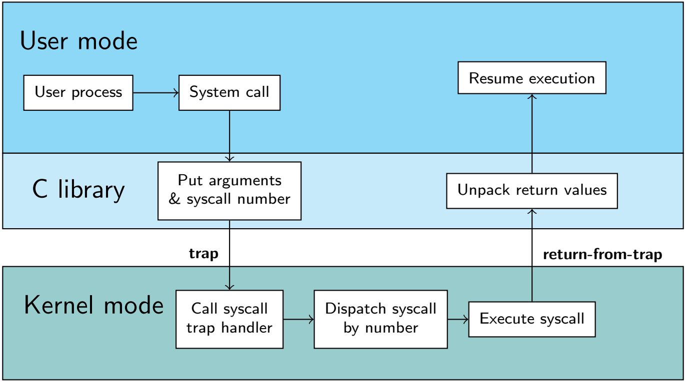
System Calls



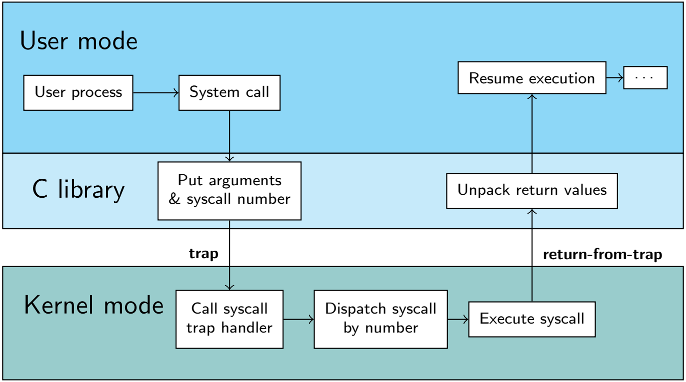
System Calls



System Calls



System Calls



Limited Execution

Two problems:

- Full access to hardware
  - A bug could corrupt the hardware, freeze the machine, etc.
- How does the OS stop a running process?
  - Implementing **time sharing**
  - What if a process goes into an infinite loop?

Dealt with problem #1

- User/kernel mode, system calls

Switching Between Processes

- A process is running on the CPU
- The OS (by definition) is not running

- How can the OS regain control of the CPU (to switch between processes)?

Switching Between Processes

- The **cooperative** approach
  - Most processes frequently make **system calls**
  - Trust the processes to behave reasonably
  - Include an explicit **yield** system call

- But what if a process goes into an infinite loop?

Switching Between Processes

- The **cooperative** approach
  - Most processes frequently make **system calls**
  - Trust the processes to behave reasonably
  - Include an explicit **yield** system call
- But what if a process goes into an infinite loop?
  - OS can't do much
  - Only recourse: reboot the machine

Timer Interrupt

- The **preemptive** approach
  - On initialization, the OS starts the timer
  - Timer device programmed to raise an interrupt regularly
  - The timer **interrupt** routine might decide to:
    - Halts current process
    - Saves state of running process
    - (Easily restored later)

Context Switch

- The **scheduler** makes a decision:
  - Continue running the currently-running process
  - Switch to a different process

Context Switch

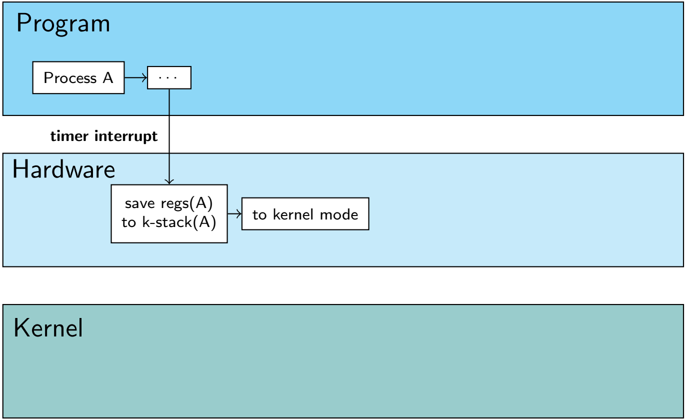
- The **scheduler** makes a decision:
  - Continue running the currently-running process
  - Switch to a different process

- A **context switch**:
  - Save a few register values for the currently-running process
  - Restore values for the upcoming process
  - A **return-from-trap** instruction:
    - Upcoming process becomes currently-running process

