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 out 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

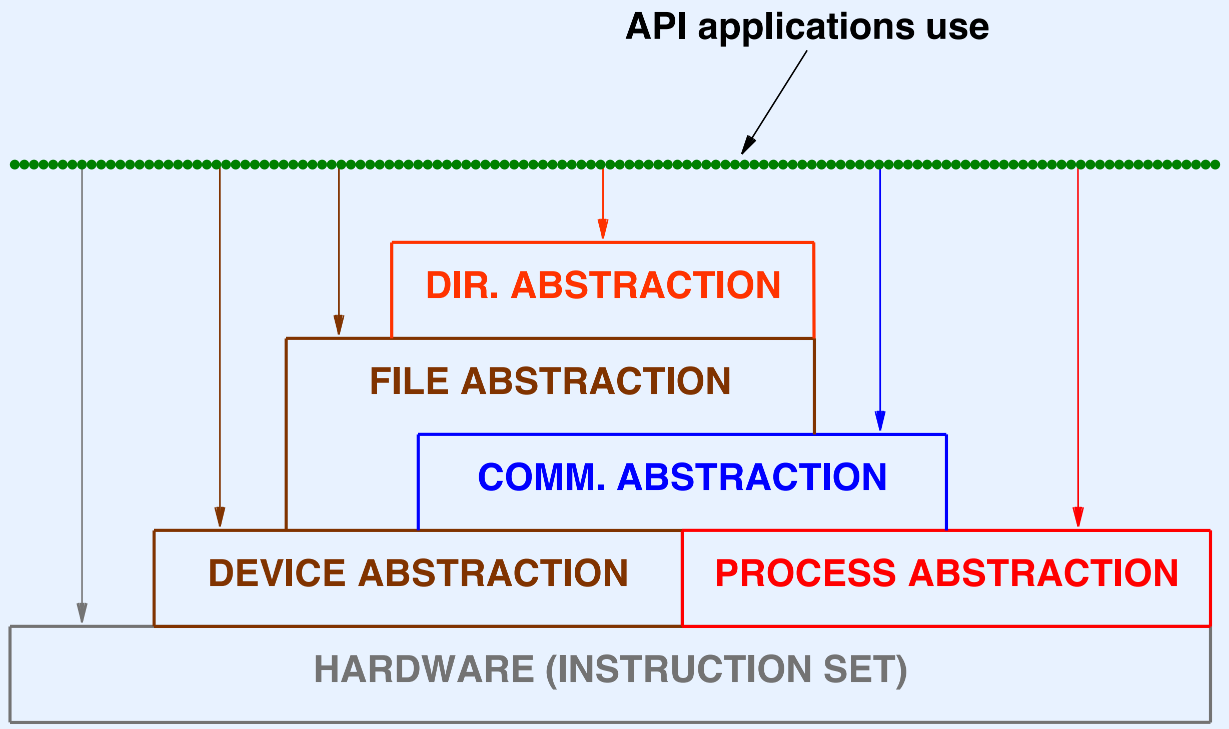
- A hardware mechanism
- A programming language
- A compiler
- A windowing system or a browser
- A command interpreter
- A library of utility functions
- A graphical desktop

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 abstractions 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

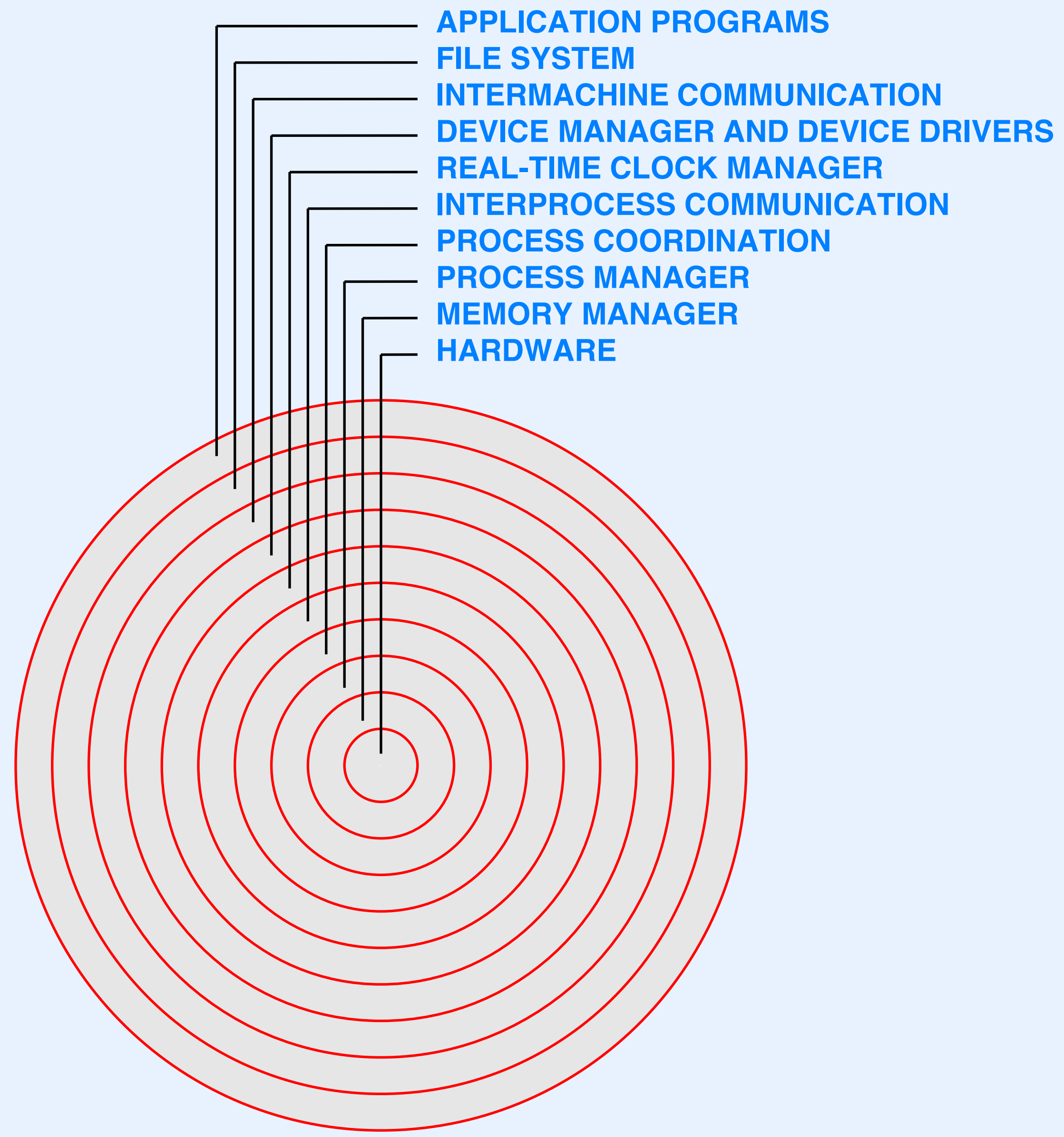
Multilevel Vs. Multilayered Organization

- 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

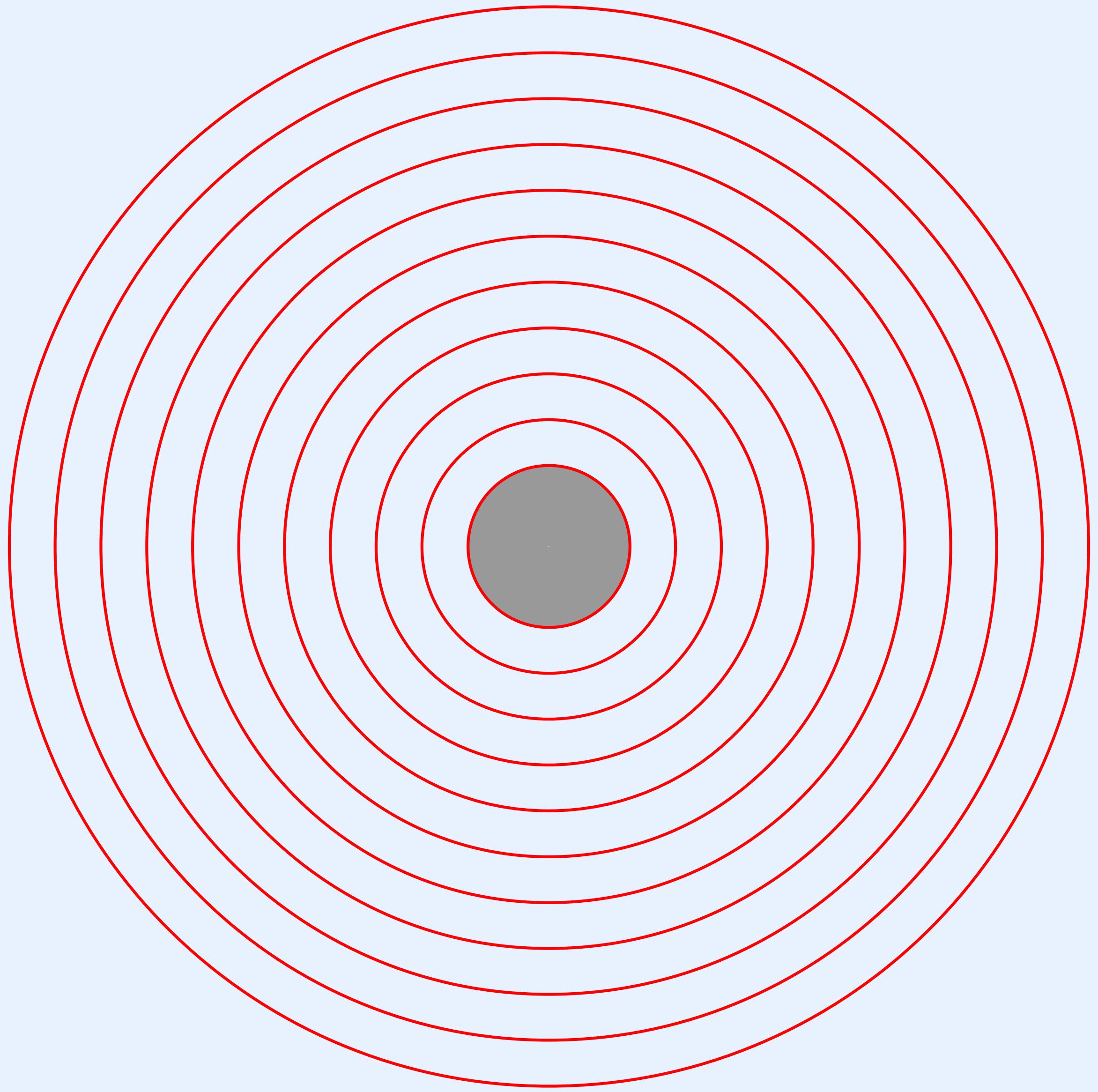
A Design Example

- 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

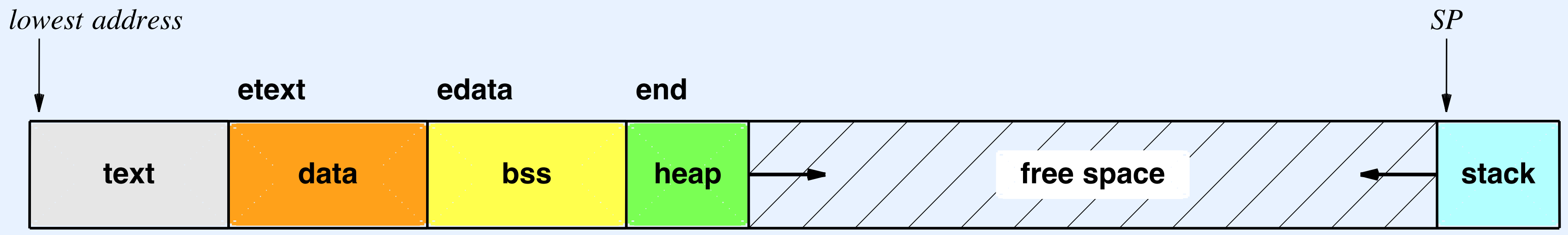
- The processor's *instruction set*
- The *general-purpose registers*
  - Used for computation
  - Saved and restored during subprogram invocation
- The main memory system
  - Consists of an array of bytes
  - Holds code as well as data
  - Imposes endianness for integers
  - May provide address mapping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

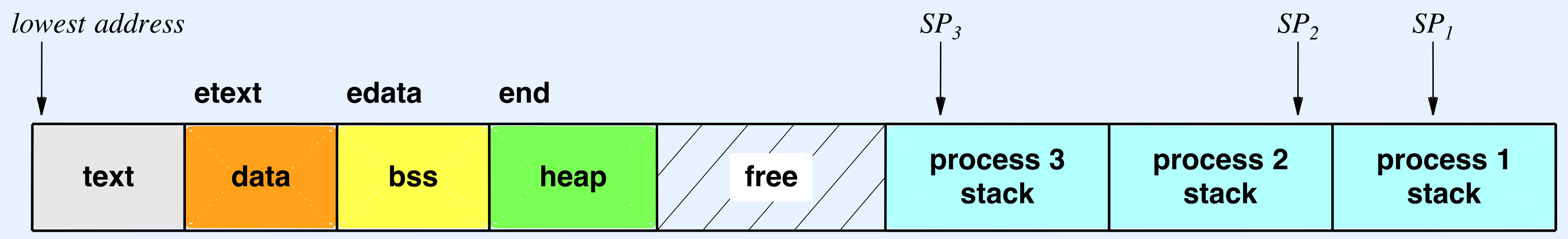
Run-Time Features Pertinent To An OS (continued)

- 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;
```

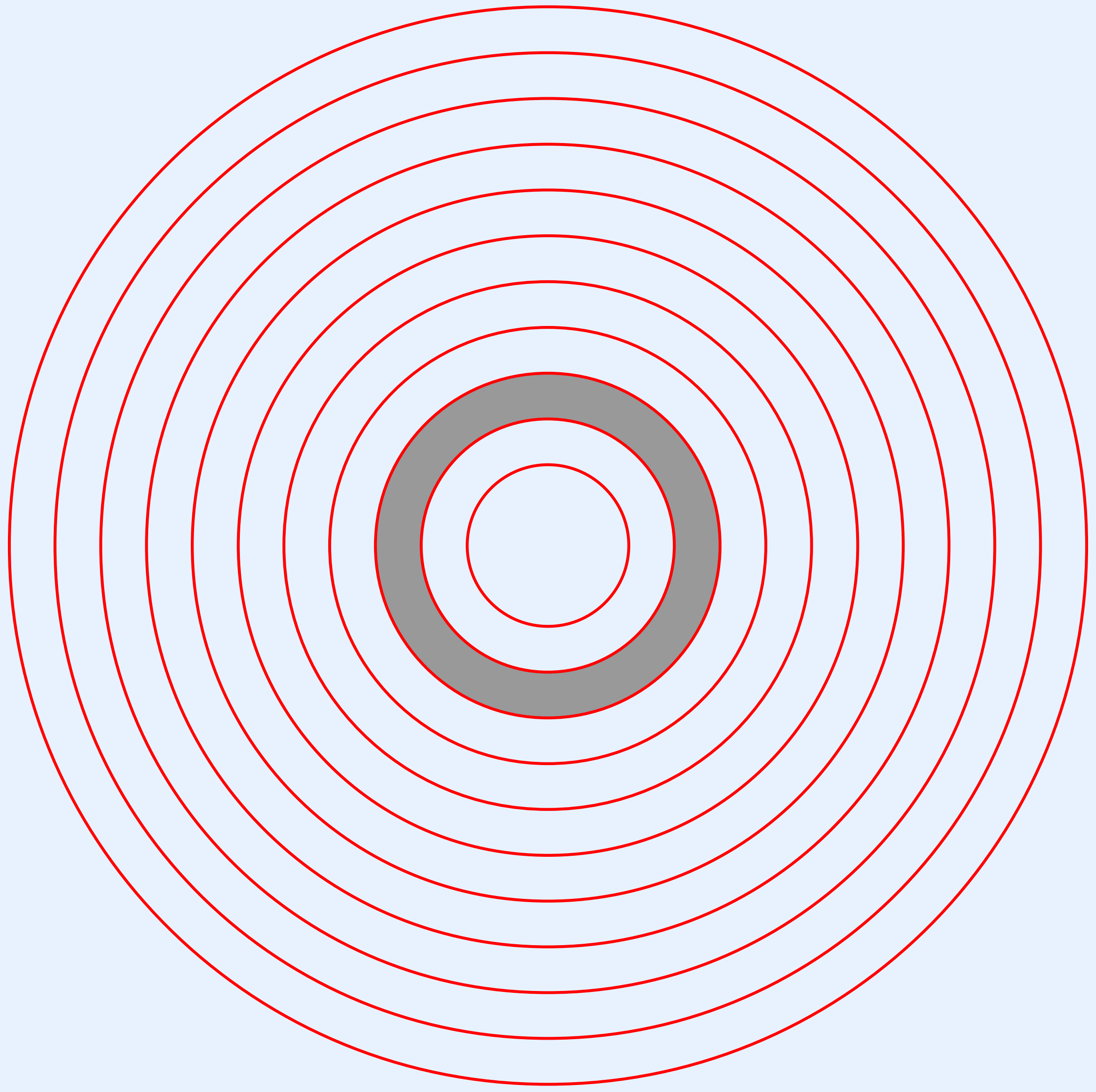
- Only the addresses are significant; the values are irrelevant
- Note: in assembly language, external names have an underscore prepended (e.g., _*end*)

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

Location Of Process Manager In The Hierarchy



Review: What Is A Process?

- An abstraction known only to operating system
- The “running” of a program
- Runs concurrently with other processes

A Fundamental Principle

- All computation must be done by a process
  - No execution can be done by the operating system itself
  - No execution can occur “outside” of a process
- Key consequence
  - At any time, a process *must* be running
  - An operating system cannot stop running a process unless it switches to another process

Process Terminology

- Various terms have been used to denote a process
  - *Job*
  - *Task*
  - *Heavyweight process*
  - *Lightweight process / thread*
- Some of the differences are
  - Address space allocation and variable sharing
  - Longevity
  - Whether the process is declared at compile time or created at run time

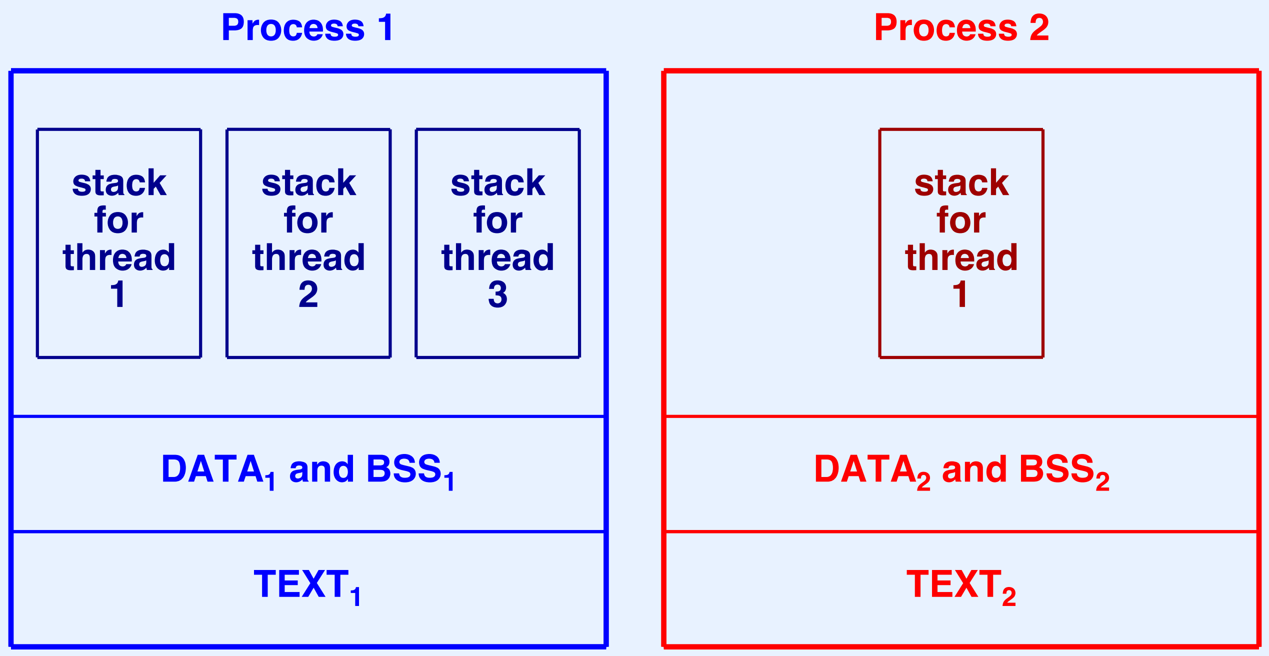
Lightweight Process

- AKA *thread of execution*
- Can share data (data and bss segments) with other threads
- Has a private stack segment for
  - Local variables
  - Function calls

Heavyweight Process

- AKA *Process* with an uppercase “P”
- Pioneered in Mach and adopted by Linux
- A single address space with one or more threads
- One data segment per Process
- One bss segment per Process
- Each thread is bound to a single Process, and cannot move to another

Illustration Of Two Heavyweight Processes And Their Threads



- Threads within a Process share *text*, *data*, and *bss*
- No sharing between Processes
- Threads within a Process cannot share stacks

Our Terminology

- The distinctions among *task*, *thread*, *lightweight process*, and *heavyweight process* are important to some groups
- For this course, we will use the term “process” unless we are specifically talking about the facilities in a specific system, such as Unix/Linux

Maintaining Processes

- Remember that a process is
  - An OS abstraction unknown to hardware
  - Created dynamically
- The pertinent information must be kept by OS
- The OS stores information in a central data structure
- The data structure that hold
  - Is called a *process table*
  - Usually part of OS address space that is not accessible to applications

Information Kept In A Process Table

- For each process
  - A unique *process identifier*
  - The owner (e.g., a user)
  - A scheduling priority
  - The location of the code and all data (including the stack)
  - The status of the computation
  - The current program counter
  - The current values for general-purpose registers

Information Kept In A Process Table
(continued)

- If a heavyweight process contains multiple threads, the process table stores for each thread
  - The owning process
  - The thread’s scheduling priority
  - The location of the thread’s stack
  - The status of the computation
  - The current program counter
  - The current values of registers
- Commercial systems may keep additional information, such as measurements of the process used for accounting

The Xinu Process Model

- Xinu uses the simplest possible scheme
- Xinu is a single-user system, so there is no ownership
- Xinu uses one global context
- Xinu places all code and data in one global address space with
  - No boundary between the OS and applications
  - No protection
- Note: a Xinu process *can* access OS data structures directly, but good programming practice requires applications to use system calls

Example Items In A Xinu Process Table

Field	Purpose
prstate	The current status of the process (e.g., whether the process is currently executing or waiting)
prprio	The scheduling priority of the process
prstkptr	The saved value of the process's stack pointer when the process is not executing
prstkbase	The address of the base of the process's stack
prstklen	A limit on the maximum size that the process's stack can grow
prname	A name assigned to the process that humans use to identify the process's purpose

Process State

- Used by the OS to manage processes
- Is set by the OS whenever process changes status (e.g., waits for I/O)
- Consists of a small integer value stored in the process table
- Is tested by the OS to determine
  - Whether a requested operation is valid
  - The meaning of an operation

The Set Of All Possible Process States

- Must be specified by designer when the OS is created
- One “state” is assigned per activity
- The value in process table is updated when an activity changes
- Example values
  - *Current* (process is currently executing)
  - *Ready* (process is ready to execute)
  - *Waiting* (process is waiting on semaphore)
  - *Receiving* (process is waiting to receive a message)
  - *Sleeping* (process is delayed for specified time)
  - *Suspended* (process is not permitted to execute)

Definition Of Xinu Process State Constants

```
/* Process state constants */

#define PR_FREE         0       /* process table entry is unused        */
#define PR_CURR         1       /* process is currently running         */
#define PR_READY        2       /* process is on ready queue            */
#define PR_RECV         3       /* process waiting for message          */
#define PR_SLEEP        4       /* process is sleeping                  */
#define PR_SUSP         5       /* process is suspended                 */
#define PR_WAIT         6       /* process is on semaphore queue        */
#define PR_RECTIM       7       /* process is receiving with timeout    */
```

- Recall: the possible states are defined as needed when an operating system is constructed
- We will understand the purpose of each state as we consider the system design

Scheduling

Process Scheduling

- A fundamental part of process management
- Is performed by the OS
- Takes three steps
  - Examine processes that are eligible for execution
  - Select a process to run
  - Switch the processor from the currently executing process to the selected process

Implementation Of Scheduling

- An OS designer starts with a *scheduling policy* that specifies which process to select
- The designer then builds a scheduling function that
  - Selects a process according to the policy
  - Updates the process table states for the current and selected processes
  - Calls a *context switch* function to switch the processor from the current process to the selected process

Scheduling Policy

- Determines how processes should be selected for execution
- The goal is usually *fairness*
- The selection may depend on
  - The user’s priority
  - How many processes the user owns
  - The time a given process has been waiting to run
  - The priority of the process
- The policy may be complex
- Note: both hierarchical and flat scheduling have been used

The Example Scheduling Policy In Xinu

- Each process is assigned a *priority*
  - A non-negative integer value
  - Assigned when a process is created
  - Can be changed at any time
- The scheduler always chooses to run an eligible process that has highest priority
- The policy is implemented by a system-wide invariant

The Xinu Scheduling Invariant

At any time, the processor must be executing a highest priority eligible process. Among processes with equal priority, scheduling is round robin.

The Xinu Scheduling Invariant

At any time, the processor must be executing a highest priority eligible process. Among processes with equal priority, scheduling is round robin.

- The invariant must be enforced whenever
  - The set of eligible processes changes
  - The priority of any eligible process changes
- Such changes only happen during a system call or an interrupt (i.e., when running operating system code)

Implementation Of Scheduling

- A process is *eligible* if its state is *ready* or *current*
- To avoid searching the process table during scheduling
  - Keep all ready processes on linked list called a *ready list*
  - Order the ready list by process priority
  - Scheduling is efficient because selection of a highest-priority process can be performed in constant time

Xinu's High-Speed Scheduling Decision

- Compare the priority of the currently executing process to the priority of first process on ready list
  - If the current process has a higher priority, do nothing
  - Otherwise, extract the first process from the ready list and perform a *context switch* to switch the processor to the extracted process

Xinu Scheduler Details

- The scheduler uses an unusual argument paradigm
- Before calling the scheduler
  - Global variable *currpid* gives ID of process that is currently executing
  - *proctab[currpid].prstate* must be set to desired *next* state for the current process
- If current process remains eligible and has highest priority, the scheduler does nothing (i.e., merely returns)
- Otherwise, the scheduler moves the current process to the specified state and runs the highest priority ready process

Round-Robin Scheduling Of Equal-Priority Processes

- When inserting a process on the ready list, insert the process “behind” other processes with the same priority
- If scheduler switches context, the first process on ready list will be selected
- Note: the scheduler should switch context if the first process on the ready list has priority *equal* to the current process
- We will see how the implementation results in switching without a special case in the code

Example Scheduler Code (resched Part 1)

