Introduction

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Rutgers Sakai: 01:198:416 Sp11

(https://sakai.rutgers.edu)

http://www.winlab.rutgers.edu/~chandrga/CS416_Spring2011.html

Logistics

Me: Gayathri Chandrasekaran
chandrga@cs.rutgers.edu

Office hour: Tuesdays, 3.30 – 4.30pm
Office: Hill 418

TA: Binh Q.Pham
binhpham@cs.rutgers.edu

Recitation: Thursdays , 12:15pm-1:10pm, SEC 204.

Resources:

Rutgers Sakai: 01:198:416 Sp11 (https://sakai.rutgers.edu)

Course Overview

Goals:

Understanding of OS and the OS/architecture interface/interaction

Prerequisites:

113 & 211

What to expect:

We will cover core concepts and issues in lectures

In Recitations, you and your TA will practice paper & pencil problems and talk about the programming assignments

3 programming assignments

2-3 small paper & pencil assignments

1 Midterm and 1 Final

Warning

Do NOT ignore this warning!

The programming assignments will take a significant amount of time! Don't take this course if you are overloaded or do not have the needed background.

Assignments will help you get hands on and learn a lot.

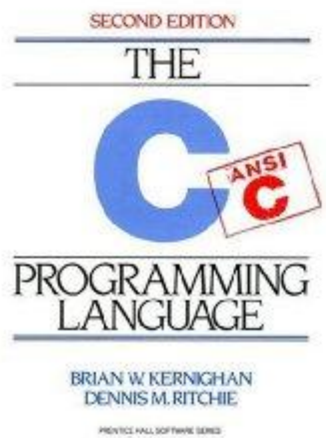
You will have to work hard!

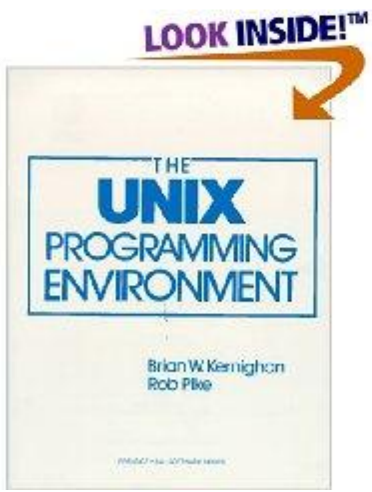
Textbook and Topics

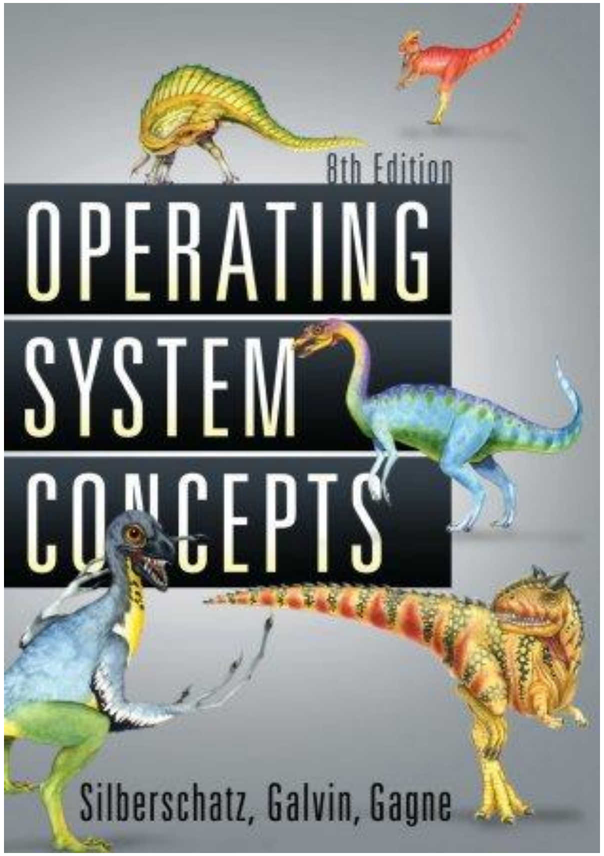
Silberschatz, Galvin, and Gagne. Operating System Concepts. John Wiley & Sons.

- Reading assignments based off of 8th edition
- Any of 6th, 7th, or 8th edition should be ok

Your favorite C book







Topics

Architecture Introduction

Processes and threads

Synchronization

Processor scheduling

Virtual memory

File systems

I/O systems

Security – (If time permits)

Grading

Rough guideline

* Assignment 1 -- 10%
* Assignment 2 -- 10%
* Assignment 3 -- 15%
* Homeworks -- 10%
* Midterm -- 20%
* Final Exam -- 35%

Grading should really be based on how much you have learned from the course

- Any concrete thing that you do towards convincing me of this may help
- For example, showing up at office hours and participating in class will likely help

Paper & Pencil Homeworks

- 2-3 paper & pencil homeworks
  - Each will have some graded and some suggested but not graded problems
  - Good practice is to do them all
- Goals
  - Understand concepts and issues
  - Practice for tests
- Homeworks are due by the end of the lecture on the due date
  - Late hand-ins will not be accepted

Cheating


OMAHA WORLD-HERALD
KOTERBA
NO. YOU MAY *NOT* OUTSOURCE YOUR HOMEWORK TO ...

