CS354
Operating Systems

Prof. Douglas Comer

Computer Science Department
Purdue University

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

PART I

Introduction

Topic And Scope

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

What We Will Cover

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

What You Will Learn

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

What We Will NOT Cover

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

A Brief History Of Computing

A Brief History Of Computing

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

A Brief History Of Computing
(continued)

- 1960s: What are the best abstractions to use?
  - A move to general-purpose hardware
  - Instruction set architectures emerges
  - Families of computers (e.g., IBM System/360)
  - I/O independent of specific hardware details
  - High-level programming languages created (e.g., parameterized functions, data types, and recursion)
  - Each program written to solve a problem
  - Researchers create operating system abstractions (Multics)

A Brief History Of Computing
(continued)

- 1970s – 1980s: How can computers be interconnected?
  - Operating system abstractions adopted widely (Unix and other systems)
  - Standards created for data representation (e.g., floating point)
  - Computer networking and the Internet emerge
  - Parallel processing studied
  - Programmer must accommodate multiple machines (e.g., byte order)

A Brief History Of Computing
(continued)

- 1990s – 2000s: How can large software systems be built?
  - Cluster hardware and grid used for scientific processing
  - Data centers devised for Web services
  - Content distribution and search become prominent applications
  - Distributed computing techniques emerge (e.g., map-reduce)
  - Shift to embedded applications and cloud paradigm

General Principles

General Principles

- Hardware is ugly

General Principles

- Hardware is ugly
- Abstractions are beautiful

General Principles

- Hardware is ugly
- Abstractions are beautiful
- Everything is distributed

General Principles

- Hardware is ugly
- Abstractions are beautiful
- Everything is distributed
- The gap between hardware and users is huge

Consequences And Conclusions

Consequences And Conclusions

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

Consequences And Conclusions

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

Consequences And Conclusions

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

Consequences And Conclusions

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

Consequences And Conclusions

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

Required Background And Prerequisites

Background Needed

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

Background Needed
(continued)

- C programming
  - At least one nontrivial program
  - Comfortable with low-level constructs (e.g., bit manipulation and pointers)
- Working knowledge of basic UNIX tools
  - Text editor (e.g., emacs)
  - Compiler / linker / loader
  - Make and Makefiles
- Desire to learn

CONSIDERING DROPPING?

How We Will Proceed

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

Questions?

A FEW THINGS
TO THINK ABOUT

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

– Antoine de Saint-Exupery

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

– source unknown

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

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

PART 2

Organization Of An Operating System

What Is An Operating System?

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

Example Services An OS Supplies

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

What An Operating System Is NOT

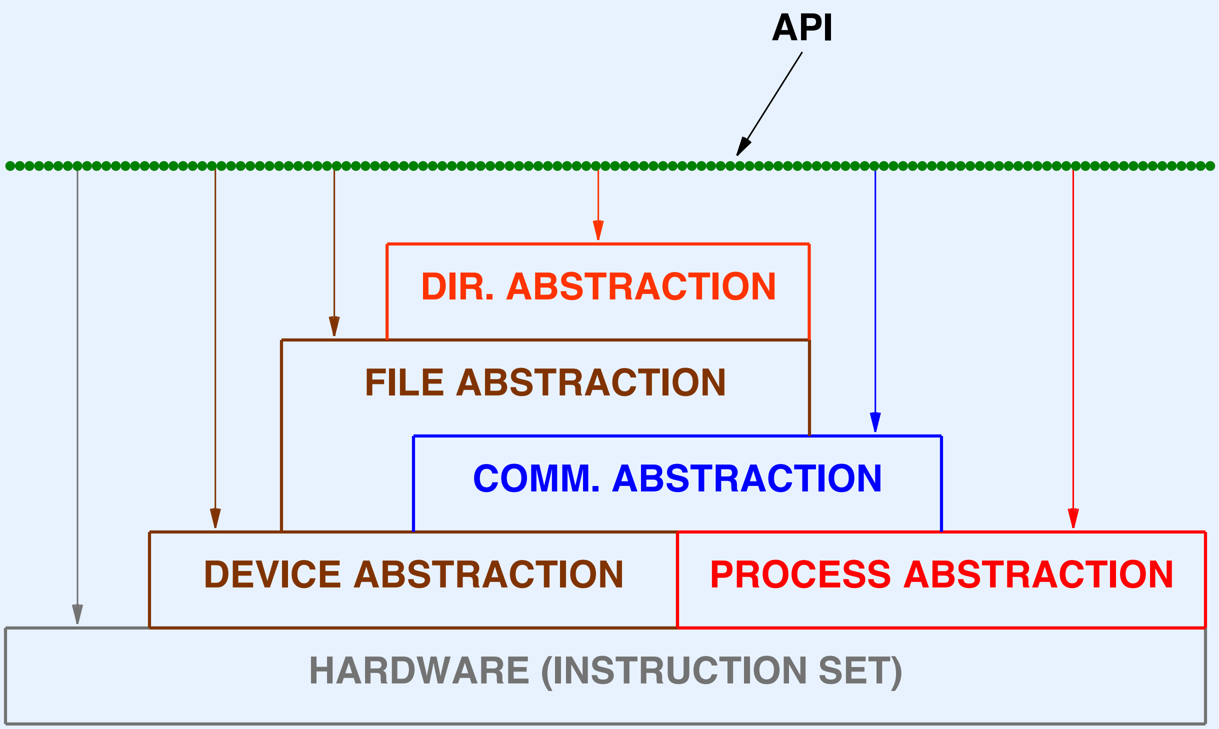
- Hardware
- Language
- Compiler
- Windowing system or browser
- Command interpreter
- Library of utility functions
- Graphical desktop

AN OPERATING SYSTEM FROM THE OUTSIDE

The System Interface

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

OS Abstractions And Application Interface



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

Interface To System Services

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

Interface To System Services
(continued)

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

Example System Call In Xinu: Write A Character On Console

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

#include <xinu.h>

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

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

OS Services And System Calls

- Each OS accessed through system call mechanism
- Most services employ a set of several system calls
- Examples
  - System may include 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* terminate use
- Internally, I/O functions coordinate

Questions

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

Concurrent Processing

- Fundamental concept that dominates OS design
- *Real concurrency* achieved by hardware
  - I/O devices operate at same time as CPU
  - Multiple CPUs/cores each operating at the same time
- *Apparent concurrency* achieved with multiprogramming
  - Multiple programs appear to operate simultaneously

Multiprogramming

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

Terminology Used With Multiprogramming

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

Process

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

Example Of Process Creation In Xinu

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

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) );
}

void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}

void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

Difference Between Function Call And Process Creation

- Normal function call
  - Synchronous execution
  - Single computation
- System call used to create process
  - Asynchronous execution
  - Two processes proceed after call

Distinction Between A Program And A Process

- Sequential program
  - Set of functions executed by a single thread of control
- Process
  - Computational abstraction not usually part of the programming language
  - Created independent of code that is executed
  - Multiple processes can execute the same code concurrently

Example Of Two Processes Sharing Code

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

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                      /* character to emit continuously */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

- Note: two processes execute *sndch* concurrently

Storage Allocation When Multiple Processes Execute

- Various memory models exist for multiprogrammed environments
- Each process requires its own
  - Run-time stack for procedure calls
  - Storage for local variables
- A process *may* have private heap storage as well

Consequence For Programmers

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

AN OPERATING SYSTEM FROM THE INSIDE

Operating System

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

Building An Operating System

Building An Operating System

- The intellectual challenge comes from the "system", not from individual pieces
- Structured design is needed
- It can be difficult to understand the consequences of choices
- We will use a hierarchical microkernel design to help control complexity

Major OS Components

- Process Manager
- Memory Manager
- Device Manger
- Clock (time) Manager
- File Manager
- Interprocess Communication
- Intermachine Communication
- Accounting

Multilevel Structure

- The design paradigm we will use
- Organizes components
- Controls interactions among subsystems
- Allows a system to be understood and built incrementally

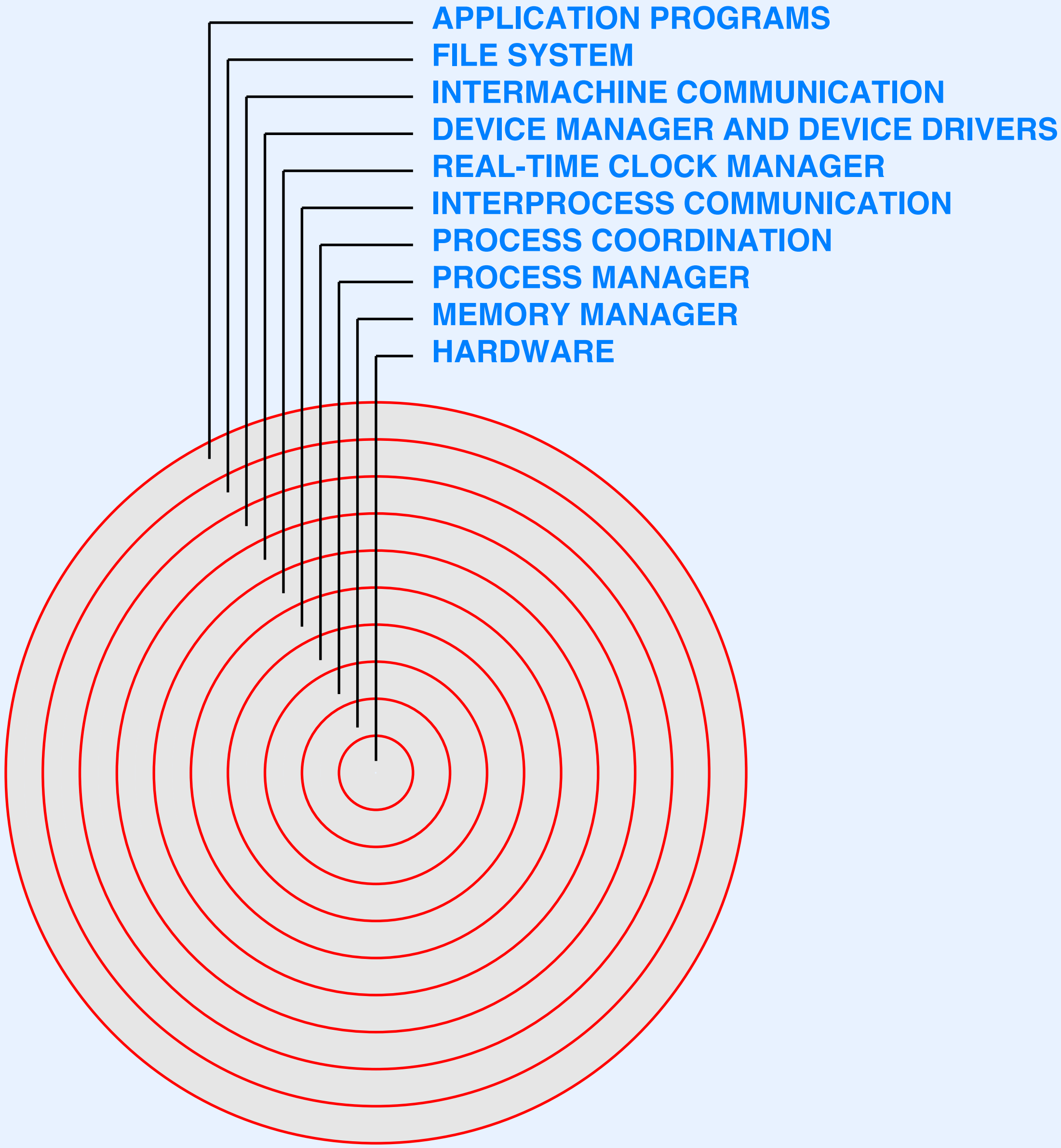
Multilevel Vs. Multilayered Organization

