CS354
Operating Systems

http://www.cs.purdue.edu/people/comer

Module I

Overview

Topics

- Introductions
- Course motivation and scope
- Xinu and the lab
- Required background
- The drop/stay decision
- Syllabus

INTRODUCTIONS

COURSE MOTIVATION AND SCOPE

Scope

This is a course about the design and structure of computer operating systems. It covers the concepts, principles, functionality, tradeoffs, and implementation of systems that support concurrent processing.

What We Will Cover

- Operating system design
- Functionality an operating system offers
- Major system components
- Interdependencies and system structure
- The key relationships between operating system abstractions and the underlying hardware (especially processes and interrupts)
- Implementation details

What You Will Learn

- Fundamental
  - Principles
  - Design options
  - Tradeoffs
- How to modify and test operating system code
- How to design and build an operating system

What We Will NOT Cover

- The course is not
  - A comparison of large commercial and open source operating systems
  - A description of features or instructions on how to use a particular operating system
  - A survey of research systems and alternative approaches that have been studied
  - A set of techniques for building operating systems on unusual hardware

How Did We Get Here?

How Did We Get Here?

- 1940s – 1950s: How can computers be built?
  - The dawn of digital computers
  - Each processor special-purpose
  - Unique interface for each I/O device
  - Software written in assembly language
  - Programs written to control the hardware

How Did We Get Here?
(continued)

- 1960s: What are the best abstractions to use?
  - A move to general-purpose hardware
  - *Instruction Set Architecture* emerges
  - Families of computers devised
  - I/O became independent of specific hardware details
  - High-level programming languages created (e.g., parameterized functions, data types, and recursion)

How Did We Get Here?
(continued)

- Late 1960s and 1970s
  - Researchers investigate operating systems
  - Multics project devises brilliant abstractions
  - Unix introduces simple, elegant, and efficient versions of the Multics abstractions
- 1980s and on
  - The Internet project discovers a set of communication abstractions that are elegant and efficient
  - All systems are connected

What Can We Conclude From History?

What Can We Conclude From History?

- The gap between hardware and users is huge

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful
- Everything is connected

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful
- Everything is connected
  - Data centers

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful
- Everything is connected
  - Data centers
  - Desktops

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful
- Everything is connected
  - Data centers
  - Desktops
  - Smart phones

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful
- Everything is connected
  - Data centers
  - Desktops
  - Smart phones
  - Devices

What Can We Conclude From History?

- The gap between hardware and users is huge
  - Hardware is ugly
  - Operating system abstractions are beautiful
- Everything is connected
  - Data centers
  - Desktops
  - Smart phones
  - Devices
  - Sensors

Consequences And Observations

Consequences And Observations

- Our job in Computer Science is to build beautiful new abstractions that fill the gap between ugly hardware and users

Consequences And Observations

- Our job in Computer Science is to build beautiful new abstractions that fill the gap between ugly hardware and users
- We cannot consider operating systems without including computer networking

Consequences And Observations

- Our job in Computer Science is to build beautiful new abstractions that fill the gap between ugly hardware and users
- We cannot consider operating systems without including computer networking
- The central questions in computing have always focused on the tradeoff between imagined beauty and performance

Consequences And Observations

- Our job in Computer Science is to build beautiful new abstractions that fill the gap between ugly hardware and users
- We cannot consider operating systems without including computer networking
- The central questions in computing have always focused on the tradeoff between imagined beauty and performance
  - It is easy to imagine new abstractions

Consequences And Observations

- Our job in Computer Science is to build beautiful new abstractions that fill the gap between ugly hardware and users
- We cannot consider operating systems without including computer networking
- The central questions in computing have always focused on the tradeoff between imagined beauty and performance
  - It is easy to imagine new abstractions
  - We must restrict ourselves to abstractions that map onto the underlying hardware efficiently (and hope that hardware engineers eventually adapt to our abstractions)

The Once And Future Hot Topic

- In the 1970s and early 1980s, operating systems was one of the hottest topics in CS
- By the mid-1990s, OS research had stagnated
- Now things have heated up again, and new operating systems are being designed for
  - Multicore systems
  - Data centers
  - Smart phones
  - Large and small embedded devices
  - The Internet of Things

XINU AND THE LAB

Motivation For Using A Real System

- Provides examples of the principles
- Makes everything concrete
- Gives students a chance to experiment
- Shows how abstractions map to real hardware

Why Xinu?

- Small — can be read and understood in a semester
- Complete — includes all the major components
- Elegant — provides an excellent example of clean design
- Powerful — has dynamic process creation, dynamic memory management, and flexible I/O
- Practical — has been used in real products

The Xinu Lab

- Innovative facility for rapid OS development and testing
- Completely automated
- Allows each student to create, download, and run code on bare hardware
- Handles hardware reboot when necessary
- Includes connections to the Internet

REQUIRED BACKGROUND AND PREREQUISITES

Background Needed

- Concepts from undergrad operating systems
  - Concurrent programming: you should have written a program that uses *fork* or the equivalent
  - Understanding of deadlock and race conditions
  - I/O: you should know the difference between standard library functions (e.g., *fopen*, *putc*, *getc*, *fread*, *fwrite*) and system calls (e.g., *open*, *close*, *read*, *write*)
  - File systems and hierarchical directories
  - Symbolic and hard links
  - File modes and protection

Background Needed
(continued)

- Understanding of runtime storage components
  - Segments (text, data, and bss) and their layout
  - Runtime stack, activation records, and argument passing
  - Basic heap storage management (free list)
- C programming
  - At least one nontrivial program
  - Comfortable with low-level constructs (e.g., bit manipulation and pointers)

Background Needed
(continued)

- Working knowledge of basic UNIX tools
  - Text editor (e.g., emacs)
  - Compiler / linker / loader
  - Tar archives
  - Make and Makefiles
- Desire to learn

SYLLABUS

How We Will Proceed

- We will examine the major components of an operating system
- For a given component we will
  - Outline the functionality needed
  - Learn the key principles involved
  - Understand one particular design choice in depth
  - Consider implementation details and the relationship to hardware
  - Discuss other possibilities and tradeoffs
- Note: we will cover components in a linear order that allows us to understand one component at a time without relying on later components

Questions?