Programming Assignments

- 3 programming assignments (all in C)
  - Shell & system call (Linux kernel 2.6)
  - Partial threads package & multi-threaded server
  - File system (Linux kernel 2.6)
- Goals
  - Improve design, implementation, and debugging skills
  - Learn to read and understand existing code
  - Learn the internals of an actual operating system
- Programming assignments must be submitted via Sakai
  - We will NOT accept any other form (in particular, NO EMAIL)
  - Late hand-ins will not be accepted

Project Resources

Linux kernel hacking

stratus.rutgers.edu

(Use your remus credentials)

Non-kernel related assignment

Cereal cluster: http://cereal.rutgers.edu

Programming Tools

- Emacs, vi-editor
- Compiler: cc (gcc)
- Project build: make (gmake)
- Debugger: gdb
- IDE: eclipse
  - For the Linux kernels, there are a number of on-line Web-based source code cross-referencing sites … choose your favorite
  - Try searching for "linux kernel cross reference" or "linux kernel browsing"

Today (and next lecture)

What is an Operating System?

Major OS components

Architectural refresher

SGG Chapter 1: 1.1 - 1.9, 1.13

What Is An Operating System?

application (user)

operating system

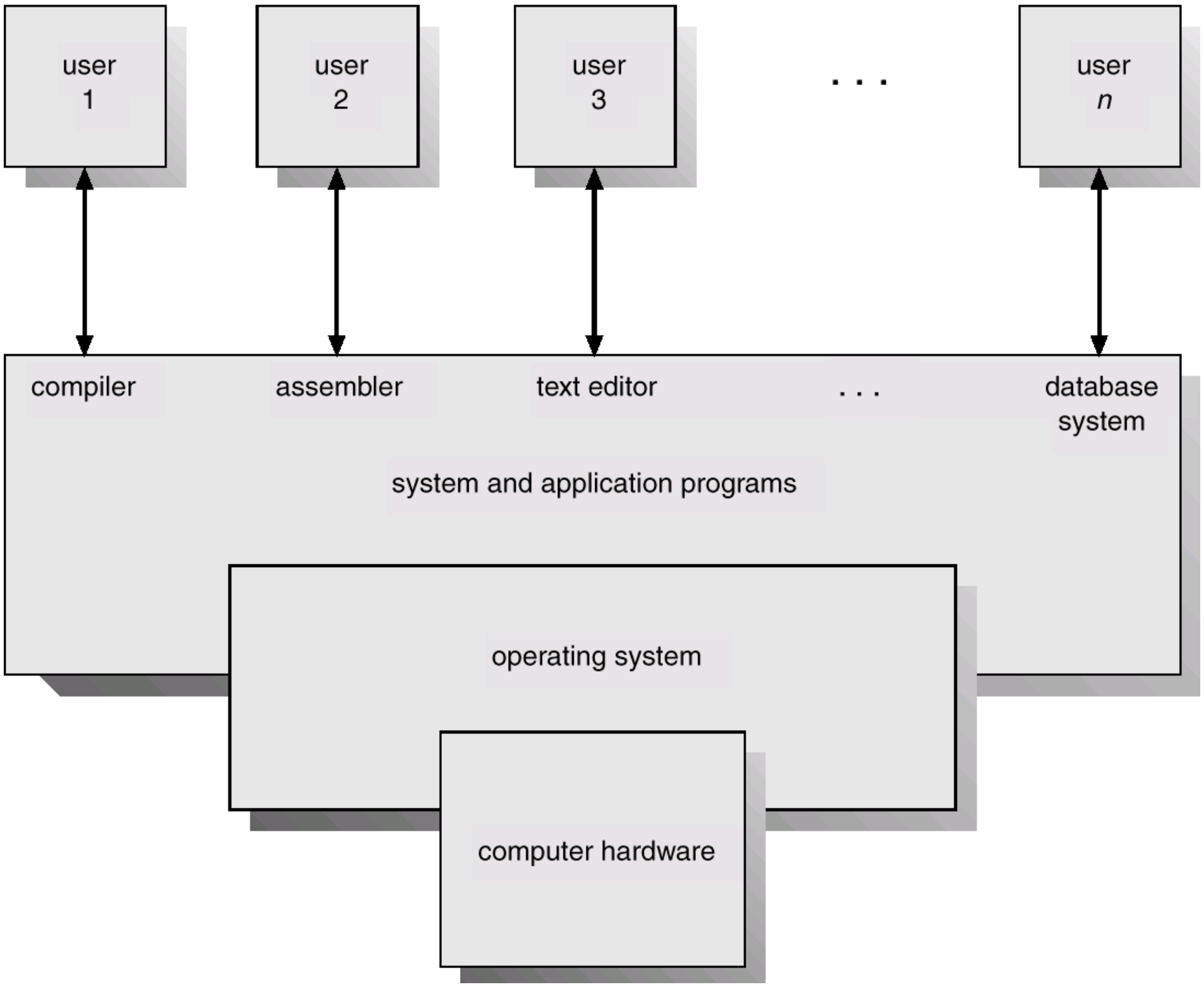
hardware

A software layer between the hardware and the application programs/users which provides a *virtual machine* interface: easy and safe

A *resource manager* that allows programs/users to share the hardware resources: fair and efficient

A set of utilities to simplify application development and execution

Abstract View of System Components



Why Do We Want An OS?