- Multilayer software
  - Visible to user as well as designer
  - Each layer uses layer directly beneath
  - Involves protection as well as data abstraction
  - Examples
    - OSI 7-layer model
    - MULTICS layered security structure
  - Can be inefficient

Multilevel Vs. Multilayered Organization (continued)

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

Multilevel Structure Of Xinu



How To Build An OS

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

Design Example

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

LIST MANIPULATION

Queues And Lists

- Fundamental throughout an operating system
- Various forms
  - FIFOs
  - Priority lists
  - Ascending and descending order
  - Event lists ordered by time of occurrence
- Operations
  - Insert item
  - Extract “next” item
  - Delete arbitrary item

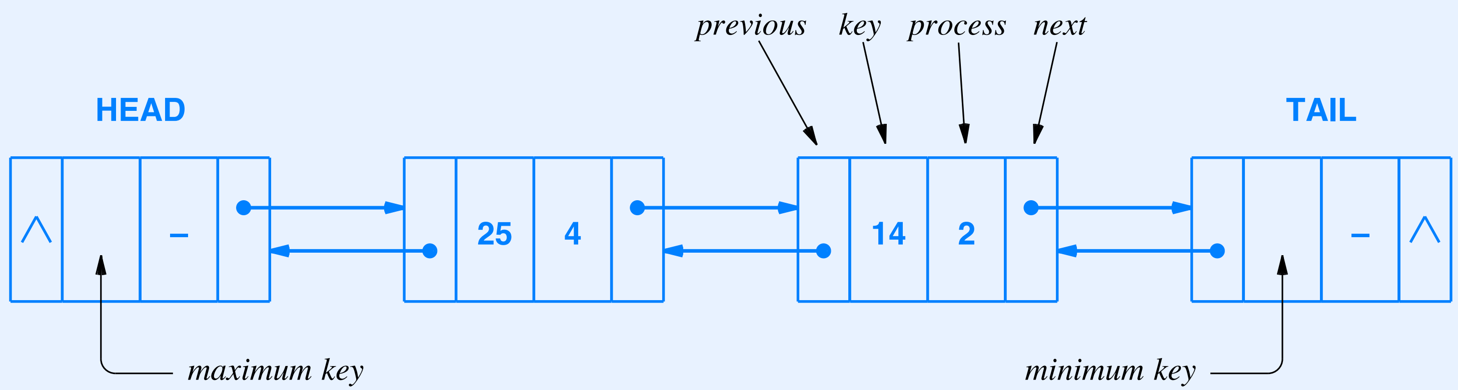
Lists And Queues In Xinu

- Important ideas
  - Many lists store processes
  - A process is known by an integer *process ID*
  - Only need to store process's ID on list
- A single data structure can be used to store many types of lists

Unified List Storage in Xinu

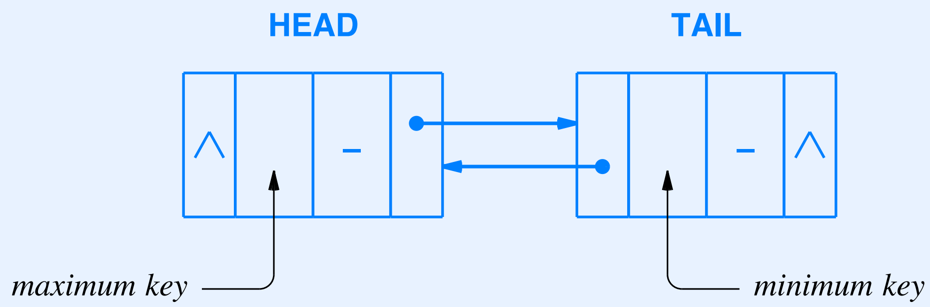
- All lists are doubly-linked, which means a node points to its predecessor and successor
- Each node stores a key as well as a process ID, even though a key is not used in a FIFO list
- Each list has a head and tail; the head and tail nodes have the same shape as other nodes
- Non-FIFO lists are ordered in descending order
- The key value in a head node is the maximum integer used as a key, and the key value in the tail node is the minimum integer used as a key

Conceptual List Structure



- Example list contains two processes, 2 and 4
- Process 4 has key 25
- Process 2 has key 14

Pointers In An Empty List



- Head and tail linked
- Eliminates special cases for insertion or deletion

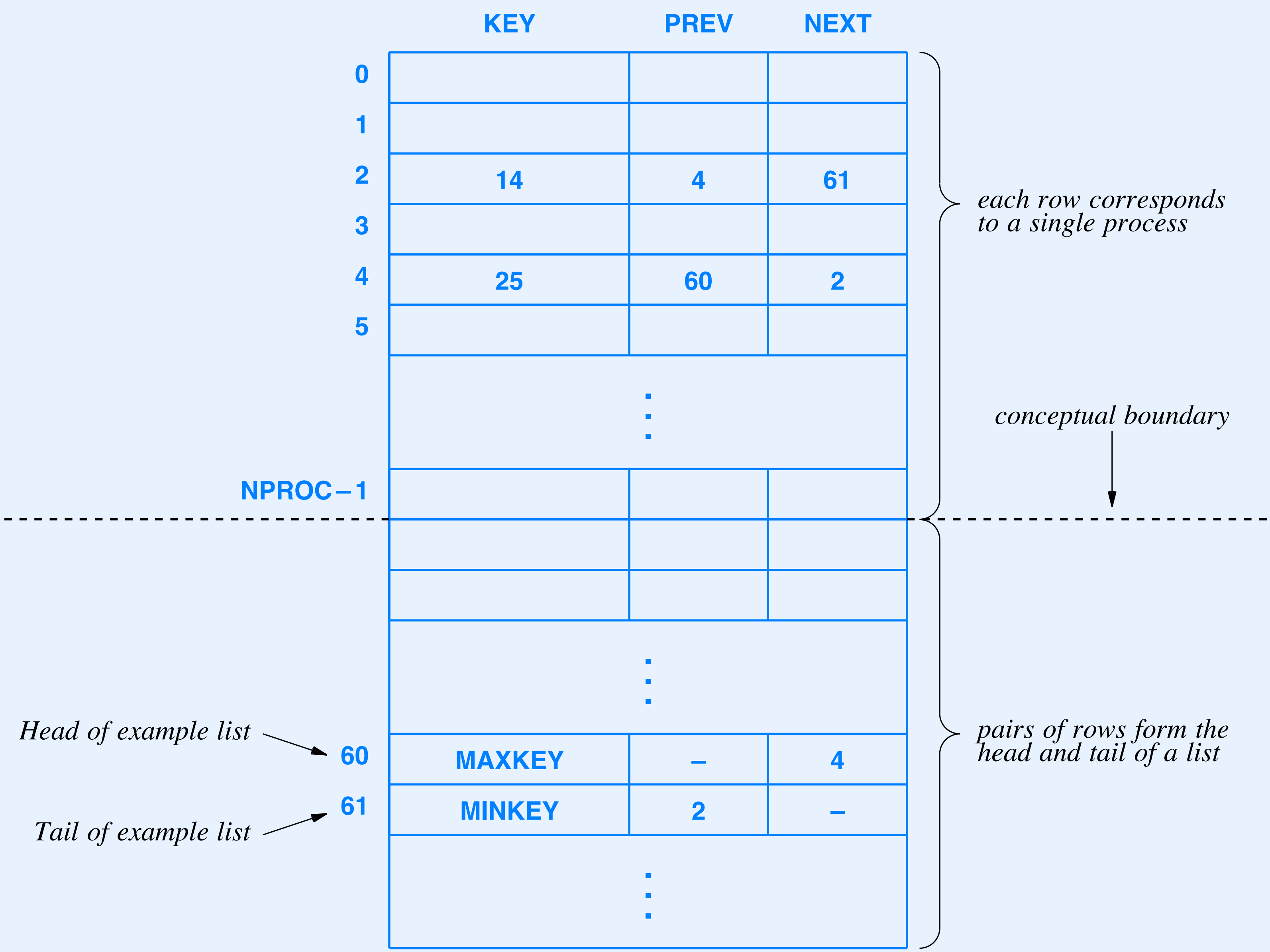
Reducing List Size

- Pointers can be expensive if a list is small
- Important concept: a process can appear on at most one list at any time
- Techniques used to reduce the size of Xinu lists
  - Relative pointers
  - Implicit data structure

Relative Pointers

- Store list elements in an array
  - Each item in array is one node
  - Use array index instead of address to identify a node
- Implicit data structure
  - Number processes 0 through $N - 1$
  - Let i^{th} element of array correspond to process i
  - Store heads and tails in same array at positions N and higher

Illustration Of Xinu List Structure



Implementation

- Data structure
  - Consists of a single array named *queuetab*
  - Is global and available throughout entire OS
- Functions
  - Include tests, such as *isempty*, as well as insertion and deletion operations
  - Implemented with inline functions when possible
- Example code shown after discussion of types

A Question About Types In C

- K&R C defines *short*, *int*, and *long* to be machine-dependent
- ANSI C leaves *int* as a machine-dependent type
- A programmer can define type names
- Question: should a type specify
  - The purpose of an item?
  - The size of an item?
- Example: should a process ID type be named
  - *processid_t* to indicate the purpose or
  - *int32* to indicate the size?