```
/* resched.c - resched */

#include <xinu.h>

struct  defer   Defer;

/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void    resched(void)           /* Assumes interrupts are disabled      */
{
        struct procent *ptold;  /* Ptr to table entry for old process   */
        struct procent *ptnew;  /* Ptr to table entry for new process   */

        /* If rescheduling is deferred, record attempt and return */

        if (Defer.ndefers > 0) {
                Defer.attempt = TRUE;
                return;
        }

        /* Point to process table entry for the current (old) process */

        ptold = &proctab[currpid];
```

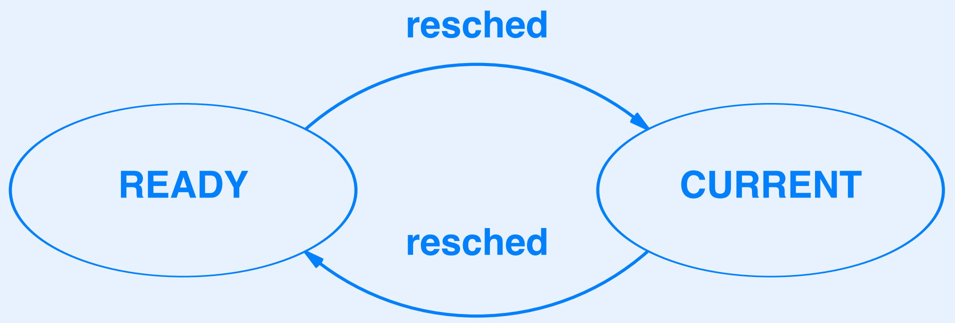
Example Scheduler Code (resched Part 2)

```
        if (ptold->prstate == PR_CURR) {  /* Process remains eligible */
                if (ptold->prprio > firstkey(readylist)) {
                        return;
                }

                /* Old process will no longer remain current */

                ptold->prstate = PR_READY;
                insert(currpid, readylist, ptold->prprio);
        }

        /* Force context switch to highest priority ready process */

        currpid = dequeue(readylist);
        ptnew = &proctab[currpid];
        ptnew->prstate = PR_CURR;
        preempt = QUANTUM;              /* Reset time slice for process */
        ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

        /* Old process returns here when resumed */

        return;
}
```

Process State Transitions

- Recall that each process has a “state”
- The state (*prstate* in the process table) determines
  - Whether an operation is valid
  - The semantics of each operation
- A transition diagram documents valid operations

Illustration Of Transitions Between The Current And Ready States



- Single function (*resched*) moves a process in either direction between the two states

Context Switch

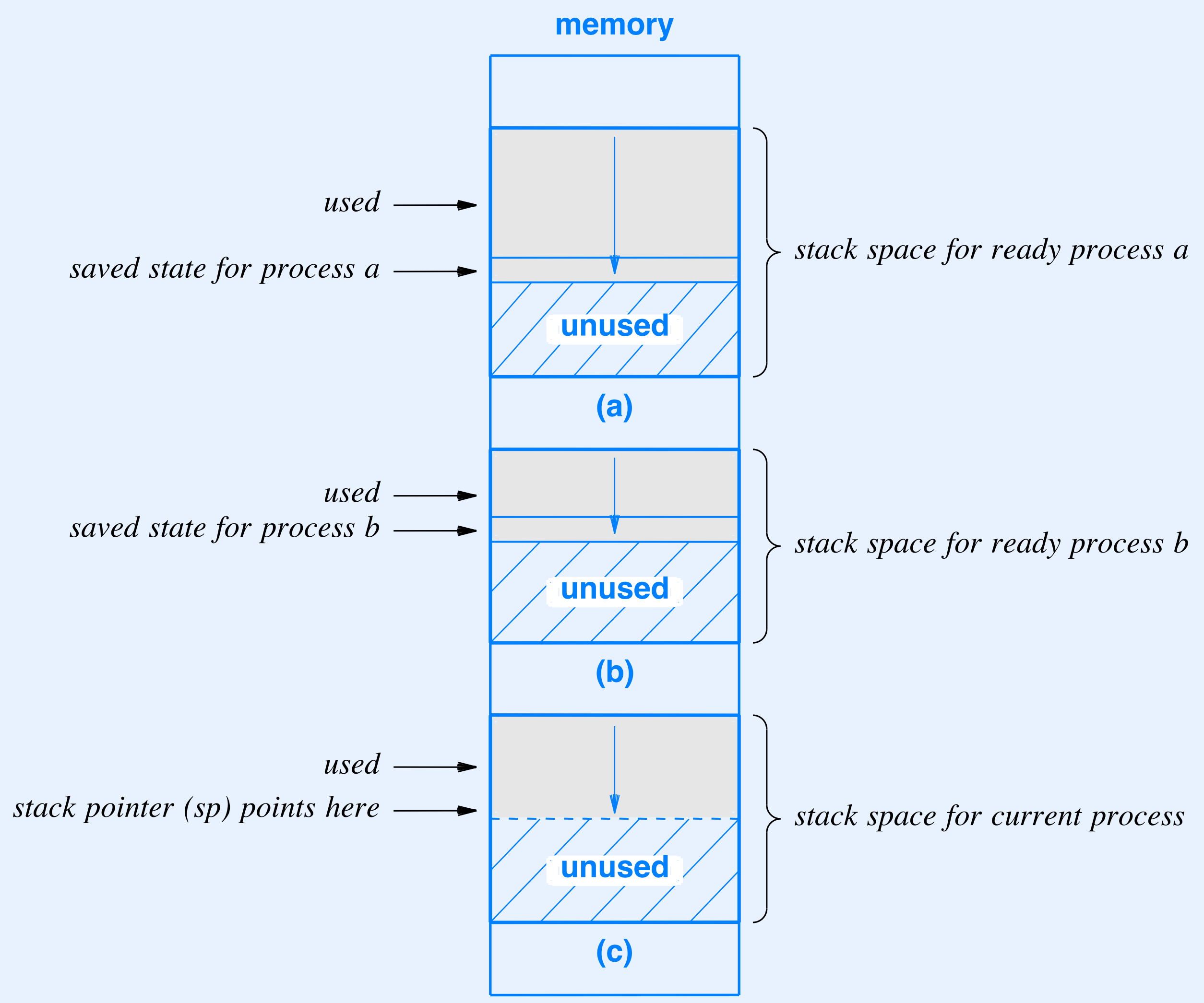
Context Switch

- Forms a basic part of the process manager
- Is low-level (i.e., manipulates the underlying hardware directly)
- Must be written in assembly language
- Is only called by the scheduler
- Actually moves the processor from one process to another

Saving State

- Recall: the processor only has one set of general-purpose registers
- The hardware may contain additional registers associated with a process (e.g., the interrupt mode)
- When switching from one process to another, the operating system must
  - Save a copy of all data associated with the current process
  - Pick up all the previously-saved data associated with the new process
- Xinu uses the process stack to save the state

Illustration Of State Saved On Process Stack


memory
used
saved state for process a
unused
stack space for ready process a
(a)
used
saved state for process b
unused
stack space for ready process b
(b)
used
stack pointer (sp) points here
unused
stack space for current process
(c)

- The stack of each *ready* process contains saved state

Context Switch Operation

- Arguments specify the locations in the process table where the “old” process’s stack and the “new” process’s stack are saved
- Push a copy of all information pertinent to the old process on its stack
  - Contents of hardware registers
  - The program counter (instruction pointer)
  - Hardware privilege level and status
  - The memory map and address space information
- Save the current stack pointer in the process table entry for the old process

...and then

Context Switch Operation
(continued)

- Pick up the stack pointer that was saved in the process table entry for the new process and set the hardware stack pointer (i.e., switch the hardware from the old process's stack to the new process's stack)

- Pop the previously saved information for the new process from its stack and place the values in the hardware registers

- Resume execution at the place where the new process was last executing (i.e., return from the context switch to *resched*)

Example Context Switch Code (Intel Part 1)

```
/* ctxsw.S - ctxsw (for x86) */

                .text
                .globl  ctxsw

/*------------------------------------------------------------------------
 * ctxsw -  X86 context switch; the call is ctxsw(&old_sp, &new_sp)
 *------------------------------------------------------------------------
 */
ctxsw:
                pushl   %ebp                /* Push ebp onto stack          */
                movl    %esp,%ebp           /* Record current SP in ebp     */
                pushfl                      /* Push flags onto the stack    */
                pushal                      /* Push general regs. on stack  */

                /* Save old segment registers here, if multiple allowed */

                movl    8(%ebp),%eax        /* Get mem location in which to */
                                            /*   save the old process's SP  */
                movl    %esp,(%eax)         /* Save old process's SP        */
                movl    12(%ebp),%eax       /* Get location from which to   */
                                            /*   restore new process's SP   */
```

Example Context Switch Code (Intel Part 2)

```
/* The next instruction switches from the old process's */
/*   stack to the new process's stack.                  */

movl    (%eax),%esp     /* Pop up new process's SP      */

/* Restore new seg. registers here, if multiple allowed */

popal                   /* Restore general registers    */
movl    4(%esp),%ebp    /* Pick up ebp before restoring */
                        /*   interrupts                 */
popfl                   /* Restore interrupt mask       */
add     $4,%esp         /* Skip saved value of ebp      */
ret                     /* Return to new process        */
```

The Null Process

- Does not compute anything useful
- Is present merely to ensure that at least one process remains ready at all times
- Simplifies scheduling (i.e., there are no special cases)

Code For The Null Process

- The easiest way to code a null process is an infinite loop:

```
while(1)
    ; /* Do nothing */
```

- A loop may not be optimal because fetch-execute takes bus cycles that compete with I/O devices using the bus
- There are two ways to optimize
  - Some processors offer a special *pause* instruction that stops the processor until an interrupt occurs
  - Other processors have an instruction cache that means fetching the same instructions repeatedly will not access the bus

Summary

- Process management is a fundamental part of an operating system
- Information about processes is kept in process table
- A state variable associated with each process records the process’s activity
  - Currently executing
  - Ready, but not executing
  - Suspended
  - Waiting on a semaphore
  - Receiving a message

Summary
(continued)

- Scheduler
  - Is a key part of the process manager
  - Implements a scheduling policy
  - Chooses the next process to execute
  - Changes information in the process table
  - Calls the context switch to change from one process to another
  - Is usually optimized for high speed

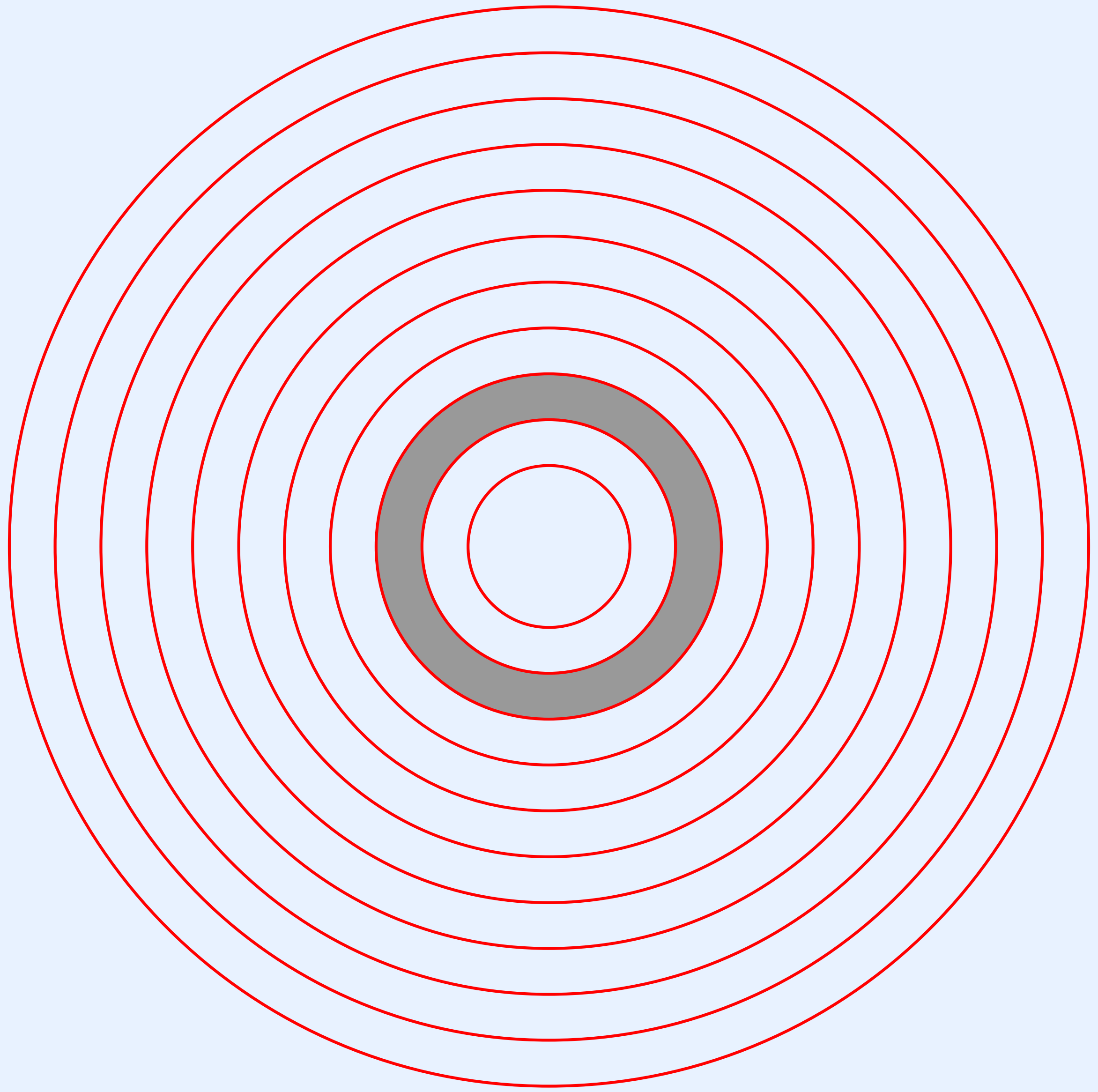
Summary
(continued)

- Context switch
  - Is a low-level part of a process manager
  - Moves the processor from one process to another
  - Involves saving and restoring hardware register contents
- The null process
  - Is needed so the processor has something to run when all user processes block to wait for I/O
  - Consists of an infinite loop
  - Runs at the lowest priority

Questions?

Process Suspension And Resumption

Location Of Process Suspension And Resumption In The Hierarchy



Process Suspension And Resumption

- The idea
  - Temporarily “stop” a process
  - Allow the process to be resumed later
- Questions
  - What happens to the process while it is suspended?
  - Can and process be suspended at any time?
  - What happens if an attempt is made to resume a process that is not suspended?

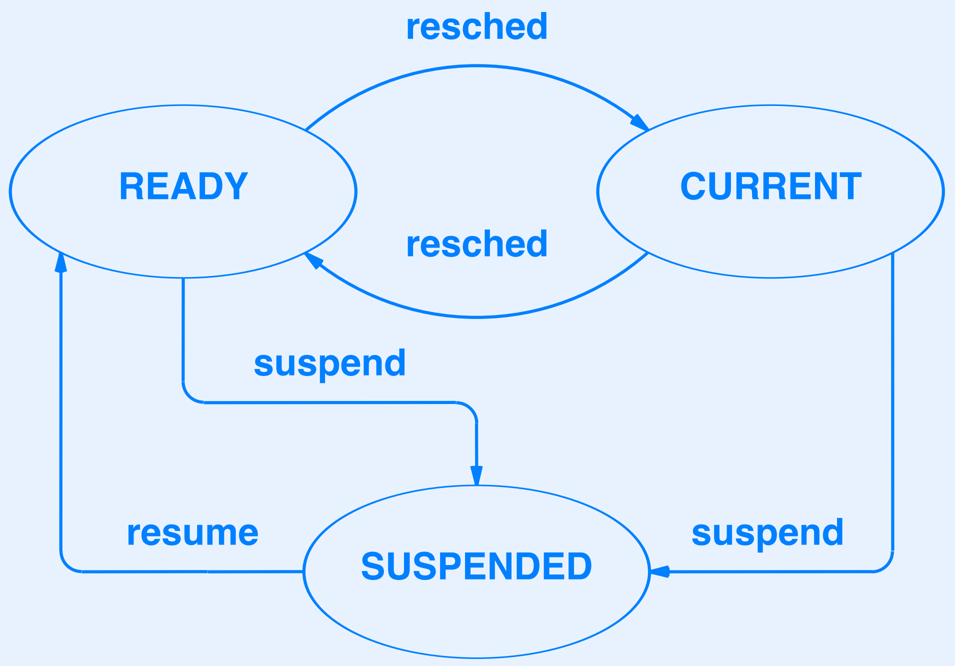
Steps In Suspension And Resumption

- Suspending a process simply means prohibiting the process from using the processor
- When suspending, the operating system must
  - Save pertinent information about the state of the process, such as where it is executing, the contents of general purpose registers, etc.
  - Set the state variable in the process table entry to indicate that the process is suspended
- When resuming, the operating system must
  - Allow the process to use the processor once again
  - Change the state to indicate that process is eligible

A State For Suspended Processes

- A suspended process is not ready, nor is it current
- Therefore, a new process state is needed
- The code uses constant *PR_SUSP* to indicate that a process is in the suspended state

State Transitions For Suspension And Resumption



- As the diagram shows, only a current or ready process can be suspended
- Only a suspended process can be resumed
- System calls *suspend* and *resume* handle the transitions

Suspended Processes

- Where is a process kept when it is suspended?
- Answer:
  - Unlike ready processes, there is no list of suspended processes
  - However, information about a suspended process remains in the process table
  - The process's stack remains allocated in memory

Suspending One's Self

- The currently executing process can suspend itself!
- Self-suspension is straightforward
- The current process
  - Finds its entry in the process table, *proctab[currpid]*
  - Sets the state in its process table entry to *PR_SUSP*, indicating that it should be suspended
  - Calls *resched* to reschedule to another process

A Note About System Calls

- An operating system contains many functions
- OS functions can be divided into two basic categories
  - Some functions are defined to be *system calls*, which means that applications can call them to access services
  - Other functions are merely internal functions used by other operating system functions
- We use the type *syscall* to distinguish system calls
- Note: although Xinu does not prohibit applications from making direct calls to internal operating system functions, good programming practice restricts applications to system calls

Concurrent Execution Of System Calls

- Important concept: multiple processes can attempt to execute a given system call concurrently
- Concurrent execution can result in problems
  - Process A starts to change variables, such as process table entries
  - The OS switches to another process, B
  - When process B examines variables, they are inconsistent

Preventing Concurrent Execution By Disabling Interrupts

- To prevent other processes from changing global data structures, a system call function disables interrupts
- A later section of the course will explain interrupts; for now, it is sufficient to know that a system call must use two functions related to interrupts
  - Function *disable* is called to turn off hardware interrupts, and the function returns a mask value
  - Function *restore* takes as an argument a mask value that was previously obtained from *distable*, and sets the hardware interrupt status according to the specified mask
- Basically, a system call uses *disable* upon being called, and uses *restore* just before it returns
- Note that *restore* must be called before *any* return
- The next slide illustrates the general structure of a system call

A Template For System Calls

```
syscall function_name ( args )   {

          intmask mask;              /* interrupt mask*/

          mask = disable( );         /* disable interrupts at start of function*/

          if  (  args are incorrect  )  {
                    restore(mask); /* restore interrupts before error return*/
                    return(SYSERR);
          }

          . . . other processing . . .

          if  (  an error occurs  )  {
                    restore(mask); /* restore interrupts before error return*/
                    return(SYSERR);
          }

          . . . more processing . . .

          restore(mask);             /* restore interrupts before normal return*/
          return(  appropriate value  );
}
```

The Suspend System Call (Part 1)

```
/* suspend.c - suspend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  suspend  -  Suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
syscall suspend(
          pid32         pid             /* ID of process to suspend     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        pri16   prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)) {
                restore(mask);
                return SYSERR;
        }
```

The Suspend System Call (Part 2)

```
        /* Only suspend a process that is current or ready */

        prptr = &proctab[pid];
        if ((prptr->prstate != PR_CURR) && (prptr->prstate != PR_READY)) {
                restore(mask);
                return SYSERR;
        }
        if (prptr->prstate == PR_READY) {
                getitem(pid);                   /* Remove a ready process   */
                                                /*   from the ready list    */
                prptr->prstate = PR_SUSP;
        } else {
                prptr->prstate = PR_SUSP;       /* Mark the current process */
                resched();                      /*   suspended and resched. */
        }
        prio = prptr->prprio;
        restore(mask);
        return prio;
}
```

Process Resumption

- The idea: resume execution of previously suspended process
- A detail: *resume* returns the priority of the resumed process
- Method
  - Make the process eligible to use the processor again
  - Re-establish scheduling invariant
- Steps
  - Move the suspended process back to the ready list
  - Change the state from *suspended* to *ready*
  - Call *resched*
- Note: resumption does *not* guarantee instantaneous execution of the resumed process

Moving A Process To The Ready List

- We will see that several system calls are needed to make a process ready
- To make it easy, Xinu includes an internal function named *ready* that makes a process ready
- *Ready* takes a process ID as an argument, and makes the process ready
- The steps are
  - Change the process's state to *PR_READY*
  - Insert the process onto the ready list
  - Ensure that the scheduling invariant is enforced

An Internal Function To Make A Process Ready

```
/* ready.c - ready */

#include <xinu.h>

qid16   readylist;                      /* Index of ready list          */

/*------------------------------------------------------------------------
 *  ready  -  Make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
status  ready(
          pid32         pid             /* ID of process to make ready  */
        )
{
        register struct procent *prptr;

        if (isbadpid(pid)) {
                return SYSERR;
        }

        /* Set process state to indicate ready and add to ready list */

        prptr = &proctab[pid];
        prptr->prstate = PR_READY;
        insert(pid, readylist, prptr->prprio);
        resched();

        return OK;
}
```

Enforcing The Scheduling Invariant

- When a process is moved to the ready list, the process becomes eligible to use the processor again
- Recall that when the set of eligible processes changes, the scheduling invariant specifies that we must be check whether a new process should execute
- Consequence: after it moves a process to the ready list, *ready* must re-establish the scheduling invariant
- Surprising, *ready* does not explicitly check the scheduling invariant, but instead simply calls *resched*
- We can now appreciate the design of *resched*: if the newly ready process has a lower priority than the current process, *resched* returns without switching context, and the current process remains running

Example Resumption Code (Part 1)

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
pri16   resume(
          pid32         pid             /* ID of process to unsuspend   */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process’ table entry  */
        pri16   prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return (pri16)SYSERR;
        }
```

Example Resumption Code (Part 2)

```
        prptr = &proctab[pid];
        if (prptr->prstate != PR_SUSP) {
                restore(mask);
                return (pri16)SYSERR;
        }
        prio = prptr->prprio;           /* Record priority to return    */
        ready(pid);
        restore(mask);
        return prio;
}
```

- Consider the code for *resume* and *ready*
- By calling *ready*, *resume* does not need code to insert a process on the ready list, and by calling *resched*, *ready* does not need code to re-establish the scheduling invariant
- The point: choosing OS functions carefully means software at successive levels will be small and elegant

Keeping Processes On A List

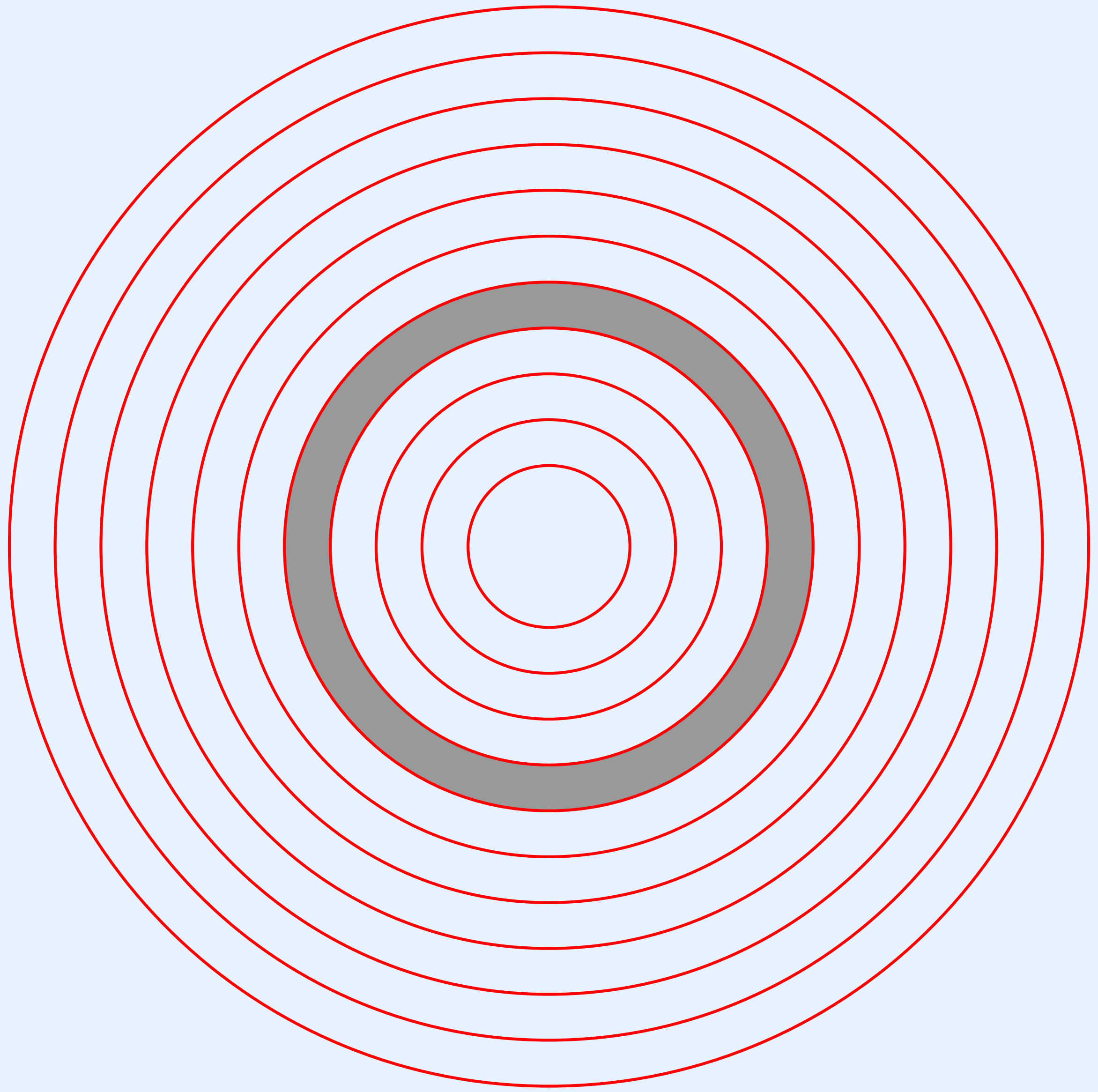
- We have seen that suspended processes are not placed on any list
- Why not?
  - Function *resume* requires the caller to supply an argument that specifies the ID of the process to be resumed
  - We will see that no other operating system functions operate on suspended processes or handle the entire set of suspended processes
- Consequence: there is no reason to keep a list of suspended processes
- In general: an operating system only places a process on a list if a function needs to handle an entire set of processes that are in a given state (e.g., like the set of all ready processes that are handled by the scheduler)

Summary Of Process Suspension And Resumption

- An OS offers functions that can change a process's state
- Xinu allows a process to be
  - Suspended temporarily
  - Resumed later
- A state variable associated with each process records the process's current status
- When resuming a process, the scheduling invariant must be re-established

Inter-Process Communication (Message Passing)

Location Of Inter-Process Communication In The Hierarchy



Inter-Process Communication

- Can be used for
  - Exchange of (nonshared) data among processes
  - Some forms of process coordination
- The general technique is known as *message passing*

An Example Design For A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- Our example facility will allow a process to send a message directly to another process

An Example Design For A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- Our example facility will allow a process to send a message directly to another process
- In principle, the design should be straightforward

An Example Design For A Message Passing Facility

- To understand the issues, we will begin with a trivial message passing facility
- Our example facility will allow a process to send a message directly to another process
- In principle, the design should be straightforward
- In practice, many design decisions arise

Message Passing Design Decisions

- Are messages fixed size or variable size?
- What is the maximum message size?
- How many messages can be outstanding at a given time?
- Where are messages stored?
- How is a recipient specified?
- Does a receiver know the sender’s identity?
- Are replies supported?
- Is the interface synchronous or asynchronous?