A FEW THINGS TO THINK ABOUT

Perfection [in design] is achieved not when there is nothing to add, but rather when there is nothing more to take away.

– Antoine de Saint-Exupery

A teacher's job is to make the agony of decision making so intense you can only escape by thinking.

– source unknown

Real concurrency — in which one program actually continues to function while you call up and use another — is more amazing but of small use to the average person. How many programs do you have that take more than a few seconds to perform any task?

(From an article about new operating systems for the IBM PC in the New York Times, 25 April 1989)

Module II

Organization Of An Operating System

What Is An Operating System?

- Provides abstract computing environment
- Supplies computational services
- Manages resources
- Hides low-level hardware details
- Note: operating system software is among the most complex ever devised

Example Services An OS Supplies

- Support for concurrent execution
- Process synchronization
- Inter-process communication mechanisms
- Message passing and asynchronous events
- Management of address spaces and virtual memory
- Protection among users and running applications
- High-level interface for I/O devices
- A file system and file access facilities
- Intermachine communication

What An Operating System Is NOT

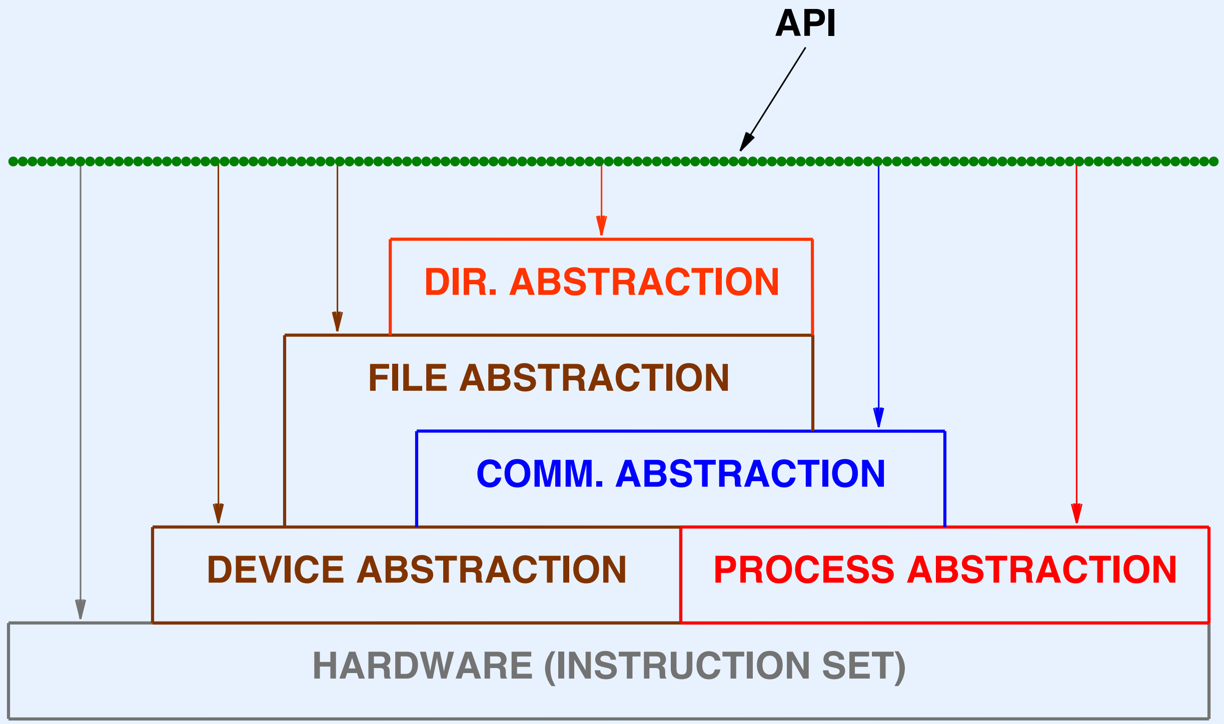
- Hardware
- Language
- Compiler
- Windowing system or browser
- Command interpreter
- Library of utility functions
- Graphical desktop

AN OPERATING SYSTEM FROM THE OUTSIDE

The System Interface

- Single copy of OS per computer
  - Hidden from users
  - Accessible only to application programs
- *Application Program Interface (API)*
  - Defines services OS makes available
  - Defines parameters for the services
  - Provides access to all abstractions
  - Hides hardware details

OS Abstractions And Application Interface



- Modules in the OS offer services
- Some services build on others

Interface To System Services

- Appears to operate like a function call mechanism
  - OS makes set of “functions” available to applications
  - Application supplies arguments using standard mechanism
  - Application “calls” one of the OS functions
- Control transfers to OS code that implements the function
- Control returns to caller when function completes

Interface To System Services
(continued)

- Requires special instruction to invoke OS function
  - Moves from application *address space* to OS
  - Changes from application *mode* or *privilege level* to OS
- Terminology used by various vendors
  - *System call*
  - *Trap*
  - *Supervisor call*
- We will use the generic term *system call*

Example System Call In Xinu: Write A Character On The Console

```
/* ex1.c - main */

#include <xinu.h>

/*------------------------------------------------------------------------
 * main  -  Write "hi" on the console
 *------------------------------------------------------------------------
 */
void    main(void)
{
        putc(CONSOLE, 'h');
        putc(CONSOLE, 'i');
        putc(CONSOLE, '\n');
}
```

- Note: we will discuss the implementation of *putc* later.