Type Names Used In Xinu

Type Names Used In Xinu

- Xinu uses a compromise to encompass both purpose and size

Type Names Used In Xinu

- Xinu uses a compromise to encompass both purpose and size
- Example: consider a variable that holds an index into queuetab

Type Names Used In Xinu

- Xinu uses a compromise to encompass both purpose and size
- Example: consider a variable that holds an index into queuetab
- The type name can specify

Type Names Used In Xinu

- Xinu uses a compromise to encompass both purpose and size
- Example: consider a variable that holds an index into queuetab
- The type name can specify
  - That the variable is a queue table index

Type Names Used In Xinu

- Xinu uses a compromise to encompass both purpose and size
- Example: consider a variable that holds an index into queuetab
- The type name can specify
  - That the variable is a queue table index
  - That the variable is a 16-bit signed integer

Type Names Used In Xinu

- Xinu uses a compromise to encompass both purpose and size
- Example: consider a variable that holds an index into queuetab
- The type name can specify
  - That the variable is a queue table index
  - That the variable is a 16-bit signed integer
- Xinu uses the type name *qid16* to specify both

Type Names Used In Xinu

- Xinu uses a compromise to encompass both purpose and size
- Example: consider a variable that holds an index into queuetab
- The type name can specify
  - That the variable is a queue table index
  - That the variable is a 16-bit signed integer
- Xinu uses the type name *qid16* to specify both
- Example declarations follows

```
/* queue.h - firstid, firstkey, isempty, lastkey, nonempty              */

/* Queue structure declarations, constants, and inline functions        */

/* Default # of queue entries: 1 per process plus 2 for ready list plus */
/*                        2 for sleep list plus 2 per semaphore         */
#ifndef NQENT
#define NQENT   (NPROC + 4 + NSEM + NSEM)
#endif

#define EMPTY   (-1)            /* null value for qnext or qprev index  */
#define MAXKEY  0x7FFFFFFF      /* max key that can be stored in queue  */
#define MINKEY  0x80000000      /* min key that can be stored in queue  */

struct  qentry  {               /* one per process plus two per list    */
        int32   qkey;           /* key on which the queue is ordered    */
        qid16   qnext;          /* index of next process or tail        */
        qid16   qprev;          /* index of previous process or head    */
};

extern  struct qentry   queuetab[];

/* Inline queue manipulation functions */

#define queuehead(q)    (q)
#define queuetail(q)    ((q) + 1)
#define firstid(q)      (queuetab[queuehead(q)].qnext)
#define lastid(q)       (queuetab[queuetail(q)].qprev)
#define isempty(q)      (firstid(q) >= NPROC)
#define nonempty(q)     (firstid(q) <  NPROC)
#define firstkey(q)     (queuetab[firstid(q)].qkey)
#define lastkey(q)      (queuetab[ lastid(q)].qkey)
```

```
/* Inline to check queue id assumes interrupts are disabled */

#define isbadqid(x)     (((int32)(x) < 0) || (int32)(x) >= NQENT-1)

/* Queue function prototypes */

pid32   getfirst(qid16);
pid32   getlast(qid16);
pid32   getitem(pid32);
pid32   enqueue(pid32, qid16);
pid32   dequeue(qid16);
status  insert(pid32, qid16, int);
status  insertd(pid32, qid16, int);
qid16   newqueue(void);
```

Summary

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

Summary
(continued)

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

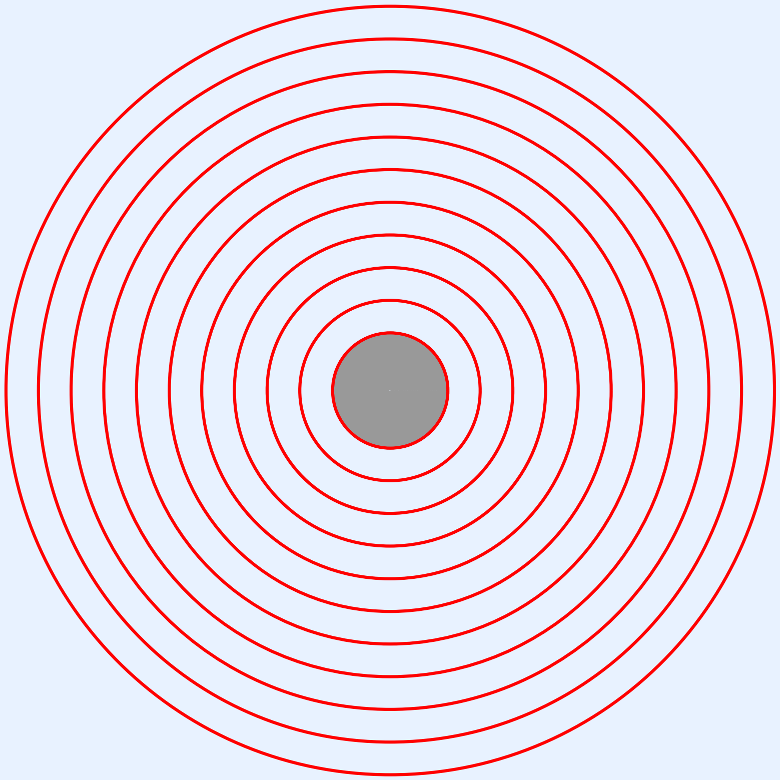
Questions?

PART 3

Hardware Architecture
And
Runtime Systems

(A Brief Overview)

Location Of Hardware In The Hierarchy



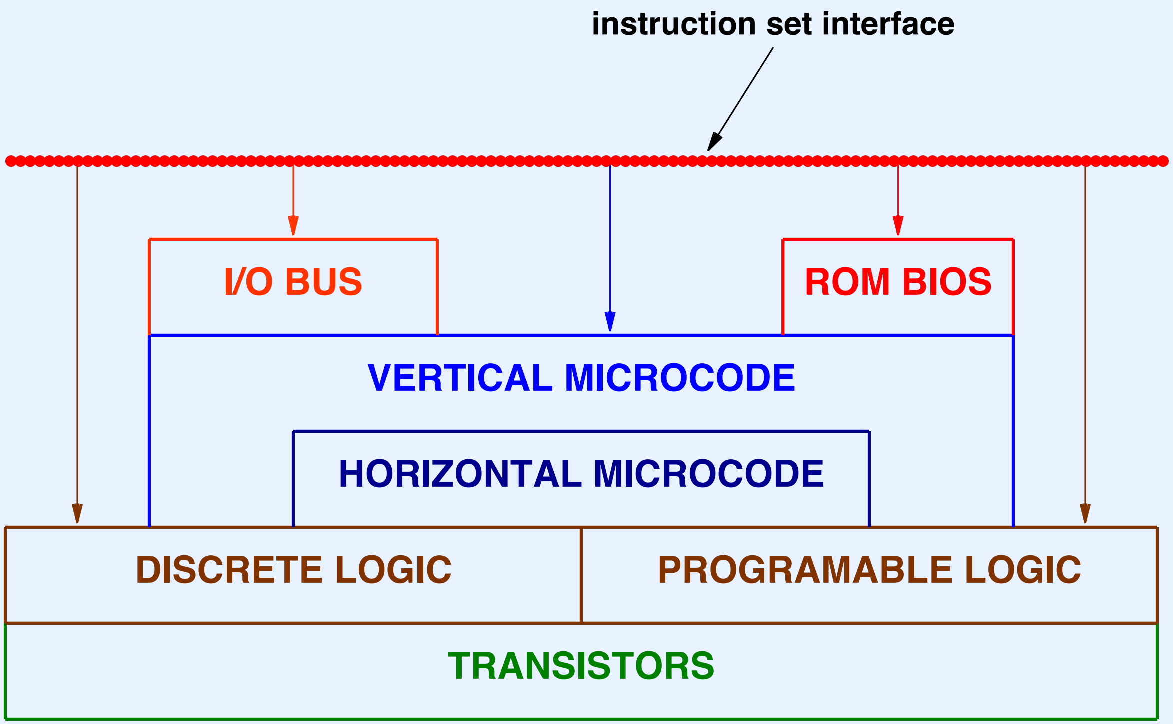
Features Of A Modern Computer

- Processor
- Memory system
- I / O Devices
- Interrupts

Processor

- Instruction set
- General-purpose and special-purpose registers
- Addressing modes
- Protection states
- Optional facilities in ROM

What Interface Does An Operating System See?



- Potentially multiple levels of hardware
- Only some levels visible
- Resulting interface is *instruction set*

General-Purpose And Special-Purpose Registers

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

Memory System

- Defines size of a *byte*, the smallest addressable unit
- Provides address space
- Typical physical address space is
  - *Monolithic*
  - *Linear*
- Includes caching
- Defines important property for programmers: endianness

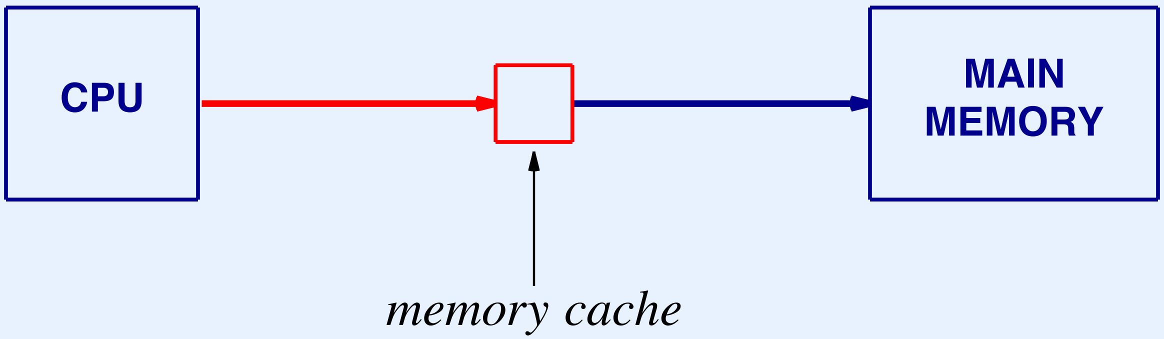
Byte Order Terminology

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

Memory Caches

- Special-purpose hardware units
- Speed memory access
- Less expensive than high-speed memory
- Placed “between” CPU and memory

Conceptual Placement Of Memory Cache



- All references (including instruction fetch) go through cache
- Multi-level cache possible
- Key question: are virtual or physical addresses cached?

I / O Devices

- Wide variety of peripheral devices available
  - Keyboard / mouse
  - Disk
  - Wired or wireless network interface
  - Printer
  - Scanner
  - Camera
  - Sensors
- Multiple transfer paradigms (character, block, packet, stream)

Communication Between Device And CPU

- I/O through *bus*
  - Conceptually parallel wires
  - One or more per computer
- CPU uses bus to
  - Interrogate device
  - Control (start or stop) device
- Device uses bus to
  - Transfer data to/from memory
  - Inform CPU of status

Bus Operations

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

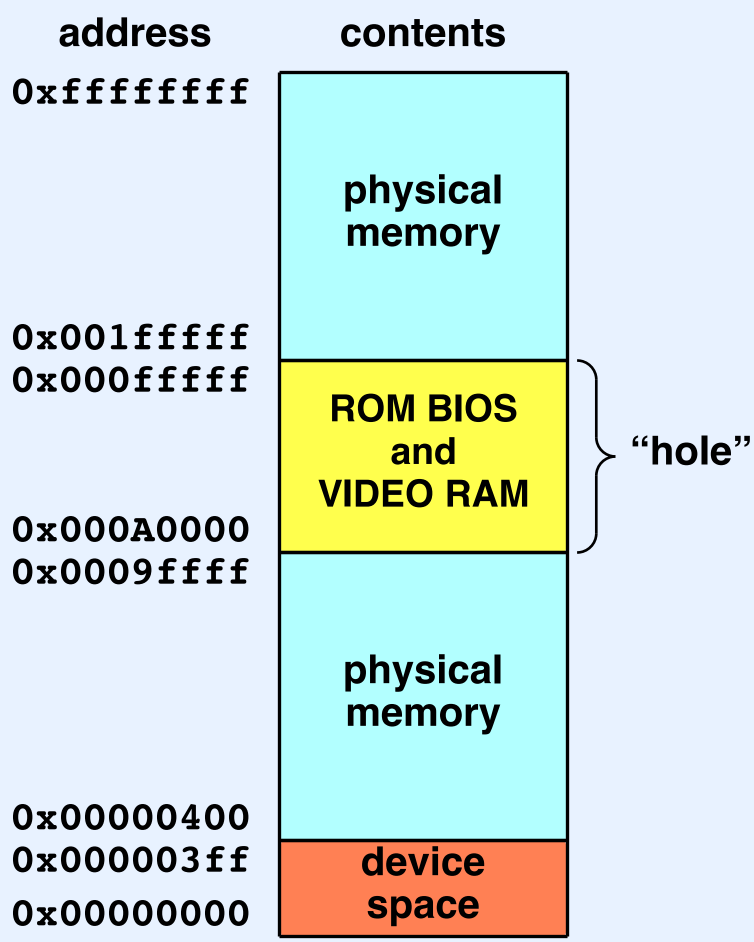
Bus Operations
(continued)