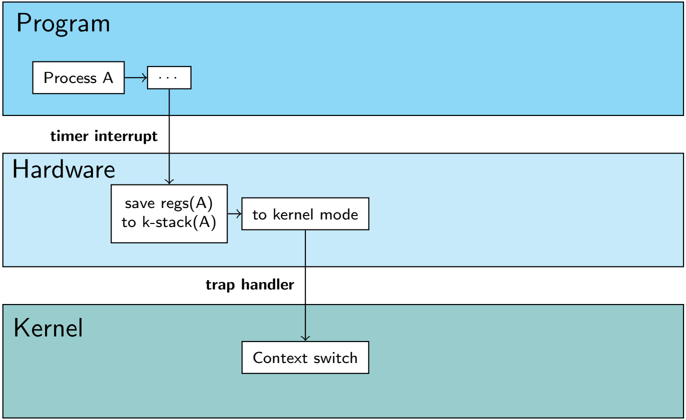
Context Switch

Context Switch

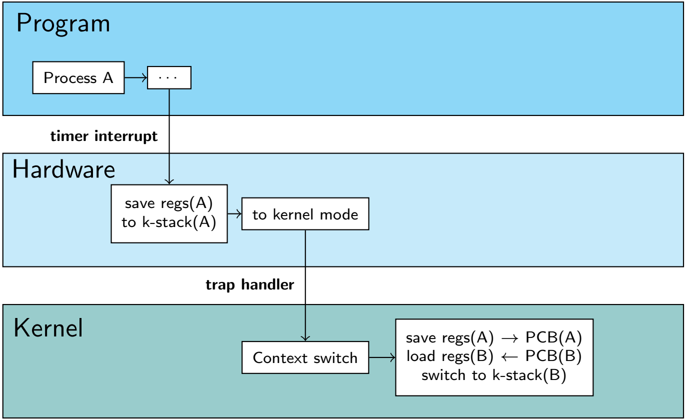
Context Switch



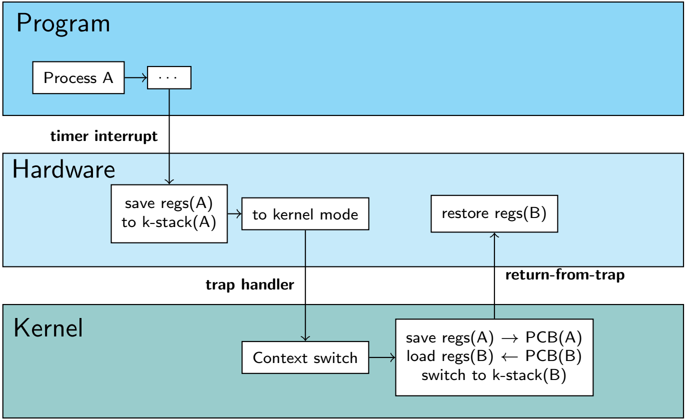
Context Switch



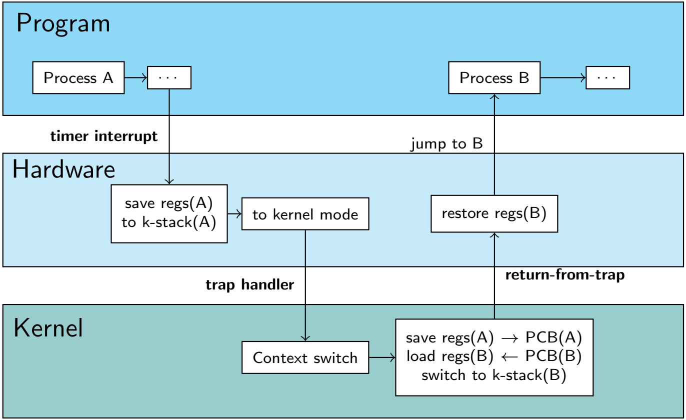
Context Switch



Context Switch



Context Switch



Context Switch

- A **context switch** should be as fast as possible
  - It is pure overhead
- Context switch is a **mechanism**
- Deciding when to switch (**scheduling**) is a **policy**

Concurrency

- You should have some concerns...

Concurrency

- You should have some concerns...
  - Timer interrupt during a system call?
  - Handling one interrupt and another one happens?

Concurrency

- You should have some concerns...
  - Timer interrupt during a system call?
  - Handling one interrupt and another one happens?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
    - Too long could lead to lost interrupts
  - make interrupt disabled part SHORT
  - Use sophisticated **locking** schemes
    - Protect concurrent access to internal data structures
- More when we get to **concurrency**

Summary

- **Limited direct execution**
  - Run program on the CPU
  - Set up hardware to limit what the process can do
  - Separate **user mode** and **kernel mode**
- **System calls**
  - **Trap** into OS code
  - Allow sensitive operations
- **Timer interrupt**
  - OS regularly regains control
  - **Context switch** when policy dictates