OS Services And System Calls

- Each OS service accessed through system call interface
- Most services employ a set of several system calls
- Examples
  - Process management service includes functions to *suspend* and then *resume* a process
  - *Socket API* used for Internet communication includes many functions

System Calls Used With I/O

- Open-close-read-write paradigm
- Application
  - Uses *open* to connect to a file or device
  - Calls functions to *write* data or *read* data
  - Calls *close* to terminate use
- Internally, the set of I/O functions coordinate
  - *Open* returns a descriptor, *d*
  - *Read* and *write* operate on descriptor *d*

Questions

- How many system calls does an OS need?
- What should they be?

Concurrent Processing

- Fundamental concept that dominates OS design
- *Real concurrency* achieved by hardware
  - I/O devices operate at same time as processor
  - Multiple processors/cores each operate at the same time
- *Apparent concurrency* achieved with multitasking (multiprogramming)
  - Multiple programs appear to operate simultaneously
  - Operating system provides the illusion

Multitasking

- Powerful abstraction
- Allows user(s) to run multiple computations
- OS switches processor(s) among available computations quickly
- All computations appear to proceed in parallel

Terminology Used With Multitasking

- A *program* consists of static code and data
- A *function* is a unit of application program code
- A *process* (also called a *thread of execution*) is an active computation (i.e., the execution or "running" of a program)

Process

- OS abstraction
- Created by OS system call
- Managed entirely by OS; unknown to hardware
- Operates concurrently with other processes

Example Of Process Creation In Xinu (Part 1)

```
/* ex2.c - main, sndA, sndB */

#include <xinu.h>

void    sndA(void), sndB(void);

/*------------------------------------------------------------------------
 * main  -  Example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndA, 1024, 20, "process 1", 0) );
        resume( create(sndB, 1024, 20, "process 2", 0) );
}

/*------------------------------------------------------------------------
 * sndA  -  Repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}
```

Example Of Process Creation In Xinu (Part 2)

```
/*------------------------------------------------------------------------
 * sndB  -  Repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

Difference Between Function Call And Process Creation

- Normal function call
  - Synchronous execution
  - Single computation
- The *create* system call that starts a new process
  - Asynchronous execution
  - Two processes proceed after the call

Distinction Between A Program And A Process

- Sequential program
  - Set of functions executed by a single thread of control
- Process
  - Computational abstraction not usually part of the programming language
  - Created independent of code that is executed
  - Key idea: multiple processes can execute the same code concurrently
- In the following example, two processes execute function *sndch* concurrently

Example Of Two Processes Sharing Code

```
/* ex3.c - main, sndch */

#include <xinu.h>

void    sndch(char);

/*------------------------------------------------------------------------
 * main  -  Example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 1024, 20, "send A", 1, 'A') );
        resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  -  Output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(
          char  ch              /* The character to emit continuously   */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

Storage Allocation When Multiple Processes Execute

- Various memory models exist for multitasking environments
- Each process requires its own
  - Runtime stack for function calls
  - Storage for local variables
  - Copy of arguments
- A process *may* have private heap storage as well

Consequence For Programmers

A copy of function arguments and local variables are associated with each process executing a particular function, *not* with the code in which they are declared.

AN OPERATING SYSTEM
FROM THE INSIDE

Operating System

- Well-understood subsystems
- Many subsystems employ heuristic policies
  - Policies can conflict
  - Heuristics can have corner cases
- Complexity arises from interactions among subsystems
- Side-effects can be
  - Unintended
  - Unanticipated
- We will see examples

Building An Operating System

Building An Operating System

- The intellectual challenge comes from the design of a "system" rather than from the design individual pieces
- Structured design is needed
- It can be difficult to understand the consequences of choices
- We will use a hierarchical microkernel design to help control complexity

Major OS Components

- Process manager
- Memory manager
- Device manger
- Clock (time) manager
- File manager
- Interprocess communication
- Intermachine communication
- Assessment and accounting

Multilevel Structure

- Organizes components
- Controls interactions among subsystems
- Allows a system to be understood and built incrementally
- Differs from traditional layered approach
- Will be employed as the design paradigm throughout the text and course

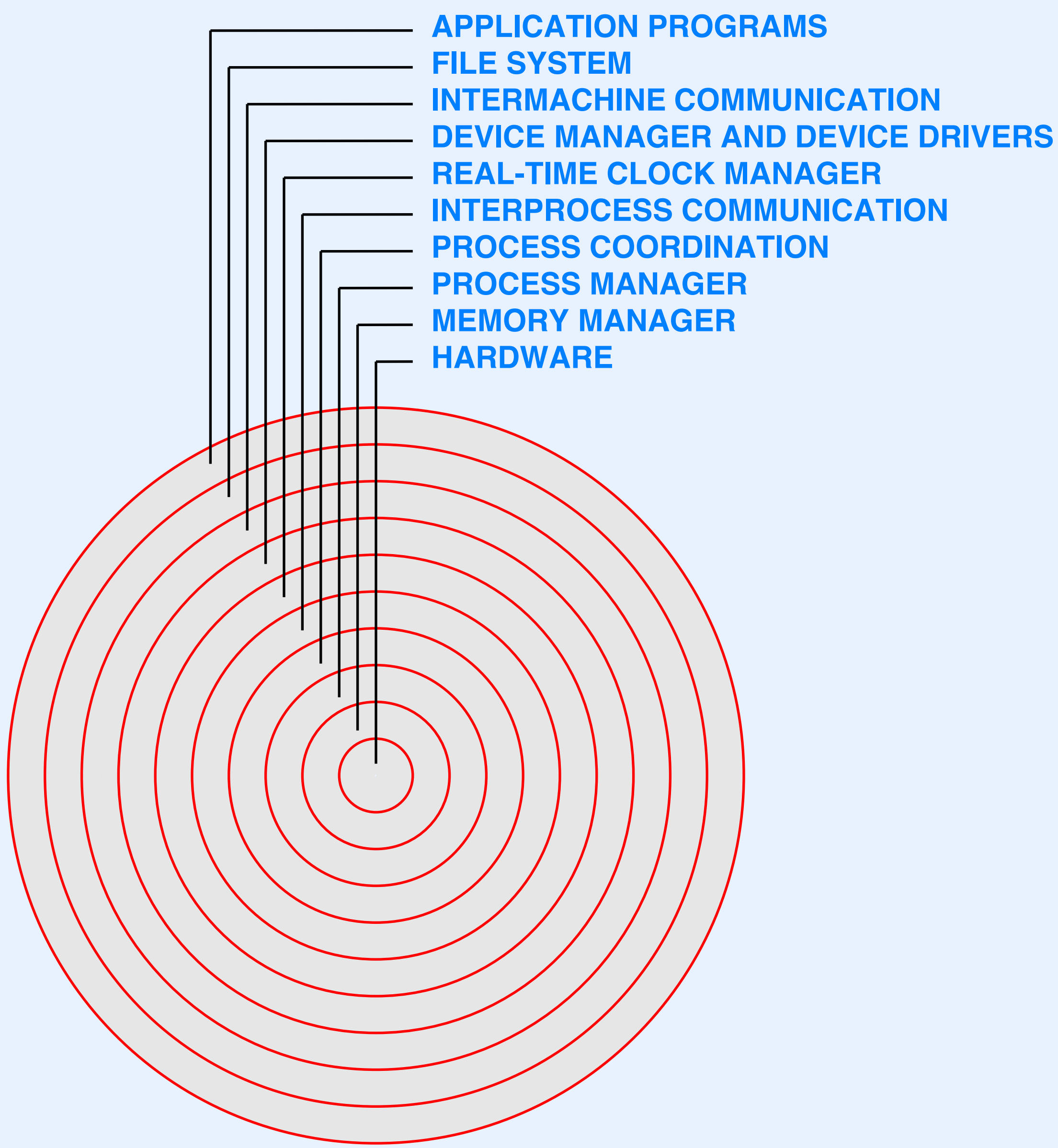
Multilevel Vs. Multilayered Organization