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

Bus Access By CPU

- Two basic approaches
- Approach 1: special instruction(s) used to access bus
  - Example hardware: Intel processor
  - Known as *port-mapped I/O*
- Approach 2: bus mapped into same address space as memory
  - Example hardware: many non-Intel processors
  - Devices placed beyond physical memory
  - CPU uses normal fetch/store memory instructions
  - Known as *memory-mapped I/O*

Illustration Of Address Space On An Intel PC



- Discontiguous for backward compatibility
- "Hole" in physical memory from 640KB to 1MB

Interrupt Mechanism

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

Interrupt-Driven I/O

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

Interrupt Details

- Bus and CPU communicate, possibly through a co-processor
- Device posts an interrupt
- CPU polls bus during fetch / execute cycle
- CPU requests interrupt cause
- Device sends unique interrupt number to CPU
- CPU saves program state (e.g., by pushing onto the stack)
- CPU uses interrupt number to fetch new program state from the interrupt vector in memory
- CPU continues the fetch / execute cycle

Interrupt Mask

- Bit mask kept in CPU status register
- Set by hardware when interrupt occurs; can be reset by OS
- Determines which interrupts are permitted
- Priorities
  - Each device assigned priority level (binary number)
  - When servicing level K interrupt, mask set to disable interrupts at level K and lower

Operating System Responsibility

- Operating system must
  - Store correct information in interrupt vector for each device
  - Arrange for interrupt code to save registers used during the interrupt
  - Arrange for interrupt code to restore registers before returning from interrupt
  - Distinguish among devices, including multiple physical copies of a given device type

Returning From An Interrupt

- Special hardware instruction used
- Atomically restores
  - Old program state
  - Interrupt mask
  - Program counter
- After a return from interrupt, the interrupted code continues and registers are unchanged

Transfer Size And Interrupts

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

Direct Memory Access (DMA)

- Hardware mechanism
- I / O device transfers data to / from memory
  - Occurs over bus
  - Does not involve CPU
- Example use
  - Transfer incoming network packet to memory

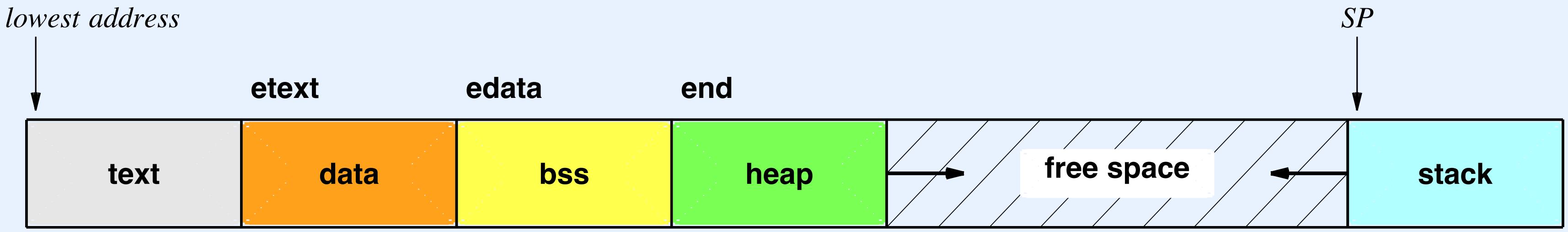
Direct Memory Access (DMA)
(continued)

- Motivation
  - Free CPU from performing I/O
- Interface hardware that uses DMA
  - More expensive
  - May contain RAM, ROM and microprocessor
  - More complex to design

Memory Segments In C Programs

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

Storage Layout For A C Program



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

Symbols For Segment Addresses

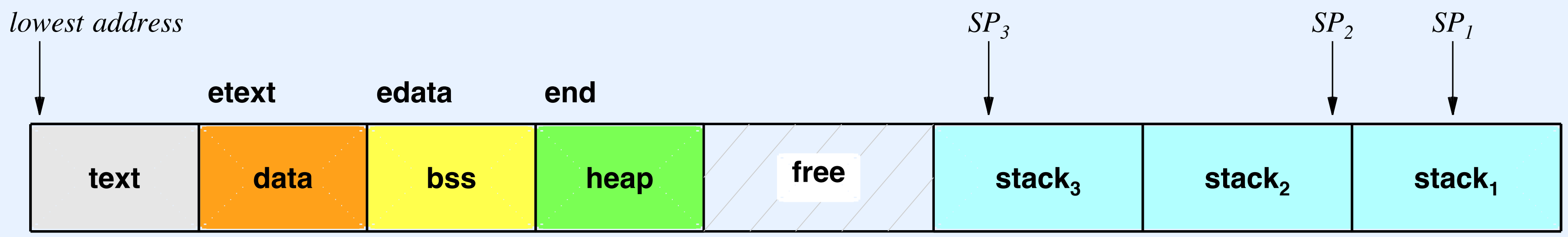
- C compiler and / or linker adds three reserved names to symbol table
- _*etext* lies beyond text segment
- _*edata* lies beyond data segment
- _*end* lies beyond bss segment
- Only the addresses are significant; values are irrelevant
- Program can use the addresses of the reserved symbol to determine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

```
extern   int     end;
```

Runtime Storage For A Process

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

Example Runtime Storage Model: Xinu



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

Summary

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

Summary
(continued)

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

Questions?

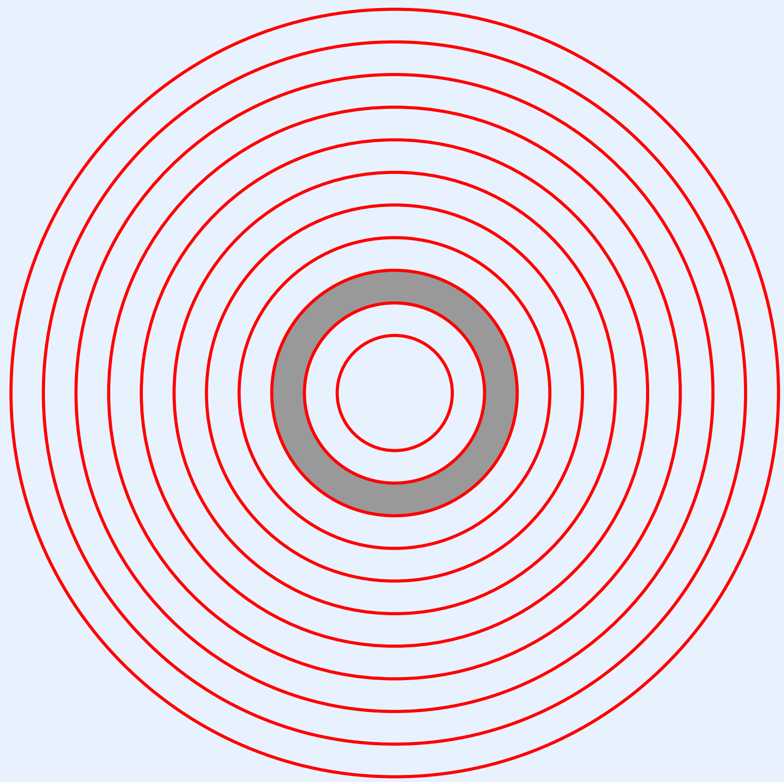
PART 4

Process Management: Scheduling, Context Switching, Process Suspension, Process Resumption, And Process Creation

Terminology

- The term *process management* has been used for decades to encompass the part of an operating system that manages concurrent execution, including both processes and the threads within them
- The term *thread management* is newer, but sometimes leads to confusion because it appears to exclude processes
- The best approach is to be aware of the controversy but not worry about it

Location Of Process Manager In The Hierarchy



Concurrent Processing

- Unit of computation
- Abstraction of a processor
  - Known only to operating system
  - Not known by hardware

A Fundamental Restriction

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

Concurrency Models

- Many variations have been used
  - *Job*
  - *Task*
  - *Thread*
  - *Process*
- Differ in
  - Address space allocation and sharing
  - Coordination and communication mechanisms
  - Longevity
  - Dynamic vs. static definition

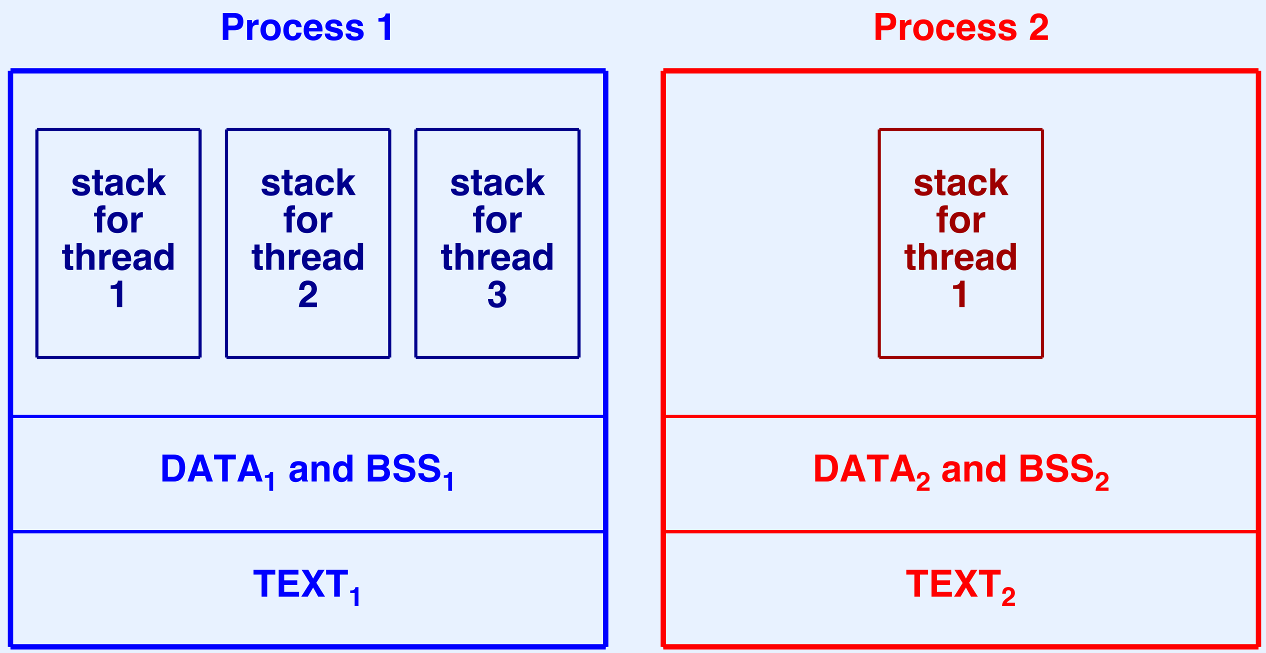
Thread Of Execution

- Single "execution"
- Sometimes called a *lightweight process*
- Can share data (data and bss segments) with other threads
- Must have private stack segment for
  - Local variables
  - Function calls

Heavyweight Process

- Pioneered in Mach and adopted by Linux
- Also called *Process* (written with uppercase "P")
- Address space in which multiple threads can execute
- One data segment per Process
- One bss segment per Process
- Multiple threads per Process
- Given 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

