

CS354

Operating Systems

Douglas Comer

**CS and ECE Departments
Purdue University**

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

Module I

Course Overview And Introduction To Operating Systems

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 fundamentals
- 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)
- A few implementation details and examples

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

- A comparison of large commercial and open source operating systems
- A description of features or instructions on how to use a particular commercial 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 Operating Systems Changed Programming

- Before operating systems
 - Only one application could run at any time
 - The application contained code to control specific I/O devices
 - The application had to overlap I/O and processing
- Once an operating system was in place
 - Multiple applications could run at the same time
 - An application is not built for specific I/O devices
 - A programmer does not need to overlap I/O and processing
 - An application is written without regard to other applications

Why Operating Systems Are Difficult To Build

- The gap between hardware and high-level services is huge
 - Hardware is ugly
 - Operating system abstractions are beautiful
- Everything is now connected by computer networks
 - An operating system must offer communication facilities
 - Distributed mechanisms (e.g., remote file access) are more difficult to create than local mechanisms

An Observation About Efficiency

- Our job in Computer Science is to build beautiful new abstractions that programmers can use
- It is easy to imagine magical new abstractions
- The hard part is that we must find abstractions that map onto the underlying hardware efficiently
- We hope that hardware engineers eventually build hardware for our abstractions (or at least build hardware that makes our abstractions more efficient)

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
 - Smart phones
 - Multicore systems
 - Data centers
 - Large and small embedded devices (the Internet of Things)

XINU AND THE LAB

Motivation For Studying A Real Operating System

- Provides examples of the principles
- Makes everything clear and concrete
- Shows how abstractions map to current hardware
- Gives students a chance to experiment and gain first-hand experience

Can We Study Commercial Systems?

- Windows
 - Millions of line of code
 - Proprietary
- Linux
 - Millions of line of code
 - Lack of consistency across modules
 - Duplication of functionality with slight variants

An Alternative: 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, flexible I/O, and basic Internet protocols
- Practical — has been used in real products

The Xinu Lab

- Innovative facility for rapid OS development and testing
- Allows each student to create, download, and run code on bare hardware
- Completely automated
- Handles hardware reboot when necessary
- Provides communication to the Internet as well as among computers in the lab

How The Xinu Lab Works

- A student
 - Logs into a conventional desktop system called a *front-end*
 - Modifies and compiles a version of the Xinu OS
 - Requests a computer to use for testing
- Lab software
 - Allocates one of the *back-end* computers for the student to use
 - Downloads the student's Xinu code into the back-end
 - Connects the console from the back-end to the student's window
 - Allows the student to release the back-end for others to use

REQUIRED BACKGROUND AND PREREQUISITES

Background Needed

- A few concepts from earlier courses
 - 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
- Concurrent programming experience: you should have written a program that uses *fork* or *threads*

Background Needed

(continued)

- An understanding of runtime storage components
 - Segments (text, data, and bss) and their layout
 - Runtime stack used for function call; argument passing
 - Basic heap storage management (malloc and free)
- 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 (needed for programming assignments)
 - Text editor (e.g., emacs)
 - Compiler / linker / loader
 - Tar archives
 - Make and Makefiles
- Desire to learn

How We Will Proceed

- We will examine the major components of an operating system
- For a given component we will
 - Outline the functionality it provides
 - Understand principles involved
 - Study one particular design choice in depth
 - Consider implementation details and the relationship to hardware
 - Quickly review 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

Introduction To Operating Systems (Definitions And Functionality)

What Is An Operating System?

- Answer: a large piece of sophisticated software that provides an abstract computing environment
- An OS manages resources and supplies computational services
- An OS hides low-level hardware details from programmers
- Note: operating system software is among the most complex ever devised

Example Services An OS Supplies

- Support for concurrent execution (multiple apps running at the same time)
- Process synchronization
- Process-to-process communication mechanisms
- Process-to-process message passing and asynchronous events
- Management of address spaces and virtual memory support
- Protection among users and running applications
- High-level interface for I/O devices
- File systems and file access facilities
- Internet communication