- Multilayer structure
  - Visible to user as well as designer
  - Each layer uses layer directly beneath
  - Involves protection as well as data abstraction
  - Examples
    - Internet protocol layering
    - MULTICS layered security structure
  - Can be inefficient

Multilevel Vs. Multilayered Organization
(continued)

- Multilevel structure
  - Form of data abstraction
  - Used during system construction
  - Helps designer focus attention on one aspect at a time
  - Keeps policy decisions independent
  - Allows given level to use *all* lower levels
  - Efficient

Multilevel Structure Of Xinu



How To Build An OS

- Work one level at a time
- Identify a service to be provided
- Begin with a *philosophy*
- Establish *policies* that follow the philosophy
- Design *mechanisms* that enforce the policies
- Construct an *implementation* for specific hardware

Design Example

- Example: access to I/O
- Philosophy: “fairness”
- Policy: FCFS resource access
- Mechanism: queue of requests (FIFO)
- Implementation: program written in C

Summary

- Operating system supplies set of services
- System calls provide interface between OS and application
- Concurrency is fundamental concept
  - Between I/O devices and processor
  - Between multiple computations
- Process is OS abstraction for concurrency
- Process differs from program or function
- You will learn how to design and implement system software that supports concurrent processing

Summary
(continued)

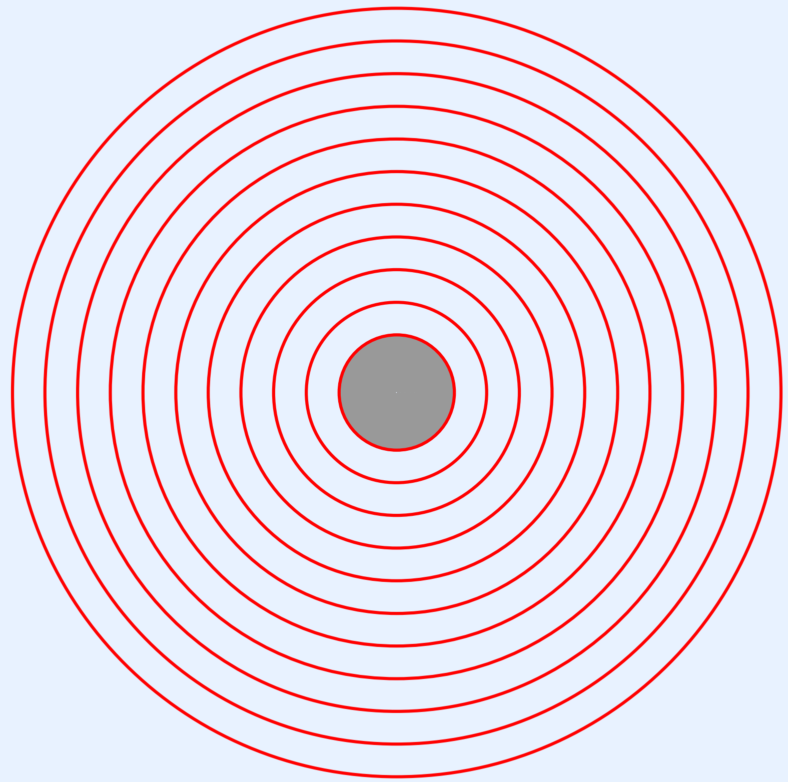
- OS has well-understood internal components
- Complexity arises from interactions among components
- Multilevel approach helps organize system structure
- Design involves inventing policies and mechanisms that enforce overall goals
- Xinu includes a compact list structure that uses relative pointers and an implicit data structure to reduce size
- Xinu type names specify both purpose and data size