Terminology

- Distinction between *process* and *Process* can be confusing
- For this course, assume generic use ("process") unless
  - Used in context of specific OS
  - Speaker indicates otherwise

Maintaining Processes

- Process
  - OS abstraction
  - Unknown to hardware
  - Created dynamically
- Pertinent information kept by OS
- OS stores information in a central data structure
  - Called *process table*
  - Part of OS address space

Information Kept In A Process Table

- For each process
  - Unique *process identifier*
  - Owner (a user)
  - Scheduling priority
  - Location of code and data (stack)
  - Status of computation
  - Current program counter
  - Current values of registers

Information Kept In A Process Table (continued)

- If a Process contains multiple threads, keep for each thread
  - Owning Process
  - Thread's scheduling priority
  - Location of stack
  - Status of computation
  - Current program counter
  - Current values of registers

Xinu Process Model

- Simplest possible scheme
- Single-user system (no ownership)
- One global context
- One global address space
- No boundary between OS and applications
- Note: all Xinu processes can share data

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

Process States

- Specified by OS designer
- One “state” assigned per activity
- Value updated in process table when 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    */
```

- States are defined as needed when a system is constructed
- We will understand the purpose of each state as we consider the system design

Scheduling And Context Switching

Scheduling

- Fundamental part of process management
- Performed by OS
- Three steps
  - Examine processes that are eligible for execution
  - Select a process to run
  - Switch the CPU to the selected process

Implementation Of Scheduling

- We need a *scheduling policy* that specifies which process to select
- We must then build a scheduling function that
  - Selects a process according to the policy
  - Updates the process table for the current and selected processes
  - Calls *context switch* to switch from current process to the selected process

Scheduling Policy

- Determines when process is selected for execution
- Goal is *fairness*
- May depend on
  - User
  - How many processes a user owns
  - Time a given process has been waiting to run
  - Priority of the process
- Note: hierarchical or flat scheduling can be used

Example Scheduling Policy In Xinu

- Each process assigned a *priority*
  - Non-negative integer value
  - Initialized when process created
  - Can be changed at any time
- Scheduler always chooses to run an eligible process that has highest priority
- Policy is implemented by a system-wide invariant

The Xinu Scheduling Invariant

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

- 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

Implementation Of Scheduling

- Process is eligible if state is *ready* or *current*
- To avoid searching process table during scheduling
  - Keep ready processes on linked list called a *ready list*
  - Order the ready list by process priority
  - Selection of highest-priority process can be performed in constant time

High-Speed Scheduling Decision

- Compare priority of current process to priority of first process on ready list
  - If 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 CPU to the process

Deferred Rescheduling

- Delays enforcement of scheduling invariant
- Used to prevent rescheduling temporarily
- Main purpose: device driver can make multiple processes ready before allowing any of them to run
- We will see an example later

Xinu Scheduler Details

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

Round-Robin Scheduling

- When inserting a process on the ready list, place the process "behind" other processes with the same priority
- If scheduler switches context, first process on ready list is selected
- Note: scheduler switches context if the first process on the ready list has priority *equal* to the current process
- Later, we will see why the equal case is important

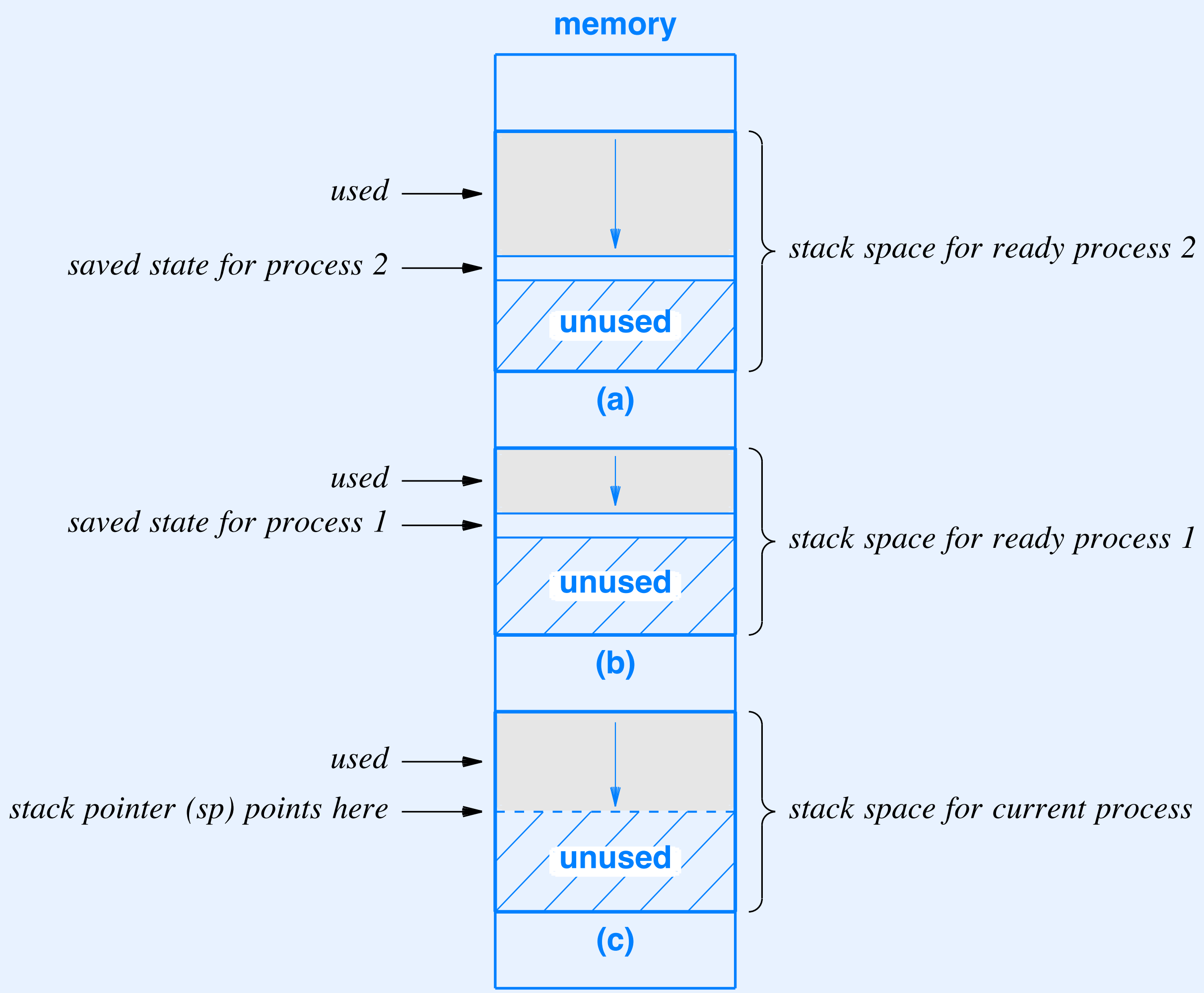
Example Scheduler Code (resched part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  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 running */
		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;
}
```

Illustration Of State Saved On Process Stack

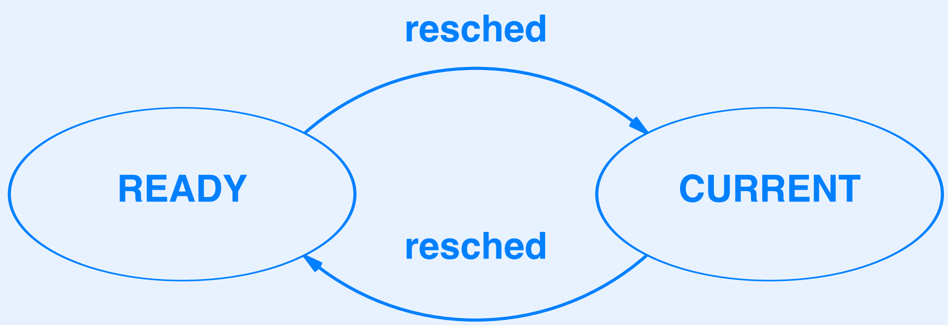


- The stack of each ready process contains saved state

Process State Transitions

- Recall each process has a “state”
- State determines
  - Whether an operation is valid
  - Semantics of each operation
- 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

- Basic part of process manager
- Low-level (must manipulate underlying hardware)
- Usually written in assembly language
- Called by scheduler
- Moves CPU from one process to another

Context Switch Operation

- Given a “new” process, *N*, and “old” process, *O*
- Save copy of all information pertinent to *O* in process table and / or on process O’s stack
  - Contents of hardware registers
  - Program counter (instruction pointer)
  - Privilege level and hardware status
  - Memory map and address space
- Load information for *N*

Context Switch On Various Architectures

- MIPS is RISC machine
  - Single instruction needed to save each register
  - Single instruction needed to restore each register
- X86 is CISC machine
  - Single instruction can save all general-purpose registers
  - Example code follows

Example Context Switch Code (X86 part 1)

```
/* ctxsw.s - ctxsw */

                .text
                .globl  ctxsw

/* excerpt from ctxsw on an X86 architecture.  */
/* arguments are: &oldsp, &newsp               */

ctxsw:
                pushl   %ebp            /* push ebp onto stack          */
                movl    %esp,%ebp       /* record current SP in ebp     */
                pushfl                  /* record flags                 */
                pushal                  /* save 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 (X86 part 2)

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

movl    (%eax),%esp     /* pick 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        */
```

Puzzle #1

- Our invariant says that at any time, a process must be executing
- Context switch code moves from one process to another
- Question: which process executes the context switch code?

Solution To Puzzle #1

- “Old” process
  - Executes first half of context switch
  - Is suspended
- “New” process
  - Continues executing where previously suspended
  - Usually runs second half of context switch

Puzzle #2

- Our invariant says that at any time, one process must be executing
- All user processes may be idle (e.g., applications all wait for input)
- Which process executes?

Solution To Puzzle #2

- Operating system needs an extra process
  - Called the *NULL process*
  - Never terminates
  - Always remains eligible to execute
  - Cannot make a system call that takes it out of ready or current state
  - Typically an infinite loop

Null Process

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

Null Process Code

- Typical null process

```
while(1)
    ;
```

- May not be optimal
  - Fetch-execute may take bus cycles
  - Competes with I/O devices

Puzzle #3

- Null process must always remain ready to execute
- Null process should avoid using bus because doing so "steals" cycles from I/O activity
- Instructions reside in memory, so merely fetching an instruction uses the bus
- How can a null process avoid using the bus?