Synchronous vs. Asynchronous Interface

- A synchronous interface
  - An operation blocks until the operation is performed
  - A sending process is blocked until the recipient accepts the message being sent
  - A receiving process is blocked until a message arrives
  - Is easy to understand and use
  - A programmer can create extra processes to obtain asynchrony

Synchronous vs. Asynchronous Interface (continued)

- An asynchronous interface
  - A process starts an operation
  - The initiating process continues execution
  - A notification arrives when the operation completes
    - The notification can arrive at any time
    - Typically, notification entails abnormal control flow (e.g., “callback” mechanism)
  - Is more difficult to understand and use
  - Polling can be used to determine the status

An Example Message Passing Facility

- We will examine a basic, low-level mechanism
- The facility provides direct process-to-process communication
- Each message is one word (e.g., an integer)
- A message is stored with the receiving process
- A process only has a one-message buffer
- Message reception is synchronous and buffered
- Message transmission is asynchronous
- The facility includes a “reset” operation

An Example Message Passing Facility
(continued)

- The interface consists of three system calls

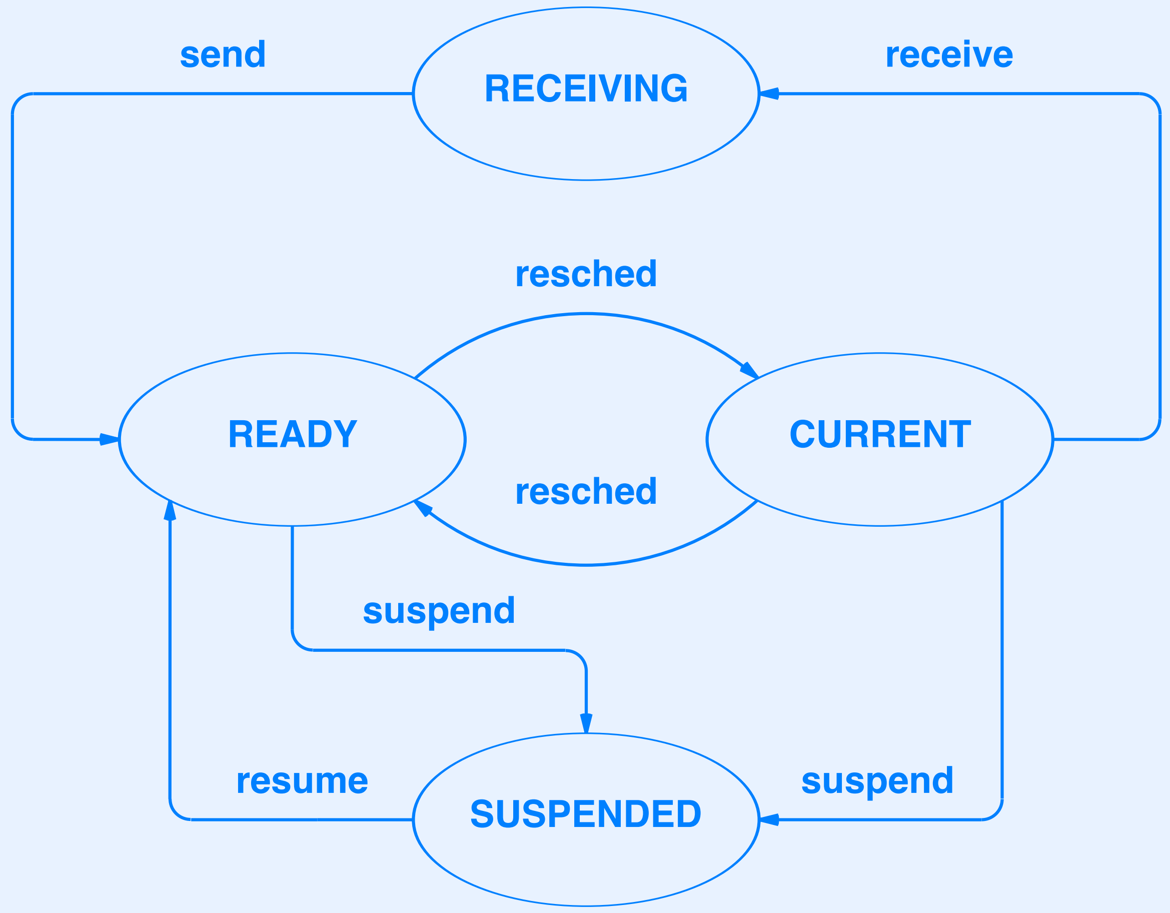
```
send(pid, msg);

msg = receive();

msg = recvclr();
```

- *Send* transmits a message to a specified process
- *Receive* blocks until a message arrives
- *Recvclr* removes an existing message, if one has arrived, but does not block
- A message is stored in the *receiver's* process table entry

A Process State For Message Reception

- While receiving a message, a process is not
  - Executing
  - Ready
  - Suspended
- Therefore, a new state is needed for message passing
- The state is named *RECEIVING*
- The state is entered when *receive* called
- The code uses constant *PR_RECV* to denote a *receiving* state

State Transitions With Message Passing



The Steps Taken To Receive A Message

- The current process calls *receive* which checks its own process table entry
- If no message has arrived, *receive* blocks the calling process to wait until a message to arrive
- Once this step has been reached, a message is present
- *Receive* extracts a copy of the message from the process table entry and resets the process table entry to indicate that no message is present
- *Receive* returns the message to its caller

Blocking To Wait For A Message

- We have seen how a process suspends itself
- Blocking to receive a message is almost the same
  - Find the current process's entry in the process table, *proctab[currpid]*
  - Set the state in the process table entry to *PR_RECV*, indicating that the process will be receiving
  - Call *resched*

Xinu Code For Message Reception

```
/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  receive  -  Wait for a message and return the message to the caller
 *------------------------------------------------------------------------
 */
umsg32  receive(void)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        umsg32  msg;                    /* Message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) {
                prptr->prstate = PR_RECV;
                resched();              /* Block until message arrives  */
        }
        msg = prptr->prmsg;             /* Retrieve message             */
        prptr->prhasmsg = FALSE;        /* Reset message flag           */
        restore(mask);
        return msg;
}
```

Message Transmission

- To send a message, a process calls *send* specifying a destination process and a message to send to the process
- The code
  - Checks arguments
  - Returns an error if the process already has a message waiting
  - Deposits the message
  - Makes the process ready if it is in the receiving state
- Note: the code also handles a receive-with-timeout state, but we will consider that state later

Xinu Code For Message Transmission (Part 1)

```
/* send.c - send */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  send  -  Pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall send(
          pid32         pid,              /* ID of recipient process      */
          umsg32        msg               /* Contents of message          */
        )
{
        intmask mask;                     /* Saved interrupt mask         */
        struct  procent *prptr;           /* Ptr to process' table entry  */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        prptr = &proctab[pid];
        if ((prptr->prstate == PR_FREE) || prptr->prhasmsg) {
                restore(mask);
                return SYSERR;
        }
```

Xinu Code For Message Transmission (Part 2)

```
        prptr->prmsg = msg;                /* Deliver message              */
        prptr->prhasmsg = TRUE;            /* Indicate message is waiting  */

        /* If recipient waiting or in timed-wait make it ready */

        if (prptr->prstate == PR_RECV) {
                ready(pid);
        } else if (prptr->prstate == PR_RECTIM) {
                unsleep(pid);
                ready(pid);
        }
        restore(mask);             /* Restore interrupts */
        return OK;
}
```

Xinu Code For Clearing Messages

```
/* recvclr.c - recvclr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvclr  -  Clear incoming message, and return message if one waiting
 *------------------------------------------------------------------------
 */
umsg32  recvclr(void)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        umsg32  msg;                    /* Message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == TRUE) {
                msg = prptr->prmsg;     /* Retrieve message             */
                prptr->prhasmsg = FALSE;/* Reset message flag           */
        } else {
                msg = OK;
        }
        restore(mask);
        return msg;
}
```

Summary Of Message Passing

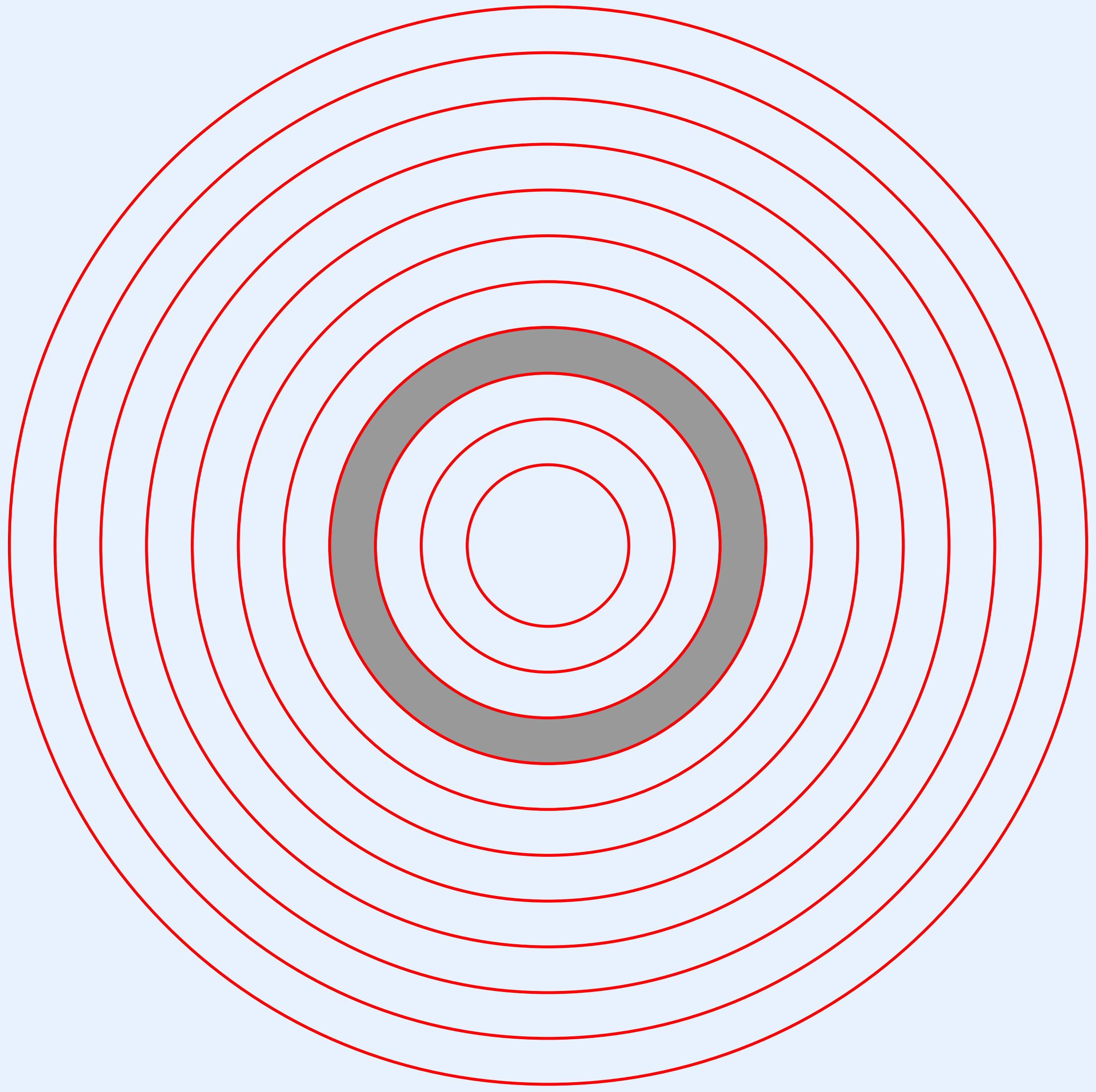
- Message passing offers an inter-process communication system
- The interface can be synchronous or asynchronous
- A synchronous interface is the easiest to use
- Xinu uses synchronous reception and asynchronous transmission
- An asynchronous operation allows a process to clear any existing message without blocking
- The Xinu message passing system only allows one outstanding message per process, and uses first-message semantics

Questions?

Module IV

Process Management: Coordination And Synchronization

Location Of Process Coordination In The Hierarchy



Coordination Of Processes

- Is necessary in a concurrent system
- Avoids conflicts when multiple processes access shared items
- Allows a set of processes to cooperate
- Can also be used when
  - A process waits for I/O
  - A process waits for another process
- An example of cooperation among processes: UNIX pipes

Two Approaches To Process Coordination

- Use a hardware mechanism
  - Most useful/important on multiprocessor hardware
  - Often relies on *busy waiting*
- Use an operating system mechanism
  - Works well with single processor hardware
  - Does not entail unnecessary execution

Note: we will mention hardware quickly, and focus on operating system mechanisms

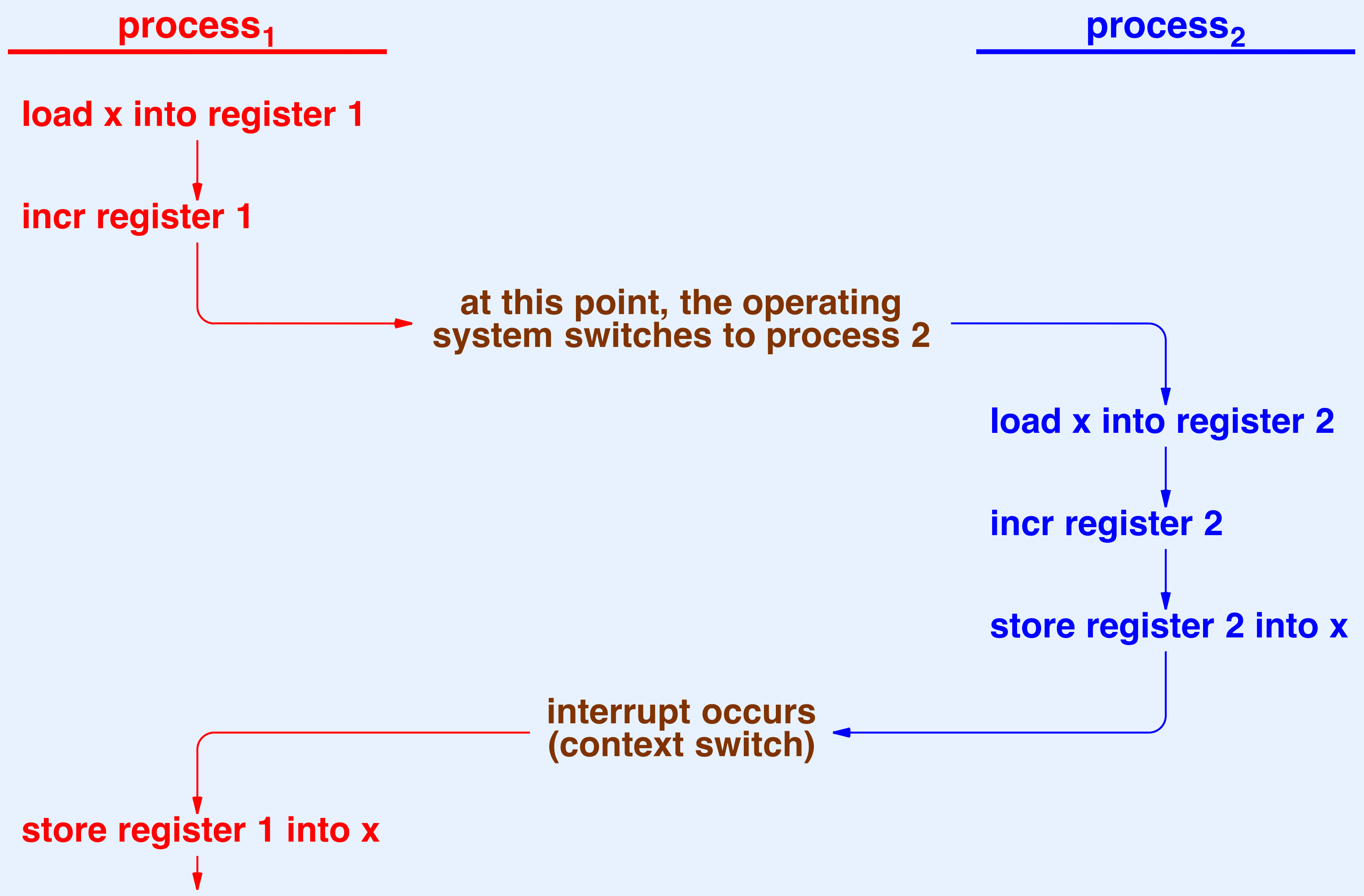
Two Key Situations That Process Coordination Mechanisms Handle

- Producer / consumer interaction
- Mutual exclusion

Mutual Exclusion

- In a concurrent system, multiple processes may attempt to access shared data items
- If one process starts to change a data item and then a context switch allows another process to run and access the data item, the results can be incorrect
- We use the term *atomic* to refer to an operation that is indivisible (i.e., the hardware performs the operation in a single instruction that cannot be interrupted)
- Programmers must take steps to ensure that when a sequence of non-atomic operations is used to change a data item, the sequence is not executed concurrently
- Even trivial operations can be non-atomic!

Illustration Of What Can Happen When Two Processes Attempt To Increment Integer **x** Concurrently



To Prevent Problems

- A programmer must ensure that only one process accesses a shared item at any time
- General approach
  - Once a process obtains access, make all other processes wait
  - When a process finishes accessing the item, grant access to one of the waiting processes
- Three techniques are available
  - Hardware spin lock instructions
  - Hardware mechanisms that disable and restore interrupts
  - Semaphores (implemented in software)

Handling Mutual Exclusion With Spin Locks

- Used in multicore CPUs; does *not* work for a single processor
- An atomic hardware operation allows a core to test a memory location and change it
- The hardware guarantees that only one core will be allowed to make the change
- It is called a *spin lock* mechanism because a core uses *busy waiting* to gain access
- Busy waiting literally means the core executes a loop that tests the spin lock repeatedly until access is granted
- The approach is also known as *test-and-set*

Handling Mutual Exclusion With Semaphores

- A programmer must allocate a semaphore for each item to be protected
- The semaphore acts as a *mutual exclusion* semaphore, and is known colloquially as a *mutex* semaphore
- All applications must be programmed to use the mutex semaphore before accessing the shared item
- The operating system guarantees that only one process can access the shared item at a given time
- The implementation avoids busy waiting

Definition Of Critical Section

- Each piece of shared data must be protected from concurrent access
- A programmer inserts mutex operations
  - Before access to the shared item
  - After access to the shared item
- The protected code is known as a *critical section*
- Mutex operations must be placed in each function that accesses the shared item

Low-Level Mutual Exclusion

- Mutual exclusion is needed in two places
  – In application processes
  – Inside operating system
- On a single-processor system, mutual exclusion can be guaranteed provided that no context switching occurs
- A context switch can only occur when
  – A device interrupts
  – A process calls *resched*
- Low-level mutual exclusion technique: turn off interrupts and avoid rescheduling

Why Interrupt Masking Is Insufficient

- It works! But...
- Stopping interrupts penalizes *all* processes when one process executes a critical section
  - It stops all I/O activity
  - It restricts execution to one process for the entire system
- Disabling interrupts can interfere with the scheduling invariant (e.g., a low-priority process can block a high-priority process for which I/O has completed)
- Disabling interrupts does not provide a policy that controls which process can access a critical section at a given time

High-Level Mutual Exclusion

- The idea is to create an operating system facility with the following properties
  - Permit applications to define multiple, independent critical sections
  - Allow processes to access each critical section independent of other sections
  - Provide an access policy that specifies how waiting processes gain access (e.g., FIFO)
- Good news: a single mechanism, the *counting semaphore*, suffices

Counting Semaphore

- An operating system abstraction
- An instance can be created dynamically
- Each instance is given a unique name
  - Typically an integer
  - Known as a *semaphore ID*
- An instance consists of a 2-tuple (count, set)
  - *Count* is an integer
  - *Set* is a set of processes that are waiting on the semaphore

Operations On Semaphores

- *Create* a new semaphore
- *Delete* an existing semaphore
- *Wait* on an existing semaphore
  - Decrements the count
  - Adds the calling process to set of waiting processes if the resulting count is negative
- *Signal* an existing semaphore
  - Increments the count
  - Makes a process ready if any are waiting

Xinu Semaphore Functions

semid = semcreate(initial_count)	Creates a semaphore and returns an ID
semdelete(semid)	Deletes the specified semaphore
wait(semid)	Waits on the specified semaphore
signal(semid)	signals the specified semaphore

Key Uses Of Counting Semaphores

- Semaphores have many potential uses
- However, using semaphores to solve complex coordination problems can be intellectually challenging
- We will consider two straightforward ways to use semaphores
  - Cooperative mutual exclusion
  - Producer-consumer synchronization (direct synchronization)

Cooperative Mutual Exclusion With Semaphores

- A set of processes use a semaphore to guard a shared item
- Initialize: create a mutex semaphore

```
sid = semcreate(1);
```

- Use: bracket each critical section in the code with calls to *wait* and *signal*

```
 wait(sid);
...critical section to use the shared item...
signal(sid);
```

- All processes must agree to use semaphores (hence the term *cooperative*)
- Only one process will access the critical section at any time (others will be blocked)

Producer-Consumer Synchronization With Semaphores

- Two semaphores suffice to control processes accessing a shared buffer
- Initialize: create producer and consumer semaphores

```
psem = semcreate(buffer-size);
csem = semcreate(0);
```

- The producer algorithm

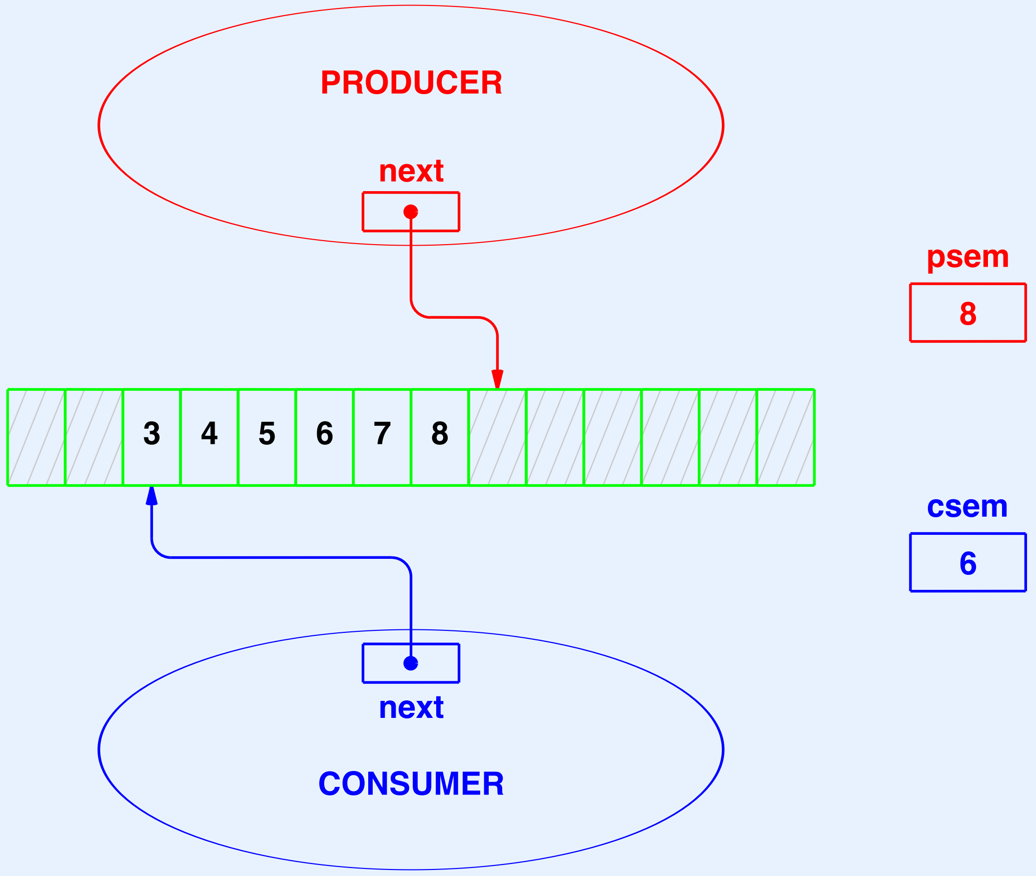
```
repeat forever {
        generate an item to be added to the buffer;
        wait(psem);
        fill_next_buffer_slot;
        signal(csem);
}
```

Producer-Consumer Synchronization With Semaphores (continued)

- The consumer algorithm

```
repeat forever {
        wait(csem);
        extract_from_buffer_slot;
        handle the item;
        signal(psem);
}
```

An Interpretation Of Producer-Consumer Semaphores



- *csem* counts the items currently in the buffer
- *psem* counts the unused slots in the buffer

The Semaphore Invariant

- Establishes a relationship between the semaphore concept and its implementation
- Makes the code easy to create and understand
- Must be re-established after each semaphore operation
- Is surprisingly elegant:

A nonnegative semaphore count means that the set of processes is empty. A count of negative *N* means that the set contains *N* waiting processes.