Module III

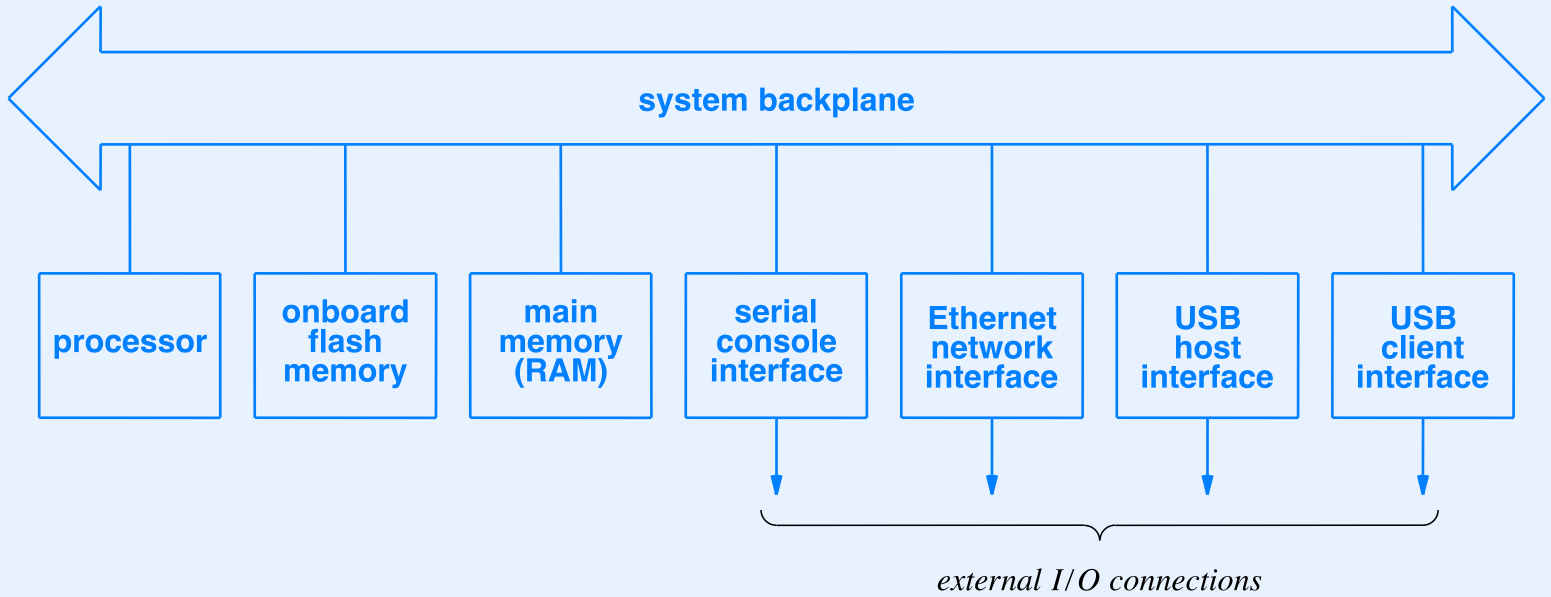
Hardware Architecture And Runtime Systems

(A Brief Overview)

Location Of Hardware In The Hierarchy



Organization Of Major Hardware Components



- Example platforms use *System on Chip* (*SoC*) approach
- Single VLSI chip contains components and interconnections
- Not relevant to operating system

Key Hardware Features From An Operating System Point Of View

- Processor
- Memory system
- Bus and address space
- Individual I/O devices
- Interrupt structure

Processor

- Instruction set
- Registers
- Optional hardware facilities
  - Firmware (e.g., BIOS in ROM)
  - Co-processor
  - Interrupt processor

Instruction Sets On The Example Platforms

- Galileo board
  - CISC design
  - Well-known Intel (x86) instruction set
- BeagleBone Black
  - RISC design
  - Well-known ARM instruction set
- Note: we will see how low-level operating system functions differ on the two

General-purpose And Special-purpose Registers

- General-purpose registers
  - Local storage inside processor
  - Hold active values during computation (e.g., used to compute an expression)
  - Saved and restored during subprogram invocation
- Special-purpose registers
  - Located inside the processor
  - Values control processor actions (e.g., mode and address space visibility)

Memory System

- Defines size of a *byte*, the smallest addressable unit
- Provides address space
- Typical physical address space is
  - *Monolithic*
  - *Linear* (but may not be contiguous)
- Includes caching
- Defines important property for programmers: endianness

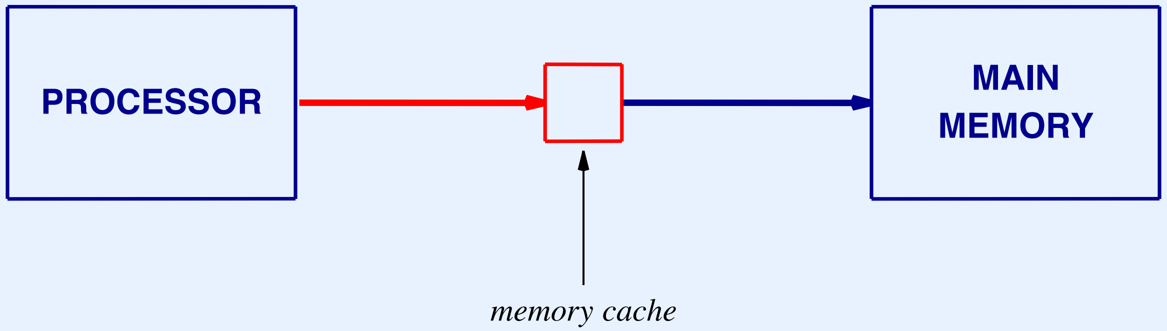
Byte Order Terminology

- Order of bytes within an integer in memory
- Irrelevant to data in registers
- *Little Endian* stores least-significant byte at lowest address
- *Big Endian* stores most-significant byte at lowest address

Memory Caches

- Special-purpose hardware units
- Speed memory access
- Less expensive than high-speed memory
- Physically associated with memory
- Conceptually placed “between” processor and memory

Conceptual Placement Of Memory Cache



- All references (including instruction fetch) go through cache
- Multi-level cache possible
- In practice, placing the cache on the processor chip improves performance
- Key question: are virtual or physical addresses cached?
- Consequence: operating system may need to
  - Flush the cache
  - Avoid the cache when accessing a device

I / O Devices

- Wide variety of peripheral devices available
  - Displays
  - Keyboards / mice
  - Disks
  - Wired and wireless network interfaces
  - Printers and scanners
  - Cameras
  - Sensors
- Multiple transfer paradigms (character, block, packet, stream)