Two Solutions To Puzzle #3

Two Solutions To Puzzle #3

- Solution #1
  - Halt the CPU until interrupt occurs
  - Special hardware instruction required

Two Solutions To Puzzle #3

- Solution #1
  - Halt the CPU until interrupt occurs
  - Special hardware instruction required
- Solution #2
  - Install an instruction cache
  - Processor fetches instructions from cache rather than memory
  - Avoids using bus when executing tight loop

More Process Management

Process Manipulation

- Need to invent ways to control processes
- Example operations
  - Suspension
  - Resumption
  - Creation
  - Termination
- Recall: state variable in process table records activity

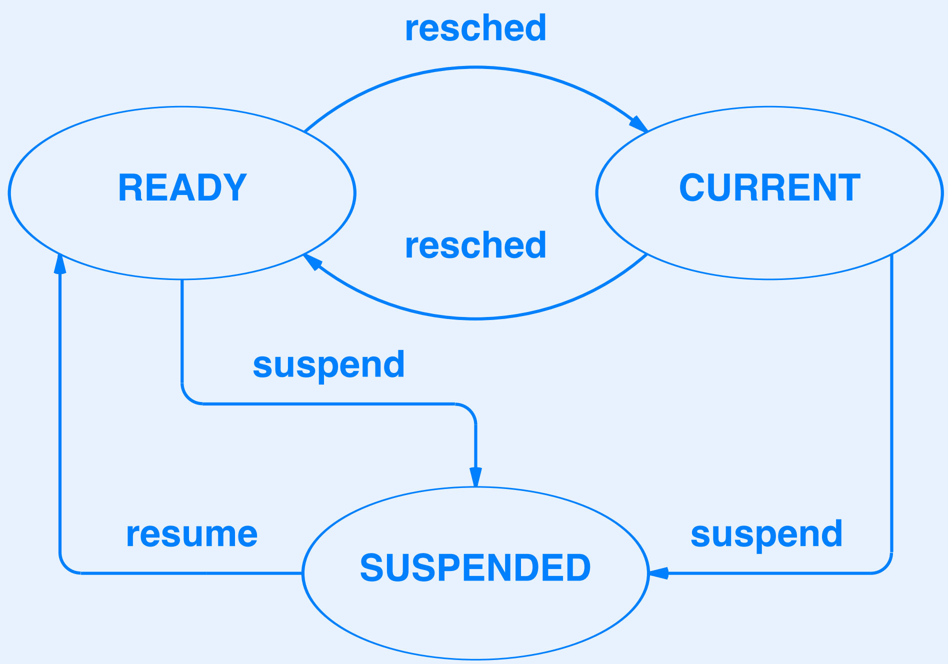
Process Suspension

- Temporarily “stop” a process
- Prohibit it from using the CPU
- To allow later resumption
  - Process table entry retained
  - Complete state of computation saved
- OS sets process table entry to indicate process is suspended

A Note About System Calls

- OS contains many functions
- Some functions correspond to system calls and others are internal
- We use the type *syscall* to distinguish

State Transitions For Suspension And Resumption



- Ether current or ready process can be suspended
- Only a suspended process can be resumed

Example Suspension Code (part 1)

```
/* excerpt from suspend.c - suspend */

/*------------------------------------------------------------------------
 *  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;
        }
```

Example Suspension Code (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 reschedule */
        }
        prio = prptr->prprio;
        restore(mask);
        return prio;
}
```

Process Resumption

- Resume execution of previously suspended process
- Method
  - Make process eligible for CPU
  - Re-establish scheduling invariant
- Note: resumption does *not* guarantee instantaneous execution

Example Resumption Code

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

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 );
}
```

Function To Make A Process Ready (part 1)

```
/* 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  */
          bool8         resch           /* reschedule afterward?        */

        )
{
        register struct procent *prptr;

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

Function To Make A Process Ready (part 2)

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

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

        if (resch == RESCHED_YES) {
                resched();
        }
        return(OK);
}
```

- Note: ready assumes that interrupts are disabled

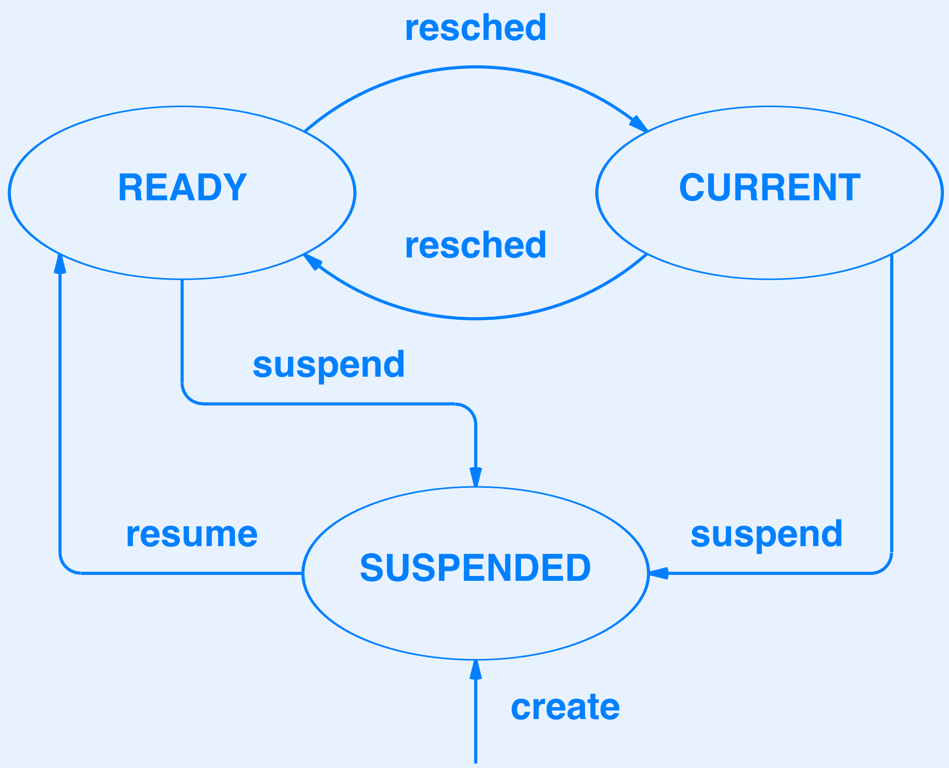
Process Termination

- Final and permanent
- Record of the process is expunged
- Process table entry becomes available for reuse
- Known as *process exit* if initiated by the thread itself
- We will see more about termination later

Process Creation

- Processes are dynamic — process *creation* refers to starting a new process
- Performed by *create* procedure in Xinu
- Method
  - Find free entry in process table
  - Fill in entry
  - Place new process in *suspended* state
- We will see more about creation later

Illustration Of State Transitions For Additional Process Management Functions



At one time, process scheduling was the primary research topic in operating systems. Why did the topic fade? Was the problem completely solved?

Summary

- Process management is a fundamental part of OS
- Information about processes 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
  - Key part of the process manager
  - Chooses next process to execute
  - Implements a scheduling policy
  - Changes information in the process table
  - Calls context switch or change from one process to another
  - Usually optimized for high speed

Summary
(continued)

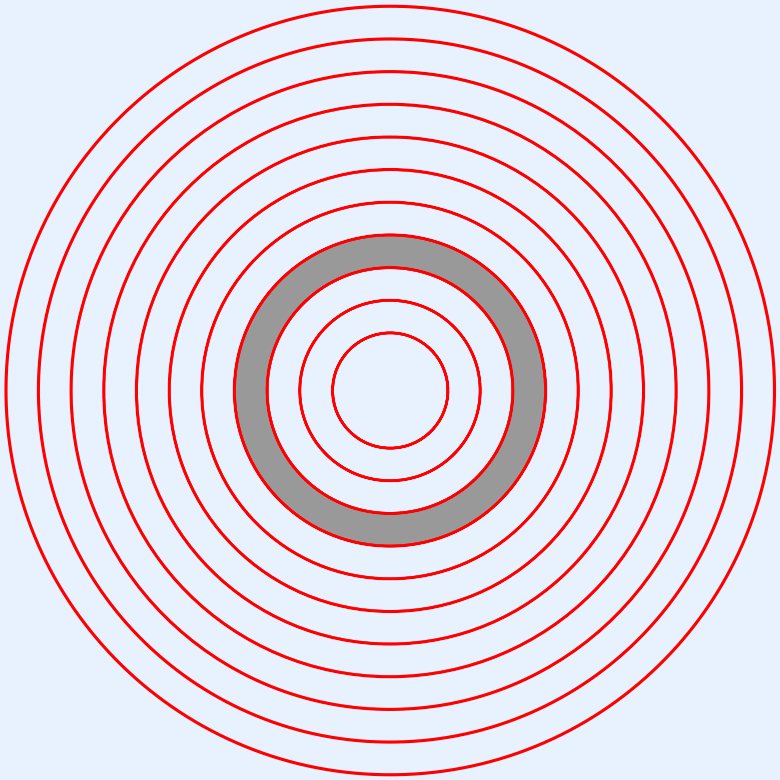
- Context switch
  - Low-level piece of a process manager
  - Moves processor from one process to another
- At any time a process must be executing
- Processes can be suspended, resumed, created, and terminated
- Special process known as *null process* remains ready to run at all times

Questions?

PART 5

Process Coordination And Synchronization

Location Of Process Coordination In The Hierarchy



Coordination Of Processes

- Necessary in a concurrent system
- Avoids conflicts when accessing shared items
- Allows multiple processes to cooperate
- Can also be used when
  - Process waits for I/O
  - Process waits for another process
- Example of cooperation among processes: UNIX pipes

Two Approaches To Process Coordination

- Use facilities supplied by hardware
  - Most useful for multiprocessor systems
  - May rely on *busy waiting*
- Use facilities supplied by the operating system
  - Can be used with single processor
  - No unnecessary execution

Note: we will focus on latter

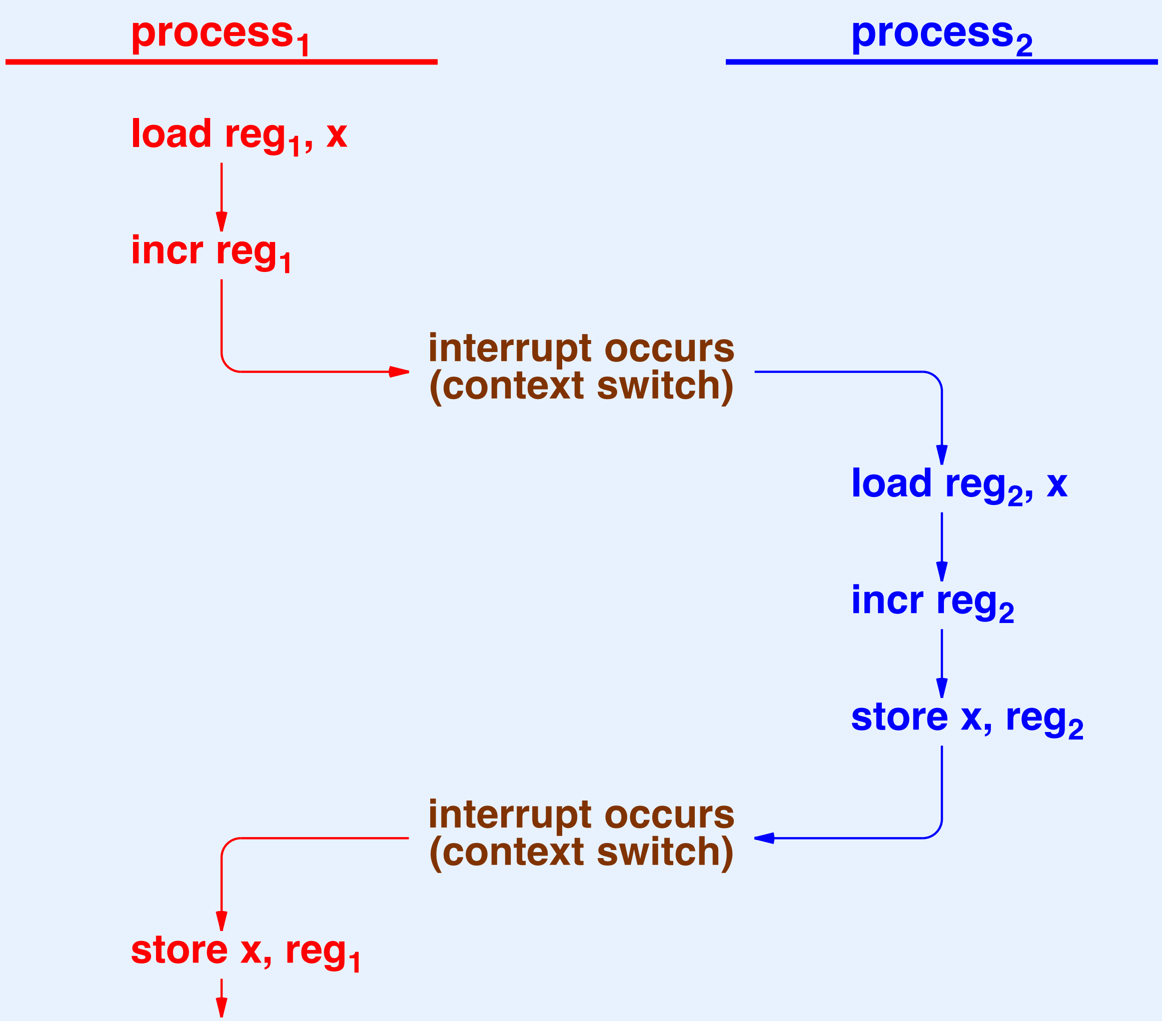
Key Situations That Process Coordination Mechanisms Handle