What An Operating System Is NOT

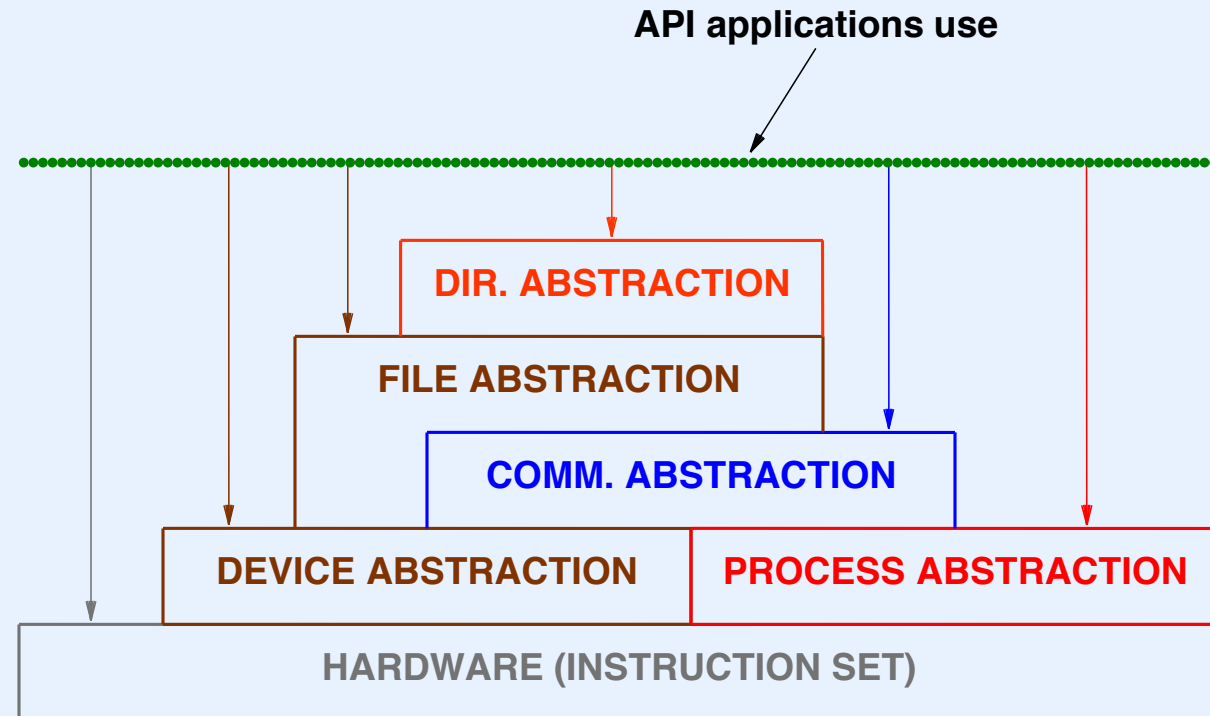
- A hardware mechanism
- A programming language
- A compiler
- A windowing system or a browser
- A command interpreter
- A library of utility functions
- A graphical desktop

AN OPERATING SYSTEM FROM THE OUTSIDE

The System Interface

- A single copy of the OS runs at any time
 - Hidden from users
 - Accessible only to application programs
- The *Application Program Interface (API)*
 - Defines services OS makes available
 - Defines arguments for the services
 - Provides access to OS abstractions and services
 - Hides hardware details

OS Abstractions And The Application Interface



- Modules in the OS offer services to applications
- Internally, 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” an OS function to access a service
- Control transfers to OS code that implements the function
- Control returns to caller when function completes

Interface To System Services

(continued)

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

An Example Of 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*

Concurrent Processing

- Fundamental concept that dominates OS design
- *Real concurrency* is only achieved when hardware operates in parallel
 - I/O devices operate at same time as processor
 - Multiple processors/cores each operate at the same time
- *Apparent concurrency* is achieved with *multitasking* (aka *multiprogramming*)
 - Multiple programs appear to operate simultaneously
 - The most fundamental role of an operating system

How Multitasking Works

- User(s) start multiple computations running
- The OS switches processor(s) among available computations quickly
- To a human, all computations appear to proceed in parallel

Terminology

- 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)