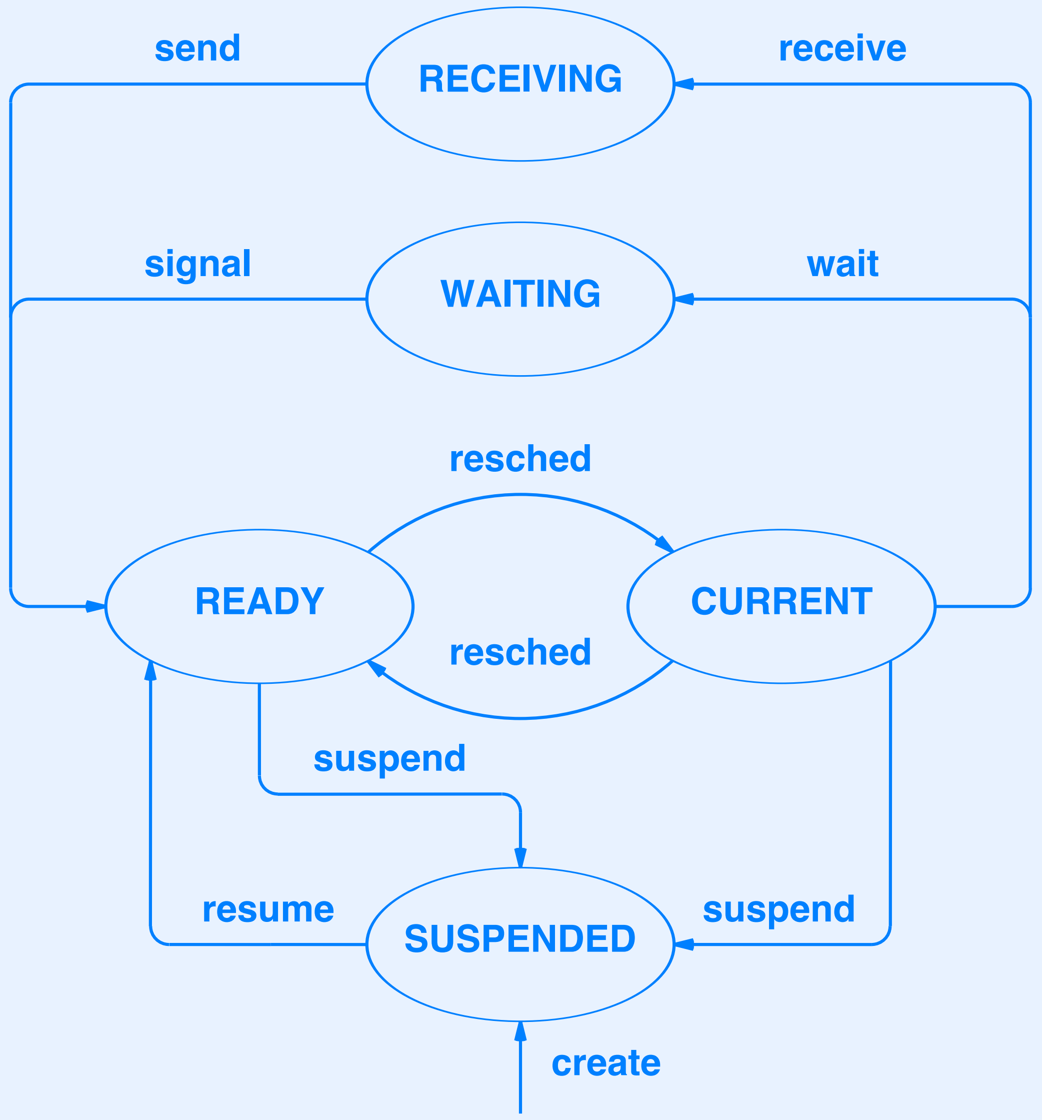
Counting Semaphores In Xinu

- Are stored in an array of semaphore entries
- Each entry
  - Corresponds to one instance (one semaphore)
  - Contains an integer count and pointer to a list of processes
- The ID of a semaphore is its index in the array
- The policy for management of waiting processes is FIFO

A Process State Used With Semaphores

- When a process is waiting on a semaphore, the process is not
  - Executing
  - Ready
  - Suspended
  - Receiving
- Note: the suspended state is only used by *suspend* and *resume*
- Therefore a new state is needed
- We will use the *WAITING* state for a process blocked by a semaphore

State Transitions With Waiting State



Semaphore Definitions

```
/* semaphore.h - isbadsem */

#ifndef NSEM
#define NSEM            120     /* Number of semaphores, if not defined */
#endif

/* Semaphore state definitions */

#define S_FREE  0               /* Semaphore table entry is available   */
#define S_USED  1               /* Semaphore table entry is in use      */

/* Semaphore table entry */
struct  sentry  {
        byte    sstate;         /* Whether entry is S_FREE or S_USED    */
        int32   scount;         /* Count for the semaphore              */
        qid16   squeue;         /* Queue of processes that are waiting  */
                                /*     on the semaphore                 */
};

extern  struct  sentry semtab[];

#define isbadsem(s)     ((int32)(s) < 0 || (s) >= NSEM)
```

Implementation Of Wait (Part 1)

```
/* wait.c - wait */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wait  -  Cause current process to wait on a semaphore
 *------------------------------------------------------------------------
 */
syscall wait(
          sid32         sem             /* Semaphore on which to wait  */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        struct  sentry *semptr;         /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
```

Implementation Of Wait (Part 2)

```
        if (--(semptr->scount) < 0) {           /* If caller must block */
                prptr = &proctab[currpid];
                prptr->prstate = PR_WAIT;       /* Set state to waiting */
                prptr->prsem = sem;             /* Record semaphore ID  */
                enqueue(currpid,semptr->squeue);/* Enqueue on semaphore */
                resched();                      /*   and reschedule     */
        }

        restore(mask);
        return OK;
}
```

- Moving a process to the waiting state only requires a few lines of code
  - Set the state of the current process to PR_WAIT
  - Record the ID of the semaphore on which the process is waiting in field *prsem*
  - Call *resched*

The Semaphore Queuing Policy

- Determines which process to select among those that are waiting
- Is only used when *signal* is called and processes are waiting
- Examples of possible policies
  - First-Come-First-Served (FCFS or FIFO)
  - Process priority
  - Random

Semaphore Policy Consequences

- The goal is “fairness”
- Which semaphore queuing policy implements the goal the best?
- In other words, how should we interpret fairness?
- The semaphore policy can interact with scheduling policy
  - Should a low-priority process be allowed to access a resource if a high-priority process is also waiting?
  - Should a low-priority process be blocked forever if high-priority processes use a resource?

The Semaphore Queuing Policy In Xinu

- First-come-first-serve
- Has several advantages
  - Is straightforward to implement
  - Is extremely efficient
  - Works well for traditional uses of semaphores
  - Guarantees all contending processes will obtain access
- Has an interesting consequence: a low-priority process can access a resource while a high-priority process remains blocked

Implementation Of A FIFO Semaphore Policy

- Each semaphore uses a list to manage waiting processes
- As we have seen Xinu manages the list of processes as a queue
  - Wait enqueues a process at one end of the queue
  - Signal chooses a process at the other end of the queue
- The code for signal follows

Implementation Of Signal (Part 1)

```
/* signal.c - signal */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  signal  -  Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
 */
syscall signal(
          sid32          sem              /* ID of semaphore to signal    */
        )
{
        intmask mask;                     /* Saved interrupt mask         */
        struct  sentry *semptr;           /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }
        semptr= &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
```

Implementation Of Signal (Part 2)

```
        if ((semptr->scount++) < 0) {   /* Release a waiting process */
                ready(dequeue(semptr->squeue));
        }
        restore(mask);
        return OK;
}
```

- Notice how little code is required to signal a semaphore

Do you understand semaphores?

Summary

- Process synchronization is used in two ways
  - As a service supplied to applications
  - As an internal facility used inside the OS itself
- Low-level mutual exclusion
  - Masks hardware interrupts
  - Avoids rescheduling
  - Is insufficient for all coordination needs

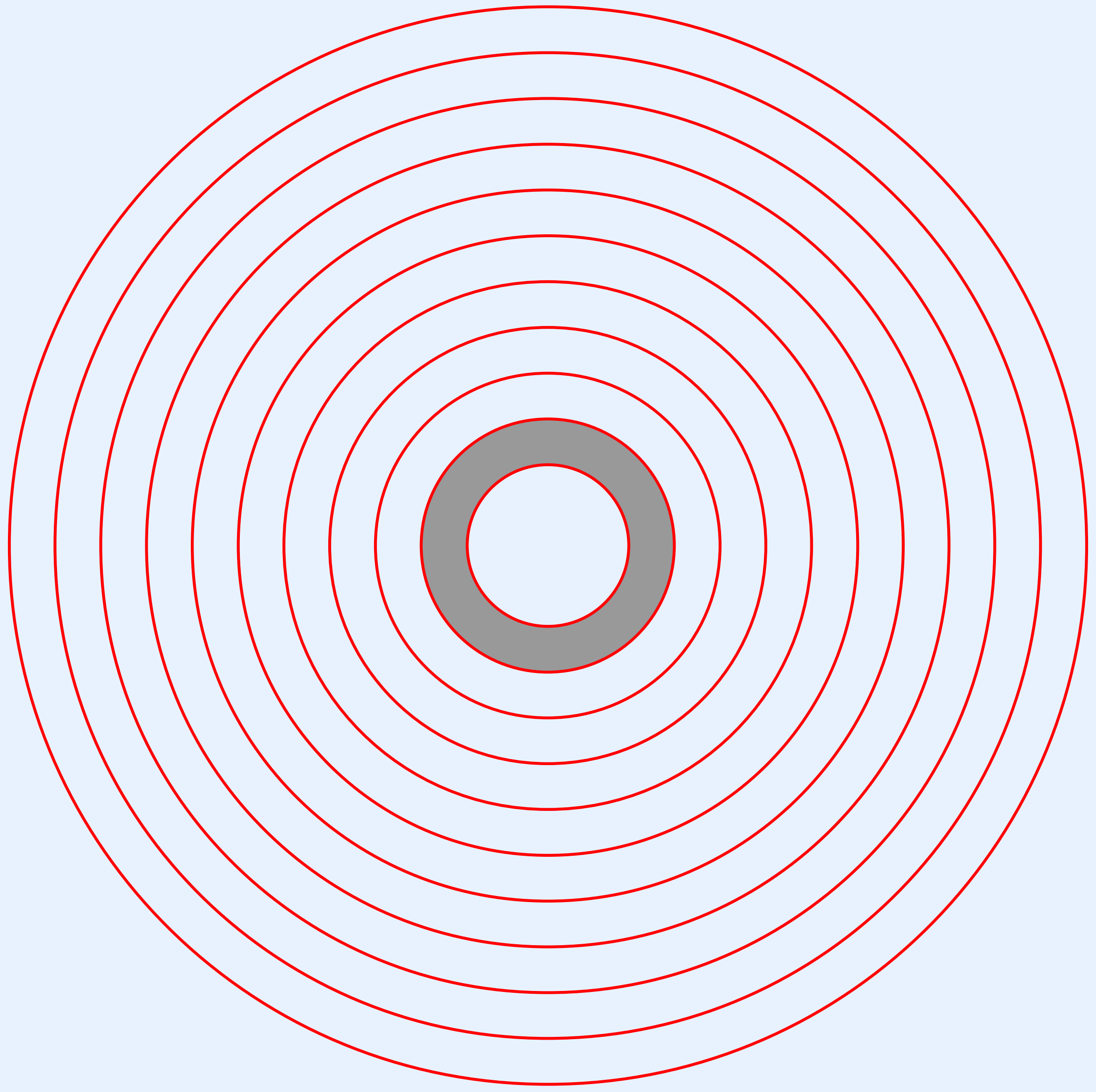
Summary
(continued)

- High-level process coordination is
  - Used by subsets of processes
  - Available inside and outside the OS
  - Implemented with counting semaphore
- Counting semaphore
  - A powerful abstraction implemented in software
  - Provides mutual exclusion and producer / consumer synchronization

Questions?

Low-Level Memory Management

Location Of Low-Level Memory Management In The Hierarchy



The Apparent Impossibility Of A Hierarchical OS Design

The Apparent Impossibility Of A Hierarchical OS Design

- A process manager uses the memory manager to allocate space for a process
- A memory manager uses the device manager to page or swap to disk
- A device manager uses the process manager to block and restart processes when they request I/O

The Apparent Impossibility Of A Hierarchical OS Design

- A process manager uses the memory manager to allocate space for a process
- A memory manager uses the device manager to page or swap to disk
- A device manager uses the process manager to block and restart processes when they request I/O
- Solution: divide the memory manager into two parts

The Two Types Of Memory Management

- Low-level memory manager
  - Manages memory within the kernel address space
  - Used to allocate address spaces for processes
  - Treats memory as a single, exhaustible resource
  - Positioned in the hierarchy below process manager
- High-level memory manager
  - Manages pages within a process’s address space
  - Positioned in the hierarchy above the device manager
  - Divides memory into abstract resources

Conceptual Uses Of A Low-Level Memory Manager

- Allocate stack space for a process
  - Performed by the process manager when a process is created
  - The memory manager must include functions to allocate and free stacks
- Allocation of heap storage
  - Performed by the device manager (buffers) and other system facilities
  - The memory manager must include functions to allocate and free heap space

The Xinu Low-Level Memory Manager

- Two functions control allocation of stack storage

```
        addr = getstk(numbytes);

        freestk(addr, numbytes);
```

- Two functions control allocation of heap storage

```
        addr = getmem(numbytes);

        freemem(addr, numbytes);
```

- Memory is allocated until none remains
- Only *getmem*/*freemem* are intended for use by application processes; *getstk*/*freestk* are restricted to the OS

Well-Known Memory Allocation Strategies

- Stack and heap can be
  - Allocated from the same free area
  - Allocated from separate free areas
- The memory manager can use a single free list and follow a paradigm of
  - First-fit
  - Best-fit
  - The free list can be circular with a roving pointer
- The memory manager can maintain multiple free lists
  - By exact size (static / dynamic)
  - By range

Well-Known Memory Allocation Strategies (continued)

- The free list can be kept in a hierarchical data structure (e.g., a tree)
  - Binary sizes of nodes can be used
  - Other sequences of sizes are also possible (e.g., Fibonacci)
- To handle repeated requests for the same size blocks, a cache can be combined with any of the above methods

Practical Considerations

- Sharing
  - A stack can never be shared
  - Multiple processes may share access to a given block allocated from the heap
- Persistence
  - A stack is associated with one process, and is freed when the process exists
  - An item allocated from a heap may persist longer than the process that created it
- Stacks tend to be one size, but heap requests vary in size
- Fragmentation can occur

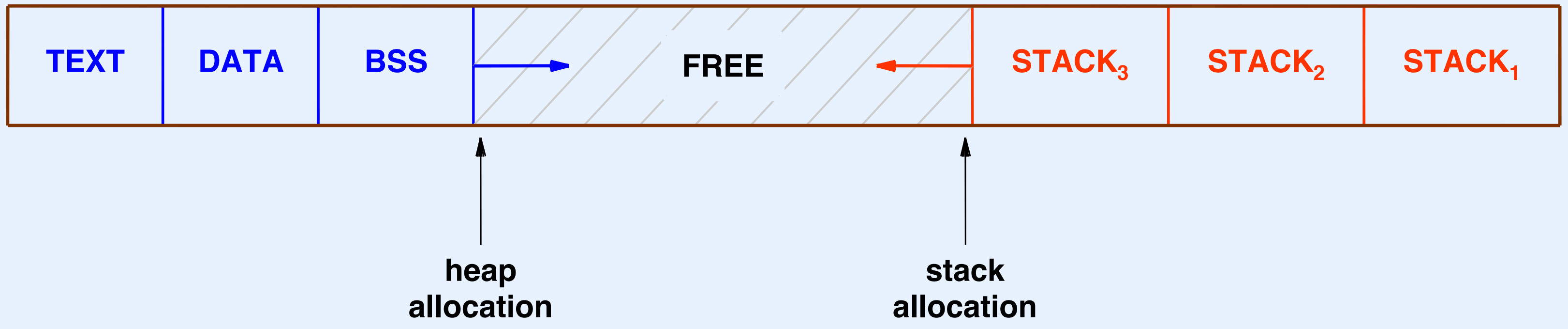
Memory Fragmentation

- Can occur if processes allocate and then free arbitrary-size blocks
- Symptom: after many requests to allocate and free blocks of memory, small blocks of allocated memory exist between blocks of free memory
- The problem: although much of the memory is free, each block on the free list is small
- Example
  - Assume a free memory consists of 1 Gigabyte total
  - A process allocates 1024 blocks of one Megabyte each (1 Gigabyte)
  - The process then frees every other block
  - Although 512 Megabytes of free memory are available, the largest free block is only 1 Megabyte

The Xinu Low-Level Allocation Scheme

- All free memory is treated as one resource
- A single free list is used for both heap and stack allocation
- The free list is
  - Ordered by increasing address
  - Singly-linked
  - Initialized at system startup to contain *all* free memory
- The Xinu allocation policies
  - Heap allocation uses the first-fit approach
  - Stack allocation uses the last-fit approach
  - The design results in two conceptual pools of memory

Consequence Of The Xinu Allocation Policy



- The first-fit policy means heap storage is allocates from lowest part of free memory
- The last-fit policy means stack storage is allocated from the highest part of free memory
- Note: because stacks tend to be uniform size, there is higher probability of reuse and lower probability of fragmentation

Protecting Against Stack Overflow

- Note that the stack for a process can grow downward into the stack for another
- Some memory management hardware supports protection
  - The memory for a process stack is assigned the process's protection key
  - When a context switch occurs the processor protection key is set
  - If a process overflows its stack, hardware will raise an exception
- If no hardware protection is available
  - Mark the top of each stack with a reserved value
  - Check the value when scheduling
  - The approach provides a little protection against overflow

Memory Allocation Granularity

- Facts
  - Memory is byte addressable
  - Some hardware requires alignment
    - For process stack
    - For I / O buffers
    - For pointers
  - Free memory blocks are kept on free list
  - One cannot allocate / free individual bytes
- Solution: choose a minimum granularity and round all requests to the minimum

Example Code To Round Memory Requests

```
/* excerpt from memory.h */

/*----------------------------------------------------------------------
 * roundmb, truncmb - Round or truncate address to memory block size
 *----------------------------------------------------------------------
 */
#define roundmb(x)      (char *)( (7 + (uint32)(x)) & (~7) )
#define truncmb(x)      (char *)( ((uint32)(x)) & (~7) )

struct  memblk  {                       /* See roundmb & truncmb        */
        struct  memblk  *mnext;         /* Ptr to next free memory blk  */
        uint32  mlength;                /* Size of blk (includes memblk)*/
        };
extern  struct  memblk  memlist;        /* Head of free memory list     */
extern  void    *minheap;               /* Start of heap                */
extern  void    *maxheap;               /* Highest valid heap address   */
```

- Note the efficient implementation
  - The size of *memblk* is chosen to be a power of 2
  - The code implements rounding and truncation with bit manipulation

The Xinu Free List

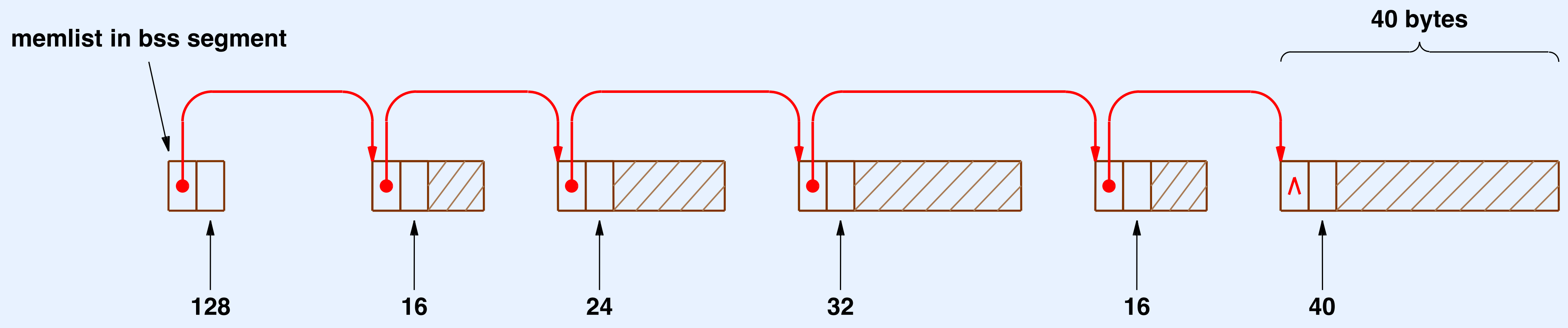
- Employs a well-known trick: to link together a list of free blocks, place all pointers *in the blocks themselves*
- Each block on the list contains
  - A pointer to the next block
  - An integer giving the size of the block
- A fixed location (*memlist* contains a pointer to the first block on the list
- Look again at the definitions in memory.h

Declarations For The Free List

```
/* excerpt from memory.h */

/*----------------------------------------------------------------------
 * roundmb, truncmb - Round or truncate address to memory block size
 *----------------------------------------------------------------------
 */
#define roundmb(x)      (char *)( (7 + (uint32)(x)) & (~7) )
#define truncmb(x)      (char *)( ((uint32)(x)) & (~7) )

struct  memblk  {                       /* See roundmb & truncmb        */
        struct  memblk  *mnext;         /* Ptr to next free memory blk  */
        uint32  mlength;                /* Size of blk (includes memblk)*/
        };
extern  struct  memblk  memlist;        /* Head of free memory list     */
extern  void    *minheap;               /* Start of heap                */
extern  void    *maxheap;               /* Highest valid heap address   */
```

- Struct *memblk* defines the two items stored in every block
- Variable *memlist* is the head of the free list
- Making the head of the list have the same structure as other nodes reduces special cases in the code

Illustration Of Xinu Free List



- Free memory blocks are used to store list pointers
- Items on the list are ordered by increasing address
- All allocations rounded to size of struct *memblk*
- The length in *memlist* counts total free memory bytes

Allocation Technique

- Round up the request to a multiple of memory blocks
- Walk the free memory list
- Choose either
  - First free block that is large enough (*getmem*)
  - Last free block that is large enough (*getstk*)
- If a free block is larger than the request, extract a piece for the request and leave the part that is left over on the free list

When Searching The Free List

- Use two pointers that point to two successive nodes on the list
- An invariant is used during the search
  - Pointer *curr* points to a node on the free list (or *NULL*)
  - Pointer *prev* points to the previous node (or *memlist*)
- The invariant is established initially by making *prev* point to *memblk* and making *curr* point to the item to which *memblk* points
- The invariant must be maintained each time pointers move along the list

Xinu Getmem (Part 1)

```
/* getmem.c - getmem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getmem  -  Allocate heap storage, returning lowest word address
 *------------------------------------------------------------------------
 */
char    *getmem(
          uint32        nbytes          /* Size of memory requested     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *prev, *curr, *leftover;

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use memblk multiples */
```

Xinu Getmem (Part 2)

```
        prev = &memlist;
        curr = memlist.mnext;
        while (curr != NULL) {                  /* Search free list     */

                if (curr->mlength == nbytes) {  /* Block is exact match */
                        prev->mnext = curr->mnext;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);

                } else if (curr->mlength > nbytes) { /* Split big block */
                        leftover = (struct memblk *)((uint32) curr +
                                        nbytes);
                        prev->mnext = leftover;
                        leftover->mnext = curr->mnext;
                        leftover->mlength = curr->mlength - nbytes;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);
                } else {                        /* Move to next block   */
                        prev = curr;
                        curr = curr->mnext;
                }
        }
        restore(mask);
        return (char *)SYSERR;
}
```

Splitting A Block

- Occurs when *getmem* chooses a block that is larger then the requested size
- *Getmem* performs three steps
  - Compute the address of the piece that will be left over (i.e., the right-hand side of the block)
  - Link the leftover piece into the free list
  - Return the original block to the caller
- Note: the address of the leftover piece is curr + nbytes (the addition must be performed using unsigned arithmetic because the high-order bit may be on)

Deallocation Technique

- Round up the specified size to a multiple of memory blocks (allows the user to specify the same value during deallocation that was used during allocation)
- Walk the free list, using *next* to point to a block on the free list, and *prev* to point to the previous block (or *memlist*)
- Stop when the address of the block being freed lies between *prev* and *next*
- Either: insert the block into the list or handle coalescing

Coalescing Blocks

- The term *coalescing* refers to the opposite of splitting
- Coalescing occurs when a block being freed is adjacent to an existing free block
- Technique: instead of adding the new block to the list, combine the new and existing block into one larger block
- Note: the code must check for coalescing with the preceding block, the following block, or both

Xinu Freemem (Part 1)

```
/* freemem.c - freemem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freemem  -  Free a memory block, returning the block to the free list
 *------------------------------------------------------------------------
 */
syscall freemem(
          char          *blkaddr,       /* Pointer to memory block      */
          uint32        nbytes          /* Size of block in bytes       */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *next, *prev, *block;
        uint32  top;

        mask = disable();
        if ((nbytes == 0) || ((uint32) blkaddr < (uint32) minheap)
                          || ((uint32) blkaddr > (uint32) maxheap)) {
                restore(mask);
                return SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use memblk multiples */
        block = (struct memblk *)blkaddr;
```