- Mutual exclusion
- Producer / consumer interaction

Mutual Exclusion

- Concurrent processes access shared data
- Nonatomic operations can produce unexpected results
- Example: multiple steps used to increment variable z
  – Load variable z into register i
  – Increment register i
  – Store register i in variable z

Illustration Of Two Processes Attempting To Increment A Shared Variable Concurrently



To Prevent Problems

- Insure that only one process accesses a shared item at any time
- Trick: once a process obtains access, make all other processes wait
- Three solutions
  - Test-and-set hardware instruction
  - Disable all interrupts
  - Semaphores (implemented in software)

Handling Mutual Exclusion With A Test-And-Set Instruction

- Atomic hardware operation, *tset*, tests whether a memory location is zero and sets it to nonzero
- Initialization (or to declare the shared item is not in use): set memory location to zero

$$m = 0;$$

- To obtain access, execute the following loop:

```
while tset(m)
        ;  /* do nothing */
```

- Only one process can access at any time
- Involves *busy waiting*
- Used in multiprocessors

Handling Mutual Exclusion With Semaphores

- Semaphore allocated for item to be protected
- Known as a *mutex* semaphore
- Applications programmed to use mutex before accessing shared item
- Operating system guarantees only one process can access the shared item at a given time
- Implementation avoids busy waiting

Definition Of Critical Section

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

At what level of granularity should mutual exclusion be applied in an operating system?

Low-Level Mutual Exclusion

- Mutual exclusion needed
  - By application processes
  - Inside operating system
- Mutual exclusion can be guaranteed provided no context switching occurs
- Context changed by
  - Interrupts
  - Calls to *resched*
- Low-level mutual exclusion: mask interrupts and avoid rescheduling

Interrupt Mask

- Hardware mechanism that controls interrupts
- Internal machine register; may be part of processor status word
- Typically, zero value means interrupts can occur
- OS can
  - Examine current interrupt mask (find out whether interrupts are enabled)
  - Set interrupt mask to prevent interrupts
  - Clear interrupt mask to allow interrupts

Masking Interrupts

- Important principle:

  No operating system function should contain code to explicitly enable interrupts.

- Technique used: given function
  - Saves current interrupt status
  - Disables interrupts
  - Proceeds through critical section
  - Restores interrupt status from saved copy
- Key insight: allows arbitrary call nesting

Why Interrupt Masking Is Insufficient

- It works! But...
- Stopping interrupts penalizes all processes when one process executes a critical section
  - Stops all I/O activity
  - Restricts execution to one process for the entire system
- Can interfere with the scheduling invariant (low-priority process can block a high-priority process for which I/O has completed)
- Does not provide a data access policy

High-Level Mutual Exclusion

- Idea is to create a facility with the following properties
  - Permit designer to specify multiple critical sections
  - Allow independent control of each critical section
  - Provide an access policy (e.g., FIFO)
- A single mechanism, the *counting semaphore*, suffices

Counting Semaphore

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

Operations On Semaphores

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

Semaphore Invariant

Semaphore Invariant

- Establishes relationship between conceptual purpose and implementation

Semaphore Invariant

- Establishes relationship between conceptual purpose and implementation
- Must be re-established after each operation

Semaphore Invariant

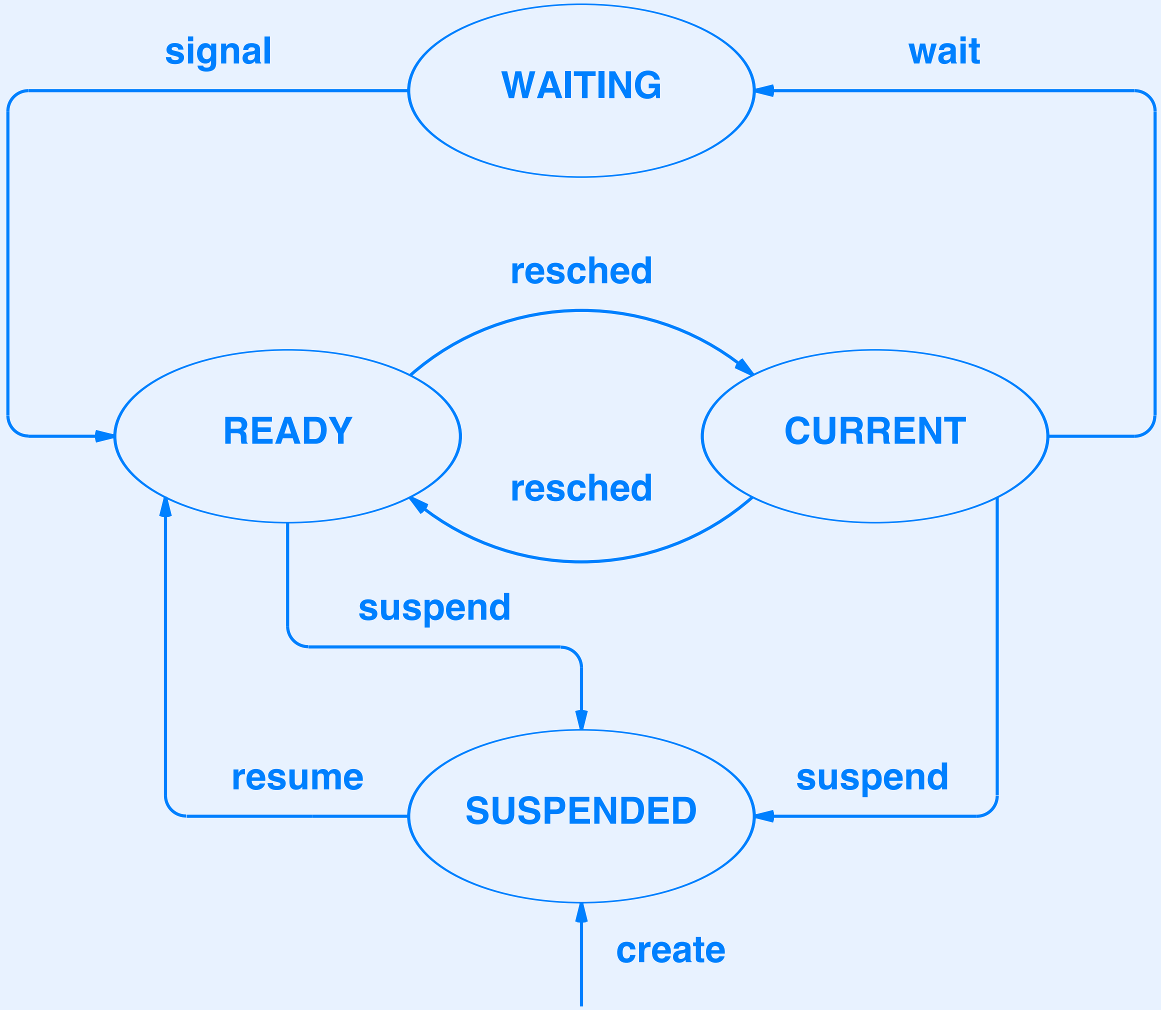
- Establishes relationship between conceptual purpose and implementation
- Must be re-established after each operation
- Surprisingly elegant:

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

Counting Semaphores In Xinu

- Stored in an array of semaphore entries
- Each entry
  - Corresponds to one instance
  - Contains an integer count and pointer to list of processes
- Semaphore ID is index into array
- Policy for management of waiting processes is FIFO
- Each process that is enqueued on a semaphore queue is in the *WAITING* state

State Transitions With Waiting State



Semaphore Definitions

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

#ifndef NSEM
#define NSEM            45      /* 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;
        }
```

Implementation Of Wait (part 2)

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

        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;
}
```

How To Use Counting Semaphores

- Two paradigms
  - Cooperative mutual exclusion
  - Direct synchronization (e.g., producer-consumer)

Cooperative Mutual Exclusion

- Initialization

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

- Use: bracket critical sections of code with calls to *wait* and *signal*

```
 wait(sid);
...critical section (use shared resource)...
signal(sid);
```

Producer-Consumer Synchronization

- Typical scenerio
  - Shared circular buffer
  - Producing process deposits items into buffer
  - Consuming process extracts items from buffer
- Must guarantee
  - Producer blocks when buffer full
  - Consumer blocks when buffer empty
- Can use two semaphores for synchronization

