VIRTUALIZING THE CPU

Kai Mast

CS 537

Fall 2022

ANNOUNCEMENTS

- Office hour schedule has been posted on Piazza
- My (Kai's) office hours are canceled today
- P1a due on Monday

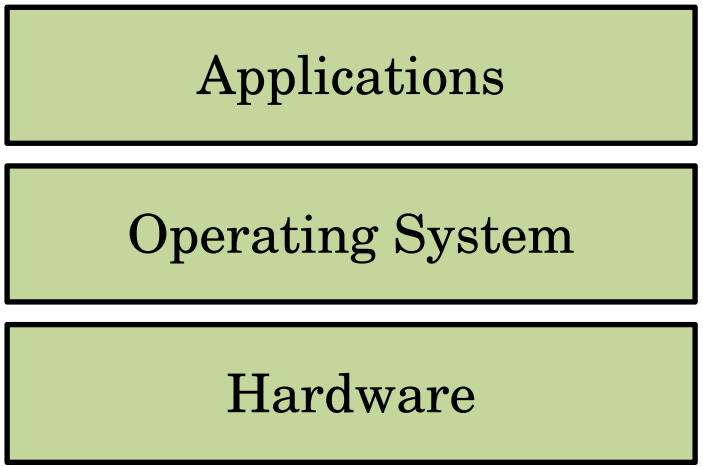
REVIEW: OPERATING SYSTEMS

Goal 1: Make hardware "easy to use"

- Hide complex device driver logic
- Make application code (mostly) independent of the underlying hardware

Goal 2: Multi-tasking

- Support multiple concurrent users
- Support multiple applications per user



REVIEW: VIRTUALIZATION

Problem: Hardware resources are physically limited

- Limited hardware concurrency (e.g., number of CPU cores)

- Fixed amount of memory, network bandwidth, disk throughput...

Virtualization: Build an illusion of unlimited hardware, e.g.,
- as many CPUs as needed
- a large, private memory region for each program

Applications should not need to worry about what resources are available when!

VIRTUALIZATION TECHNIQUES

Time-Sharing

Multiple applications use *the same resource* at *different times*

Space-Sharing

Multiple applications use *parts of the same resource* at *the same time*

Both techniques can be combined!

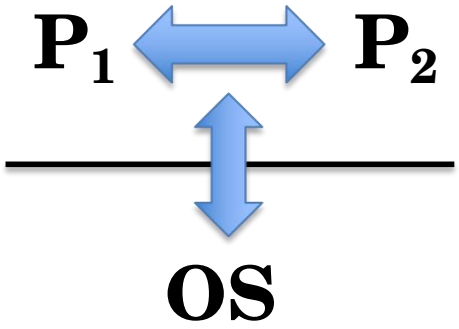
VIRTUALIZATION GOALS

Goal 1: Efficiency

- Virtualization should not create a significant overhead

Goal 2: Security

- Protect processes form each other
- Protect the OS from processes



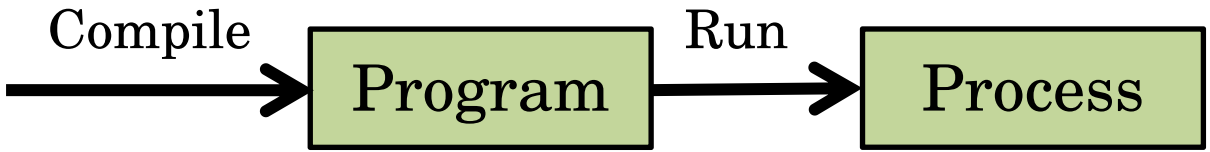
PROGRAMS VS. PROCESSES

Code

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    char *str = argv[1];