Xinu Freemem (Part 2)

```
prev = &memlist;                        /* Walk along free list */
next = memlist.mnext;
while ((next != NULL) && (next < block)) {
        prev = next;
        next = next->mnext;
}

if (prev == &memlist) {         /* Compute top of previous block*/
        top = (uint32) NULL;
} else {
        top = (uint32) prev + prev->mlength;
}

/* Ensure new block does not overlap previous or next blocks    */

if (((prev != &memlist) && (uint32) block < top)
    || ((next != NULL)  && (uint32) block+nbytes>(uint32)next)) {
        restore(mask);
        return SYSERR;
}

memlist.mlength += nbytes;
```

Xinu Freemem (Part 3)

```
        /* Either coalesce with previous block or add to free list */

        if (top == (uint32) block) {    /* Coalesce with previous block */
                prev->mlength += nbytes;
                block = prev;
        } else {                        /* Link into list as new node   */
                block->mnext = next;
                block->mlength = nbytes;
                prev->mnext = block;
        }

        /* Coalesce with next block if adjacent */

        if (((uint32) block + block->mlength) == (uint32) next) {
                block->mlength += next->mlength;
                block->mnext = next->mnext;
        }
        restore(mask);
        return OK;
}
```

Xinu Getstk (Part 1)

```
/* getstk.c - getstk */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getstk  -  Allocate stack memory, returning highest word address
 *------------------------------------------------------------------------
 */
char    *getstk(
          uint32        nbytes          /* Size of memory requested     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *prev, *curr;   /* Walk through memory list     */
        struct  memblk  *fits, *fitsprev; /* Record block that fits     */

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use mblock multiples */

        prev = &memlist;
        curr = memlist.mnext;
        fits = NULL;
```

Xinu Getstk (Part 2)

```
        while (curr != NULL) {                  /* Scan entire list     */
                if (curr->mlength >= nbytes) {  /* Record block address */
                        fits = curr;            /*   when request fits  */
                        fitsprev = prev;
                }
                prev = curr;
                curr = curr->mnext;
        }

        if (fits == NULL) {                     /* No block was found   */
                restore(mask);
                return (char *)SYSERR;
        }
        if (nbytes == fits->mlength) {          /* Block is exact match */
                fitsprev->mnext = fits->mnext;
        } else {                                /* Remove top section   */
                fits->mlength -= nbytes;
                fits = (struct memblk *)((uint32)fits + fits->mlength);
        }
        memlist.mlength -= nbytes;
        restore(mask);
        return (char *)((uint32) fits + nbytes - sizeof(uint32));
}
```

Xinu Freestk

```
/* excerpt from memory.h */

/*------------------------------------------------------------------------
 *  freestk  --  Free stack memory allocated by getstk
 *------------------------------------------------------------------------
 */
#define freestk(p,len)  freemem((char *)((uint32)(p)           \
                                - ((uint32)roundmb(len))       \
                                + (uint32)sizeof(uint32)),     \
                                (uint32)roundmb(len) )
```

- Implemented as an inline function
- Technique: convert address from the highest address in block being freed to the lowest address in the block, and call *freemem*

Virtual Memory

Definition Of Virtual Memory

- An abstraction of physical memory
- It provides separation from underlying hardware
- Primarily used with applications (user processes)
- Provides an application with an address space that is independent of
  - Physical memory size
  - A position in physical memory
- Many mechanisms have been proposed and used

General Approach

- Typically used with a heavyweight process
  - The process appears to run in an isolated address space
  - All addresses are *virtual*, meaning that each process has an address space that starts at address zero
- The operating system
  - Establishes policies for memory use
  - Creates a separate virtual address space for each process
  - Configures the hardware as needed
- The underlying hardware
  - Dynamically translates from virtual addresses to physical addresses
  - Provides support to help the operating system make policy decisions

A Virtual Address Space

- Can be smaller than the physical memory
  - Example: a 32-bit computer with more than 2^{32} bytes (four GB) of physical memory
- Can be larger than the physical memory
  - Example: a 64-bit computer with less than 2^{64} bytes (16 million terabytes) of memory
- Historic note: on early computers, physical memory was larger. Then, virtual memory was larger until physical memory caught up. Now, 64-bit architectures mean virtual memory is once again larger than physical memory.

Multiplexing Virtual Address Spaces Onto Physical Memory

- General idea
  - Store a complete copy of each process's address space on secondary storage
  - Move pieces of the address space to main memory as needed
  - Write pieces back to disk to create space in memory for other pieces
- Questions
  - How much of a process's address space should reside in memory?
  - When should a particular piece be loaded into memory?
  - When should a piece be written back to disk?

Approaches That Have Been Used

- *Swapping*
  - Transfer an entire address space (complete process image)
- *Segmentation*
  - Divide the image into large segments
  - Transfer segments as needed
- *Paging*
  - Divide image into small, fixed-size pieces
  - Transfer individual pieces as needed

Approaches That Have Been Used (continued)

- *Segmentation with paging*
  - Divide an image into large segments
  - Further subdivide segments into fixed-size pages

A Widely-Used Approach

- Paging has emerged as the most widely used approach for virtual memory
- The reasons are
  - Choosing a reasonable page size (e.g., 4K bytes) reduces page faults for most applications
  - Efficient hardware is possible

Choosing a page size that is a power of two makes it possible to build extremely efficient address mapping hardware.

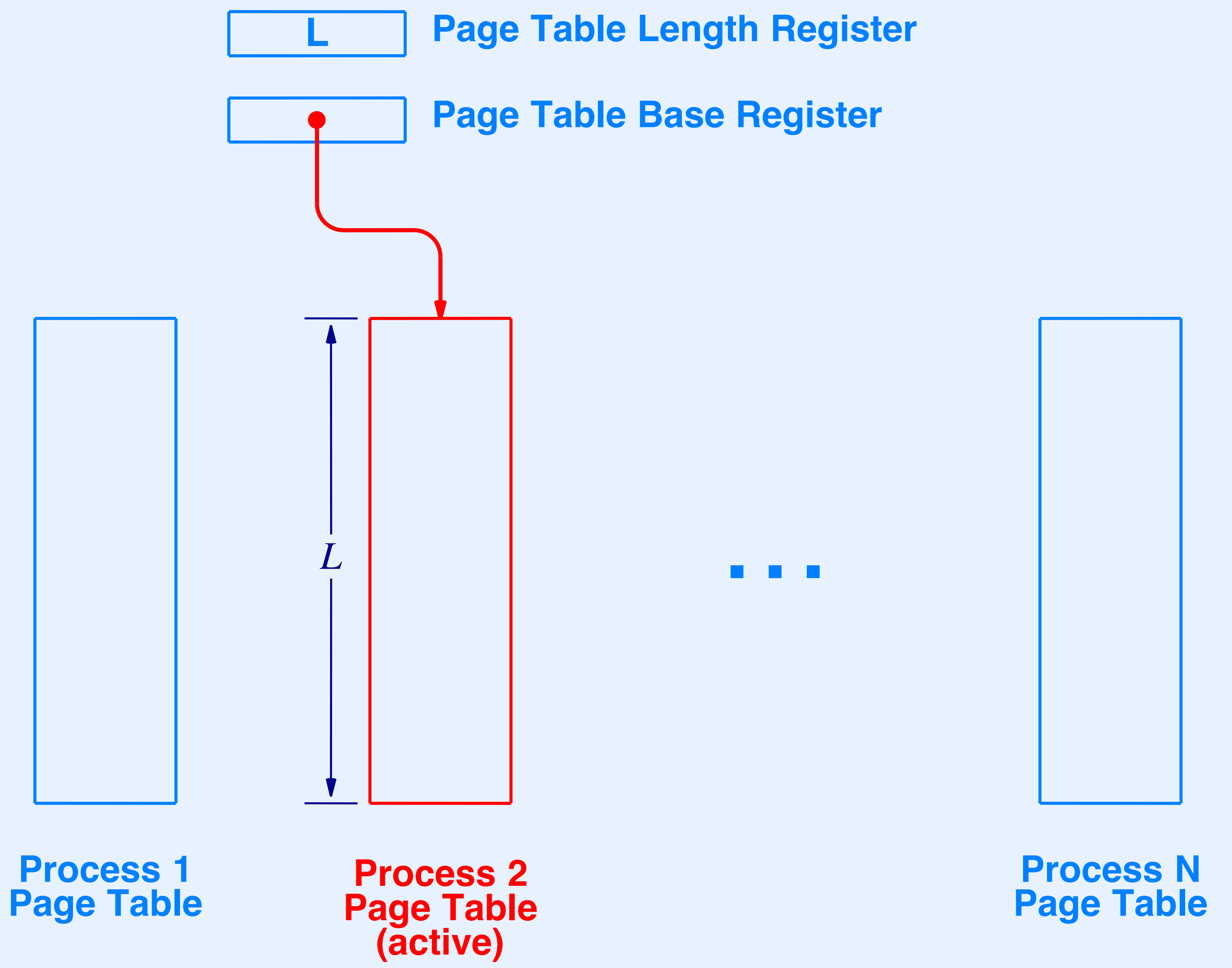
Hardware Support For Paging

- Page tables
  - The operating system allocates one page table per process
  - The location at which a page table is stored depends on the hardware
    - Kernel memory (typical)
    - *Memory Management Unit (MMU)* hardware (on some systems)
- A page table base register
  - Internal to the processor
  - Specifies the location of the page table currently being used (i.e., the page table for the current process)
  - Must be changed during a context switch

Hardware Support For Paging
(continued)

- A page table length register
  - Internal to the processor
  - Specifies the number of entries in the current page table
  - Can be changed during context switch if the size of the virtual address space differs among processes
  - Can be used to limit the size of a process's virtual address space

Illustration Of VM Hardware Registers



- Only one page table is active at a given time (the page table for the current process)

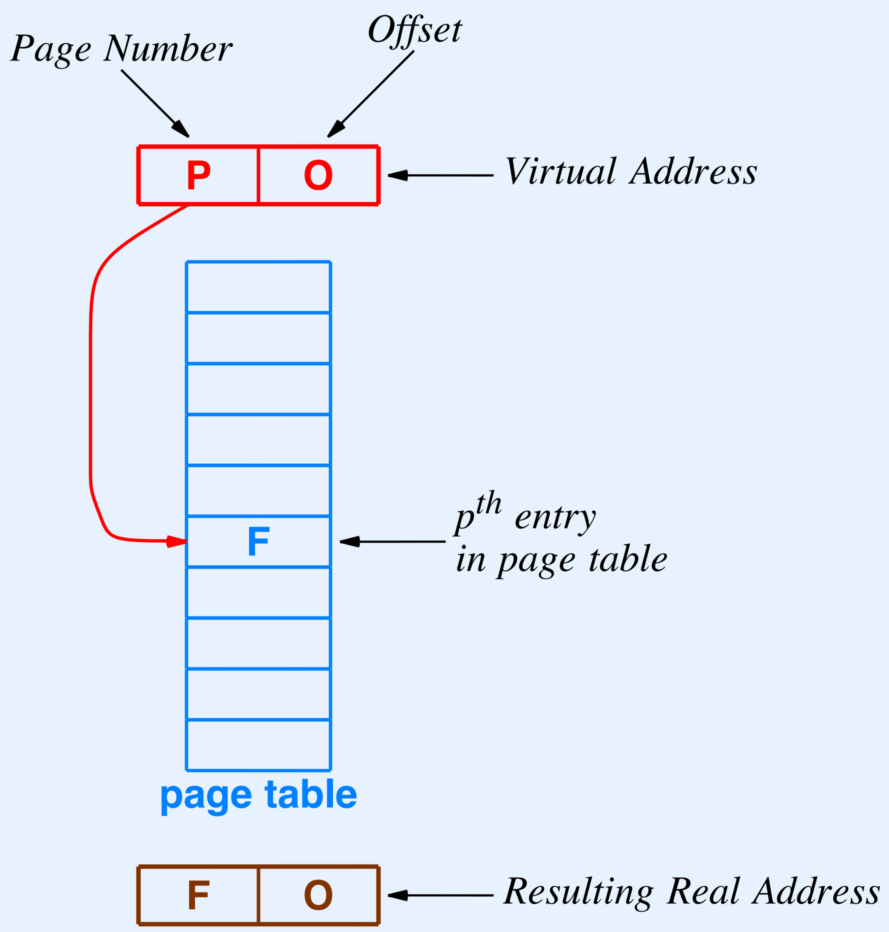
Address Translation

- A key part of virtual memory
- Refers to the translation from the virtual address a process uses to the corresponding physical memory address
- Is performed by memory management hardware
- Must occur on *every* memory reference
- A hardware unit performs the translation

Address Translation With Paging

- For now, we will assume
  - The operating system is not paged
  - The physical memory area beyond the operating system kernel is used for paging
  - Each page is 4 Kbytes (typical of current virtual memory hardware)
- Think of the physical memory area used for paging as a giant array of *frames*, where each frame can hold one page (i.e., a frame is 4K bytes)

Illustration Of Address Translation



- Each page table entry contains a physical frame address
- Choosing the page size to be a power of 2 eliminates division and modulus

In Practice

- The size of virtual space may be limited to physical memory size
- Some hardware offers separate page tables for text, data, and stack segments
  - The chief disadvantage: extra complexity
  - The advantage: the three can operate independently
- The kernel address space can also be virtual (but it hasn’t worked well in practice)

Page Table Sizes And 32 and 64 Bit Computers

- For a 32-bit address space where each page is 4 Kbytes
  - There are 2^{20} page table entries of 4 bytes per entry
  - The total page table size for one process: 4 Mbytes
- For a 64-bit address space where each page is 4 Kbytes
  - There are 2^{52} page table entries of 4 bytes per entry
  - The total page table size for one process: 16,777,216 Gbytes!
- Conclusion: we cannot have complete page tables for a 64-bit address space

Paging In A 64-Bit System

- To reduce page table size, use multiple levels of page tables
  - The high-order bits of an address form an index into the top-level page table
  - The next bits form an index into the second-level page table (but only a few second-level page tables are defined)
- Key idea: only the lowest and highest pieces of the address space need to be mapped (heap and stack)
- The same technique can be applied to 32-bit address spaces to reduce page table size

The Concept Of Demand Paging

- Keep the entire memory image of each process on secondary storage
- Treat main memory as cache of recently-referenced pages
- Allocate space in memory for new pages dynamically (when needed)
- Copy a page from the secondary store to main memory on demand (when referenced)
- To make space for newly-referenced pages, move pages from the memory cache that have been changed but are not being referenced back to their place on secondary storage

The Importance Of Hardware Support For Virtual Memory

- Every memory reference must be translated from a virtual address to a physical address, including
  - The address of an instruction as well as data
  - Branch addresses computed as a *jump* instruction executes
  - Indirect addresses that are generated at runtime
- Hardware support is essential
  - For efficiency
  - For recovery if a fault occurs
  - To record which pages are being used

In Practice

- A single instruction may reference many pages!
  - To fetch the instruction
  - To fetch each operand
  - To follow indirect references
  - To store results
- On CISC hardware, a memory copy instruction can reference *multiple* pages
- The point: hardware support is needed to make references efficient

Hardware Support For Address Mapping

- Special-purpose hardware speeds page lookup and makes paging practical
- The hardware
  - Is called a *Translation Look-aside Buffer* (*TLB*)
  - Is implemented with T-CAM
- A TLB caches most recent address translations
- Good news: many applications tend to make repeated references to the same page (i.e., a high locality of reference), so a TLB works well
- Some VM mappings must change during a context switch (e.g., multiple processes may each have a virtual address 0, but the mapping differs)
- The point: the mappings in the TLB will not be valid for the new process

Managing The TLB

- When the operating system switches context
  - On some hardware, the operating system must flush the TLB to remove old entries
  - On other hardware, tags are used to distinguish among address spaces
- Using tags
  - A tag is assigned to each page by the OS (typically, the process ID)
  - The operating system tells the VM hardware the tag to use (i.e., the current process)
  - When placing a mapping in the TLB, the hardware appends the current tag to the address
  - When searching the TLB, the hardware appends the tag to the address
  - Advantage: the OS does not need to flush the TLB during a context switch

Can Page Tables Be Paged?

- On some hardware, yes
- The tables must be stored in memory
- The current page table must be locked into memory
- Paging page tables is so extremely inefficient that is it impractical

Bits That Record Page Status

- Each page table entry contains status bits that are understood by the hardware
- The *Use Bit*
  - Set by the hardware whenever the page is referenced
  - Applies to both *fetch* and *store* operations
- The *Modify Bit*
  - Set by the hardware when a *store* operation occurs
- The *Presence Bit*
  - Set by the operating system, to indicate that it has placed the page in memory (we say the page is *resident*)
  - Tested by the hardware when the page is referenced

Page Replacement

- The hardware
  - Generates a page fault when a referenced page is not resident
  - Raises an exception to inform the operating system
- The operating system
  - Receives the exception
  - Allocates a frame in physical memory
  - Retrieves the needed page (allowing other processes to execute while page is being fetched)
  - Once the page arrives, marks the page table entry to indicate the page is resident, and restarts the process that caused the page fault

Researchers Have Studied Many Aspects Of Paging

- Which replacement policies are most effective?
- Which pages from a given address spaces should be in memory at any time?
- Should some pages be locked in memory? (If so, which ones?)
- How does a VM policy interact with other policies (e.g., scheduling?)
- Should high-priority processes/threads have guarantees about the number of resident pages?
- If a system supports libraries that are shared among many processes, which paging policy applies to a library?

A Critical Trade-off For Demand Paging

- Paging overhead and latency for a given process can be reduced by giving the process more physical memory (more frames)
- Processor utilization and overall throughput are increased by increasing level of multitasking (concurrent processes)
- Extremes
  - Paging is minimized when the current process has maximal memory
  - Throughput is maximized when all ready processes are resident
- Researchers considered the question, “What is the best tradeoff?”

Frame Allocation

- When a page fault occurs, the operating system must obtain a frame to hold the page
- If at least one frame is unused, the selection is trivial — select an unused frame
- If all frames are currently occupied by pages from various processes, the operating system must
  - Select one of the resident pages and save a copy on disk
  - Mark the page table entry to indicate that the page is no longer resident
  - Select the frame that has been vacated
  - Obtain the page that caused the page fault, and fill in the appropriate page table entry
- Question: which frame should be selected when all are in use?

Choosing A Frame

- Researchers have studied
  - Global competition: consider frames from all processes when choosing a frame
  - Local competition: choose one of the frames of the process that caused the page fault
- They have also studied various policies
  - *Least Recently Used* (*LRU*)
  - *Least Frequently Used* (*LFU*)
  - *First In First Out* (*FIFO*)
- In the end, a basic approach has been adopted: *global clock*

The Global Clock Algorithm

- Originated in the MULTICS operating system
- Allows all processes to compete with one another (hence the term *global*)
- Has relatively low overhead
- Has become the most popular practical method

Global Clock Paradigm

- The clock algorithm is activated when a page fault occurs
- It searches through all frames in memory, and selects a frame to use
- The term *clock* is used because the algorithm starts searching where it left off the last time
- A frame containing a referenced page is given a “second chance” before being reclaimed
- A frame containing a modified page is given a “third chance” before being reclaimed
- In the worst case: the clock sweeps through all frames twice before reclaiming one
- Advantage: the algorithm does *not* require any external data structure other than the standard page table bits

Operation Of The Global Clock

- The clock uses a global pointer that picks up where it left off previously
  - It sweeps through all frames in memory
  - It only starts moving when a frame is needed
  - It stops moving once a frame has been selected
- During the sweep, the algorithm checks *Use* and *Modify* bits of each frame
- It reclaims the frame if the *Use* / *Modify* bits are *(0,0)*
- It changes *(1,0)* into *(0,0)* and bypasses the frame
- It changes *(1,1)* into *(1,0)* and bypasses the frame
- The algorithm keeps a copy of the actual modified bit to know whether a page is dirty

In Practice

- A global clock is usually configured to reclaim a small set of frames when one is needed
- The reclaimed frames are cached for subsequent references
- Advantage: collecting multiple frames means the clock will run less frequently

A Problem With Paging: Thrashing

- Imagine a large set of processes each referencing their pages at random
- At first, free frames in memory can be used to hold pages
- Eventually, the frames in memory fill up, and each new reference causes a page fault, which results in
  - Choosing a frame (the clock algorithm runs)
  - Writing the existing page to secondary storage (disk I/O)
  - Fetching a new page from secondary storage (more disk I/O)
- The processor spends most of the time paging and waiting for I/O, so little computation can be performed
- We use the term *thrashing* to describe the situation
- Having a large memory on a computer helps avoid thrashing

The Importance/Unimportance Of Paging Algorithms

- Facts
  - At one time, page replacement algorithms were the primary research topic in operating systems
  - Sophisticated mathematical analysis was done to understand their behavior
  - By the 1990s, interest in page replacement algorithms faded
  - Now, almost no one uses complex replacement algorithms

The Importance/Unimportance Of Paging Algorithms

- Facts
  - At one time, page replacement algorithms were the primary research topic in operating systems
  - Sophisticated mathematical analysis was done to understand their behavior
  - By the 1990s, interest in page replacement algorithms faded
  - Now, almost no one uses complex replacement algorithms
- Why did the topic fade?
- Was the problem completely solved?

The Importance/Unimportance Of Paging Algorithms

- Facts
  - At one time, page replacement algorithms were the primary research topic in operating systems
  - Sophisticated mathematical analysis was done to understand their behavior
  - By the 1990s, interest in page replacement algorithms faded
  - Now, almost no one uses complex replacement algorithms
- Why did the topic fade?
- Was the problem completely solved?
- Answer: physical memories became so large that very few systems need to replace pages
- A computer scientist once quipped that paging only works if systems don’t page

Summary

- We considered two forms of high-level memory management
- Inside the kernel
  - Define a set of abstract resources
  - Firewalling memory used by each subsystem prevents interference
  - The mechanism uses buffer pools
  - A buffer is referenced by single address
- Outside the kernel
  - Swapping, segmentation, and paging have been used

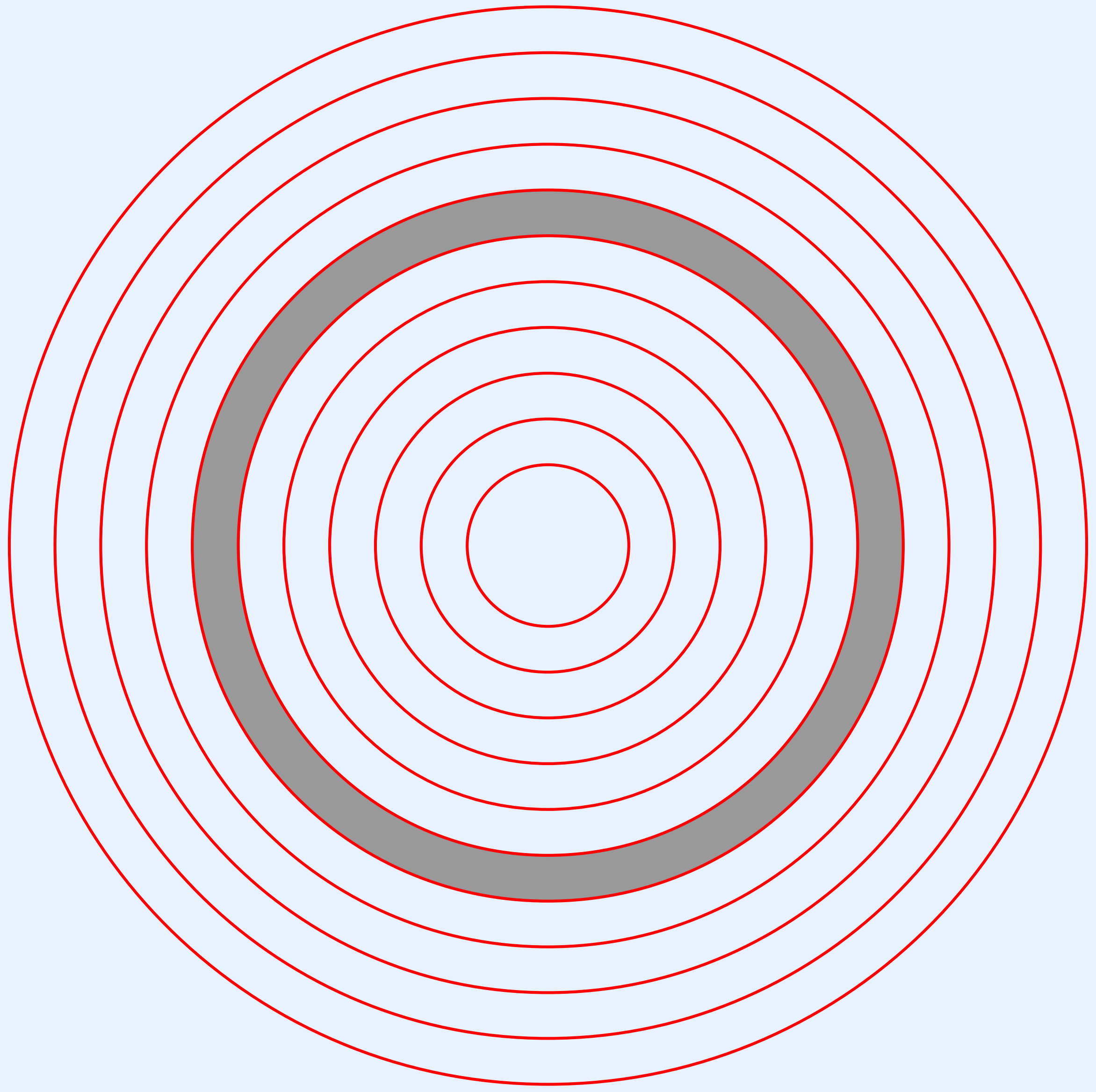
Summary
(continued)

- Demand paging is the most popular VM technology
  – It uses fixed size pages (typically 4K bytes)
  – A page is brought into memory when referenced
- The global clock algorithm is widely used for page replacement

Module VIII

Device Management
Interrupts, Device Drivers, Clocks. And Clock Management

Location Of Device Management In The Hierarchy



Ancient History

- Each device had a unique hardware interface
- Code to communicate with device was built into applications
- An application polled the device; interrupts were not used
- Disadvantages
  - It was painful to create a program
  - A program could not use arbitrary devices (e.g., specific models of a printer and a disk were part of the program)

The Modern Approach

- A device manager is part of an operating system
- The operating system presents applications with a uniform interface to all devices (as much as possible)
- All I / O is *interrupt-driven*

A Device Manager In An Operating System

- Manages peripheral resources
- Hides low-level hardware details
- Provides an API that applications use
- Synchronizes processes and I/O

A Conceptual Note

One of the most intellectually difficult aspects of operating systems arises from the interaction between processes (an operating system abstraction) and devices (a hardware reality). Specifically, the connection between interrupts and scheduling can be tricky because an interrupt that occurs in one process can enable another.

