System Security

Security is Found at the Boundary (Kernel/User Modes)

Mohamed Sabt

Univ Rennes, CNRS, IRISA

2023 / 2024

Part I

Limited Direct Execution

CPU Virtualization

- The basic idea of CPU virtualization is simple
  - Time sharing the CPU: run one process for a little while, then run another one, and so forth.

- Challenge #1 (Performance)
  - how can we implement virtualization without adding excessive overhead to the system?
- Challenge #2 (Control)
  - how can we run processes efficiently while retaining control over the CPU?

Direct Execution (without Limits)

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	

Raised Problems

- Problem #1
  - If we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do?

- Problem #2
  - When we are running a process, how does the operating system stop it from running and switch to another process

Part II

Problem #1: Restricted Operations

Problem Statement

- Direct execution has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect.

- A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.

Failing Approach

- One approach would simply be to let any process do whatever it wants in terms of I/O and other related operations.

- However, doing so would prevent the construction of many kinds of systems that are desirable.
  - For example, if we wish to build a file system that checks permissions before granting access to a file, we can't simply let any user process issue I/Os to the disk;
  - if we did, a process could simply read or write the entire disk and thus all protections would be lost.

OS API

- Many executables need OS services.
- OS services are accessible through system call
  - System Calls are OS APIs to users' applications.
  - Program passes relevant information to the OS.
  - The OS performs the service if it is permitted for this program (or the executing user) at the time of the call.
- System calls are just a C kernel space function that user space programs call to handle some request.
- It is important to note that programs usually call a function in a library (system function) that actually calls the system call.

Example 1/2

- Let's say that a user program has the following line of code:

  int count = read(fd, buf, nbytes)

- Can be viewed as special function calls
  - executed in kernel, they perform a privileged operation

- The system call interface represents abstract machine provided by the operating system

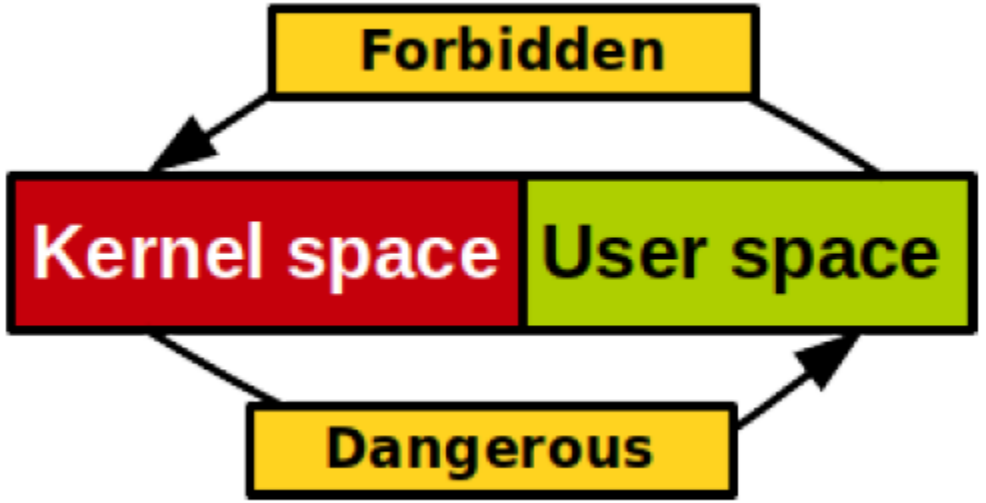
Example 2/2

- Some issues to be addressed:
  - How are parameters passed?
  - How are results provided to the user program?
  - How is control given to the system call and the operating system?

- To understand how these issues addressed, we will briefly introduce some important notions on processes and CPUs

Kernel-Space vs User-Space Memory

- User space cannot access kernel memory
  - There is an implicit assumption that the OS is trusted, but applications are not.
  - Prevent a process from wrecking the OS.



- Kernel code must never blindly follow a pointer into user-space.
  - Accessing incorrect user address can make kernel crash!

Hardware Execution Modes

- **user mode;** code that runs in user mode is restricted in what it can do.
  - For example, when running in user mode, a process can't issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process.

- **kernel mode**; code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.

Privileged Instructions

- Which of these instructions should run in supervisor mode?
  - Read time-of-day instruction.
  - Set time-of-day clock.
  - Change the memory map.

