CSIS 429 Operating Systems

Lecture 2: CPU Mechanisms

September 9th 2020

Textbook chapters

Read “Processes” and “Process API”

Intro	**Virtualization**		**Concurrency**	**Persistence**	**Appendices**
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	Flash-based SSDs
		21 Swapping: Mechanisms	34 *Summary*	44 Data Integrity and Protection	
		22 Swapping: Policies		45 *Summary*	
		23 Case Study: VAX/VMS		46 *Dialogue*	
		24 *Summary*		47 Distributed Systems	
				48 Network File System (NFS)	
				49 Andrew File System (AFS)	
				50 *Summary*	

Understanding Operating Systems

- Operating system
  - Software that helps other programs control computer hardware and interact with users
- Application
  - Software program that provides service for computer user
  - Cannot act without “permission” from operating system

Operating System Components

Major operating system components:

- Kernel
- Device drivers
- Shell
- Utility programs
- Graphical user interface (GUI)

Operating System Functions

- Abstraction: provide system calls and libraries for using resources
- Manage system resources

“Crux of the problem”

The main idea of the entire course

How does an OS virtualize:

- the CPU
- Memory
- Disk

$\left.\begin{matrix}\\ \\ \\ \end{matrix}\right\}$ Resources

efficiently?

Operating System: Abstraction

Provide system calls and libraries for using resources

- Resources are things like CPU, memory, disk, etc.
- How does an OS “abstract” resources?
  - CPU: abstracted as processes or threads
  - Memory: process address space
  - Disk: files
- Why use abstraction?
  - Make the system easy for developers to use
  - Can make different devices look the same – e.g. files

Operating System: Managing Resources

Why should an OS manage resources?

- Security: protect applications from each other
- Fairness
  - users should be able to access & use the system
  - Disk: files
- Efficiency
  - Allow efficient (cost, time, energy) access to h/w

Operating System == Virtual Machine

An OS provides APIs – a standard library for applications

Other names for Oses:

- Supervisor
- Master Control Program

The UNIX Operating System

- UNIX
  - An operating system
  - Originally created at AT&T Bell Labs in early 1970s
  - Designed to control networked computers that were shared by many users
  - Features and low cost of Linux effectively driving UNIX out of market

Three Easy Pieces

- The three easy pieces are the three main themes in operating systems:
  1) Virtualization – make each application believe it has the resource to itself.
  2) Concurrency – handle interactions between processes running at the same time
  3) Persistence – data should stick around, beyond the lifetime of an application run

Hardware Resource: CPU

- What did you learn about CPUs from CSIS 355?
- What does a CPU do?

Hardware Resource: CPU

CPUs perform a very simple set of tasks:

1) Fetch an instruction from memory
2) Decode the instruction
3) Execute

Repeat!

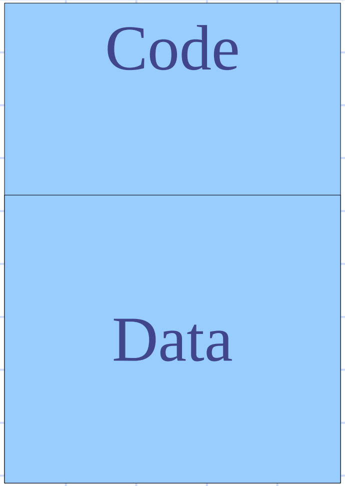
Very quickly – at about a billion instructions/second

Programs

For all their complexity, programs also perform a simple set of tasks:

1) Load code and data into memory
2) Fetch an instruction from memory
3) Execute

Repeat steps 2 & 3

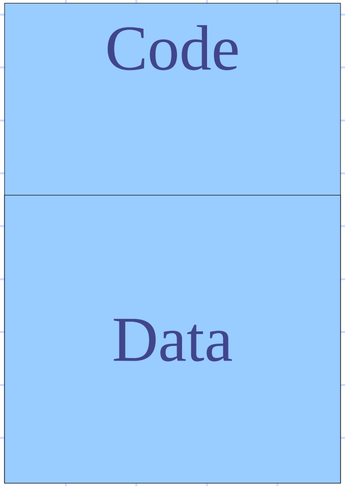


Programs

For all their complexity, programs also perform a simple set of tasks:

1) Load code and data into memory
2) Fetch an instruction from memory
3) Execute

Repeat steps 2 & 3



Notice:

A program does the same thing as the CPU

The OS “virtualizes” the CPU so that each program thinks it is the only thing running

Virtualization

Let’s take a look at

- `cpu.c`

Compile using `gcc cpu.c -lpthread`

Can run multiple instances:

```
./cpu P1 & ./cpu P2 & ./cpu P3
```

Virtualizing the CPU

The main way to virtualize the CPU is to use the "process" abstraction:

A process is an execution stream within the context of an execution state.

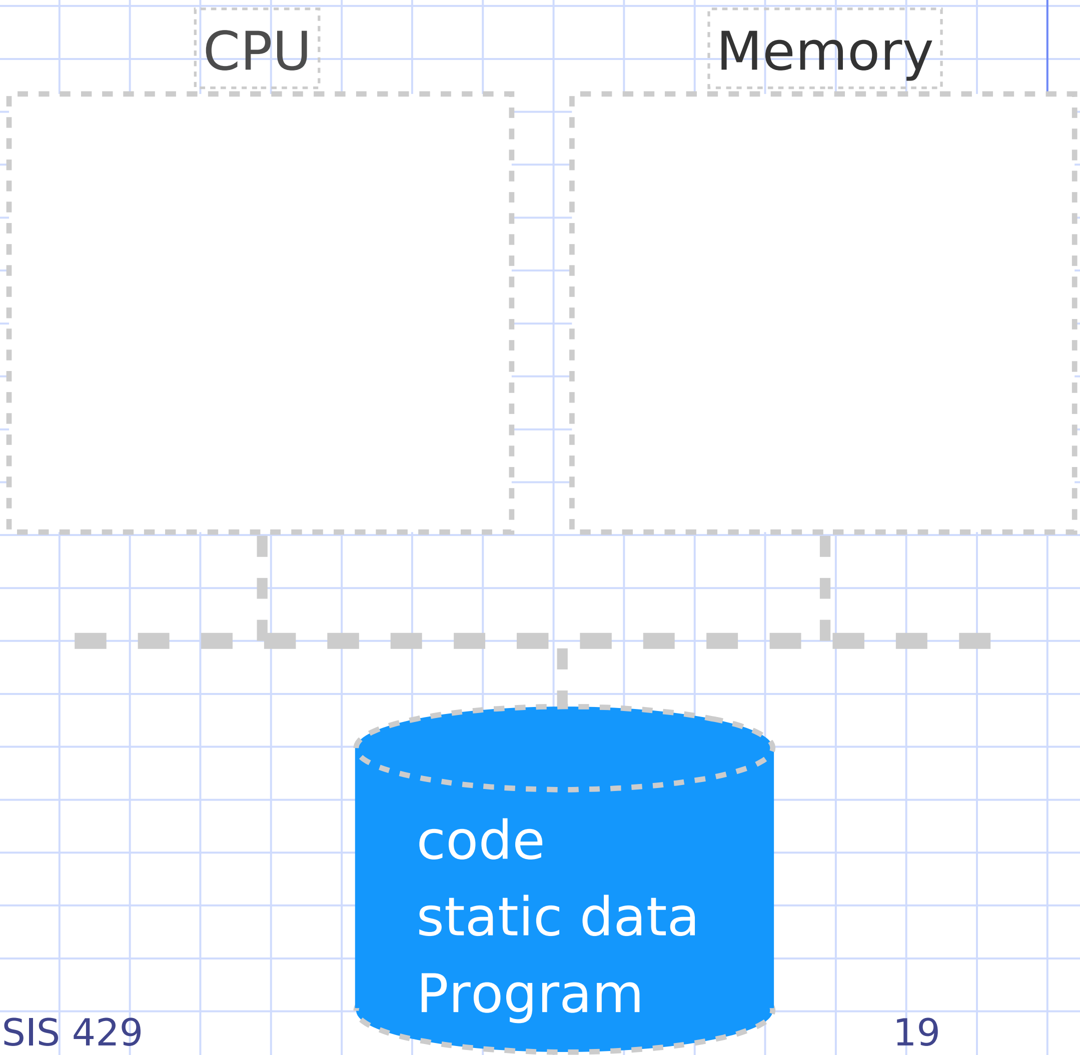
Execution stream == stream of instructions executing, thread of control