Review Of I/O Using Interrupts

- The processor
  - Starts a device
  - Enables interrupts and continues with other computation
- The device
  - Performs the requested operation
  - Raises an interrupt on the bus
- Processor hardware
  - Checks for interrupts after each instruction is executed, and invokes an interrupt function if an interrupt is pending
  - Has a special instruction used to return from interrupt mode and resume normal processing

Processes And Interrupts

- Key ideas
  - Recall that at any time, a process is running
  - We think of an interrupt as a function call that occurs “between” two instructions
  - Processes are an operating system abstraction, not part of the hardware
  - An operating system cannot afford to switch context whenever an interrupt occurs
- Consequence:

The current process executes interrupt code

Historic Interrupt Software

- A separate interrupt function was created for each device
  - Very low-level code
  - Handles many details
    - Saves / restores registers
    - Sets the interrupt mask
  - Finds the interrupting device on the bus
  - Interacts with the device to transfer data
  - Resets the device for the next interrupt
  - Returns from the interrupt to normal processing

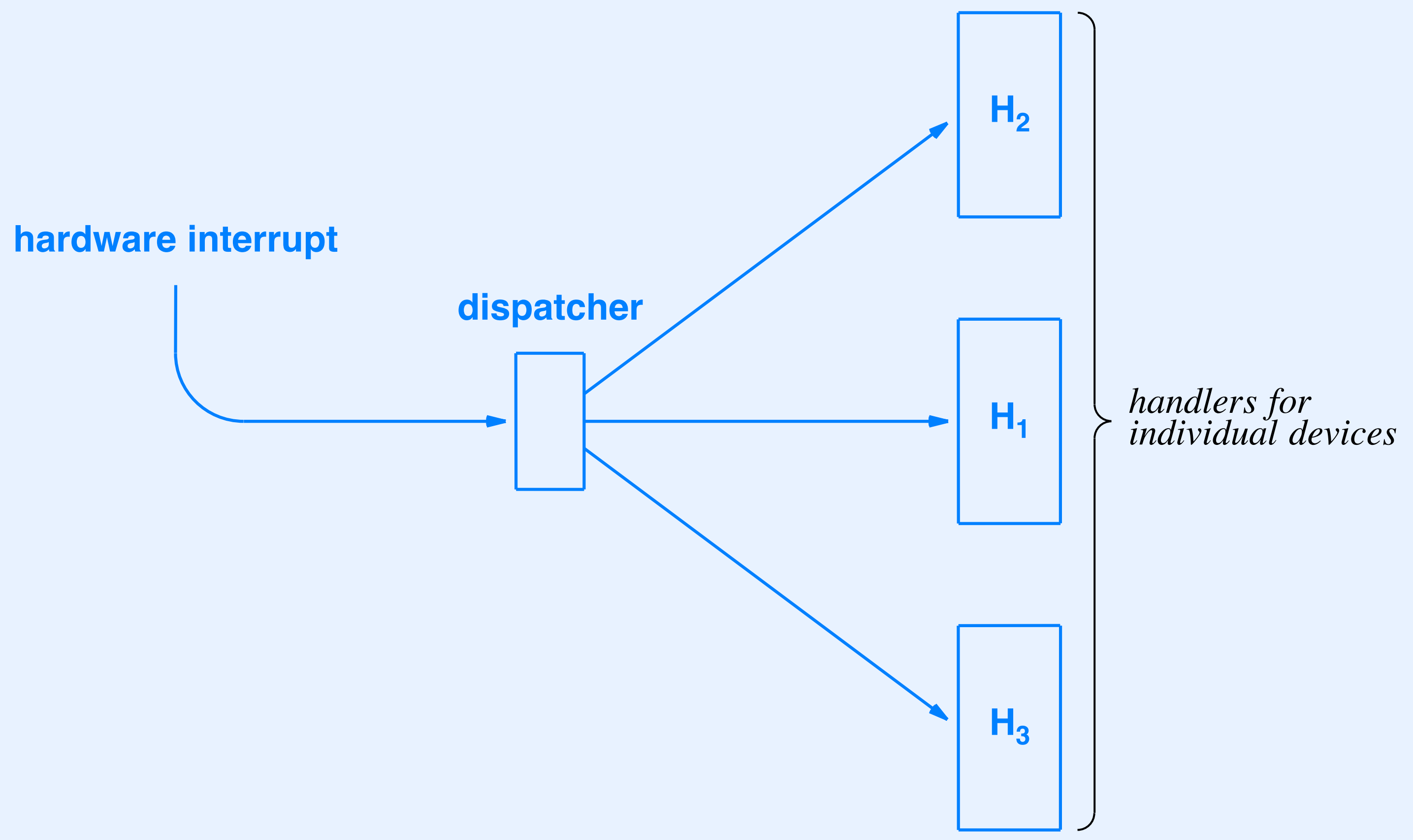
Modern Interrupt Software (Two Pieces)

- An *interrupt dispatcher*
  - Is a single function common to all interrupts
  - It handles low-level details, such as finding the interrupting device on the bus
  - It sets up the environment needed for a function call and calls a device-specific function
  - Some functionality may be incorporated into an *interrupt controller chip*
- An *interrupt handler*
  - One handler for each device
  - Is invoked by the dispatcher
  - Performs all interaction with a specific device

Interrupt Dispatcher

- A low-level piece of code
- Is invoked by the hardware when interrupt occurs
  - Runs in interrupt mode (i.e., with further interrupts disabled)
  - The hardware has saved the instruction pointer for a return
- The dispatcher
  - Saves other machine state as necessary
  - Identifies the interrupting device
  - Establishes the high-level runtime environment needed by a C function
  - Calls a device-specific *interrupt handler*

Conceptual View Of Interrupt Dispatching



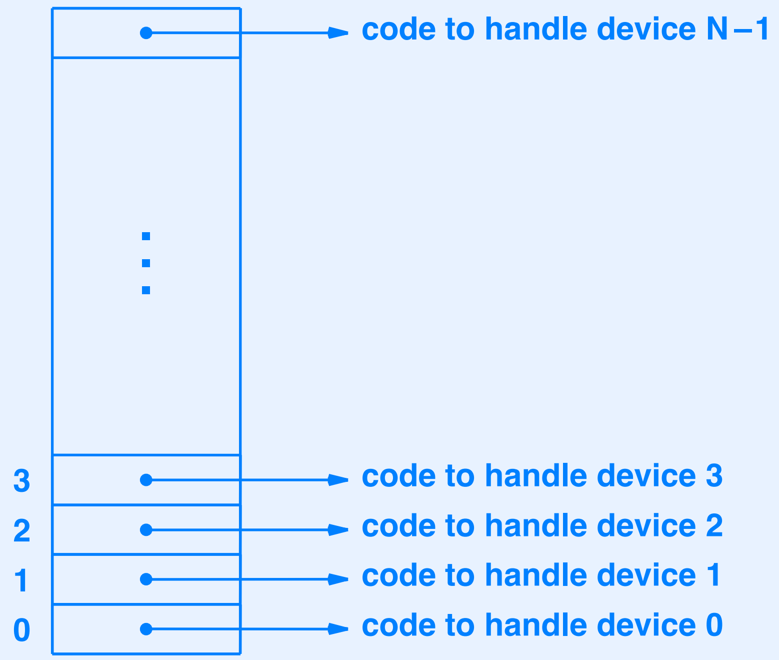
- Note: the dispatcher is typically written in assembly language

Return From Interrupt

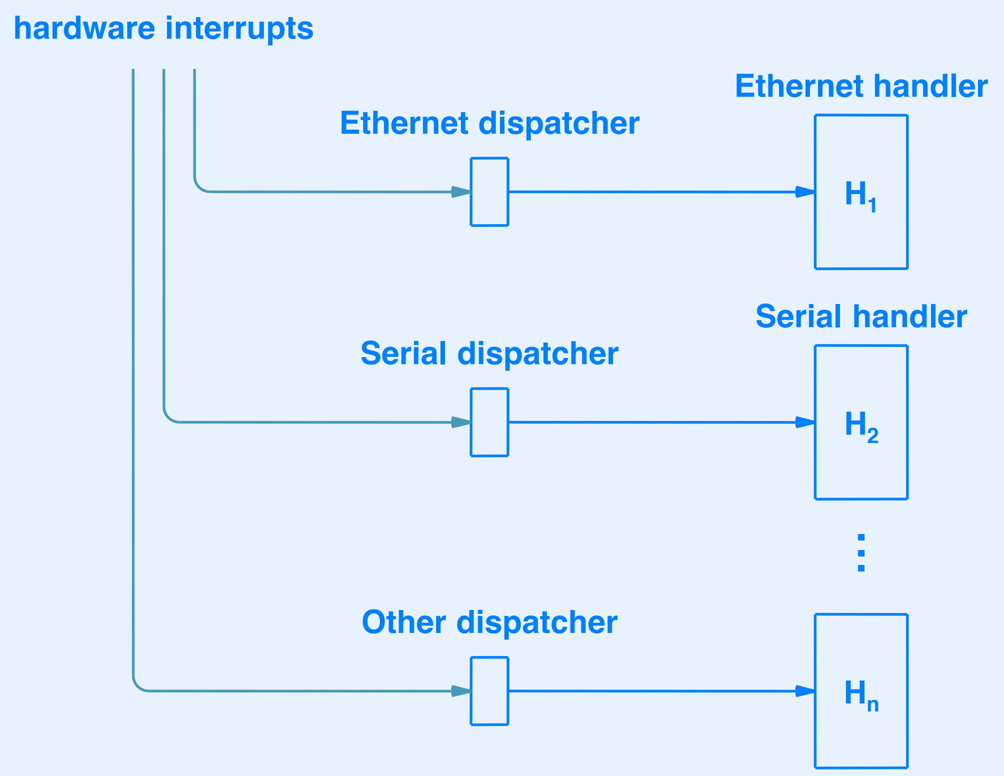
- The interrupt handler
  - Communicates with the device
  - May restart the next operation on the device
  - Eventually returns to the interrupt dispatcher
- The interrupt dispatcher
  - Executes a special hardware instruction known as *return from interrupt*
- The *return from interrupt* instruction atomically
  - Resets the instruction pointer to the saved value
  - Enables interrupts

The Mechanism Used For Interrupts: A Vector

- Each possible interrupt is assigned a unique integer, sometimes called an *IRQ*
- The hardware uses the IRQ as an index into an *interrupt vector* array
- Conceptual organization of an interrupt vector

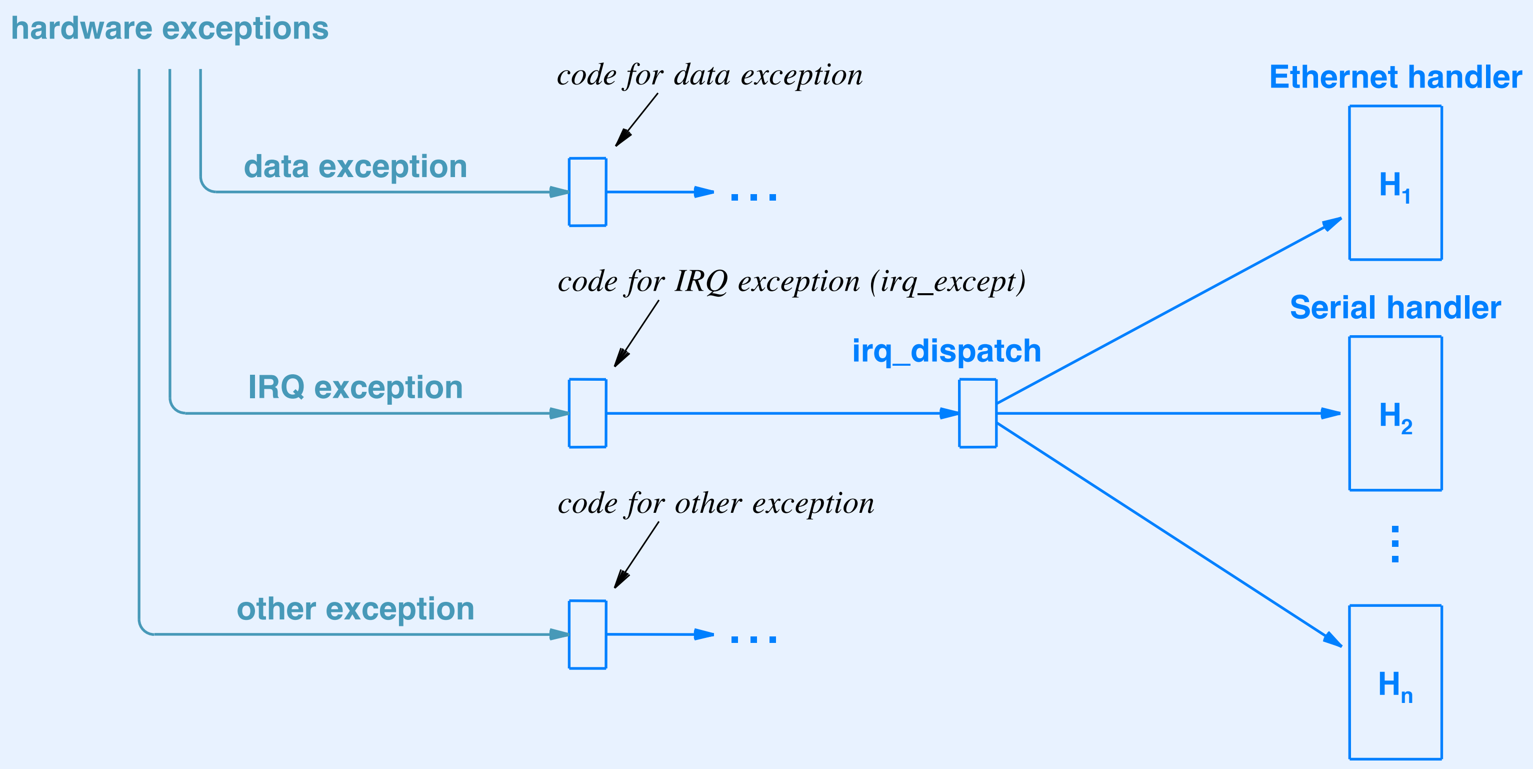


Interrupts On A Galileo (x86)



- The operating system preloads the interrupt controller with the address of a dispatcher for each device
- The controller invokes the correct dispatcher

Interrupts On A BeagleBone Black (ARM)



- Uses a two-level scheme where the controller hardware raises an *IRQ exception* for any device interrupt
- The IRQ exception code invokes the IRQ dispatcher, which calls the correct handler

A Basic Rule For Interrupt Processing

- Facts
  - The processor disables interrupts before invoking the interrupt dispatcher
  - Interrupts remain disabled when the dispatcher calls a device-specific interrupt handler
- Rule
  - To prevent interference, an interrupt handler must keep interrupts disabled until it finishes touching global data structures, ensures all data structures are in a consistent state, and returns
- Note: we will consider a more subtle version of the rule later

Interrupts And Processes

- When an interrupt occurs, I/O has completed
- Either
  - The device has received incoming data
  - Space has become available in an output buffer because the device has finished sending outgoing data
- A process may have been blocked waiting
  - To read the data that arrived
  - To write more outgoing data
- The blocked process may have a higher priority than the currently executing process
- The scheduling invariant *must* be upheld

The Scheduling Invariant

- Suppose process *X* is executing when an interrupt occurs
- We said that process *X* remains executing when the interrupt dispatcher is invoked and when the dispatcher calls a handler
- Suppose data has arrived and a higher-priority process, process *Y*, is waiting for the data
- If the hander merely returns from the interrupt, process *X* will continue to execute
- To maintain the scheduling invariant, the handler must call *resched*

Interrupts And The Null Process

- In the concurrent processing world
  - A process is always running
  - An interrupt can occur at any time
  - The currently executing process executes interrupt code
- An important consequence: the null process may be running when an interrupt occurs, which means the null process will execute the interrupt handler
- We know that the null process must always remain eligible to execute

A Restriction On Interrupt Handlers Imposed By The Null Process

Because an interrupt can occur while the null process is executing, an interrupt hander can only call functions that leave the executing process in the current or ready states. For example: an interrupt handler can call send **or** signal**, but cannot call** wait**.**

A Question About Scheduling And Interrupts

- Recall that
  - The hardware disables further interrupts before invoking a dispatcher
  - Interrupts remain disabled when the dispatcher calls a device-specific interrupt handler
- To remain safe
  - A device-specific interrupt handler must keep further interrupts disabled until it completes changes to global data structures
- What happens if an interrupt calls a function that calls *resched* and the new process has interrupts enabled?

An Example Of Rescheduling During Interrupt Processing

- Suppose
  - An interrupt handler calls *signal*
  - *Signal* calls *resched*
  - *Resched* switches to a new process
  - The new process executes with interrupts enabled
- Will interrupts pile up indefinitely?

An Example

- Let *T* be the current process
- When interrupt occurs, *T* executes an interrupt handler
- The interrupt handler calls *signal*
- *Signal* calls *resched*
- A context switch occurs and process *S* runs
- *S* may run with interrupts enabled

The Answer

Rescheduling during interrupt processing is safe provided that each interrupt handler leaves global data in a valid state before rescheduling and no function enables interrupts unless it previously disabled them (i.e., uses disable/restore **rather than** enable**).**

Device Drivers

Definition Of A Device Driver

- A *device driver* consists of a set of functions that perform I/O operations on a given device
- The code is device-specific
- The set includes
  - An interrupt handler function
  - Functions to control the device
  - Functions to read and write data
- The code is divided into two conceptual parts

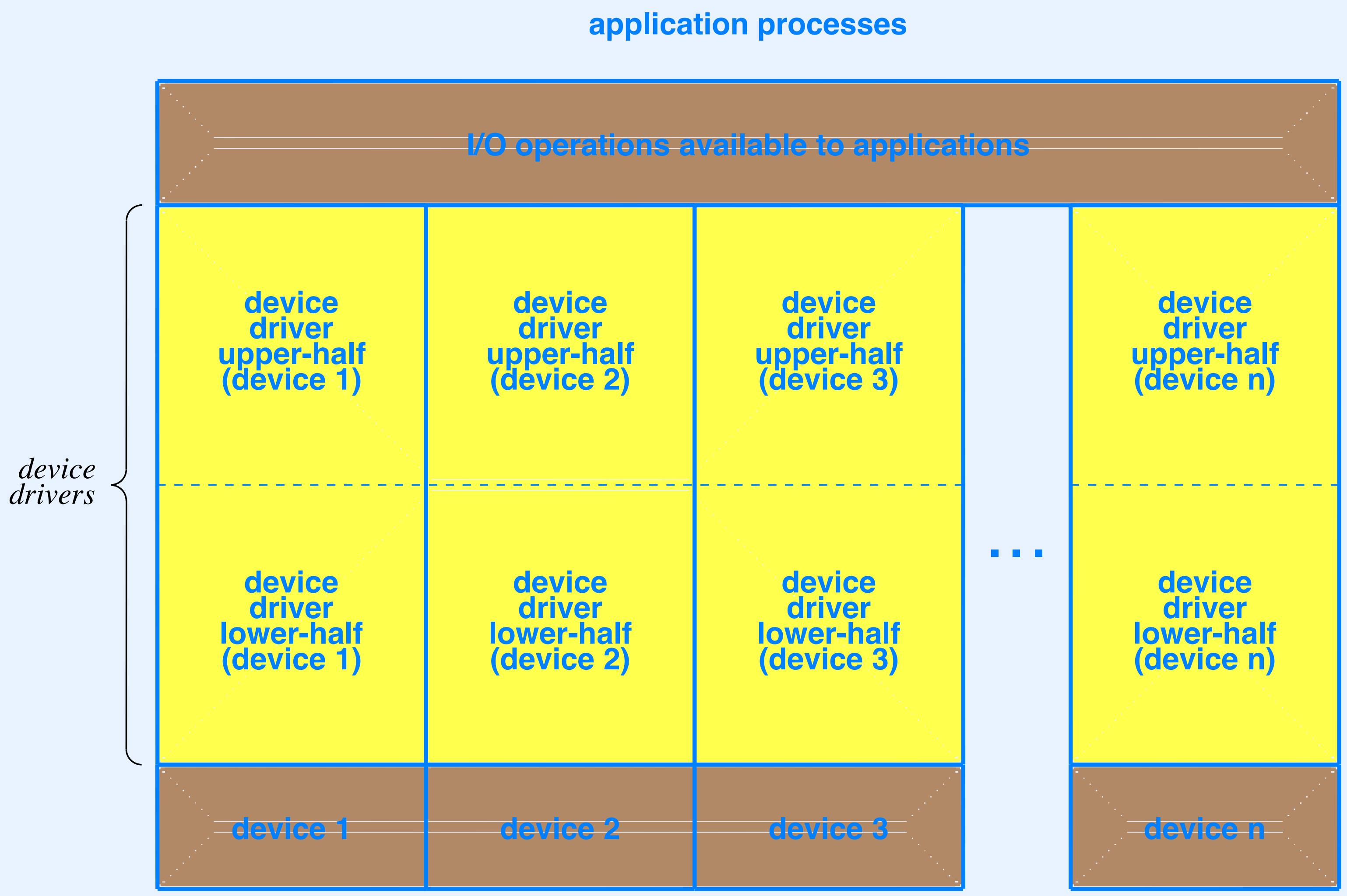
The Two Conceptual Parts Of A Device Driver

- The upper-half
  - Functions that are executed by an application
  - The functions usually perform data transfer (*read* or *write*)
  - The code copies data between the user and kernel address spaces
- The lower-half
  - Is invoked by the hardware when an interrupt occurs
  - Consists of a device-specific interrupt handler
  - May also include dispatcher code, depending on the architecture
  - Executed by whatever process is executing
  - May restart the device for the next operation

Division Of Duties In A Driver

- The upper-half functions
  - Have minimal interaction with device hardware
  - Enqueue a request, and may start the device
- The lower-half functions
  - Have minimal interaction with application
  - Interact with the device to
    - Obtain incoming data
    - Start output
  - Reschedule if a process is waiting for the device

Conceptual Organization Of Device Software


application processes
I/O operations available to applications
device drivers
device driver upper-half (device 1)
device driver upper-half (device 2)
device driver upper-half (device 3)
device driver upper-half (device n)
device driver lower-half (device 1)
device driver lower-half (device 2)
device driver lower-half (device 3)
. . .
device driver lower-half (device n)
device 1
device 2
device 3
device n

Synchronous Interface I/O

- Most systems provide a *synchronous* I/O interface to applications
- For input, the calling process is blocked until data arrives
- For output, the calling process is blocked until the device driver has buffer space to store the outgoing data

Coordination Of Processes Performing I/O

- A device driver must be able to block and later unblock application processes
- Good news: there is no need to invent new coordination mechanisms because standard process coordination mechanisms suffice
  - Message passing
  - Semaphores
  - Suspend / resume
- We will see examples later

Summary Of Interrupts And Device Drivers

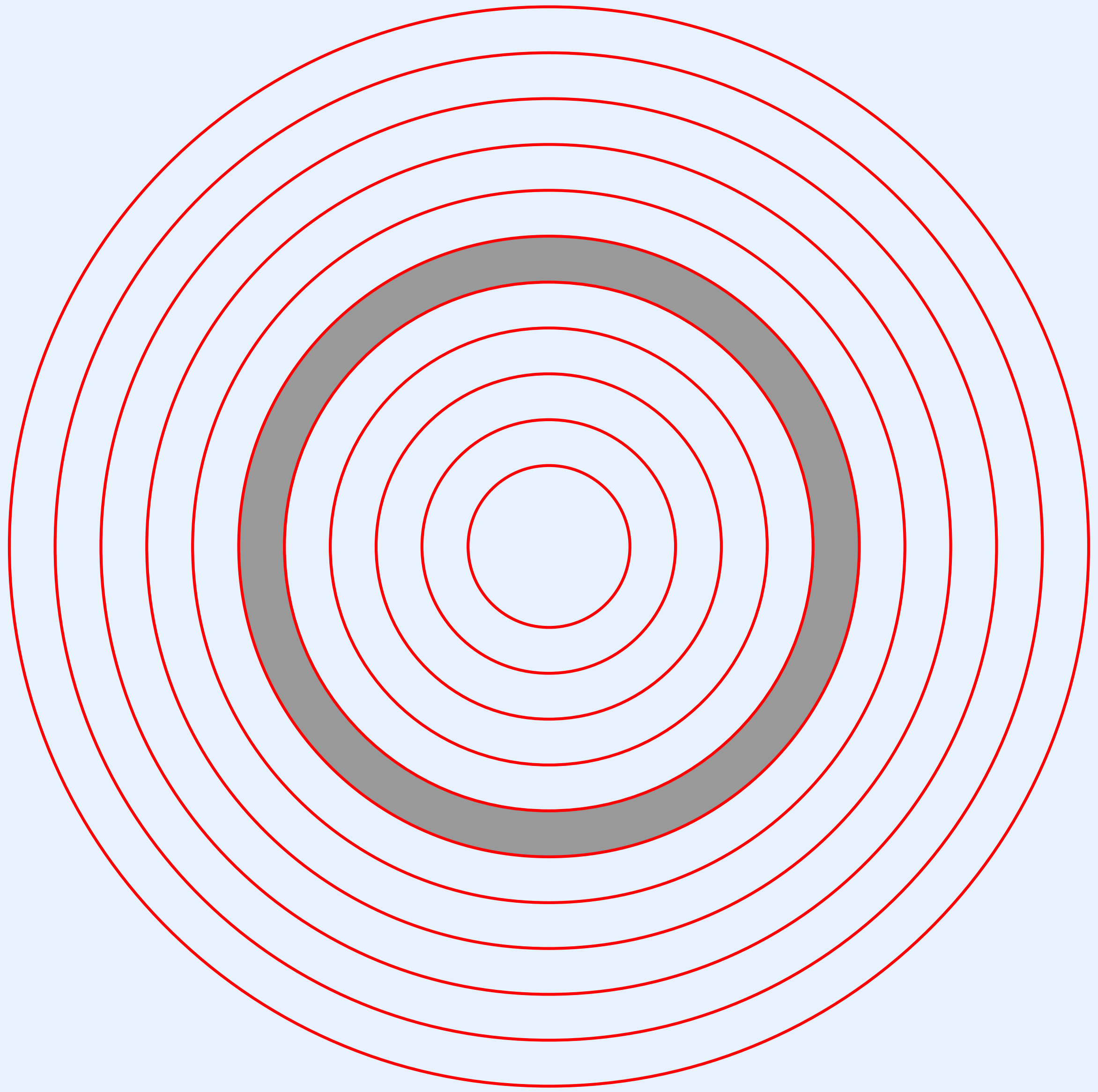
- The *device manager* in an operating system handles I/O
- Device-independent routines
  - Provide uniform interface
  - Define generic operations that must be mapped to device-specific functions
- Interrupt code
  - Consists of single dispatcher and handler for each device
  - Is executed by whatever process was running when interrupt occurred
- To accommodate null process, interrupt handler must leave executing process in *current* or *ready* states

Summary

- Rescheduling during interrupt is safe provided
  - Global data structures valid
  - No process explicitly enables interrupts
- Device driver functions
  - Are divided into upper-half and lower-half
  - Can use existing primitives to block and unblock processes

Clocks And
Clock Management

Location Of Clock Management In The Hierarchy



Various Types Of Clock Hardware Exist

- Processor clock (rate at which instructions execute)
- Real-time clock
  - Pulses regularly
  - Interrupts the processor on each pulse
  - Called *programmable* if rate can be controlled by OS
- Interval timer
  - The processor sets a timeout and the device interrupts after the specified time
  - Can be used to pulse regularly
  - May have an automatic restart mechanism

Timed Events

- Two types of timed events are important to an operating system
- A *preemption event*
  - Known as *timeslicing*
  - Guarantees that a given process cannot run forever
  - Switches the processor to another process
- A *sleep event*
  - Is requested by a process to delay for a specified time
  - The process resumes execution after the time passes

A Note About Timeslicing

Most applications are I/O bound, which means the application is likely to perform an operation that takes the process out of the current state before its timeslice expires.