Benefits for application writers

- Easier to write programs
  - See high level abstractions instead of low-level hardware details
    - E.g. files instead of disk blocks
- Portability – Works for any underlying hardware

Benefits for users

- Easier to use computers
  - Can you imagine trying to use a computer without the OS?
- Safety
  - OS protects programs from each other
  - OS protects users from each other

Mechanism And Policy

application (user)

operating system: *mechanism+policy*

hardware

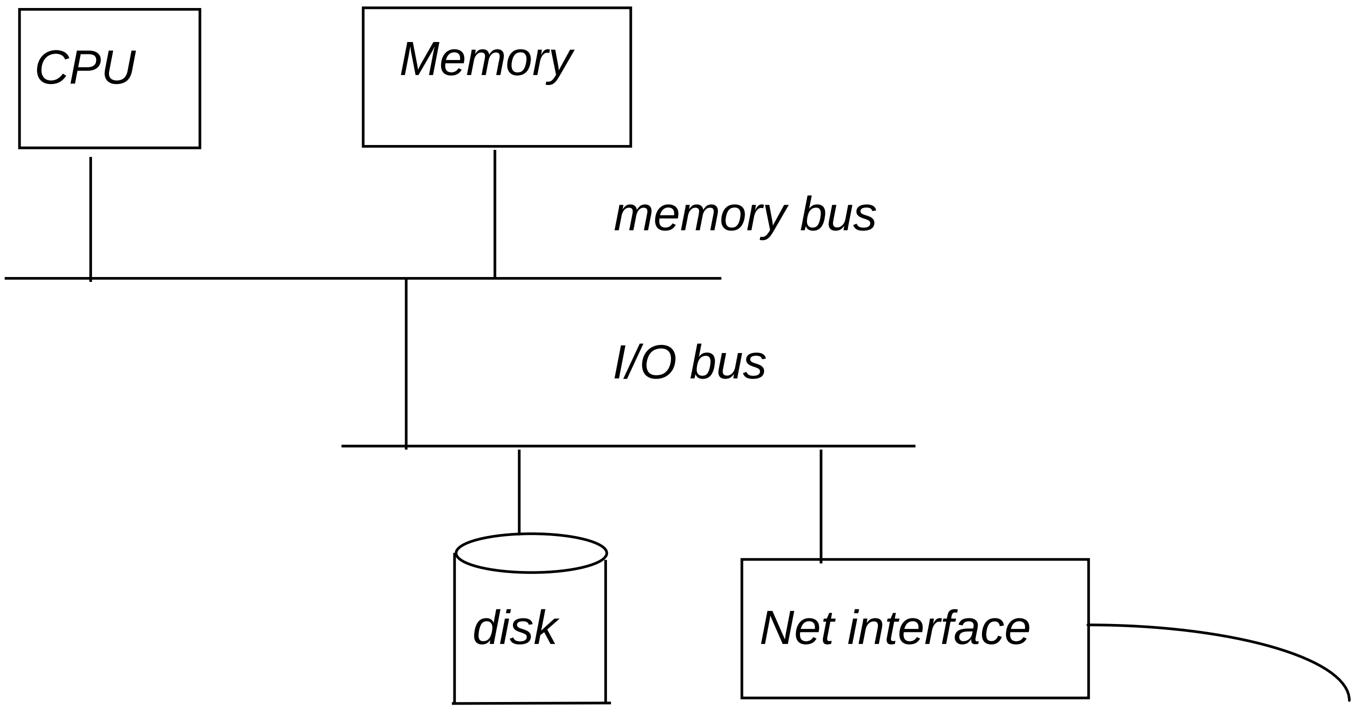
Mechanisms: data structures and operations that implement an abstraction (e.g. the buffer cache)

Policies: the procedures that guides the selection of a certain course of action from among alternatives (e.g. the replacement policy for the buffer cache)

Want to separate mechanisms and policies as much as possible

- Different policies may be needed for different operating environments

Basic computer structure



Virtual Machine Abstractions

Processes: system abstraction – illusion of being the only job executing in the system

Threads: CPU abstraction – illusion of having a dedicated CPU

Virtual memory: memory abstraction – illusion of having an unlimited memory

File system: storage abstraction – illusion of structured, persistent storage system

Messaging: communication abstraction – illusion of reliable, ordered communication

Character and block devices: I/O abstraction – standardized I/F for devices

Major Issues In OS Design

Programming API: what should the VM look like?

Resource management: how should resources be shared among multiple users?

Security: how to protect users from each other? How to protect programs from each other? How to project the OS from applications and users?

Communication: how can applications exchange information?

Concurrency: how do we deal with the concurrency that's inherent in OS'es?

Major Issues In OS Design

Performance: how to make it all run fast?

Reliability: how do we keep the OS from crashing?

Persistence: how can we make data last beyond program execution?

Accounting: how do we keep track of resource usage?

Distribution: how do we make it easier to use multiple computers in conjunction?

Scaling: how do we keep the OS efficient and reliable as the imposed load and so the number of computers grow?