Execution state == everything that affects the instruction stream: CPU registers, heap, stack, open files, other data.

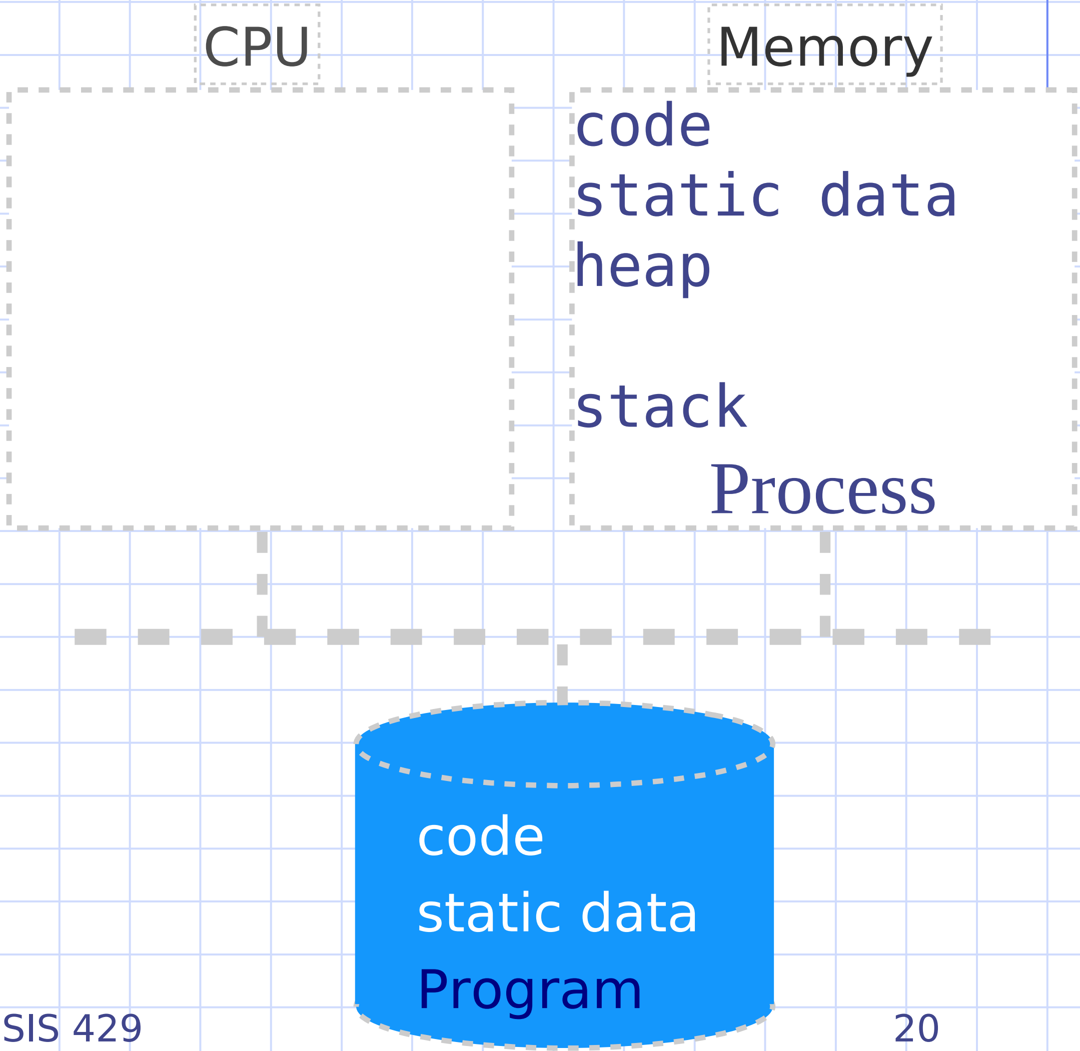
Processes vs. Programs

- Program: static code, static data
- Process: code undergoing execution, changing data
- We can run multiple instances (processes) of the same program

Processes Creation



Processes Creation



In Chapter 4 (Processes) read section 4.3 (pages 4-5) on **Process Creation** for more details

Virtualizing Memory

Let’s take a look at

- `mem.c`

Compile using `gcc`

`./mem X`

Look at addresses printed by mem

ASLR – can be turned off in **`gdb`**

```
(gdb) set disable-randomization off
(gdb) show disable-randomization
```

Based on what you learned in CSIS 248 (Operating System Programming):

When we run multiple instances of **`mem`**, are we accessing the same physical RAM location?

Virtualizing Memory

When we run multiple instances of **`mem`**, are we accessing the same physical RAM location?

OS provides a “process address space”

Also called “virtual address space” or “virtual memory”

→ OS “maps” these to physical RAM locations

Direct vs. Controlled program execution

Direct: allow a user process to run directly on hardware - “DOS” model. Can break:

- Security – can change files, settings, etc.
- Fairness – can use CPU forever
- Efficiency – may busy-wait for slow I/O

Solution: OS and hardware maintain some control over what processes can do.

How can we control program execution

Set up two levels: user (restricted) mode and kernel (unrestricted) mode

- User processes run in user mode
- OS runs in kernel mode

How can we enforce security?

- Processes have to use special “system calls” to access valuable resources
- System calls are checked by kernel

OS tracks program execution

- Each process is in one state at any single time:

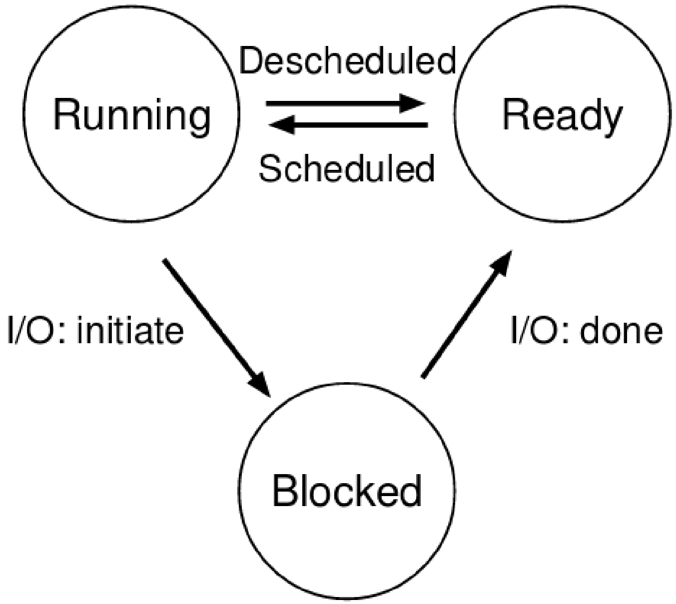


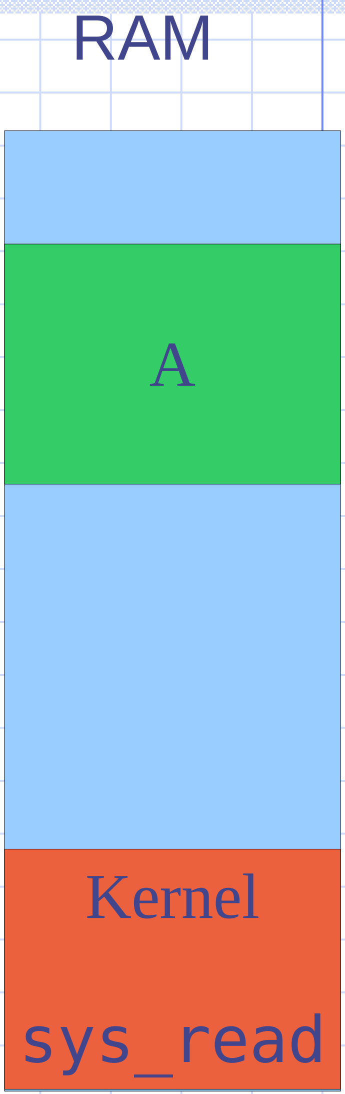
Figure 4.2: **Process: State Transitions**

System Calls

Process A wants to call sys_read() to access disk

A is in user space and can only see its own memory: other processes and kernel are “hidden” → no way to call sys_read directly.

Instead, it calls the read() library function