A Process

- Is an OS abstraction
- Can be created when needed (an OS system call allows a running process to create a new process)
- Is managed entirely by the OS and is unknown to the 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');  
}
```


The Difference Between Function Call And Process Creation

- A normal function call
 - Only involves a single computation
 - Executes synchronously (caller waits until the call returns)
- The *create* system call
 - Starts a new process and returns
 - Both the old process and the new process proceed to run after the call

The Distinction Between A Program And A Process

- A sequential program is
 - Declared explicitly in the code (e.g., with the name *main*)
 - Is executed by a single thread of control
- A process
 - Is an OS abstraction that is not visible in a programming language
 - Is created independent of code that is executed
 - Important idea: multiple processes can execute the same code concurrently
- In the following example, two processes execute function *sndch* concurrently

Example Of Two Processes Running The Same 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 concurrent processes
- Each process requires its own storage for
 - A runtime stack of function calls
 - Local variables
 - Copies of arguments passed to functions
- A process *may* have private heap storage as well

Consequence For Programmers

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

AN OPERATING SYSTEM FROM THE INSIDE

Operating System Properties

- An OS contains well-understood subsystems
- An OS must handle dynamic situations (processes come and go)
- Unlike most applications, an OS uses a heuristic approach
 - A heuristic can have corner cases
 - Policies from one subsystem can conflict with policies from others
- Complexity arises from interactions among subsystems, and the side-effects can be
 - Unintended
 - Unanticipated, even by the OS designer