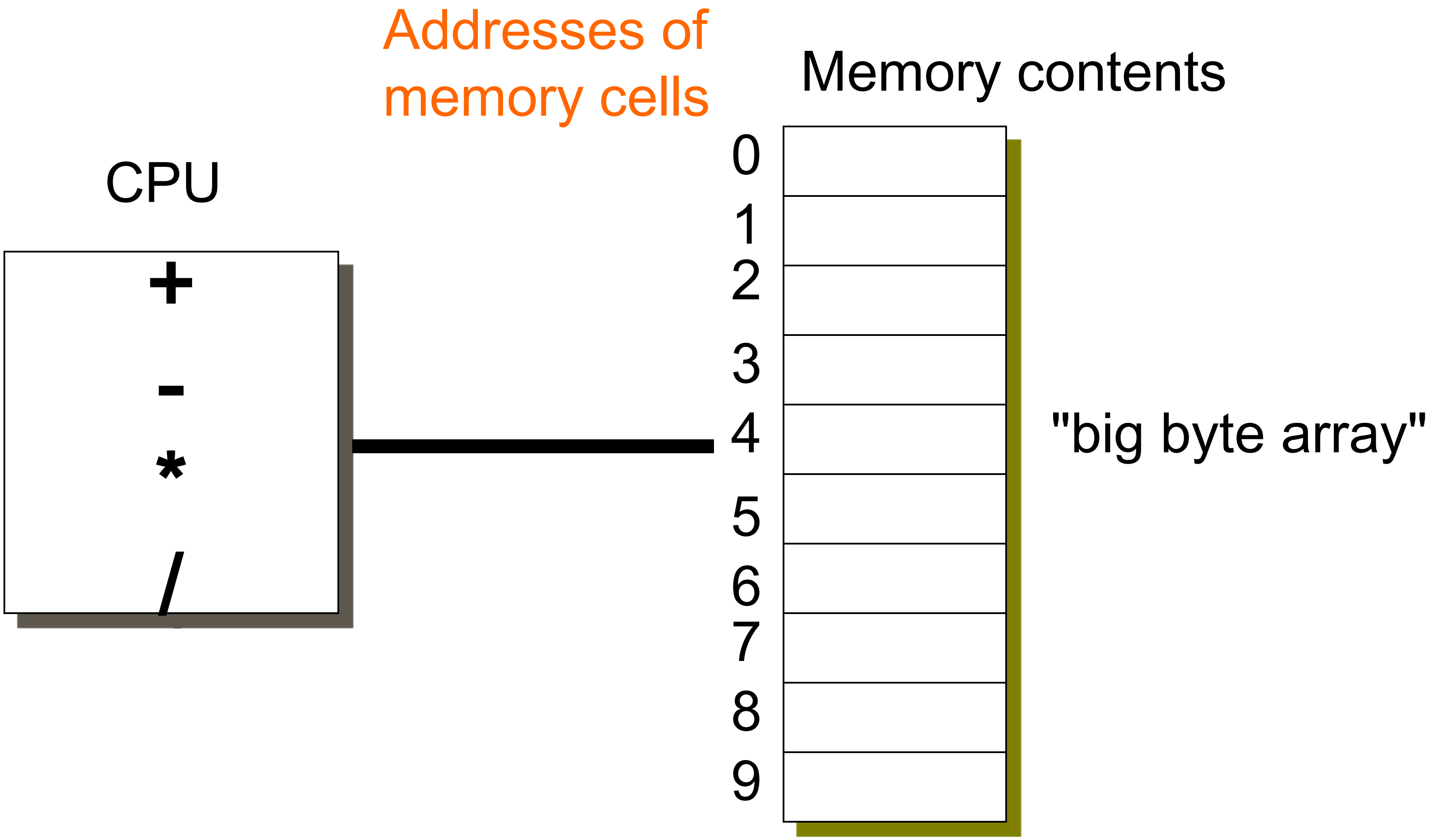
Architectural Refresher

- View of a computer from an Operating System's designer perspective
- Operating system is a layer of software that creates a virtual machine
  - OS also manages the resources of this machine but this mostly involves sharing policies so will be discussed later
- These lectures will familiarize you with
  - The underlying machine
  - The extra hardware mechanisms needed for virtualization

Topics

- The von Neumann architecture
  - CPU + memory
- Hardware support for abstracting the basic machine
  - Modes, Exceptions, Traps and Interrupts
- Input and Output
  - Network, storage and graphics

Conceptual Model



Operating System Perspective

A computer is a piece of hardware which runs the fetch-decode-execute loop

Next slides: walk through a very simple computer to illustrate

- Machine organization
  - What are the pieces and how they fit together
- The basic fetch-decode-execute loop
- How higher-level constructs are translated into machine instructions

At its core, the OS builds what looks like a more complex machine on top of this basic hardware

Fetch-Decode-Execute

Computer as a large, general purpose calculator

- want to program it for multiple functions

All von Neumann computers follow the same loop:

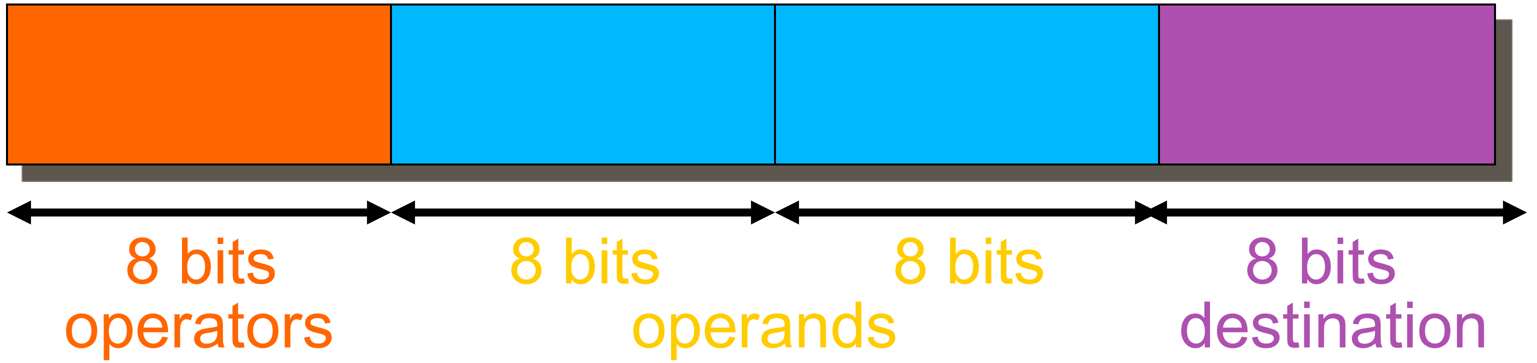
- Fetch the next instruction from memory
- Decode the instruction to figure out what to do
- Execute the instruction and store the result

Instructions are simple. Examples:

- Increment the value of a memory cell by 1
- Add the contents of memory cells X and Y and store in Z
- Multiply contents of memory cells A and B and store in B

Instruction Encoding

How to represent instructions as numbers?



+: 1

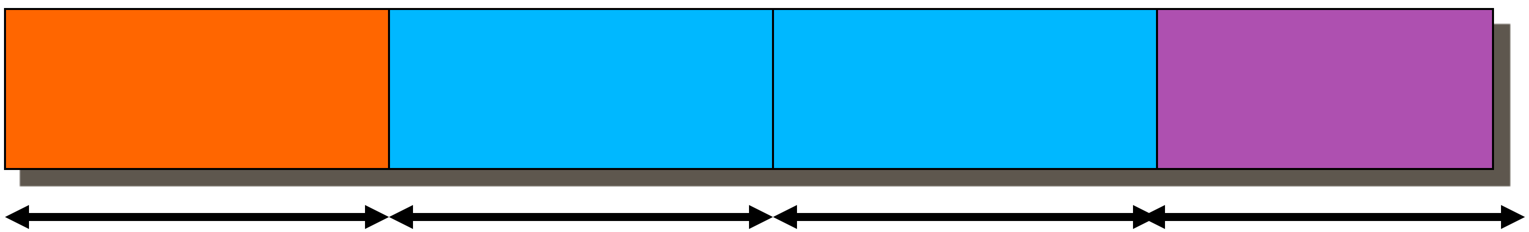
-: 2

*: 3

/: 4

Example Encoding

Add cell 28 to cell 63 and place result in cell 100:



Instruction as a number in:

Decimal: 1:28:63:100

Binary: 00000001:00011100:00111111:01100100

Hexadecimal: 01:1C:3F:64

Example Encoding (cont)

How many instructions can this encoding have?

- 8 bits, 2^8 combinations = 256 instructions

How much memory can this example instruction set support?

- Assume each memory cell is a byte (8 bits) wide
- Assume operands and destination come from the same memory
- 8 bits per source/dest = 2^8 combinations = 256 bytes

How many bytes did we use per instruction?

- 4 bytes per instruction

How could we get more memory without changing the encoding?

Why is this simple encoding not realistic?

The Program Counter

Where is the “next instruction” held in the machine?