    while (1) {
      printf("%s\n", str);
      Spin(1);
    }
    return 0;
}
```



ABSTRACTION: PROCESS

High-level Goal:
Give each *running program* the impression it alone is actively using the CPU and other resources

What changes while a program runs?

Input/Output

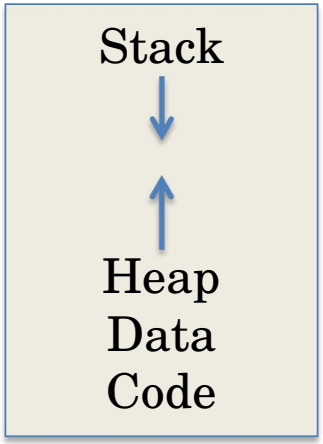
- Open file descriptors
- Writing to disk, network, screen, ...

CPU

Registers

- Program Counter
- General Purpose
- ...

Memory



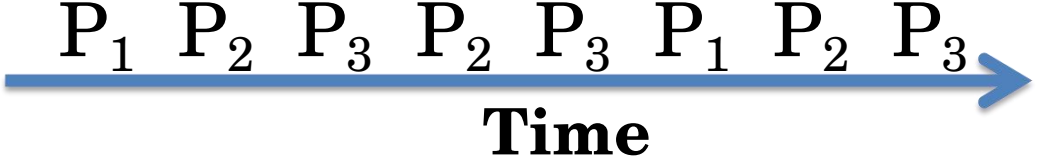
VIRTUALIZING THE CPU

Space Sharing

Allocate a certain number of CPU (cores) to a process

Time Sharing

Share the same CPU core among multiple processes



Today

TIME-SHARING THE CPU

Mechanisms (Low-Level):
How to switch between processes?

Policies (High-Level): Which process to run when?

Examples for scheduling policies?

- Oldest process first
- Newest process first
- Shortest process first

FIRST ATTEMPT: DIRECT EXECUTION

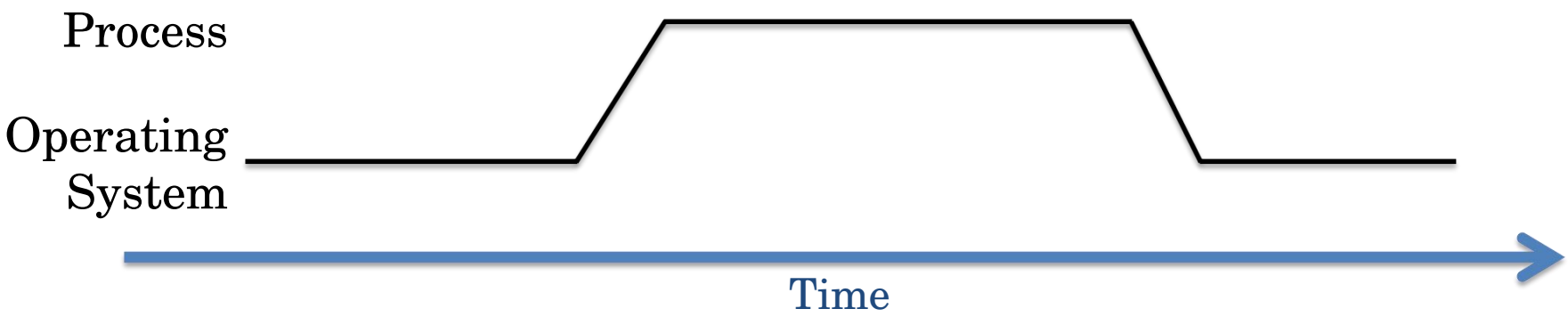
OS is the first program to run

Boot Loader  **Operating System**

Needs to set up data structures:

- Free list (to track memory)
- Process list
- etc.

FIRST ATTEMPT: DIRECT EXECUTION



Operating System
- Updates data structures, e.g. process list
- Allocates initial memory segment for process
- Copies program code and data to process memory

Process
runs until termination

Operating System
- Updates data structures
- Frees process memory

PROBLEMS WITH DIRECT EXECUTION