- 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 of individual pieces
- Structured design is needed
- It can be difficult to understand the consequences of individual choices
- We will study a hierarchical microkernel design that helps control complexity and provides a unifying architecture

Major OS Components

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

Our Multilevel Structure

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

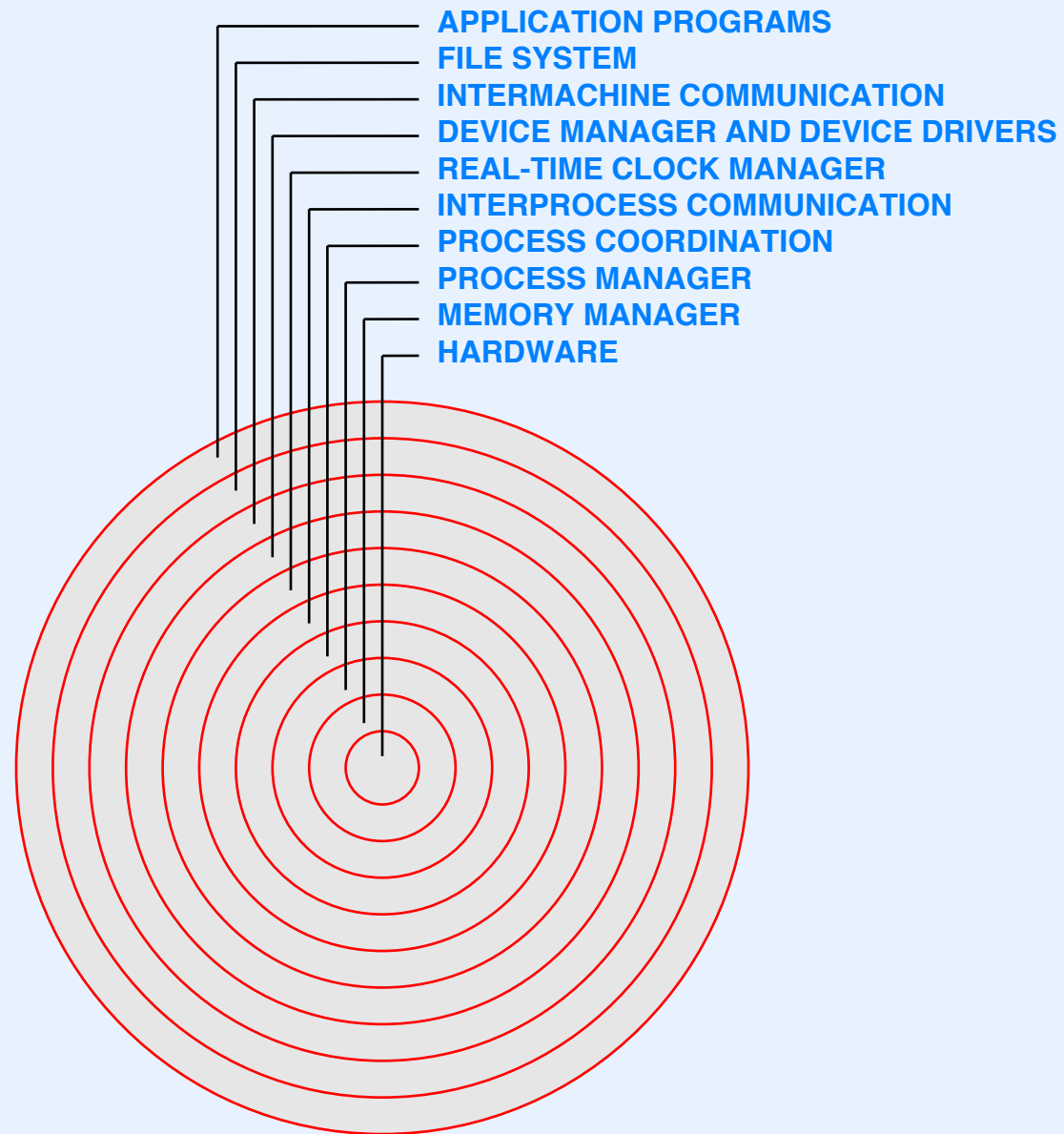
Multilevel Vs. Multilayered Organization

- Multilayer structure
 - Visible to the user as well as designer
 - Software at a given layer only uses software at the layer directly beneath
 - Examples
 - * Internet protocol layering
 - * MULTICS layered security structure
- Can be extremely inefficient

Multilevel Vs. Multilayered Organization (continued)

- Multilevel structure
 - Separates all software into multiple levels
 - Allows software at a given level to use software at *all* lower levels
 - Especially helpful during system construction
 - Focuses a designer's attention on one aspect of the OS at a time
 - Helps keeps policy decisions independent and manageable
 - Is efficient

Multilevel Structure Of Xinu



How To Understand An OS

- Use the same approach as when designing a system
- Work one level at a time
- Understand the service to be provided at the level
- Consider the overall *goal* for the service
- Examine the *policies* that are used to achieve the goal
- Study the *mechanisms* that enforce the policies
- Look at an *implementation* that runs on specific hardware

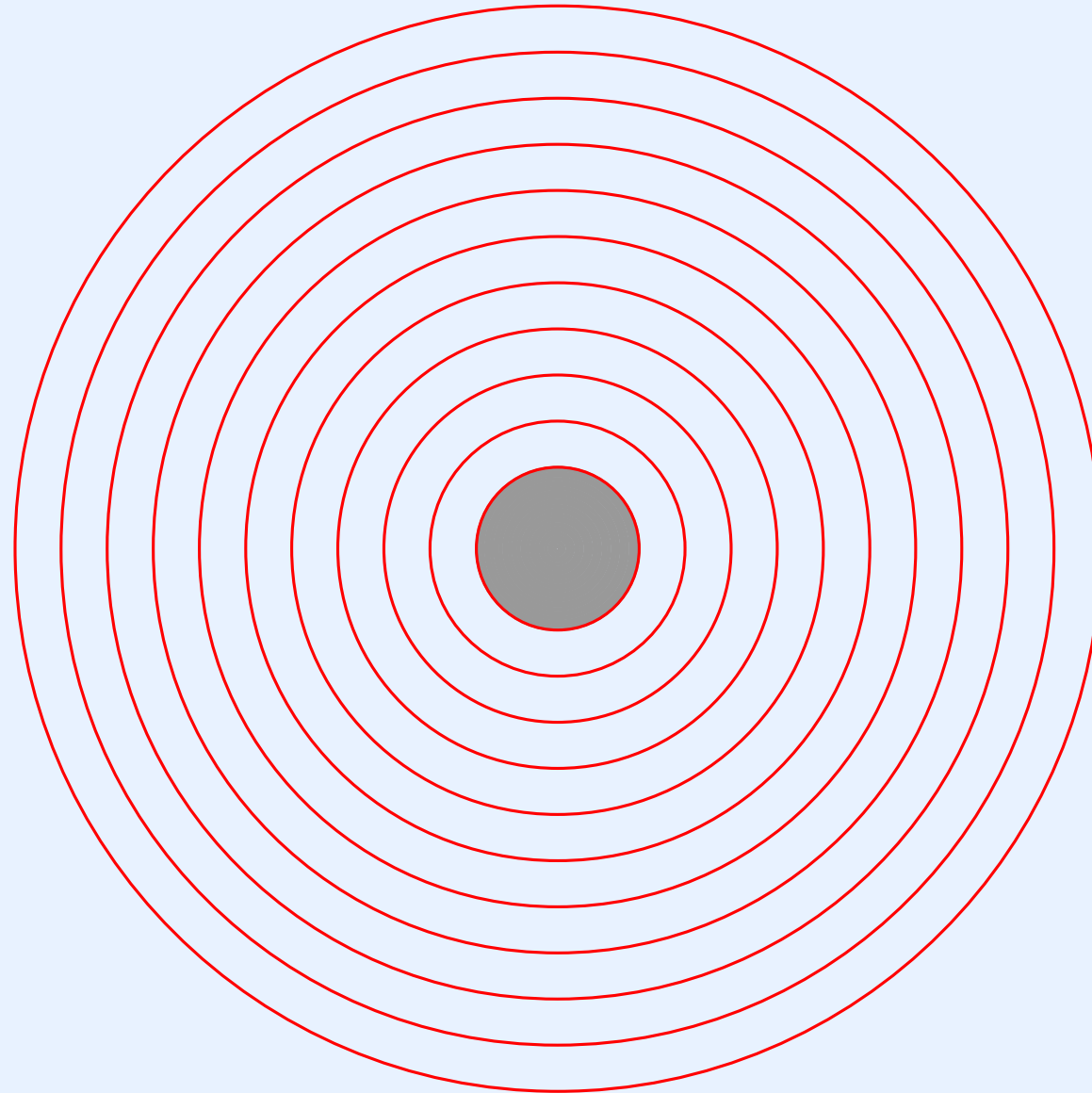
A Design Example

- Example: access to I/O
- Goal: “fairness”
- Policy: First-Come-First-Served access to a given I/O device
- Mechanism: a queue of pending requests (FIFO order)
- Implementation: program written in C

Module II