Managing Timed Events

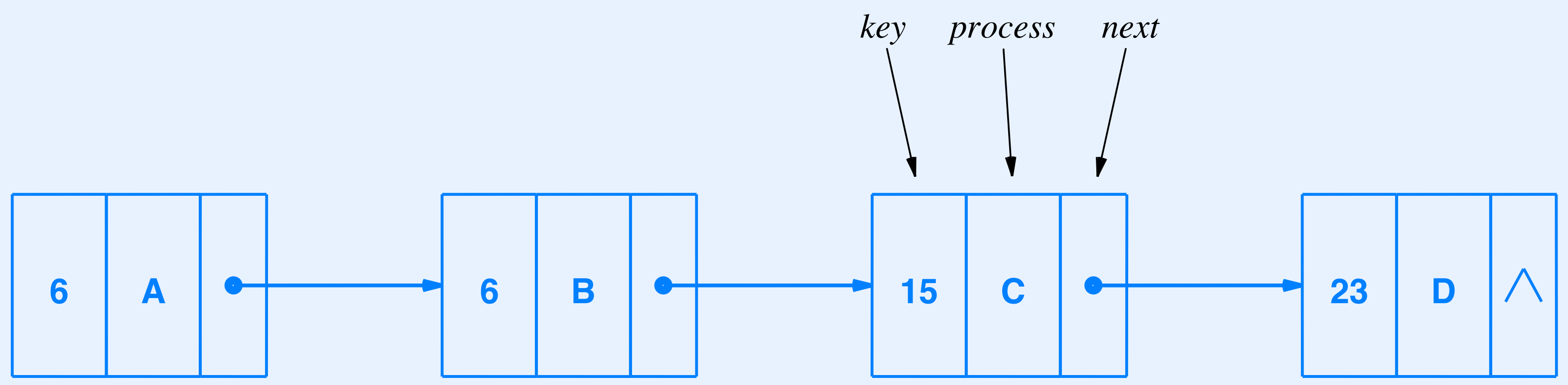
- The code must be efficient because
  - Clock interrupts occur frequently and continuously
  - More than one event may occur at a given time
  - The clock interrupt code should avoid searching a list
- An efficient mechanism
  - All timed events are kept on a list
  - The list is known as an *event queue*

The Delta List

- A data structure used for timed events
- Items on a delta list are ordered by the time they will occur
- Trick to make processing efficient: use *relative* times
- Implementation: the key in an item stores the difference (*delta*) between the time for the event and time for the previous event
- The key in first event stores the delta from “now”

Delta List Example

- Assume events for processes *A* through *D* will occur *6*, *12*, *27*, and *50* ticks from now
- The delta keys are *6*, *6*, *15*, and *23*



Real-time Clock Processing In Xinu

- The clock interrupt handler
  - Decrements the preemption counter and calls *resched* if the timeslice has expired
  - Processes the sleep queue
- The sleep queue
  - Is a delta list
  - Each item on the list is a sleeping process
- Global variable *sleepq* contains the ID of the sleep queue

Keys On The Xinu Sleep Queue

- Processes on *sleepq* are ordered by time at which they will awaken
- Each key tells the number of clock ticks that the process must delay beyond the preceding one on the list
- The relationship must be maintained whenever an item is inserted or deleted

Sleep Timer Resolution

- A process calls *sleep* to delay
- Question: what resolution should be used for sleep?
  - Humans typically think in seconds or minutes
  - Some applications may need millisecond accuracy (or more, if available)
- The tradeoff: using a high resolution, such as microseconds, means long delays will overflow a 32-bit integer

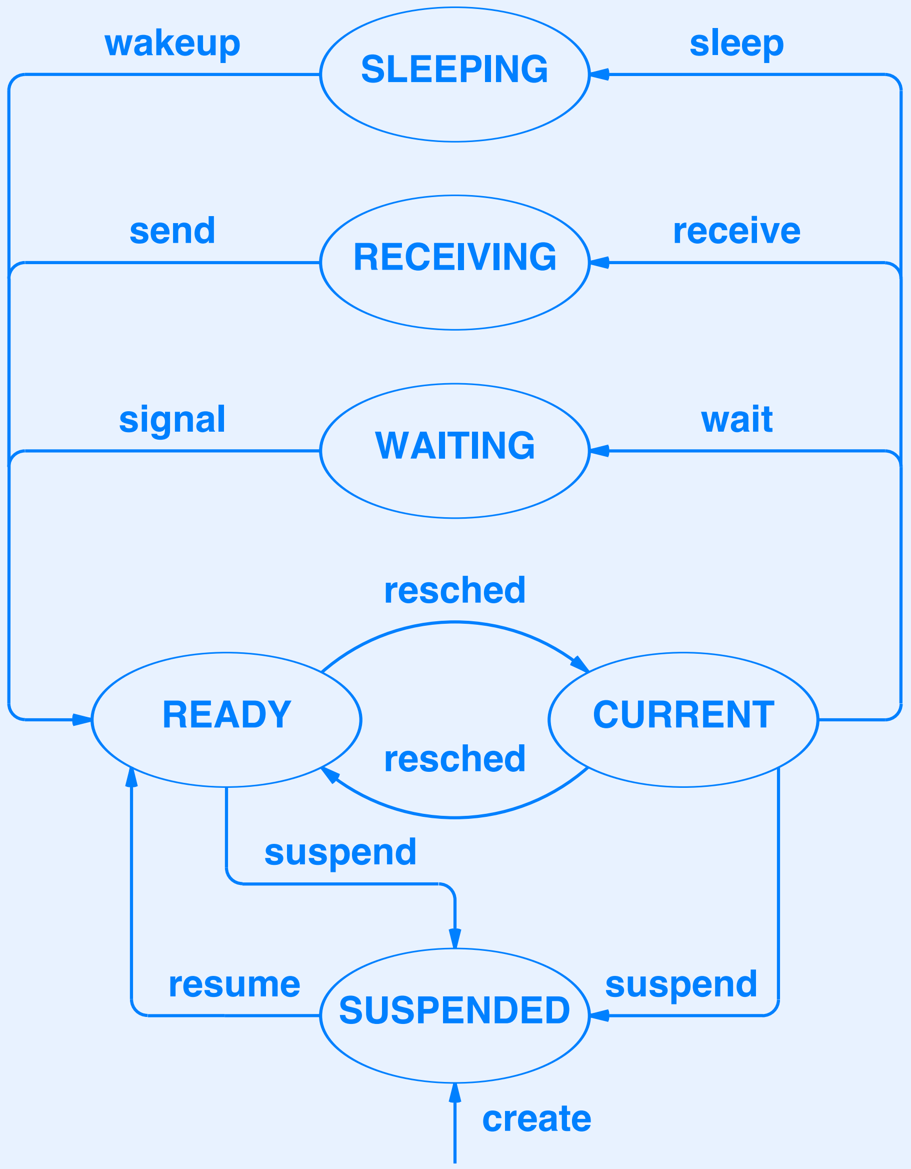
Xinu Sleep Primitives

- Xinu offers a set of functions to accommodate a range of possible resolutions

sleep	– the delay is given in seconds
sleep10	– the delay is given in tenths of seconds
sleep100	– the delay is given in hundredths of seconds
sleepms	– the delay is given in milliseconds

- The smallest resolution is milliseconds because the clock operates at a rate of one millisecond per tick

A New Process State For Sleeping Processes



Xinu Sleep Function (Part 1)

```
/* sleep.c - sleep sleepms */

#include <xinu.h>

#define MAXSECONDS      2147483         /* Max seconds per 32-bit msec  */

/*------------------------------------------------------------------------
 *  sleep  -  Delay the calling process n seconds
 *------------------------------------------------------------------------
 */
syscall sleep(
          int32 delay           /* Time to delay in seconds     */
        )
{
        if ( (delay < 0) || (delay > MAXSECONDS) ) {
                return SYSERR;
        }
        sleepms(1000*delay);
        return OK;
}
```

Xinu Sleep Function (Part 2)

```
/*------------------------------------------------------------------------
 *  sleepms  -  Delay the calling process n milliseconds
 *------------------------------------------------------------------------
 */
syscall sleepms(
          int32 delay                   /* Time to delay in msec.       */
        )
{
        intmask mask;                   /* Saved interrupt mask         */

        if (delay < 0) {
                return SYSERR;
        }

        if (delay == 0) {
                yield();
                return OK;
        }
```

Xinu Sleep Function (Part 3)

```
        /* Delay calling process */

        mask = disable();
        if (insertd(currpid, sleepq, delay) == SYSERR) {
                restore(mask);
                return SYSERR;
        }

        proctab[currpid].prstate = PR_SLEEP;
        resched();
        restore(mask);
        return OK;
}
```

Inserting An Item On Sleepq

- The current process calls *sleepms* or *sleep* to request a delay
- *Sleepms*
  - The underlying function that takes action
  - Inserts current process on *sleepq*
  - Calls *resched* to allow other processes to execute
- Method
  - Walk through *sleepq* (with interrupts disabled)
  - Find the place to insert the process
  - Adjust remaining keys as necessary

Xinu Insertd (Part 1)

```
/* insertd.c - insertd */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insertd  -  Insert a process in delta list using delay as the key
 *------------------------------------------------------------------------
 */
status  insertd(                        /* Assumes interrupts disabled  */
          pid32         pid,            /* ID of process to insert      */
          qid16         q,              /* ID of queue to use           */
          int32         key             /* Delay from "now" (in ms.)    */
        )
{
        int32   next;                   /* Runs through the delta list  */
        int32   prev;                   /* Follows next through the list*/

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }
```

Xinu Insertd (Part 2)

```
        prev = queuehead(q);
        next = queuetab[queuehead(q)].qnext;
        while ((next != queuetail(q)) && (queuetab[next].qkey <= key)) {
                key -= queuetab[next].qkey;
                prev = next;
                next = queuetab[next].qnext;
        }

        /* Insert new node between prev and next nodes */

        queuetab[pid].qnext = next;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[next].qprev = pid;
        if (next != queuetail(q)) {
                queuetab[next].qkey -= key;
        }

        return OK;
}
```

The Invariant Used During Sleepq Insertion

At any time during the search, both key **and** queuetab[next].qkey **specify a delay relative to the time at which the predecessor of the "next" process awakens.**

A Clock Interrupt Handler

- Updates the time-of-day (which counts seconds)
- Handles sleeping processes
  - Decrements the key of the first process on the sleep queue
  - Calls *wakeup* if the counter reaches zero
- Handles preemption
  - Decrements the preemption counter
  - Calls *resched* if the counter reaches zero

A Clock Interrupt Handler
(continued)

- When sleeping processes awaken
  - More than one process may awaken at a given time
  - The processes may not have the same priority
  - If the clock interrupt handler starts a process running immediately, a higher priority process may remain on the sleep queue, even if its time has expired
- Solution: *wakeup* awakens *all* processes that have zero time remaining before allowing any of them to run

Summary

- Two types of timed events are especially important in an operating system
  - Preemption
  - Process delay (sleep)
- A delta list provides an elegant and efficient data structure to store a set of sleeping processes
- If multiple processes awaken at the same time, rescheduling must be deferred until all have been made ready
- *Recvtime* allows a process to wait a specified time for a message to arrive

Questions?

Achieving Device-independent I/O

- Define a set of abstract operations
- Build a function for each operation
- Have each function include an argument that a programmer can use to specify a particular device
- Arrange an efficient way to map generic operation onto code for a specific device

Xinu's Device-Independent I/O Primitives

- Follow the Unix open-read-write-close paradigm

init	–	initialize a device (invoked once, at system startup)
open	–	make a device ready for use
close	–	terminate use of a device
read	–	input arbitrary data from a device
write	–	output arbitrary data to a device
getc	–	input a single character from a device
putc	–	output a single character to a device
seek	–	position a device (primarily a disk)
control	–	control a device and / or its driver

- Note: some abstract functions may not apply to a given device

Implementation Of Device-Independent I/O In Xinu

- An application process
  - Makes calls to device-independent functions (e.g., *read*)
  - Supplies the device ID as parameter (e.g., ETHER)
- The device-independent I/O function
  - Uses the device ID to identify the correct hardware device
  - Invokes the appropriate device-specific function to perform the specified operation (e.g., *ethread* to read from an Ethernet)

Mapping A Generic I/O Function To A Device-Specific Function

- The mapping must be extremely efficient
- Solution: use a two-dimensional array
- The array is called a *device switch table*
  - A kernel data structure that is initialized when system loaded
  - Each row in the array corresponds to one device
  - Each column in the array corresponds to an operation
- An entry in the table points to a function to be called to perform the operation
- The device ID is chosen to be a index into rows of the table

Entries In The Device Switch Table

- Each device-independent operation is generic
- However, a given operation may not make sense for a given device
  - *Seek* on keyboard, network, or display screen
  - *Close* on a mouse
- How should I/O functions handle the exceptions?

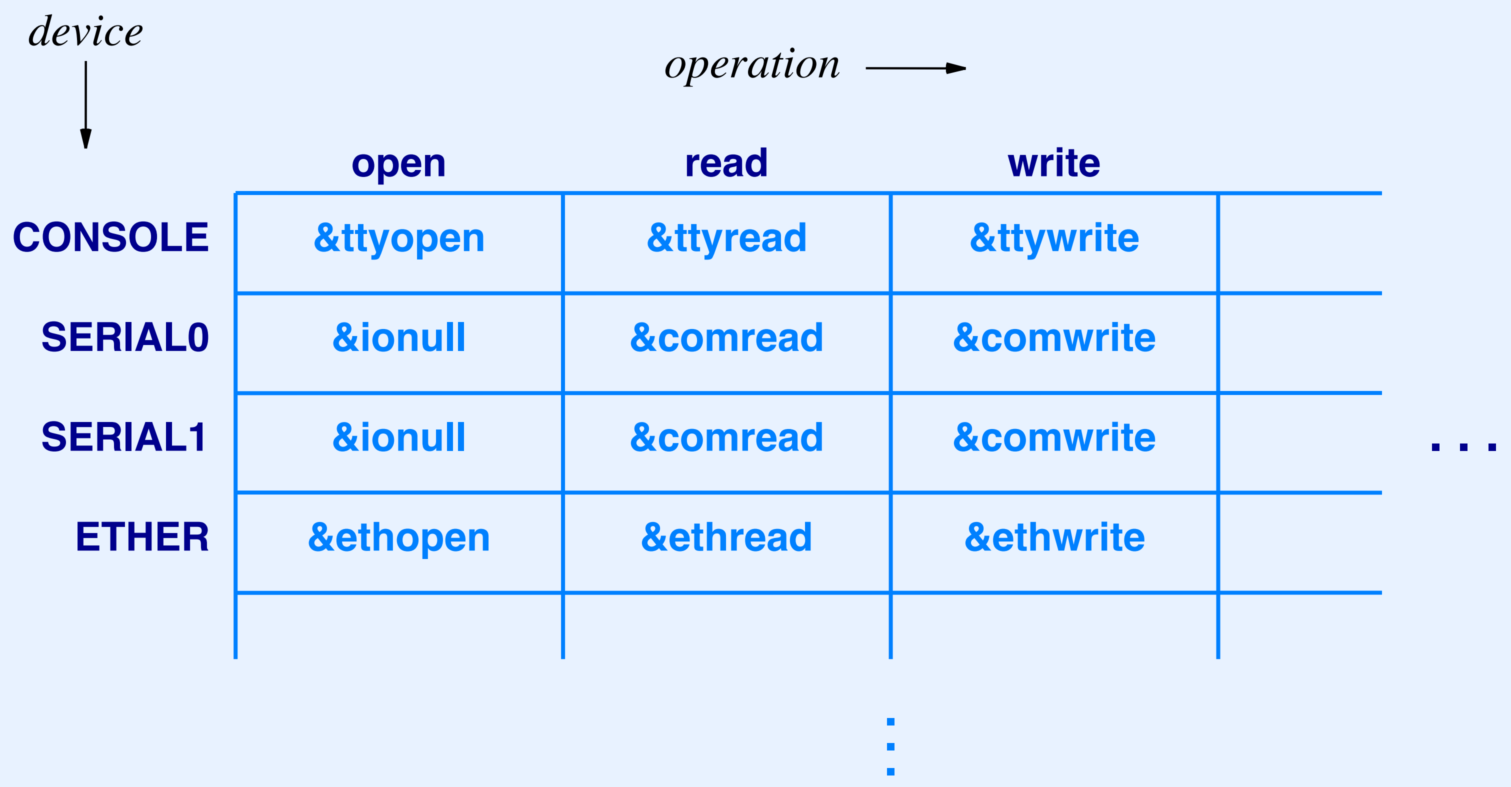
Entries In The Device Switch Table

- Each device-independent operation is generic
- However, a given operation may not make sense for a given device
  - *Seek* on keyboard, network, or display screen
  - *Close* on a mouse
- How should I/O functions handle the exceptions?
- To avoid special cases
  - Fill in *all* entries in the table
  - Place a valid function pointer in each entry
  - Create special functions for cases where an operation does not apply to a specific device

Special Entries Used In The Device Switch Table

- *ionull*
  - Used for an innocuous operation (e.g., *open* for a device that does not really require opening)
  - Simply returns *OK*
- *ioerr*
  - Used for an incorrect operation (e.g., *putc* on disk)
  - Simply returns *SYSERR*

Illustration Of Device Switch Table



	open	read	write
CONSOLE	&ttyopen	&ttyread	&ttywrite
SERIAL0	&ionull	&comread	&comwrite
SERIAL1	&ionull	&comread	&comwrite
ETHER	&ethopen	&ethread	&ethwrite

- Each row corresponds to a device and each column corresponds to an operation
- An entry specifies the address of a function to invoke
- The example uses *ionull* for *open* on devices *SERIAL0* and *SERIAL1*

Replicated Devices And Device Drivers

- A computer may contain multiple copies of a given physical device
- Examples
  - Two Ethernet NICs
  - Two USB devices
- Goal have one copy of device driver code for the device and use the code with multiple devices

Parameterized Device Drivers

- A device driver must
  - Know which physical copy of a device to use
  - Keep information about the device separate from information for other copies
- To accommodate multiple copies of a device
  - Assign each instance a unique minor number (0, 1, 2, ...) known as its *minor device number*
  - Store the minor device number in the device switch table

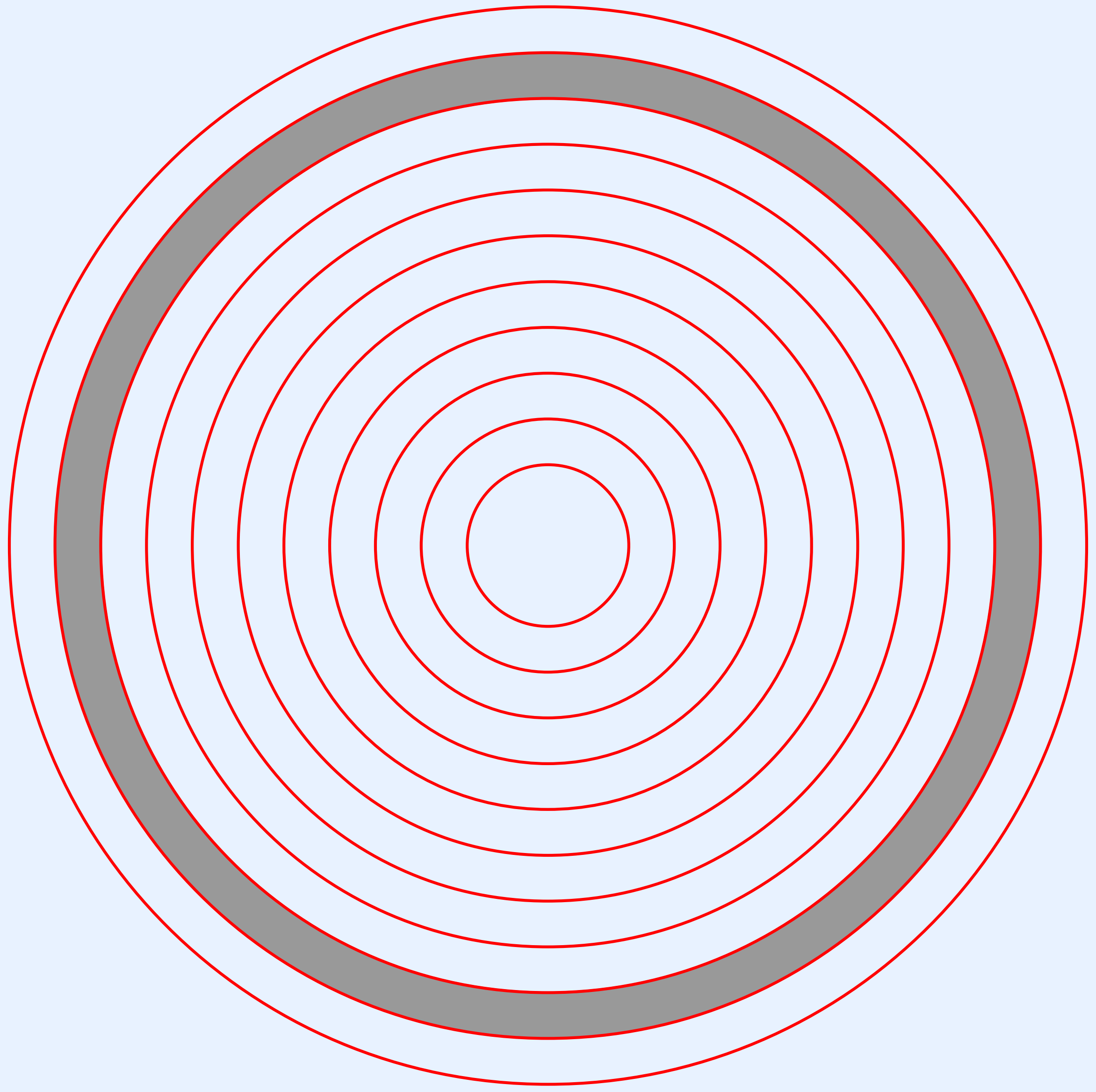
Device Names

- Previous examples have shown examples of device names used in code (e.g., *CONSOLE*, *SERIAL0*, *SERIAL1*, *ETHER*)
- The device switch table is an array, and each device name is really an index into the array
- How does the system know how many rows to allocate in the table?
- How are unique values assigned to device names?
- How are minor device numbers assigned for replicated devices?
- Answer: a configuration program takes device information as input, including names to be used for devices, and generates the definitions and the device switch table entries automatically
- We will see more details later

Module XII

File Systems

Location Of File Systems In The Hierarchy



Purpose Of A File System

- Manages data on nonvolatile storage
- Allows user to name and manipulate semi-permanent files
- Provides mechanisms used to organize files directories (aka folders)
- Stores metadata associated with a file
  - Size
  - Ownership
  - Access rights
  - Location on the storage system

Aspects Of A File System

- The relatively straightforward aspect
  - Allow applications to read and write data to files on local storage
- More difficult aspects
  - Control sharing on a multiuser system
  - Handle caching (important for efficiency)
  - Manage a distributed file system that allows applications on many computers to create, access, and change files

Sharing

- The most difficult aspects of file sharing revolve around the semantics of concurrent access
- An example: consider three applications that all have access to a given file
  - Application 1 opens the file, and is therefore positioned at byte 0
  - Before Application 1 reads or writes the file, Application 2 opens the file and reads 10 bytes
- At that point in time, Application 3 deletes the file
- Application 1 tries to read from the file
- What should happen?

File Sharing In A Unix System

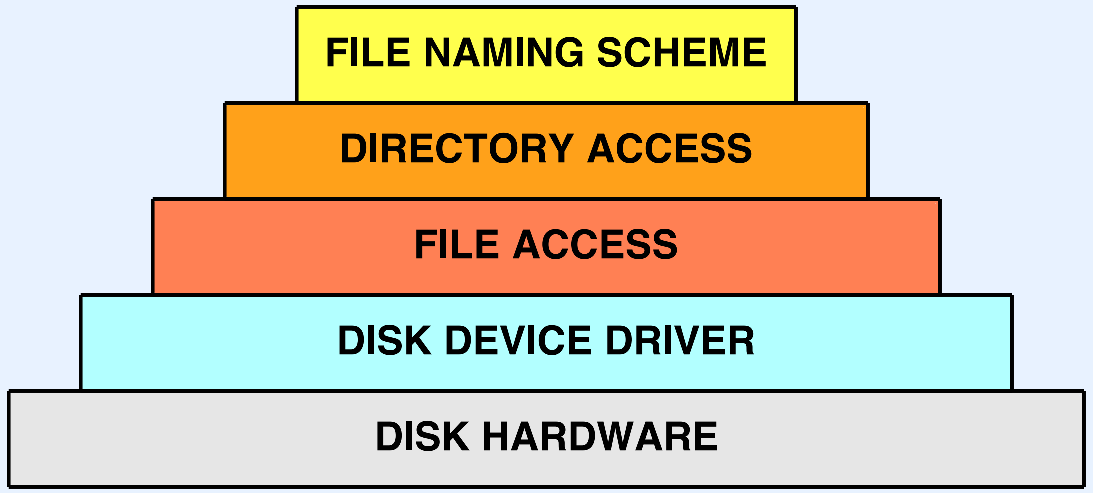
- What happens if
  - A file is deleted after it has been opened?
  - File permissions change *after* a file has been opened?
  - A file is moved to a new directory *after* it has been opened?
  - File ownership changes *after* a file has been opened?
- What happens to the file position in open files after a *fork()*?
- What happens if two processes open a file and concurrently write data
  - To different locations?
  - To the same location?