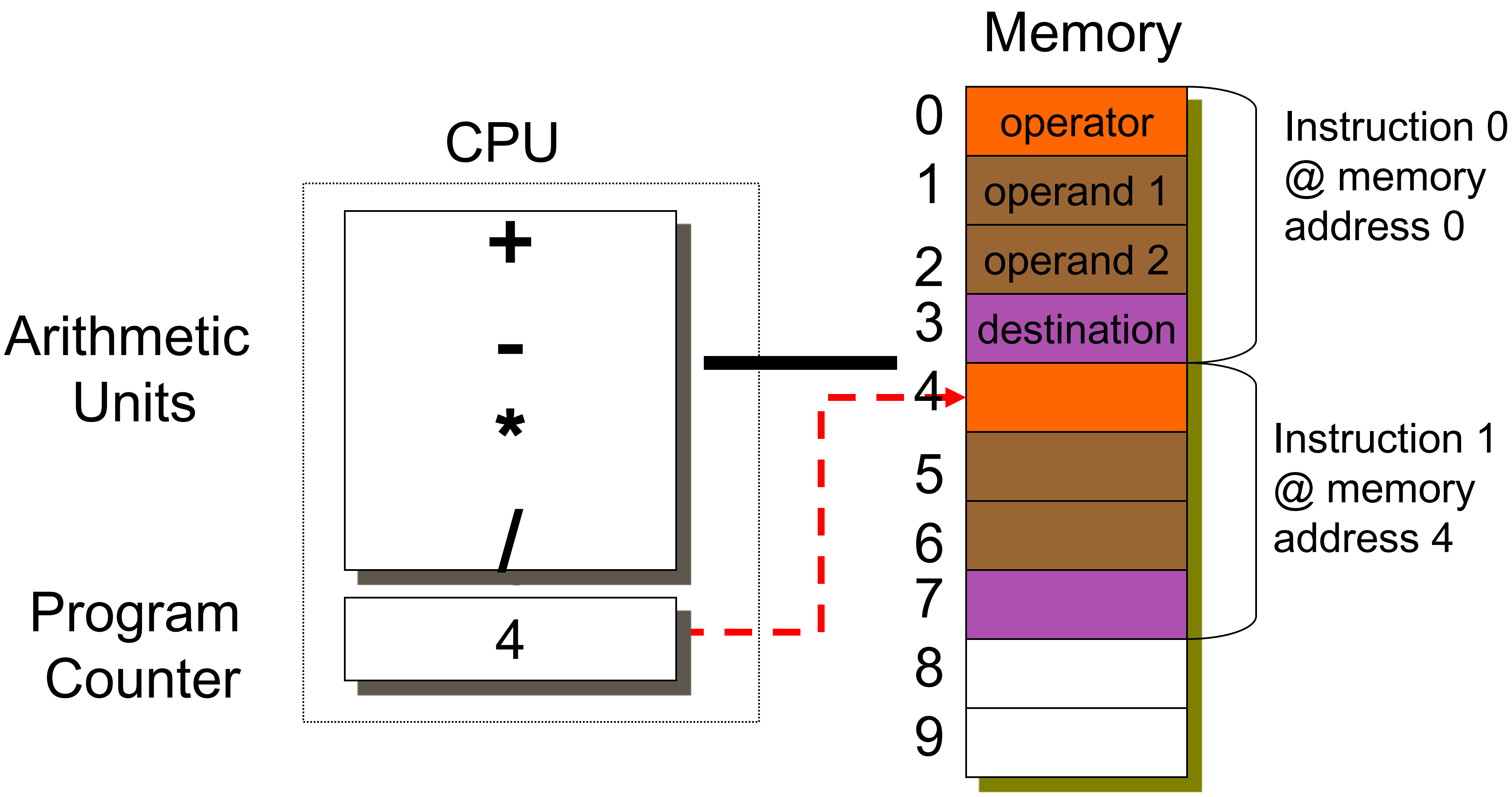
In a special memory cell in the CPU called the “program counter" (the PC)

- Special purpose memory in the CPU and devices are called registers

Naive fetch cycle: Increment the PC by the instruction length (4) after each execute

- Assumes all instructions are the same length

Conceptual Model



Memory Indirection

How do we access array elements efficiently if all we can do is name a cell?

Modify the operand to allow for fetching an operand "through" a memory location

- E.g.: LOAD [5], 2 means fetch the contents of the cell whose address is in cell 5 and put it into cell 2
- So if cell 5 had the number 100, we would place the contents of cell 100 into cell 2

This is called indirection

- Fetch the contents of the cell “pointed to” by the cell in the opcode

Steal an operand bit to signify if an indirection is desired

Conditionals and Looping

Instructions that modify the Program Counter

Branch Instructions

- If the content of this cell is [zero, non zero, etc.], set the PC to this location
- jump is an unconditional branch

Example: While Loop

```
while (counter > 0) {
    sum = sum + Y[counter];
    counter--;
};
```

Variables to memory cells:

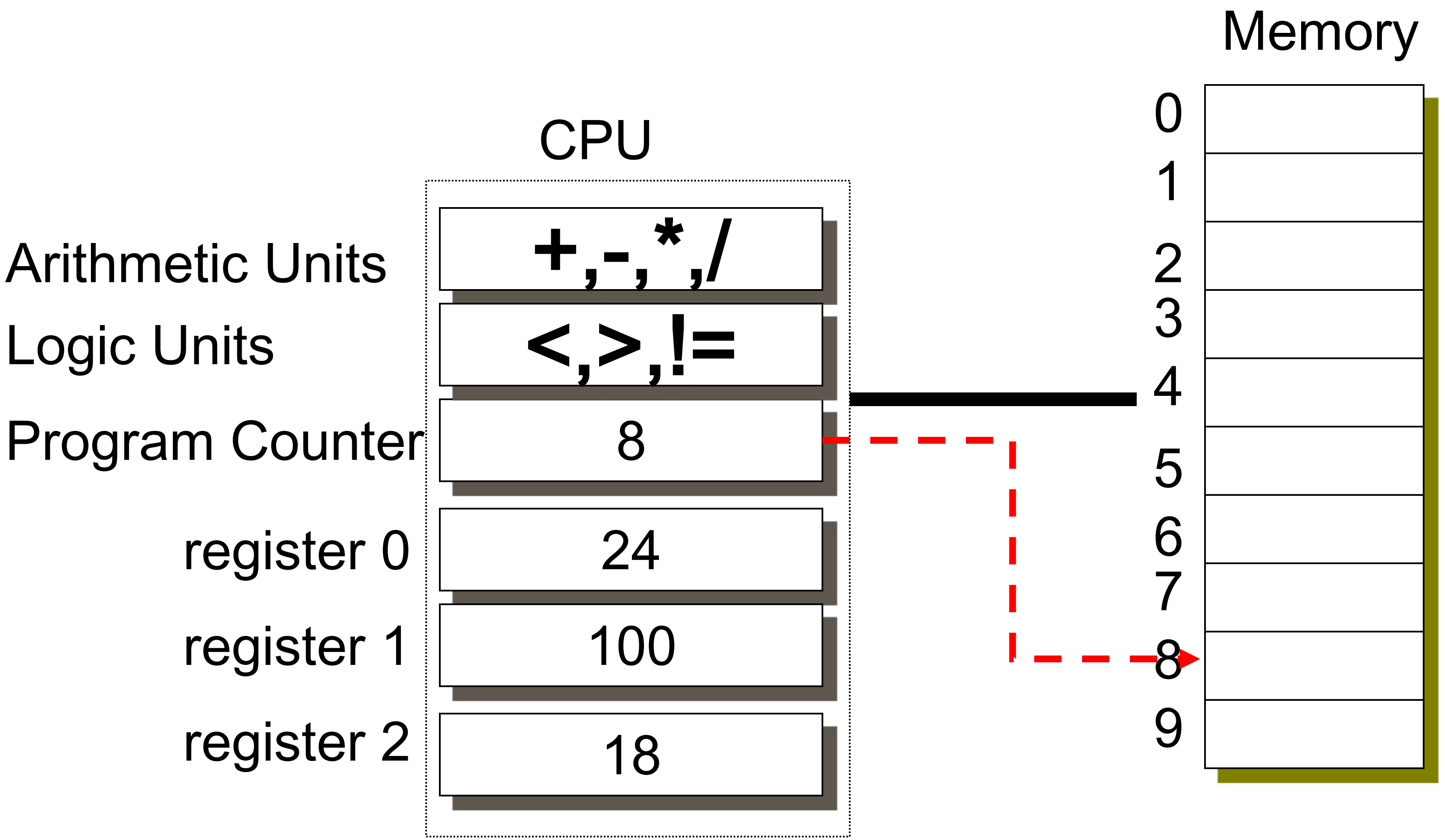
```
counter is cell 1
sum is cell 2
index is cell 3
Y[0]= cell 4, Y[1]=cell 5…
```