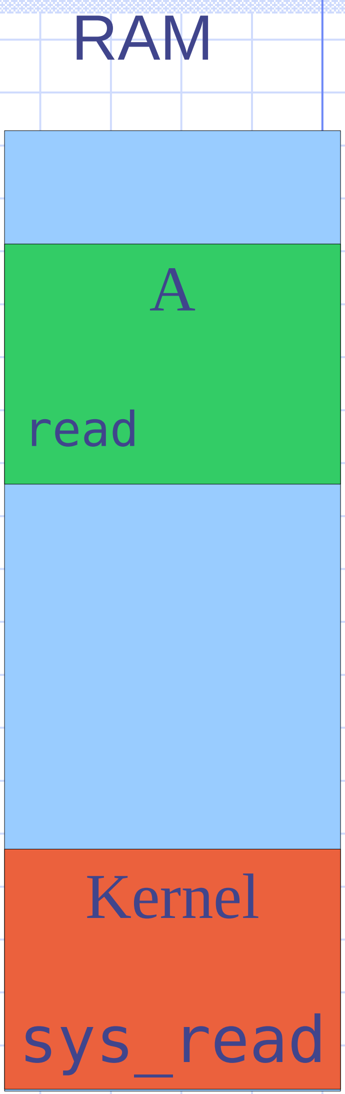


System Calls

Process A has to call the read() library function

read():

```
movl $6, %eax; %syscall table index
int $64; %trap table index
```



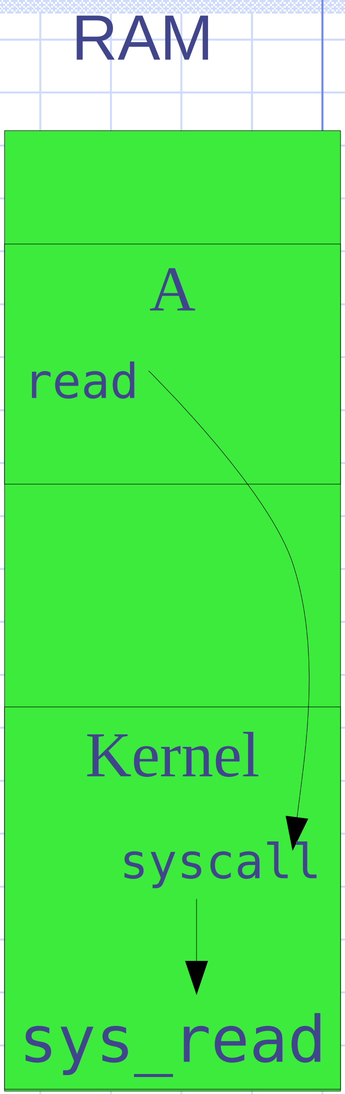
System Calls

Once process A executes the interrupt, we switch to Kernel Mode

→ executing as kernel, we can do anything!