Exceptions 1/2

- Exceptions: an abrupt change in control flow in response to a change in processor state.
  - Note: Exceptions in OS ≠ exceptions in Java

- There are four classes of exceptions
  - Interrupts
  - Traps
  - Faults
  - Aborts

Exceptions 2/2

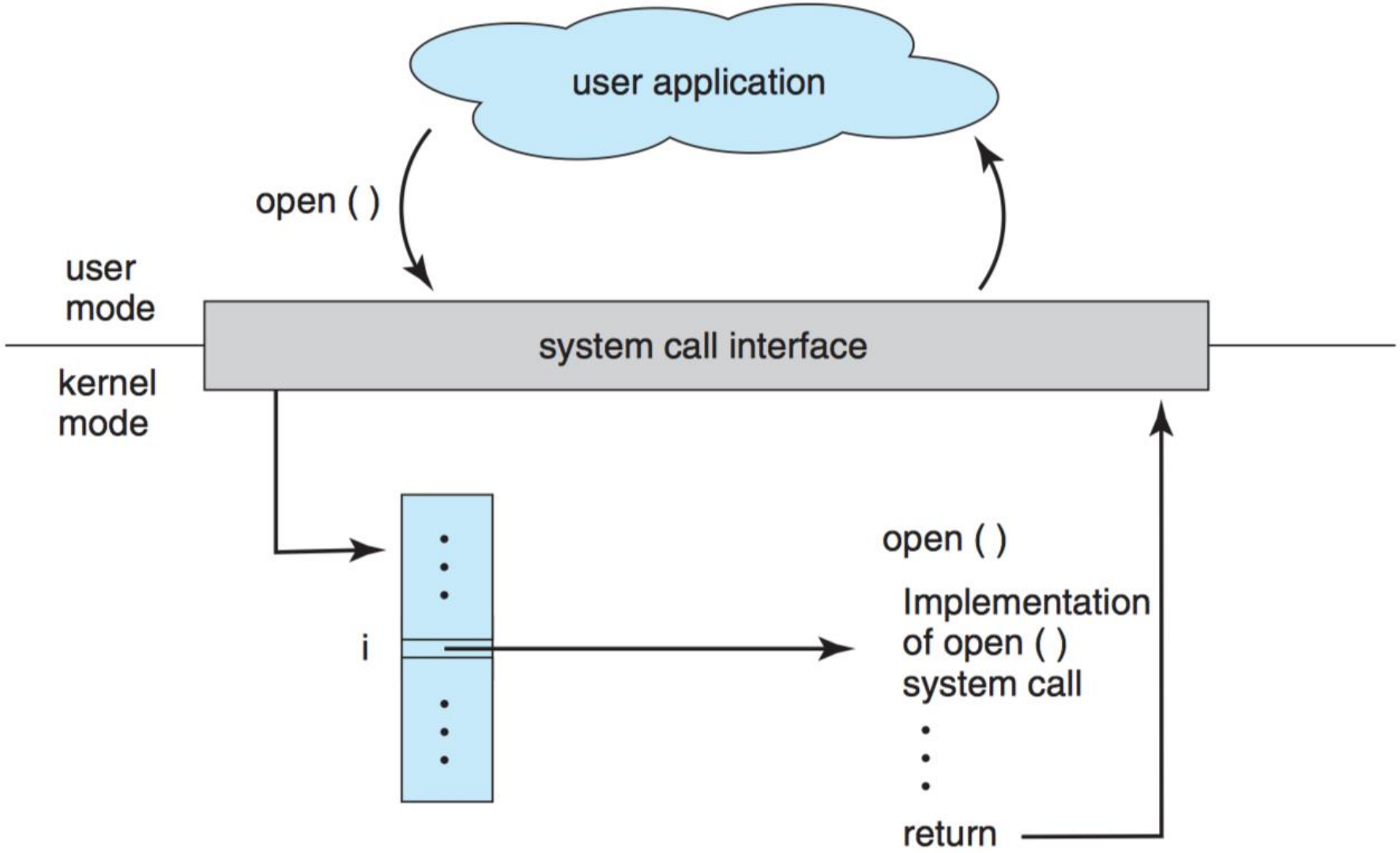
Class	Cause	Asynch/Sync	Return Behavior
Interrupt	Signal from I/O device	Asynch	Return to next instr
Trap	intentional	Sync	Return to next instr
Fault	(Maybe) recoverable error	Sync	(maybe) return to current instr
Abort	Non-recoverable error	Sync	Do not return

System Calls

- User programs are allowed to perform a **system call.**

- To execute a system call, a program
  - must execute a special **trap** instruction.
  - This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode;
  - Once in the kernel, the system can now perform whatever privileged operations are needed (if allowed).
  - When finished, the OS calls a special **return-from-trap** instruction, which, as you might expect, returns into the calling user program while simultaneously reducing the privilege level back to user mode.