Communication Between Device And Processor

- Communication with I/O devices is *memory-mapped*
- Processor can
  - Interrogate device status
  - Control (e.g., shutdown or awaken) a device
  - Start a data transfer
- Device uses bus to
  - Reply to commands from the processor
  - Transfer data to/from memory
  - Interrupt when operation completes

Bus Operations

- Only two basic operations supported
  - Fetch
  - Store
- All I/O uses fetch-store paradigm
- Fetch
  - Processor places address on bus
  - Processor uses control line to signal *fetch request*
  - Device senses its address
  - Device puts specified data on bus
  - Device uses control line to signal *response*

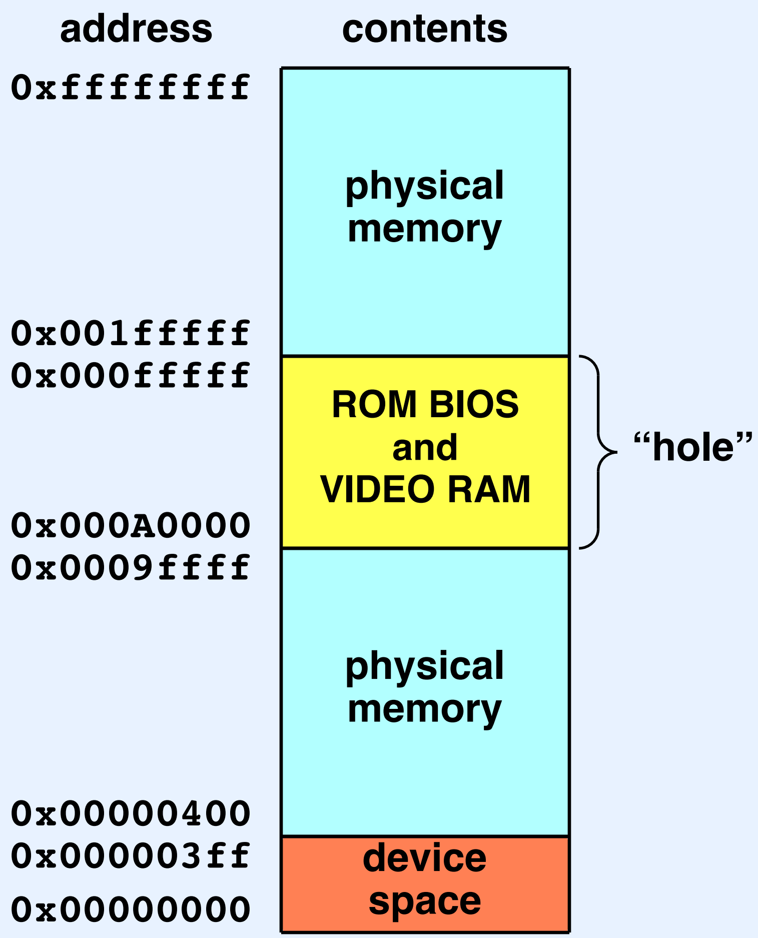
Bus Operations
(continued)

- Store
  - Processor places data and address on bus
  - Processor uses control line to signal *store request*
  - Device senses its address
  - Device extracts data from the bus
  - Device uses control line to signal *data extracted*

Two basic Approaches For I/O

- *Port-mapped I/O*
  - Special instructions used to access devices
  - Used on earlier Intel machines
- *Memory-mapped I/O*
  - Devices placed beyond physical memory
  - Processor uses normal fetch / store memory instructions
  - On later Intel machines as well as most others