Memory cell address	Assembler label	Assembler "mnemonic"	English
100	LOOP:	BNZ 1,END	// branch to address of END // if cell 1 is not 0.

104		ADD 2,[3],2	// Add cell 2 and the value
			// of the cell pointed to by
			// cell 3 then place the
			// result in cell 2

108		DEC 3	// decrement cell 3 by 1
112		DEC 1	// decrement cell 1 by 1
116		JUMP LOOP	// start executing from the // address of LOOP
120	END:	<next code block>	

Register Machine Model



Registers (cont)

- Most CPUs have 16-32 “general purpose” registers
  - All look the “same”: combination of operators, operands and destinations possible
- Operands and destination can be in:
  - Registers only (Sparc, PowerPC, Mips, Alpha)
  - Registers & 1 memory operand (x86 and clones)
  - Any combination of registers and memory (Vax)
- Only memory operations possible in "register-only" machines are load from and store to memory
- Operations 100-1000 times faster when operands are in registers compared to when they are in memory
- Save instruction space too
  - Only address 16-32 registers, not GB of memory

Typical Instructions

Add the contents of register 2 and register 3 and place result in register 5

ADD r2,r3,r5

Add 100 to the PC if register 2 is not zero

Relative branch

BNZ r2,100

Load the contents of memory location whose address is in register 5 into register 6

LDI r5,r6

Abstracting the Machine

Bare hardware provides a computation device

How to share this expensive piece of equipment between multiple users?*

Sign up during certain hours?   Give program to an operator?

Software to give the illusion of having it all to yourself while actually sharing it with others (time-sharing)!

This software is the Operating System

Need hardware support to "virtualize" machine

* Software that makes it easy for 1 expensive user to use multiple devices!!

Architecture Features for the OS

Next we'll look at the mechanisms the hardware designers add to allow OS designers to abstract the basic machine in software

- Processor modes
- Exceptions
- Traps
- Interrupts

These require modifications to the basic fetch-decode-execute cycle in hardware

Processor Modes

- OS code is stored in memory … von Neumann model, remember?
  - What if a user program modifies OS code or data?
- Introduce modes of operation
  - Instructions can be executed in user mode or system mode
- A special register holds which mode the CPU is in
- Certain instructions can only be executed when in system mode
- Likewise, certain memory location can only be written when in system mode
  - Only OS code is executed in system mode
    - Only OS can modify its memory
  - The mode register can only be modified in system mode