System Calls Overview



Which Code to Return To

- The hardware must make sure to save enough of the caller's registers in order to be able to return correctly when the OS issues the return-from-trap instruction.

- In Intel, the processor will push the program counter, flags, and a few other registers onto a per-process kernel stack;
  - the return-from-trap will pop these values off the stack and resume execution of the user mode

Which Code to Jump To

- The calling process can't specify an address to jump to (as you would when making a procedure call);
  - doing so would allow programs to jump anywhere into the kernel

- The kernel does so by setting up a **trap table** at boot time.

- One of the first things the OS does is to tell the hardware what code to run when certain exceptional events occur.
  - The OS informs the hardware of the locations of these **trap handlers**.

Boundary Parameters

- To specify the exact system call, a **system-call number** is usually assigned to each system call.

- The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack;
  - The OS, when handling the system call inside the trap handler, examines this number, ensures it is valid, and, if it is, executes the corresponding code.

- **Input sanitization**: the OS must check what the user passes in and ensure that arguments are properly specified, or otherwise reject the call.
  - Otherwise, it would be possible for a user to read all of kernel memory.

System Calls vs. Function Calls

- System calls are similar to function calls
  - Control transfer from original code to other code
  - Other code executes
  - Control returns original code

- System calls are different from function calls
  - Processor pushes data onto OS's stack, not application's stack
  - Handler runs in privileged mode, not in user mode
  - Control might return to next instruction – sometimes it does not return at all!

Baby Proofing

- **Limited Direct Execution:** just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what the process can do without OS assistance.

- **Baby proofing:** locking cabinets containing dangerous stuff and covering electrical sockets.
  - Then, you can let your baby roam freely, secure in the knowledge that the most dangerous aspects of the room have been restricted.

Part III

Problem #2: Switching Between Processes

Problem Statement

- Switching between processes should be simple, right?
  - The OS should just decide to stop one process and start another.

- If a process is running on the CPU, this by definition means the OS is not running. If the OS is not running, how can it do anything at all?

- Crux:
  - How can the operating system regain control of the CPU so that it can switch between processes?

Cooperative Approach

- The OS **trusts** the processes of the system to behave reasonably.
  - Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.

- How: transfer control of the CPU to the OS quite frequently by making system calls, or more broadly issuing exceptions.

- Exercise: what can possibly go wrong?

Reboot is Useful

- It is not uncommon in large-scale cluster Internet services for system management software to periodically reboot sets of machines.

- Advantage:
  - reclaim stale or leaked resources (e.g., memory) which may otherwise be hard to handle.

Non-Cooperative Approach

- **Timer interrupt (hardware feature)**:
  - timer device can be programmed to raise an **interrupt** every so many milliseconds;
  - when the interrupt is raised, the currently running process is halted, and a pre-configured **interrupt handler** in the OS runs.

- The principle of Baby proofing again.

Saving and Restoring Context

- **Definitions:**
  - **Scheduler**: decide whether to continue currently-running process, or to switch to a different one.
  - **Context switch**: save the interrupted process and restore the soon-to-be-executing one.

- There are two types of register saves/restores
  - When the timer interrupt occurs: the user registers of the running process are implicitly saved by the hardware, using the kernel stack of that process.
  - When the scheduler decides to switch: the kernel registers are explicitly saved by the software (i.e., the OS), but this time into memory in the process structure of the process.