Address Space On An Intel System

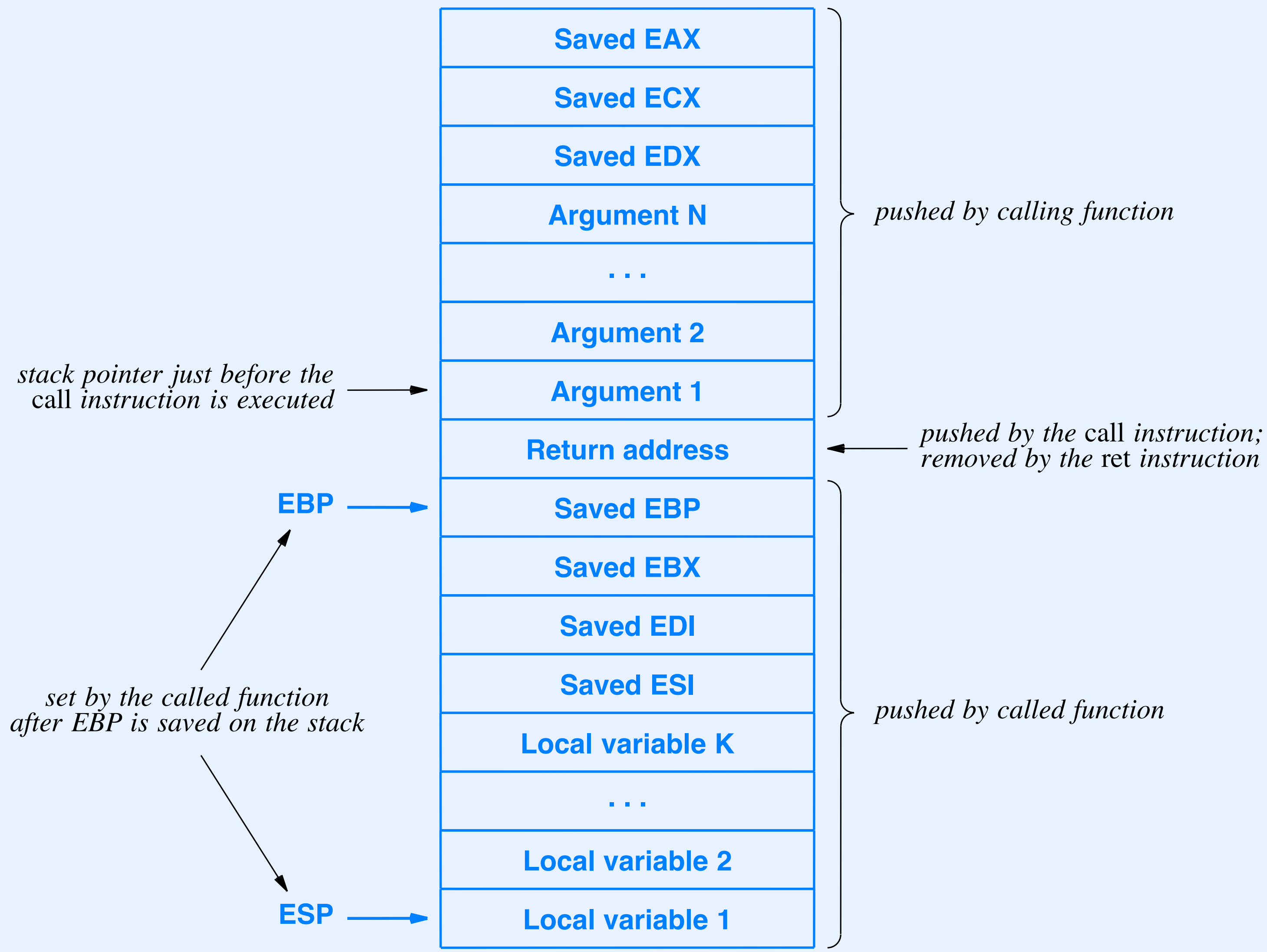


- Often discontiguous
- Traditional PCs have a "hole" from 640KB to 1MB

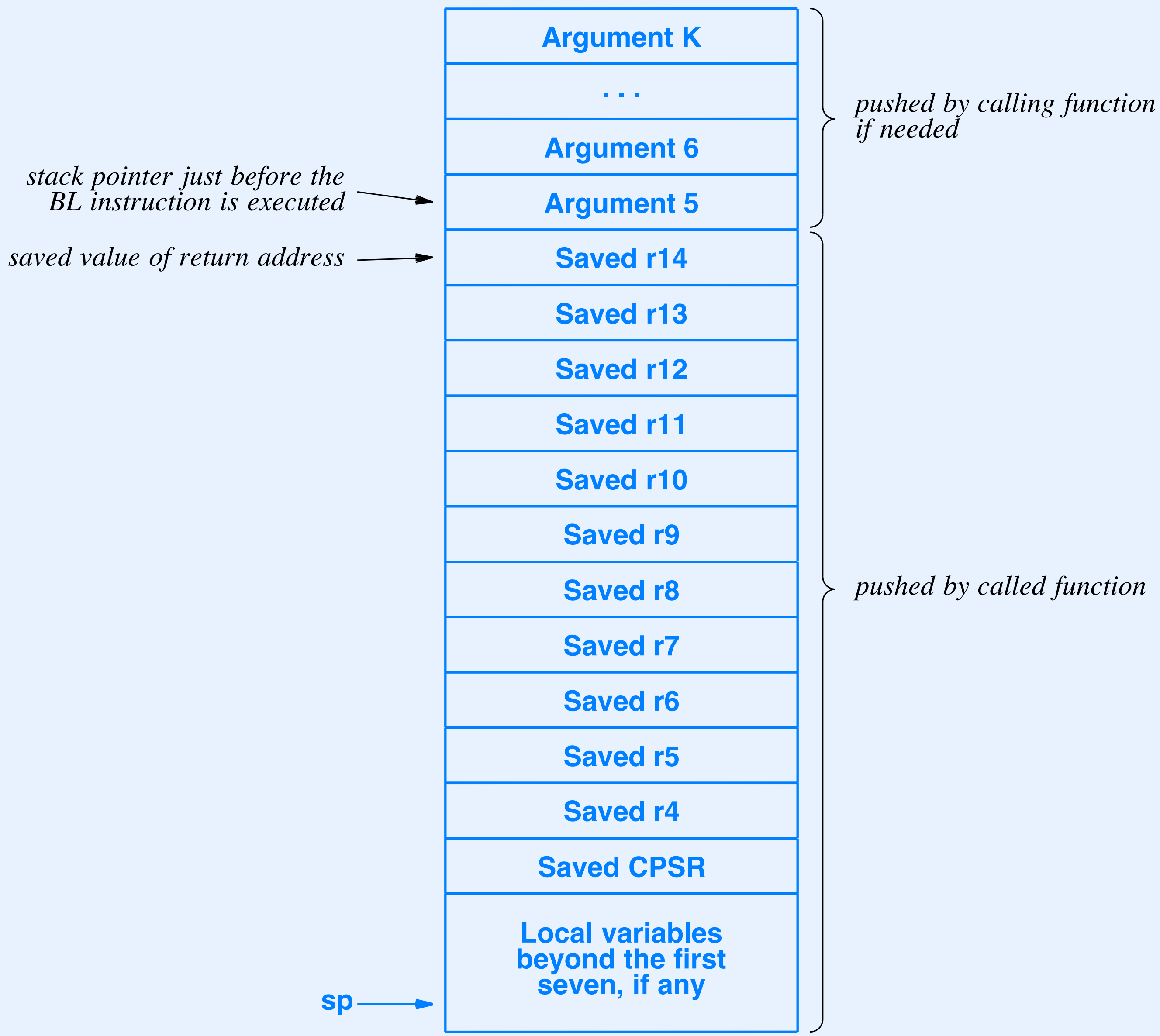
Calling Conventions

- Refer to the set of steps taken during a function call
- The conventions specify
  - Which registers must be saved by caller
  - Which registers must be saved by called function
  - Where registers are saved (e.g., on the stack)
- Specific details may depend on
  - The hardware
  - The compiler being used
- We will see that key operating systems functions cannot be implemented unless the calling conventions are known