Simple Protection Scheme

All addresses < 100 are reserved for operating system use

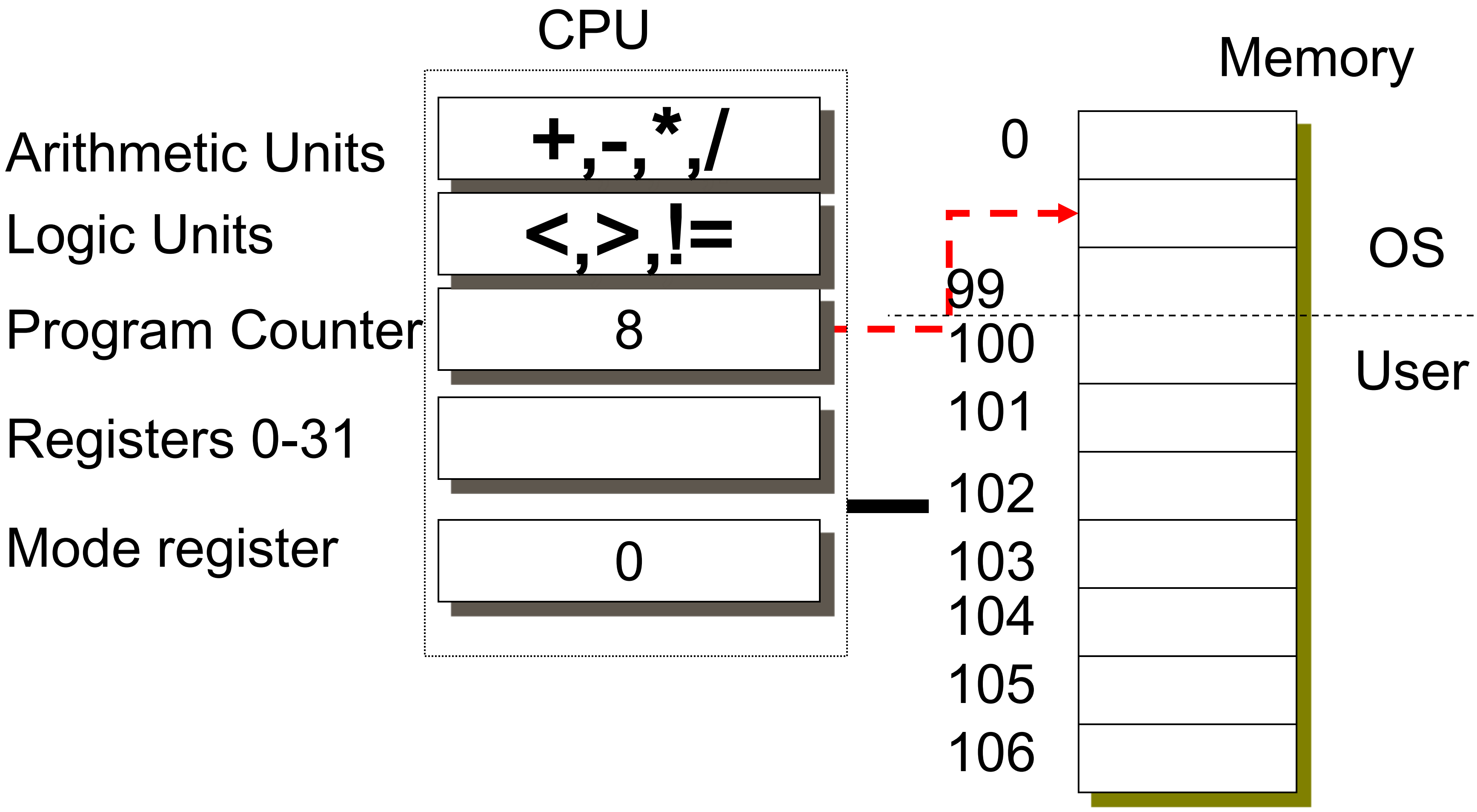
Mode register provided

- zero = CPU is executing the OS (in system mode)
- one = CPU is executing in user mode

Hardware does this check:

- On every fetch, if the mode bit is 1 and the address is less than 100, then do not execute the instruction
- When accessing operands, if the mode bit is 1 and the operand address is less than 100, do not execute the instruction
- Mode register can only be set if mode is 0

Simple Protection Model



Fetch-decode-execute Revised

Fetch:

if ((the PC < 100) && (the mode register == 1)) then

Error! User tried to access the OS

else

fetch the instruction at the PC

Decode:

if ((destination register == mode) && (the mode register == 1)) then

Error! User tried to set the mode register

< more decoding >

Execute:

if ((an operand < 100) && (the mode register == 1) then

error! User tried to access the OS

else

execute the instruction

Exceptions

What happens when a user program tries to access memory holding the operating system code or data?

Answer: exceptions

An exception occurs when the CPU encounters an instruction which cannot be executed

Modify fetch-decode-execute loop to jump to a known location in the OS when an exception happens

Different errors jump to different places in the OS (are "vectored" in OS speak)

Fetch-decode-execute with Exceptions

Fetch:

if ((the PC < 100) && (the mode bit == 1)) then

set the PC = 60 ← 60 is the well known entry point for a memory violation

set the mode = 0

fetch the instruction at the PC

Decode:

if ((destination register == mode) && (the mode register == 1)) then

set the PC = 64 ← 64 is the well known entry point for a mode register violation

set the mode = 0

goto fetch

< more decoding >

Execute:

< check the operands for a violation>

Access Violations

Notice both instruction fetch from memory and data access must be checked

Execute phase must check both operands

Execute phase must check again when performing an indirect load

This is a very primitive memory protection scheme. We'll cover more complex *virtual memory* mechanisms and policies later in the course

Recovering from Exceptions

The OS can figure out what caused the exception from the entry point

But how can it figure out where in the user program the problem was?

Solution: add another register, the PC'

- When an exception occurs, save the current PC to PC' before loading the PC with a new value

OS can examine the PC' and perform some recovery action

- Stop user program and print an error message: error at address PC'
- Run a debugger

Fetch-decode-execute with Exceptions & Recovery

```
Fetch:
        if (( the PC < 100) && ( the mode bit == 1)) then
                set the PC' = PC
                set the PC = 60
                set the mode = 0
Decode:
        if (( destination register == mode) && ( the mode register == 1)) then
                set the PC' = PC
                set the PC = 64
                set the mode = 0
                goto fetch
        < more decoding >
Execute:
         …
```