Security

The process might want to do something restricted, e.g., read from disk

Efficiency

The process might do something very slow, e.g., network I/O

Process might run forever

e.g., `while(true) {}`

LIMITED DIRECTION EXECUTION

How can we ensure a user process cannot harm others?

Solution: Privilege levels supported by the CPU (single bit)

- Applications run in **user mode** (restricted mode)
  - Only allowed to execute certain instructions, access single address space

- OS runs in **kernel mode** (not restricted, privileged)
  - Able to execute all instructions, access all of memory

HOW TO PAUSE A PROCESS?

On Pause:
Process state is stored on its *kernel stack*.
The kernel stack is managed by the OS among other process metadata

On Resume:
Process state is restored from kernel stack to the CPU

What process state needs to be stored on pause?

Registers (program counter, general purpose, etc.)

SWITCHING BETWEEN USER AND KERNEL MODE

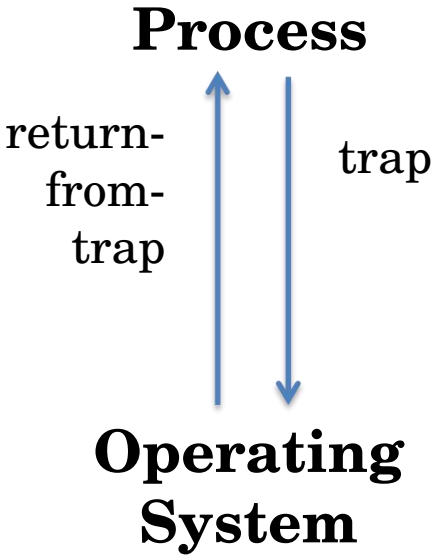
Trap: Enters kernel mode from user mode

- Code invokes *system calls*
- Code performs illegal/restricted operation

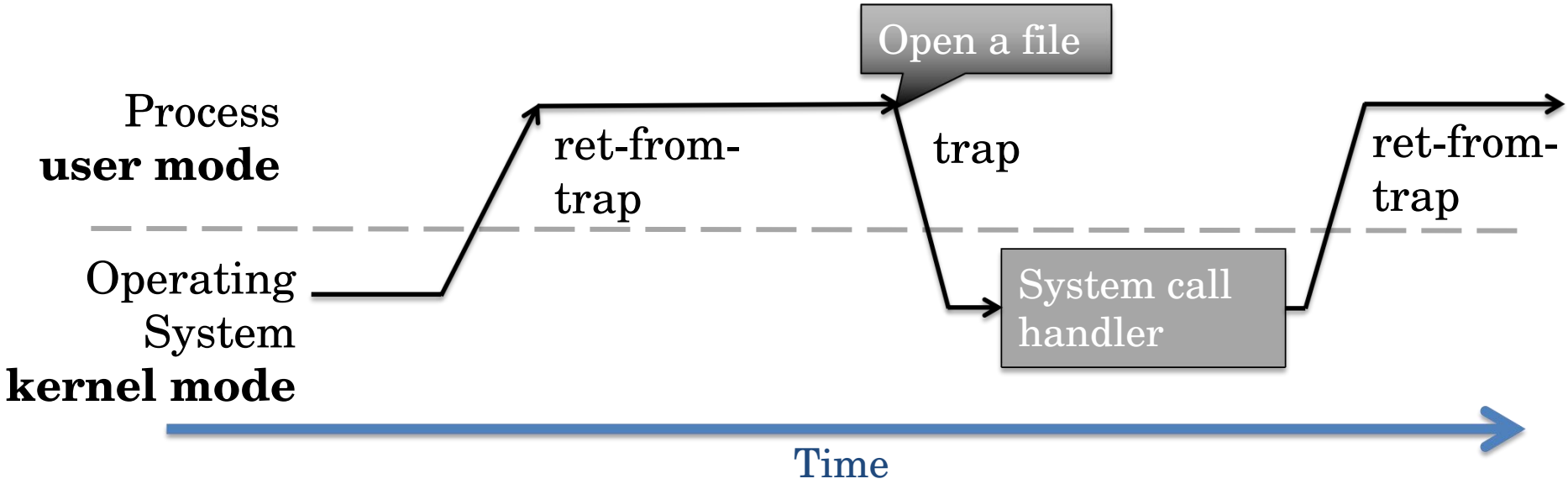
Return-from-trap:
Executed after returning the process

Trap handlers:

- Set up at boot time by the OS
- Executed after a process calls a trap



SYSTEM CALLS



HOW TO RUN MANY PROCESSES "AT ONCE?"

So far: Cooperative multi-tasking

- Can only switch to another task if the process says so