Calling Conventions For Intel



Calling Conventions For ARM



Interrupt Mechanism

- Fundamental role in modern system
- Permits I/O concurrent with execution
- Allows device priority
- Informs processor when I/O finished
- *Software interrupt* also possible

Interrupt-Driven I/O

- Processor starts device
- Device operates concurrently
- Device interrupts the processor when finished with the assigned task
- Interrupt timing
  - Asynchronous wrt computation
  - Synchronous wrt an individual instruction (occurs between instructions)

Interrupt Mask

- Bit mask kept in a processor status register
- Determines whether interrupts are enabled
- Hardware sets mask during interrupt to prevent further interrupts
- Interrupt priority scheme
  - Offered by some hardware
  - Each device assigned priority level (binary number)
  - When servicing level *K* interrupt, hardware sets mask to disable interrupts at level *K* and lower

Interrupt Processing

- Operating system must
  - Store address of interrupt code in *vector* for each device
  - Arrange for interrupt code to save registers used during the interrupt and restore registers before returning
- More details later in the course

Returning From An Interrupt

- Special hardware instruction used
- Atomically restores
  - Old program state
  - Interrupt mask
  - Instruction pointer (program counter)
- Hardware returns to location where interrupt occurred, and processing continues exactly as if no interrupt happened

Transfer Size And Interrupts

- Interrupt occurs after I/O operation completes
- Transfer size depends on device
  - Serial device transfers one character
  - Disk controller transfers one block (e.g., 512 bytes)
  - Network interface transfers one packet
- Large transfers use *Direct Memory Access* (*DMA*)

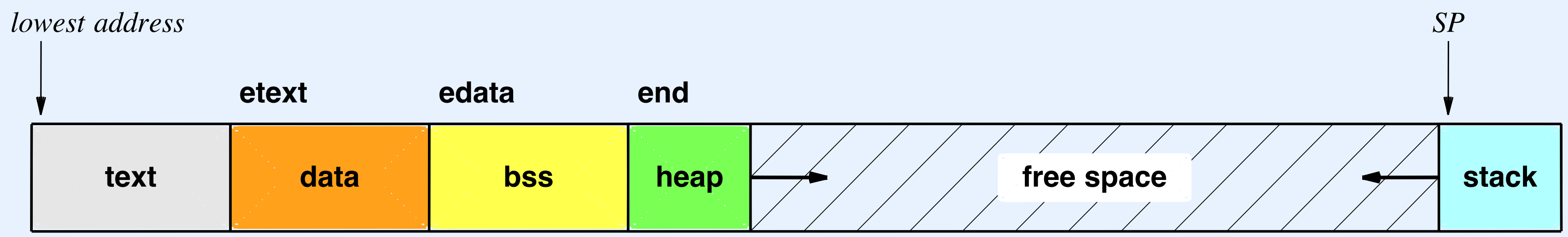
Direct Memory Access (DMA)

- Hardware mechanism
- I/O device transfers large block of data to/from memory without using the processor
- Example: network interface places incoming network packet in memory buffer
- Advantage: allows processor to execute during I/O transfer
- Disadvantages
  - More expensive
  - More complex to design and program

Memory Segments In C Programs

- C Program has four primary data areas called *segments*
- Text segment
  - Contains program code
  - Usually unwritable
- Data segment
  - Contains initialized data values (globals)
- Bss segment
  - Contains uninitialized data values (set to zero)
- Stack segment
  - Used for function calls

Storage Layout For A C Program



- Stack grows downward (toward lower memory addresses)
- Heap grows upward

Symbols For Segment Addresses

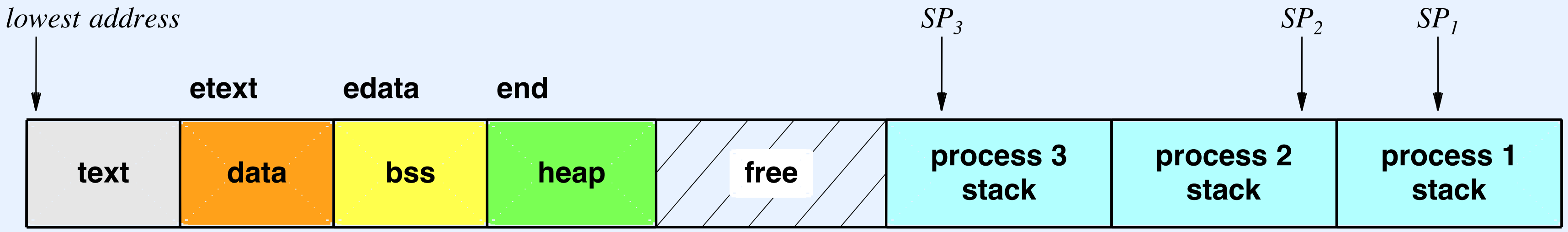
- C compiler and / or linker adds three reserved names to symbol table
- _*etext* lies beyond text segment
- _*edata* lies beyond data segment
- _*end* lies beyond bss segment
- Only the addresses are significant; values are irrelevant
- Program can use the addresses of the reserved symbol to determine the size of segments
- Note: names are declared to be *extern* without the underscore:

```
extern  int   end;
```

Runtime Storage For A Process

- Text is shared
- Stack cannot be shared
- Data area *may* be shared
- Exact details depend on address space model OS offers

Example Runtime Storage Model: Xinu



- Single, shared copy of
  - Text segment
  - Data segment
  - Bss segment
- One stack segment per process
  - Allocated when process created
  - Each process has its own stack pointer

Summary

- Components of third generation computer
  - Processor
  - Main memory
  - I / O Devices
    * Accessed over bus
    * Operate concurrently with processor
    * Can be memory-mapped or port-mapped
    * Can use DMA
    * Employ interrupts

Summary
(continued)

- Interrupt mechanism
  - Informs processor when I/O completes
  - Permits asynchronous device operation
- C uses four memory areas: text, data, bss, and stack segments
- Multiple concurrent computations
  - Can share text, data, and bss
  - Cannot share stack

Questions?