Sharing In A Unix System (Answers)

- Permissions are only checked when a file is opened
- Each process has its own position for a file; if two processes access the same file, chancing the position in one does not affect the position in the other
- In Unix, a file is separate from the directory entry for the file
  - Removing a file from a directory does not delete the file itself
  - When a file is removed, actual deletion is deferred until the last process that has opened the file closes it
  - Consequence: even if a file has been removed from the directory system, processes that have it open will be able to read/write it

File System Internals

The Conceptual Organization Of A File System



- Each level adds functionality
- An implementation may integrate multiple levels

The Function Of Each Level Of Software

- Naming level
  - Deals with name syntax
  - May determine the location of a file (e.g., whether file is local or remote)
- Directory access level
  - Maps a name to a file object
  - May be completely separate from naming or integrated
- File access level
  - Implements basic operations on files
  - Includes creation and deletion as well as reading and writing
- Disk driver level
  - Performs block I/O operations on a specific type of hardware

Two Fundamental Philosophies Have Been Used

- Typed files (MVS)
  - The operating system defines a set of types that specify file format / contents
  - A user chooses a type when creating file
  - The type determines operations that are allowed
- Untyped files (Unix)
  - A file is a “sequence of bytes”
  - The operating system does not understand contents, format, or structure
  - A small set of operations apply to all files

An Assessment Of Typed Files

- Pros
  - Types protect user from application / file mismatch
  - File access mechanisms can be optimized
  - A programmer can choose whichever file representation is best for a given need
- Cons
  - Extant types may not match new applications
  - It is extremely difficult to add a new file type
  - No “generic” commands can be written (e.g., *od*)

An Assessment Of Untyped Files

- Pros
  - Untyped files permit generic commands and tools to be used
  - The file system design is separate from the applications and the structure of data they use
  - There is no need to change the operating system when new applications need a different file format
- Cons
  - The operating system cannot prevent mismatch errors (e.g., *cat a.out* garbles the screen)
  - The file system may not be optimal for any particular application
  - The operating system owner does not know how files are being used

An Example Of Operations For Untyped Files

- The classic open-close-read-write interface is defined by Unix
- Conceptually, there are eight main functions

create	– start a fresh file object
destroy	– remove existing file
open	– provide access path to file
close	– remove access path
read	– transfer data from file to application
write	– transfer data from application to file
seek	– move to a specified file position
control	– miscellaneous operations (e.g., change protection modes)

File Allocation Choices

- How should files be allocated?
- Static allocation
  - The historic approach
  - Space is allocated before the file is used
  - The file size cannot change
  - Easy to implement; difficult to use
- Dynamic allocation
  - Files grow as needed
  - Easy to use; more difficult to implement
  - Has the potential for starvation (one file takes all the space)

The Desired Cost Of File Operations

- Read / write
  - The most common operations performed
  - Provide sequential data transfer
  - The desired cost is *0(t)*, where *t* is size of transfer
- Move to an arbitrary position in the file
  - Needed for random access
  - Not often used
  - The desired cost is *0(log n)*, where *n* is file size

A Few Factors That Affect File System Design

- Many files are small; few are large
- Most access is sequential; random access is uncommon
- Overhead is important (e.g., the latency required to open a file and move to the first byte)
- Clever data structures are needed

The Underlying Hardware

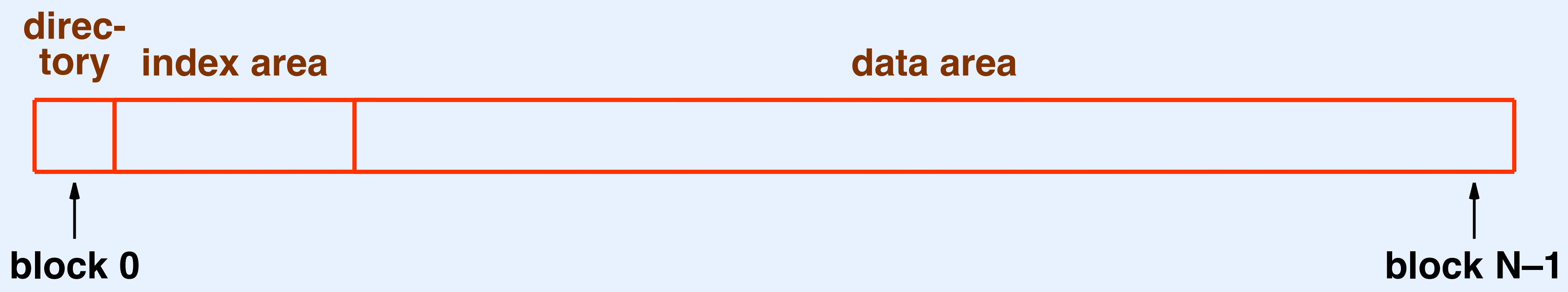
- Most files systems assume a traditional disk
  - The disk has fixed-size sectors that are numbered 0, 1, 2, ...
  - The standard sector size is 512 bytes, even for solid-state disks
- The disk interface
  - The hardware can only transfer (read or write) a complete block
  - The hardware provides random access by sector number
- An important point, especially for metadata

Disk hardware cannot perform partial-block transfers.

An Example File System

The Xinu File System

- Views the underlying disk as an array of disk blocks, where each block is 512 bytes
- Takes a simplistic approach by partitioning a disk into three areas
  - Directory area (one block)
  - File index area (a small number of blocks)
  - Data area (the rest of the disk)



The Data Area

- The file system treats the entire data area as an array of *data blocks*
  - Data blocks are numbered from *0* to $D-1$
  - Each data block is 512 bytes long, and occupies one physical disk sector
  - Blocks in the data area only store file contents
  - Currently unused data blocks are linked on a free list

The Index Area

- The file system treats the index area as an array of *index blocks* (*i-blocks*)
  - Index blocks are numbered from *0* to $I-1$
  - Because an i-block is smaller than 512 bytes, multiple blocks occupy a given disk sector
  - Each index block stores
    - Pointers to data blocks
    - The offset in file of first data byte indexed by the i-block
  - Currently unused index blocks are linked on free list

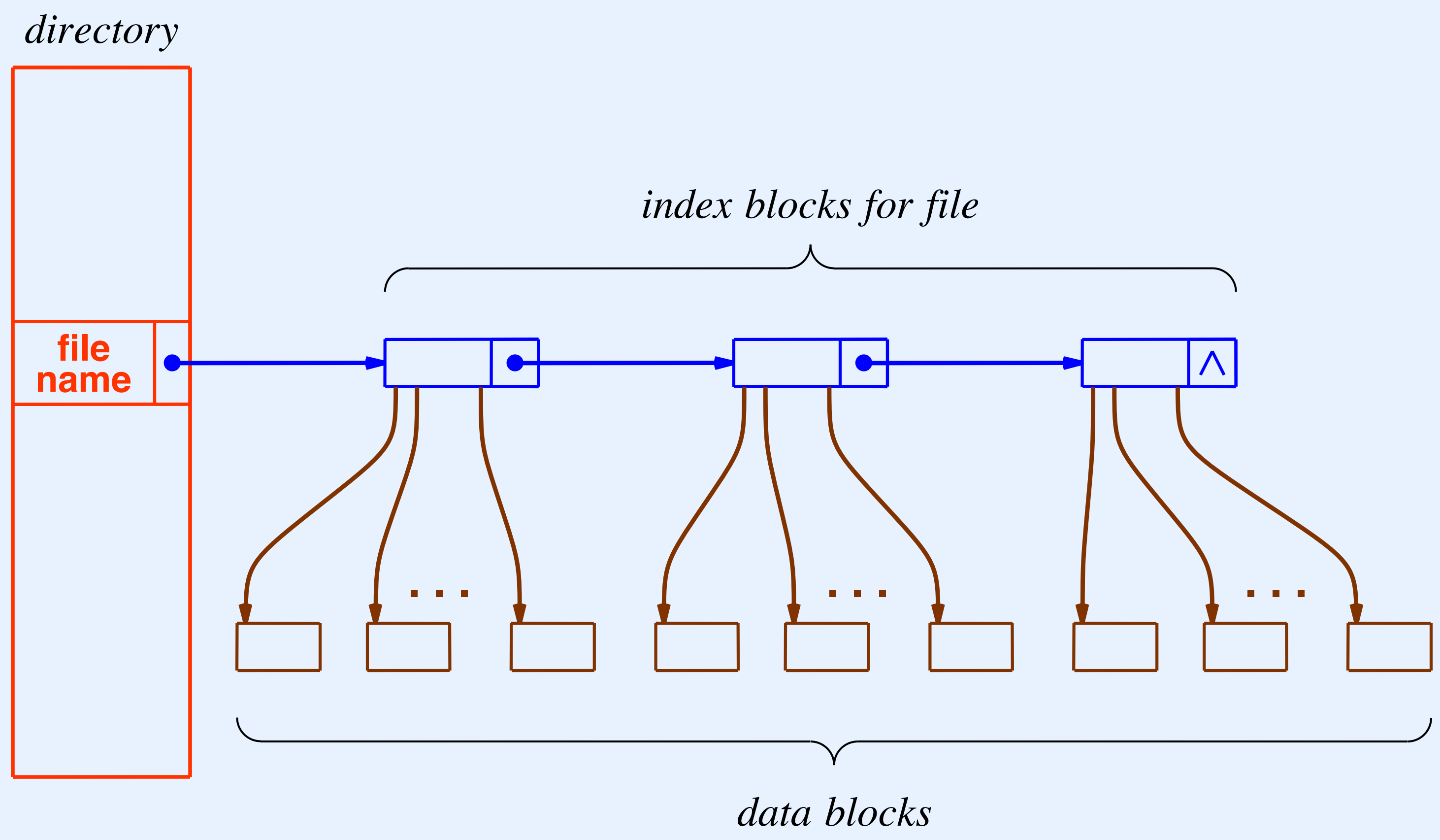
The Directory Area

- The file system treats the directory as an array of pairs:

  (file name, first index block for the file)

- Conceptually, a directory entry provides a mapping from a name to the actual file
- The entire directory occupies the first physical sector on the disk
- The directory is limited, but has sufficient size for a small embedded system

The Xinu File System Data Structure



- Index blocks for a file are linked together, and each index block points to a set of data blocks
- The figure is not drawn to scale (a data block is actually larger than an index block)

Important Concept

Within the operating system, a file is referenced by the i-block number of the first index block, not by name.

(A name is only needed when opening a file.)

File Access In Xinu

- In Xinu, everything is a device
- The file access paradigm uses
  - A set of “file pseudo devices” defined when system configured
  - A single pseudo device, *LFILESYS*, is used to open files, and a set of *K* additional pseudo devices are used for data transfer
  - The device driver for a data transfer pseudo device implements *read* and *write* operations
  - The device driver for the *LFILESYS* pseudo device implements *open*

Using The Xinu Local File System

- To open a file, an application calls

  desc = open(LFILESYS, name, mode);

- The call sets *desc* to the device descriptor of one of the data transfer pseudo devices, and associates the pseudo device with the named file

- The application calls *read*, *write*, and possibly *seek*, passing *desc* as the device descriptor

- The device driver for the data transfer pseudo device performs *read* and *write* operations on the file that has been opened

- When it finishes using the file, the application calls *close*

The Xinu File Access Paradigm

- When an application opens a file, the code takes the following steps
  - Obtain a copy of the directory from disk if it is not already in memory
  - Search the directory to find the i-block number for the file
  - Allocate a data transfer pseudo-device for the application to use
  - Set the initial file position to zero
  - Obtain the data block that contains byte zero of the file
    - Read the first i-block to find first data block ID
    - Read the first data block into a buffer
    - Set the byte pointer to first byte in the buffer

The Xinu File Access Paradigm
(continued)

- When the application reads or writes data
  - If the current file position has moved outside the current data block, fetch the data block for the current position
  - Read or write data from/to the current data block buffer, incrementing the buffer position for each byte
- Note: even if all data in a buffer is consumed, the file system does not fetch the “next” data block until it is needed

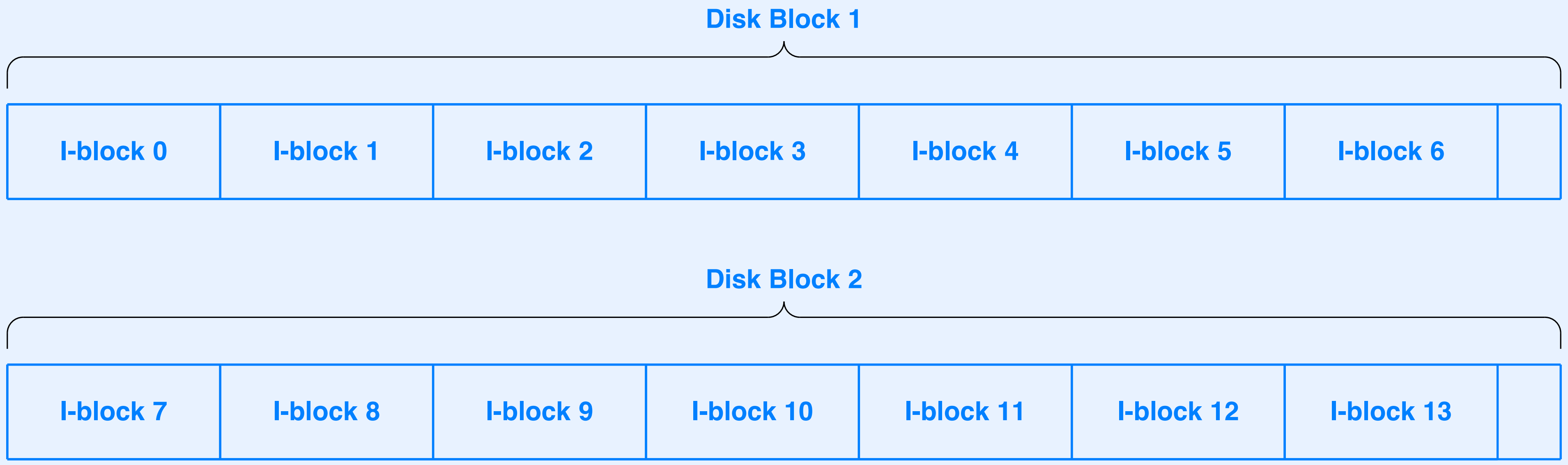
Concurrent Access To A Shared File

- The chief design difficulty: shared file position
- Ambiguity can arises when
  - A set of processes open a file for reading
  - Other processes open the same file for writing
  - Each process issues *read* and *write* calls without specifying a file position
  - The file position depends on when processes execute
- To avoid the problem, Xinu prohibits concurrent access
  - Only one active open can exist on a given file at a given time
  - A programmer must choose how to share a file among processes

Index Block Access And Disk I/O

- Recall
  - The hardware always transfers a complete physical block
  - An index block is smaller than a physical block
- To store index block number *i*
  - Map *i* to a physical block, *p*
  - Read disk block *p*
  - Copy i-block *i* to the correct position in *p*
  - Write physical block *p* back to disk
- Unix i-nodes use the same paradigm (discussed later)

Illustration Of Index Blocks In A Disk Block



- Xinu stores seven I-blocks in each disk block
- To find the disk block number in which an I-block resides, divide the I-block number by 7 (integer arithmetic) and add 1
- To find the byte position within a disk block, calculate *r*, the remainder of dividing the I-block number by 7, and multiply *r* times the size of an I-block

Questions

- What should be cached?
  - Individual index blocks?
  - The disk block in which an index block is contained?
- How can the Xinu file system be extended to
  - Allow concurrent file access?
  - Use a file to store the directory?
  - Provide better caching?

The Unix™ File System

The Unix File Access Paradigm

- The operating system maintains an *open file table*
  - Internal to the operating system
  - One entry for each open file
  - Uses a reference count for concurrent access
- Each process has a *file descriptor table*
  - An array where each entry points to an entry in the open file table
  - Each entry contains a position in the file for the process
- A file descriptor
  - Is a small integer returned by *open*
  - Provides an index into the process's file descriptor table
  - Is meaningless outside the process

The Generalization Of Unix File Descriptors

- Unix file descriptors provide access to mechanisms other than local files
- A descriptor can refer to
  - An I/O device
  - A network socket
  - A remote file
- The open-read-write-close paradigm is used for all descriptors

Inheritance, Sharing, And Reference Counts

- Recall: a reference count is kept for each entry in the open file table
- The reference count is initialized to *1* when a file is first opened
- When a process uses *fork* to create a new process
  - The new process gains a copy of each descriptor
  - The reference count in the open file table is incremented
- When a process calls *close*, the reference count in the open file table is decremented, and the entry in the process's file descriptor table is released for reuse
- When a reference count in open file table reaches zero, the entry is released
- Unix closes all open descriptors automatically when a process exits, so the above steps are followed whether a process explicitly closes a file or merely exits

Unix File System Properties

- The design accommodates both small and large files
- It has highly tuned access mechanisms
- The overhead is logarithmic in the size of allocated files
- It provides a hierarchical directory system (like *MULTICS*)
- The data structure uses index nodes (*i-nodes*) and data blocks
- An interesting twist: directories are actually files!

Embedding directory in a file is possible because inside the operating system, files are known by their index rather than by name

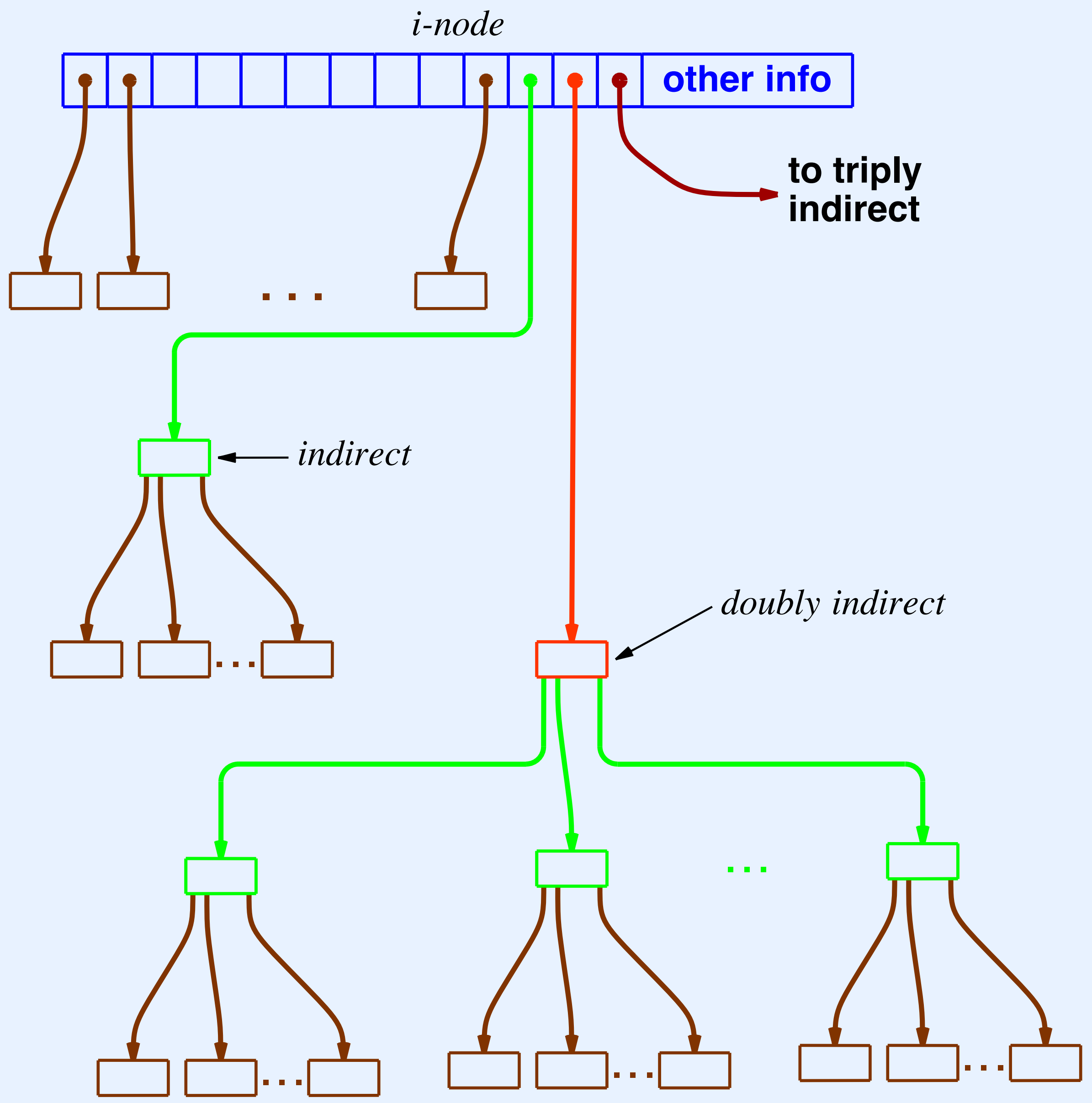
The Contents Of A Unix I-node

- The owner’s user ID
- A group ID
- The current file size
- The number of links (how many directory entries point to the file)
- Permissions (i.e., read, write, and execute protection bits)
- Timestamps for creation, last access, and last update
- A set of 13 pointers that lead to the data blocks of the file

The 13 Pointers In An I-node

- Ten *direct* pointers each point to a data block
- One *indirect* pointer points to a block of *128* pointers to data blocks
- One *doubly indirect* pointer points to a block of *128* indirect pointers
- One *triply indirect* pointer points to a block of *128* doubly indirect pointers
- The scheme accommodates
  – Rapid access to small files
  – Fairly rapid access to intermediate files
  – Reasonable access to large files

Illustration of Pointers In A Unix I-node



Unix File Sizes

- The data accessible using direct pointers
  - Up to 5,120 bytes
- The data accessible via the indirect pointer
  - Up to 70,656 bytes
- The data accessible via the doubly indirect pointer
  - Up to 8,459,264 bytes
- The data accessible via the triply indirect pointer
  - 1,082,201,088 bytes
- Note: maximum size file seemed immense when Unix was designed; FreeBSD increased sizes to use 64-bit pointers, making the maximum size 8ZB.

Unix Hierarchical Directory Mechanism

- Provides the scheme used to organize file names
- Was derived from the *MULTICS* system
- Allows a hierarchy of *directories* (aka *folders*)
- A given directory can contain
  - Files
  - Subdirectories
- The top-level directory is called the *root*

A Unix File Name

- A name is a text string
- Each name corresponds to a specific file
- The name specifies a *path* through the hierarchy
- Example
  - / u / u5 / dec / stuff
- Two special names are found in each directory
  - The current directory is named “**.**”
  - The parent directory is named “**. .**”

Unix Hierarchical Directory Implementation

- A directory is implemented as a file
  - Files that contain directories have a special file type (*directory*)
  - Each directory contains a set of triples

(type, file name, i-node number)

- The *root directory* is always at i-node *2*
- A path is resolved one component at a time, starting with i-node *2*
- The directory system is general enough for an arbitrary graph; restrictions are added to simplify administration

Advantages Of Unix File System

- Imposes very little overhead for sequential access
- Allows random access to specified position
  - Especially fast search in a short file
  - Logarithmic search in a large files
- Files can grow as needed
- Directories can grow as needed
- Economy of mechanism is achieved because directories are embedded in files

Disadvantages Of Unix File System

- The protections are restricted to three sets: *owner*, *group*, and *other*
- The single access mechanism may not be optimized for any particular purpose
- The data structures can be corrupted during system crash
- The integration of directories into the file system makes a distributed file system more difficult

Caching

- Recall that

The most difficult aspects of file system design arise from the tension between efficient concurrent access, caching, and the need to guarantee consistency on disk.

Caching, Locking Granularity, And Efficiency Questions

- To be efficient, a file system must cache data items in memory
- To guarantee mutual exclusion, cached items must be locked
- What granularity of locking works best?
  - Should an entire directory be locked?
  - Should individual i-nodes be locked?
  - Should individual disk blocks be locked?
- Does it make sense to lock a disk block that contains i-nodes from multiple files?
- Can locking at the level of disk blocks lead to a deadlock?

Caching, Locking Granularity, And Efficiency Questions (continued)

- A file system cannot afford to write every change to disk immediately
- When should updates be made?
  - Periodically?
  - After a significant change?
- How can a file system maintain consistency on disk?
  - Must an i-node be written first?
  - When should the i-node free list be updated on disk?
  - In which order should indirect blocks be written to disk?

The Importance Of Caching

- An i-node cache eliminates the need to reread the index
- A disk block cache tends to keep the directories near the root in memory because they are searched often
- Caching provides dramatic performance improvements

Memory-mapped Files

- The idea
  - Map a file into part of a process's virtual address space
  - Allow the process to manipulate the entire file as an array of bytes in memory
  - Use the virtual memory paging system to fetch pages of the file from disk when they are needed
- The approach works best with a large virtual address space (e.g., a 64-bit address space)

File System Partitions

- The idea: divide a physical disk into multiple areas, and place a separate file system in each area
- Glue all partitions together by using *mount* to link all partitions into a single, unified directory hierarchy
- Motivation
  - Higher reliability: fewer files tend to be lost in a crash
  - Higher performance: keeps i-nodes closer to data blocks, which speeds up performance on an electromechanical disk
  - Lower maintenance cost: a smaller file system is much faster to check or repair

Summary

- A file system manages data on non-volatile storage
- The functionality includes
  - A naming mechanism
  - A directory system
  - Individual file access
- The Xinu file system contains files and a directory
- Files are implemented with index blocks that point to data blocks
- Unix embeds directories in files, a technique that is possible because files are identified by i-node numbers
- Caching is essential for high performance
- Memory-mapped files are feasible, especially with a large virtual address space

Questions?