Quick Review Of Hardware And Runtime Features Process Management: Scheduling And Context Switching

Location Of Hardware In The Hierarchy



Hardware Features An OS Uses Directly

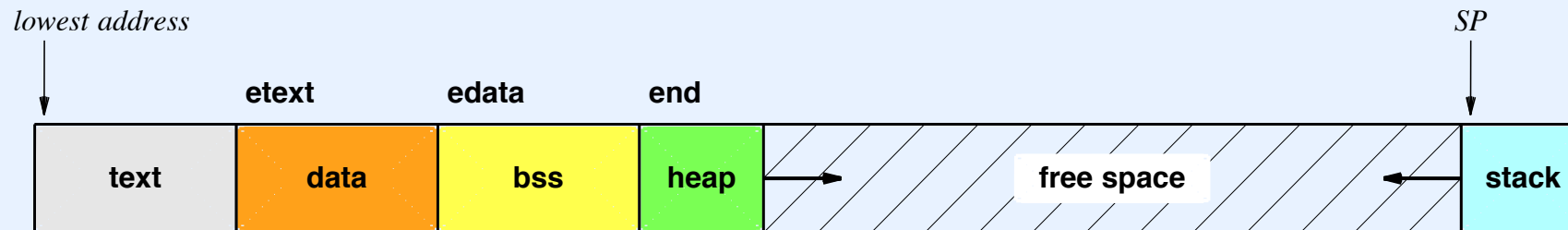
- The processor's *instruction set*
- The *general-purpose registers*
 - Used for computation
 - Saved and restored during subprogram invocation
- The main memory system
 - Consists of an array of bytes
 - Holds code as well as data
 - Imposes endianness for integers
 - May provide address mapping for virtual memory

Hardware Features An OS Uses Directly (continued)

- I/O devices
 - Accessed over a bus
 - Can be *port-mapped* or *memory-mapped* (we will see more later)
- Calling Conventions
 - The set of steps taken during a function call
 - The hardware specifies ways that function calls can operate; a compiler may choose among possible variants

Run-Time Features Pertinent To An OS

- A program is compiled into four segments in memory: text, data, bss, stack



- The stack grows downward (toward lower memory addresses)
- The heap grows upward

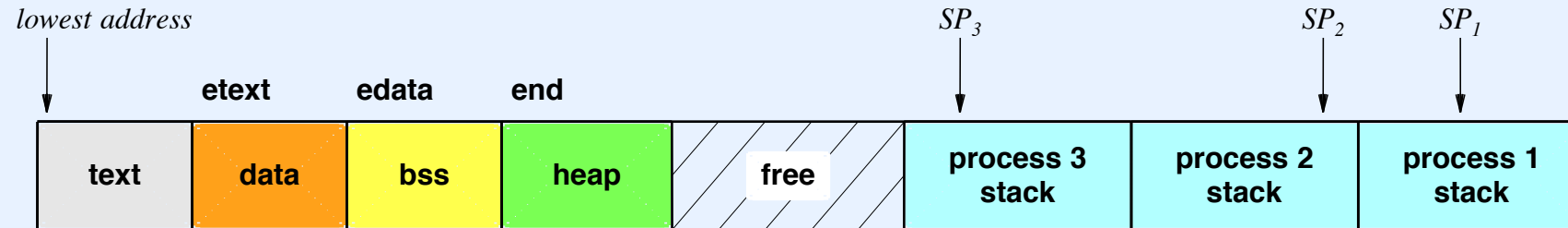
Run-Time Features Pertinent To An OS

(continued)

- A compiler includes global variable names that a program can use to find segment addresses
 - Symbol *etext* lies beyond text segment
 - Symbol *edata* lies beyond data segment
 - Symbol *end* lies beyond bss segment
- A programmer can access the names by declaring them *extern*

```
extern int end;
```
- Only the addresses are significant; the values are irrelevant
- Note: in assembly language, external names have an underscore prepended (e.g., *_end*)

Runtime Storage For Xinu Processes



- All processes share
 - A single text segment
 - A single data segment
 - A single bss segment
- Each process has its own stack segment
 - The stack for a process is allocated when the process is created
 - The stack for a process is released when the process terminates