Part IV

System Calls in Unix

Unix System Calls in C/C++ Program

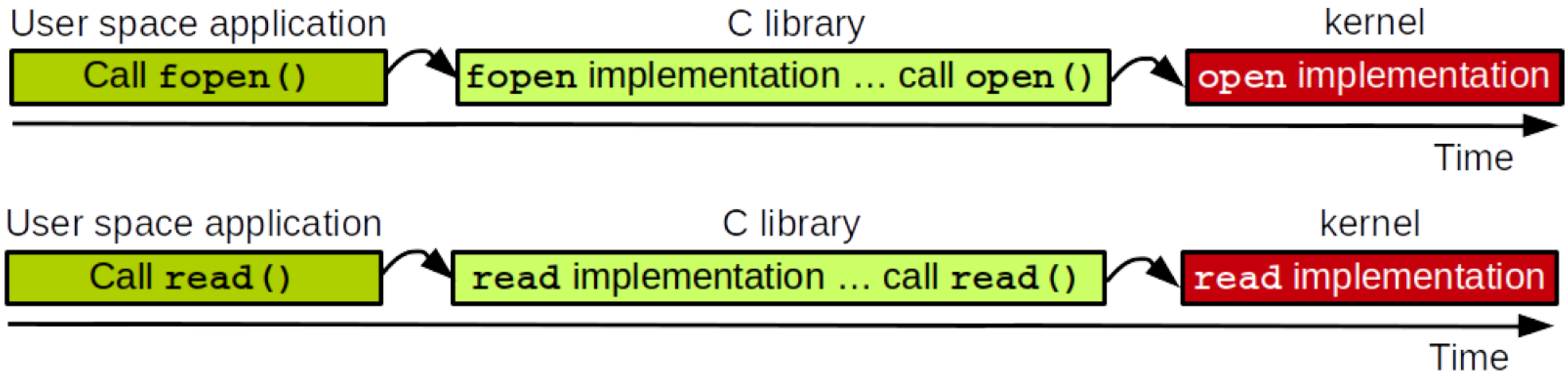
- In C/C++ program, system calls can be invoked almost directly.
  - Needs assembly support.
  - C does not support special machine instructions.
  - Linux uses GCC extensions: asm

- The majority of the system calls have a library version with the same name and the same parameters.
  - The library version acts as function wrapper.

- Other library functions present a more user friendly version of system calls to programmers
  - Fewer number of parameters, more flexible values, etc.

Calling System Calls: Using libc

- Real C program unlike Unix Utilities.
- OS API calls hidden within library.
- Portable to OS implementing libc.



Calling System Calls: Using Posix API

- Uses standard Posix API
- Are used directly to call OS
  - C wrapper functions for the real system calls.

Calling System Calls: Unix Utilities

- **system()**: creates a shell to run command line
  - int system(const char * command_line)
  - Executes command in *command_line* using the shell:

    /bin/sh –c command_line

- Shell does the work: actually in another process.
- No real programming, restrictive but easy!

Example 1

```
#include <unistd.h>
#include <stdio.h>

int main()
{
      int pid;

      /* get Process ID */
      pid = getpid();

      printf("process id = %d\n", pid);

      return 0;
}
```

Library call that has the same name as a system call

Library call that make a system call

- System Calls invoked in this example:
  - **getpid()**
  - **write()** – made by **printf()** library call
  - **exit()** – made by return

Example 2

- A syscall can be directly called through syscall
  - See man syscall → C library uses syscall

```
#include <unistd.h>
#include <sys/syscall.h>   /* for SYS_xxx definitions */

int main(void)
{
    char    msg[] = "Hello, world!\n";
    ssize_t bytes_written;

    /* ssize_t write(int fd, const void *msg, size_t count);      */
    bytes_written = syscall(1, 1, msg, 14);
    /*                       \  \                                 */
    /*                        \  +-- fd: standard output          */
    /*                         +-- write syscall id (or SYS_write) */
    return 0;
}
```

Example 3

- x86_64 architecture has a syscall instruction.

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    mov  $1, %rax      # syscall id: write
    mov  $1, %rdi      # 1st arg: fd (standard output)
    mov  $msg, %rsi    # 2nd arg: msg
    mov  $len, %rdx    # 3rd arg: length of msg
    syscall            # switch from user space to kernel space

    mov  $60, %rax     # syscall id: exit
    xor  %rdi, %rdi    # 1st arg: 0
    syscall            # switch from user space to kernel space
```

Transition from user space to kernel space

- x86 instruction for system call
  - int 0x80: raise a software interrupt (old)
  - sysenter: fast system call (x86_32)
  - syscall: fast system call (x86_64)

- Passing a syscall ID and parameters
  - syscall ID: %rax
  - Parameters (x86_64): rdi, rsi, rdx, r8, r9 and r10

Returning from the syscall interrupt

- Passing a return value: rax

- x86 instruction for system call
  - iret: interrupt return (x86-32 bit, old)
  - sysexit: fast return from fast system call (x86-32 bit)
  - sysretq: return from fast system call (x86-64 bit)

Strace

- strace is well-known system call tracer.
- Target is stopped at every system call twice (entry, exit)
- Tracing is slow
  - Very intrusive and not suitable for diagnosing production issues
- Very good reporting

Improving System Call Performance

- System call performance is critical in many applications
  - gettimeofday(), getpid()

- Hardware: add a new fast system call instruction
  - int 0x80 → syscall

- Software: vDSO (virtual dynamically linked shared object)
  - A kernel mechanism for exporting a kernel space routines to user space applications.
  - No context switching overhead.
  - E.g. the kernel allows the page containing the current time to be mapped read-only into user space.