Non-cooperative multi-tasking

- Time-share the same CPU core among multiple processes
- Switch between processes *every few milliseconds*

Requires another CPU feature: timer interrupts

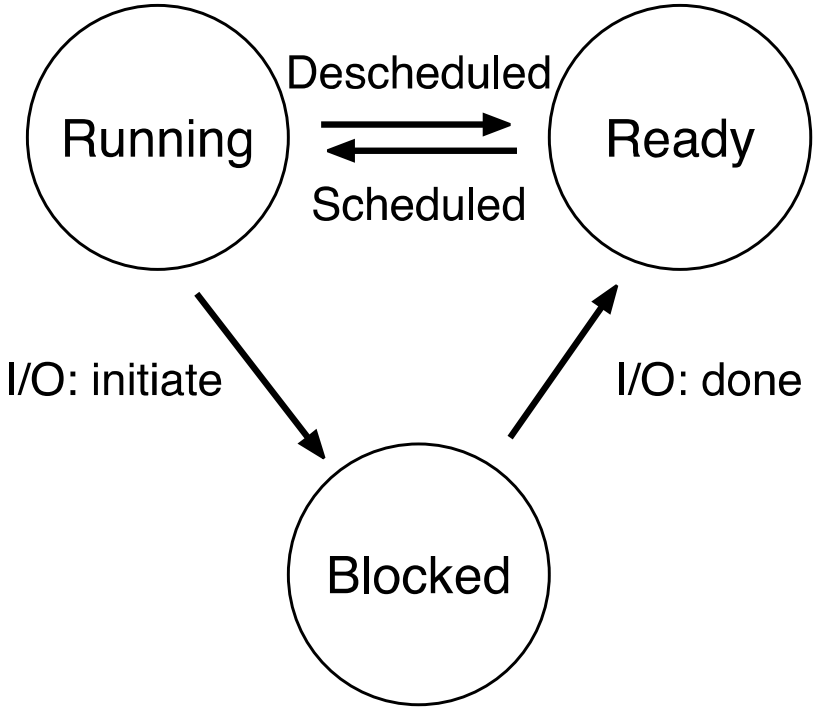
TIMER INTERRUPTS

- Setup up by the operating system at boot time
- Timer interrupt the CPU every few milliseconds
- Operating System may run scheduler* to decide which process to run next

* next lecture

Operating System	Hardware	Program
		Process A runs
	timer interrupt save regs(A) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call switch() routine save regs(A) → proc t(A) restore regs(B) ← proc t(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B runs

PROCESS STATES



What if a process does something slow?

Do not schedule/run a blocked processes

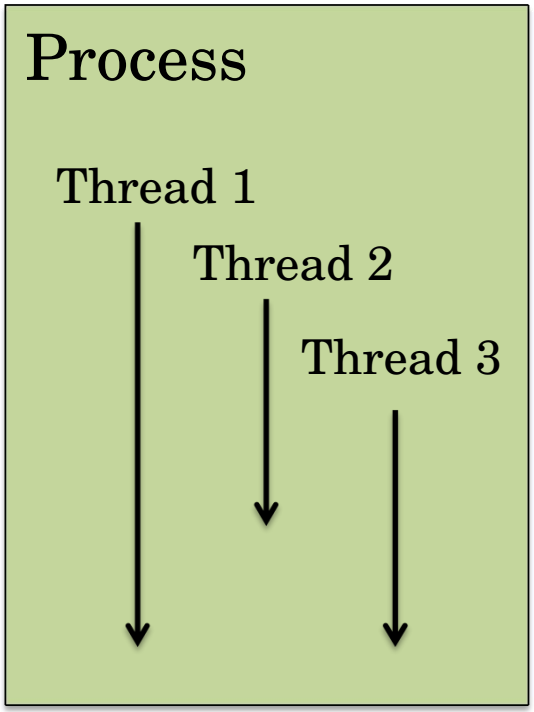
THREADS

Threads: "Lightweight process"

- A process can have multiple threads
- All threads of a process read/write the processes memory

They are handled (almost) the same as processes by the OS

- Each thread has their own stack and kernel stack
- Threads can be ready, running, or blocking



SUMMARY

Process: Abstraction to virtualize CPU

Limited Direct Execution

- Direct execution = runs directly on the CPU
- Limited = hardware-level protections are in place
- Use system calls to control access to devices
- Context-switch using interrupts for multi-tasking (mechanism)

Next class: Scheduling Policies