read():

```
movl $6, %eax; %syscall table index
int $64; %trap table index
```

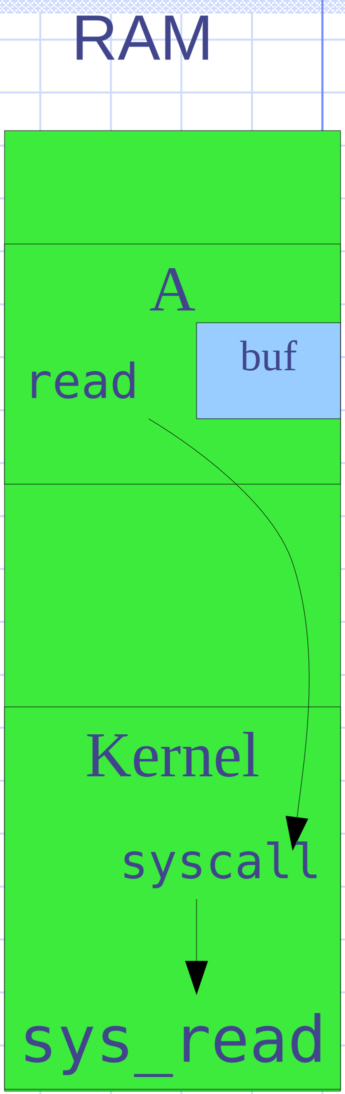


System Calls

Once process A executes the interrupt, we switch to Kernel Mode

→ executing as kernel, we can do anything!

Kernel can access user memory to fill the buf buffer space when done reading from disk.



What are user processes not allowed to do?

User processes are not allowed to

- Access memory beyond process address space
- Access disk
- Execute privileged (ring 0) x86 instructions

Privileged (ring 0) x86 instructions

LGDT	Loads an address of a GDT into GDTR
LLDT	Loads an address of a LDT into LDTR
LTR	Loads a Task Register into TR
MOV Control Register	Copy data and store in Control Registers
LMSW	Load a new Machine Status WORD
CLTS	Clear Task Switch Flag in Control Register CR0
MOV Debug Register	Copy data and store in debug registers
INVD	Invalidate Cache without writeback
INVLPG	Invalidate TLB Entry
WBINVD	Invalidate Cache with writeback
HLT	Halt Processor
RDMSR	Read Model Specific Registers (MSR)
WRMSR	Write Model Specific Registers (MSR)
RDPMC	Read Performance Monitoring Counter
RDTSC	Read time Stamp Counter

How does an OS control access to CPU?

In order to allow an OS to switch between processes, the OS has to be able to “stop” one process, save its “execution state” and switch to another process:

- Run process A for its “time slice”
- Stop process A execution and save context of A
- Load context of process B
- Start process B execution

OS controlling access to CPU

Operating System	Hardware	Program
		Process A
	timer interrupt context-switch: save regs(A) to kernel-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call **switch()** routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	Context-switch: restore regs(B) from k-stack(B) move to user mode jump to B’s IP	
		Process B

CPU Virtualization

The goal of CPU virtualization is to run N processes on M CPUs. N >> M

Abstraction used: “process”

Each process “thinks” it has access to the entire machine.

CPU Virtualization

- To support the "process" abstraction, we need a lot of support in the kernel and CPU.

- Each process "thinks" it has access to the entire machine.

Concurrency

Let's take a look at

- `threadv0.c`
- `threadv1.c`

Compile using `gcc`
May need `-lpthread`