Producer-Consumer Synchronization

- Initialization

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

- Producer algorithm

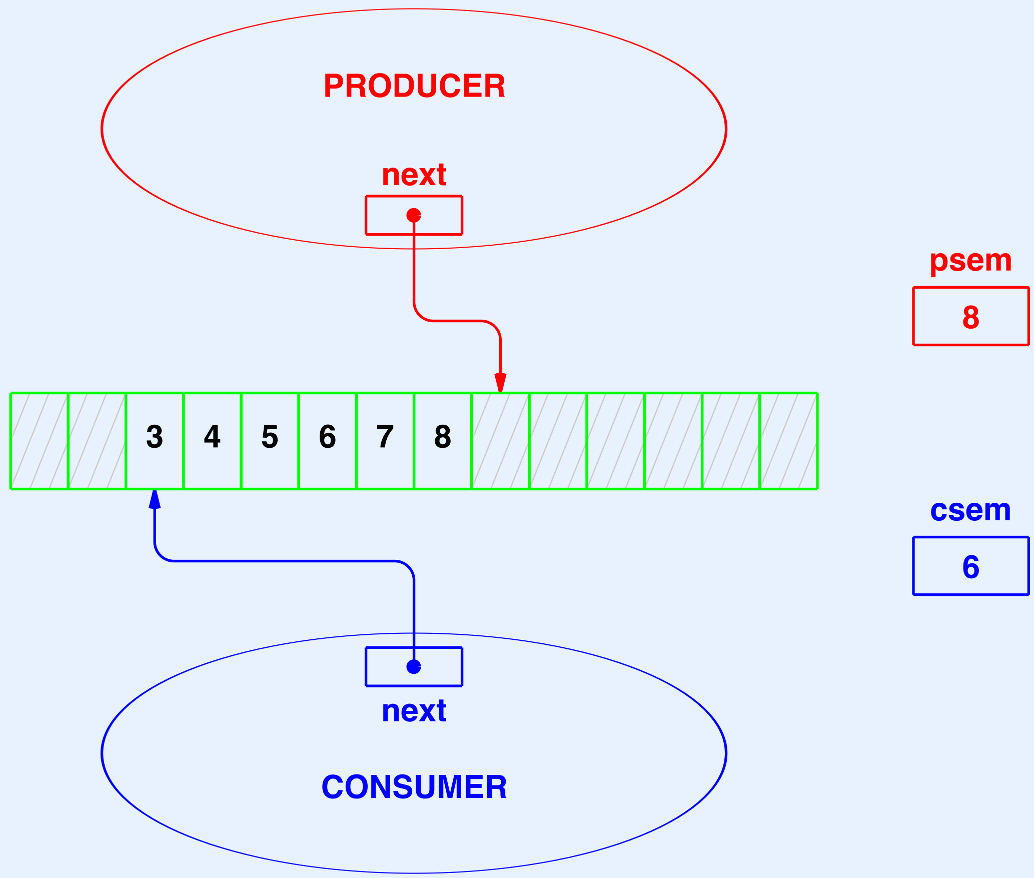
```
repeat forever {
        wait(psem);
        fill_next_buffer_slot;
        signal(csem);
}
```

Producer-Consumer Synchronization (continued)

- Consumer algorithm

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

Illustration Of Producer-Consumer



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

Semaphore Queuing Policy

- Determines which process to select among those waiting
- Needed when *signal* called
- Examples
  - First-Come-First-Served (FCFS or FIFO)
  - Process priority
  - Random

Question

Question

- The goal is “fairness”

Question

- The goal is “fairness”
- Which semaphore queuing policy implements goal best?

Question

- The goal is “fairness”
- Which semaphore queuing policy implements goal best?
- In other words, how should we interpret fairness?

Question

- The goal is "fairness"
- Which semaphore queuing policy implements goal best?
- In other words, how should we interpret fairness?
  - Should a low-priority process be allowed to run if a high-priority process is also waiting?
  - Should a low-priority process be blocked forever if high-priority processes use a resource?

Choosing A Semaphore Queueing Policy

- Difficult
- No single best answer
  - Fairness not easy to define
  - Scheduling and coordination interact
  - May affect other OS policies
- Interactions of heuristics may produce unexpected results

Example Semaphore Queuing Policy

- First-come-first-serve
- Straightforward to implement
- Works well for traditional uses of semaphores
- Potential problem: low-priority process can access while high-priority process remains waiting

Implementation Of FIFO Semaphore Policy

- Each semaphore uses a list to manage waiting processes
- List is run as a queue: insertions at one end and deletions at the other
- Example implementation 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;
        }
```

Implementation Of Signal (Part 2)

```
        semptr= &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        if ((semptr->scount++) < 0) {   /* release a waiting process */
                ready(dequeue(semptr->squeue), RESCHED_YES);
        }
        restore(mask);
        return OK;
}
```

Semaphore Allocation

- Static
  - Semaphores defined at compile time
  - More efficient, but less powerful
- Dynamic
  - Semaphore created at runtime
  - More flexible
- Xinu supports dynamic allocation

Xinu Semcreate (part 1)

```
/* semcreate.c - semcreate, newsem */

#include <xinu.h>

local   sid32   newsem(void);

/*------------------------------------------------------------------------
 *  semcreate  -  create a new semaphore and return the ID to the caller
 *------------------------------------------------------------------------
 */
sid32   semcreate(
          int32          count          /* initial semaphore count      */
        )
{
        intmask mask;                   /* saved interrupt mask         */
        sid32   sem;                    /* semaphore ID to return       */

        mask = disable();

        if (count < 0 || ((sem=newsem())==SYSERR)) {
                restore(mask);
                return SYSERR;
        }
        semtab[sem].scount = count;     /* initialize table entry       */

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

Xinu Semcreate (part 2)

```
/*------------------------------------------------------------------------
 *  newsem  -  allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
local   sid32   newsem(void)
{
        static  sid32   nextsem = 0;    /* next semaphore index to try  */
        sid32   sem;                    /* semaphore ID to return       */
        int32   i;                      /* iterate through # entries    */

        for (i=0 ; i<NSEM ; i++) {
                sem = nextsem++;
                if (nextsem >= NSEM)
                        nextsem = 0;
                if (semtab[sem].sstate == S_FREE) {
                        semtab[sem].sstate = S_USED;
                        return sem;
                }
        }
        return SYSERR;
}
```

Semaphore Deletion

- Wrinkle: one or more processes may be waiting when semaphore is deleted
- Must choose a disposition for each
- Xinu policy: make process ready

Xinu Semdelete (part 1)

```
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 * semdelete  --  Delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
syscall semdelete(
          sid32          sem              /* ID of semaphore to delete    */
        )
{
        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;
        }
        semptr->sstate = S_FREE;
```

Xinu Semdelete (part 2)

```
        while (semptr->scount++ < 0) {  /* free all waiting processes   */
                ready(getfirst(semptr->squeue), RESCHED_NO);
        }
        resched();
        restore(mask);
        return OK;
}
```

Do you understand semaphores?

Thought Problem
(The Convoy)

- One process creates a semaphore

```
mutex = screate(1);
```

- Three processes execute the following

```
process convoy(char_to_print)
   do forever {
       think (i.e., use CPU);
       wait(mutex);
       print(char_to_print);
       signal(mutex);
   }
```

- The processes print characters A, B, and C, respectively

Convoy Problem
(continued)

- Initial output
  - 20 A's, 20 B's, 20 C's, 20 A's, etc.
- After tens of seconds
  ABCABCABC...
- Facts
  - Everything is correct
  - No other processes are executing
  - Print is nonblocking (polled I/O)

Convoy Problem
(continued)

- Questions
  - How long is thinking time?
  - Why does convoy start?
  - Will output switch back given enough time?
  - Did knowing the policies or the implementation of the scheduler and semaphore mechanisms make the convoy behavior obvious?

Summary

- Process synchronization fundamental
  - Supplied to applications
  - Used inside OS
- Low-level mutual exclusion
  - Masks hardware interrupts
  - Avoids rescheduling
  - Insufficient for all coordination

Summary
(continued)

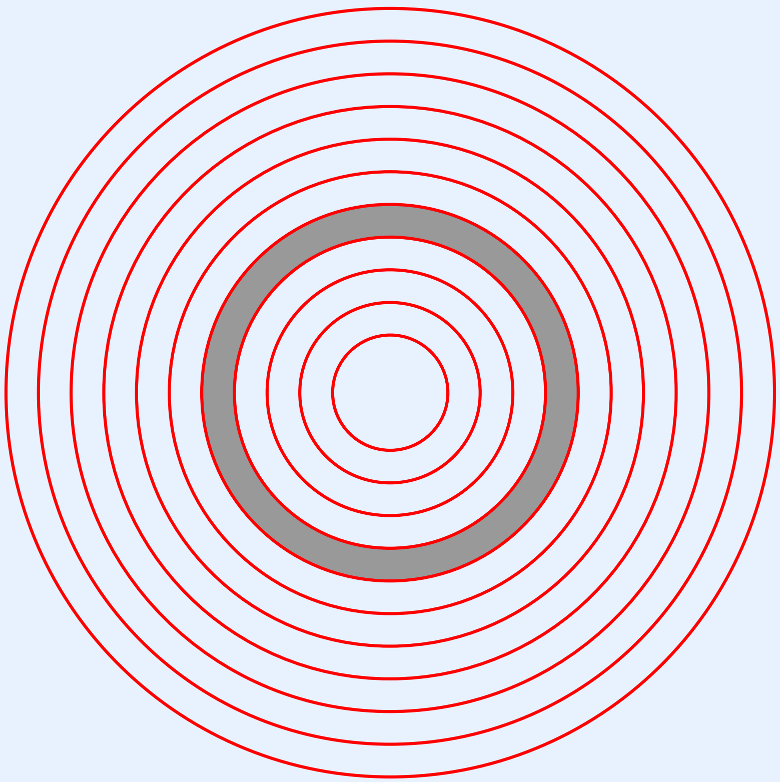
- High-level coordination
  - Used by subsets of processes
  - Available inside and outside OS
  - Implemented with counting semaphore
- Counting semaphore
  - Powerful abstraction
  - Provides mutual exclusion and producer / consumer synchronization

Questions?

PART 6

Inter-Process Communication

Location Of Inter-process Communication In The Hierarchy



Inter-process Communication

- Used for
  - Exchange of (nonshared) data
  - Process coordination
- General technique: *message passing*

Two Approaches To Message Passing

- Approach #1
  - Message passing is one of many services
  - Messages are separate from I/O and process synchronization services
  - Implemented using lower-level mechanisms, such as semaphores
- Approach #2
  - The entire operating system is *message-based*
  - Messages, not function calls, provide the fundamental building block
  - Messages, not semaphores, used for process synchronization

Design Of A Message Passing Facility

Design Of A Message Passing Facility

- To understand the issue, begin with a trivial message passing facility

Design Of A Message Passing Facility

- To understand the issue, begin with a trivial message passing facility
- Allow a process to send a message directly to another process

Design Of A Message Passing Facility

- To understand the issue, begin with a trivial message passing facility
- Allow a process to send a message directly to another process
- In principle,

Design Of A Message Passing Facility

- To understand the issue, begin with a trivial message passing facility
- Allow a process to send a message directly to another process
- In principle, the design should be straightforward

Design Of A Message Passing Facility

- To understand the issue, begin with a trivial message passing facility
- Allow a process to send a message directly to another process
- In principle, the design should be straightforward
- In practice,

Design Of A Message Passing Facility

- To understand the issue, begin with a trivial message passing facility
- 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 or variable size?
- What is the maximum message size?
- How many messages 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

- Synchronous interface
  - Blocks until operation performed
  - Easy to understand / program
  - Extra processes can be used to obtain asynchrony

Synchronous vs. Asynchronous Interface (continued)

- Asynchronous interface
  - Starts an operation
  - Allows initiating process to continue execution
  - Notification
    - Arrives when operation completes
    - May entail abnormal control (e.g., software interrupt or “callback” mechanism)
  - Polling can be used to determine status

Why Is A Message Passing Facility So Difficult To Design?

- Interacts with
  - Process coordination subsystem
  - Memory management subsystem
- Affects user’s perception of system

An Example Inter-process Message Passing

- Simple, low-level mechanism
- Direct process-to-process communication
- One-word messages
- One-message buffer
- Synchronous, buffered reception
- Asynchronous transmission and “reset” operation

An Example Inter-process Message Passing (continued)

- Three functions

```
send(msg, pid);

msg = receive();

msg = recvclr();
```

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

An Example Inter-process Message Passing (continued)

- First-message semantics
  - First message sent to a process is stored until it has been received
  - Subsequent attempts to send fail
- Typical idiom

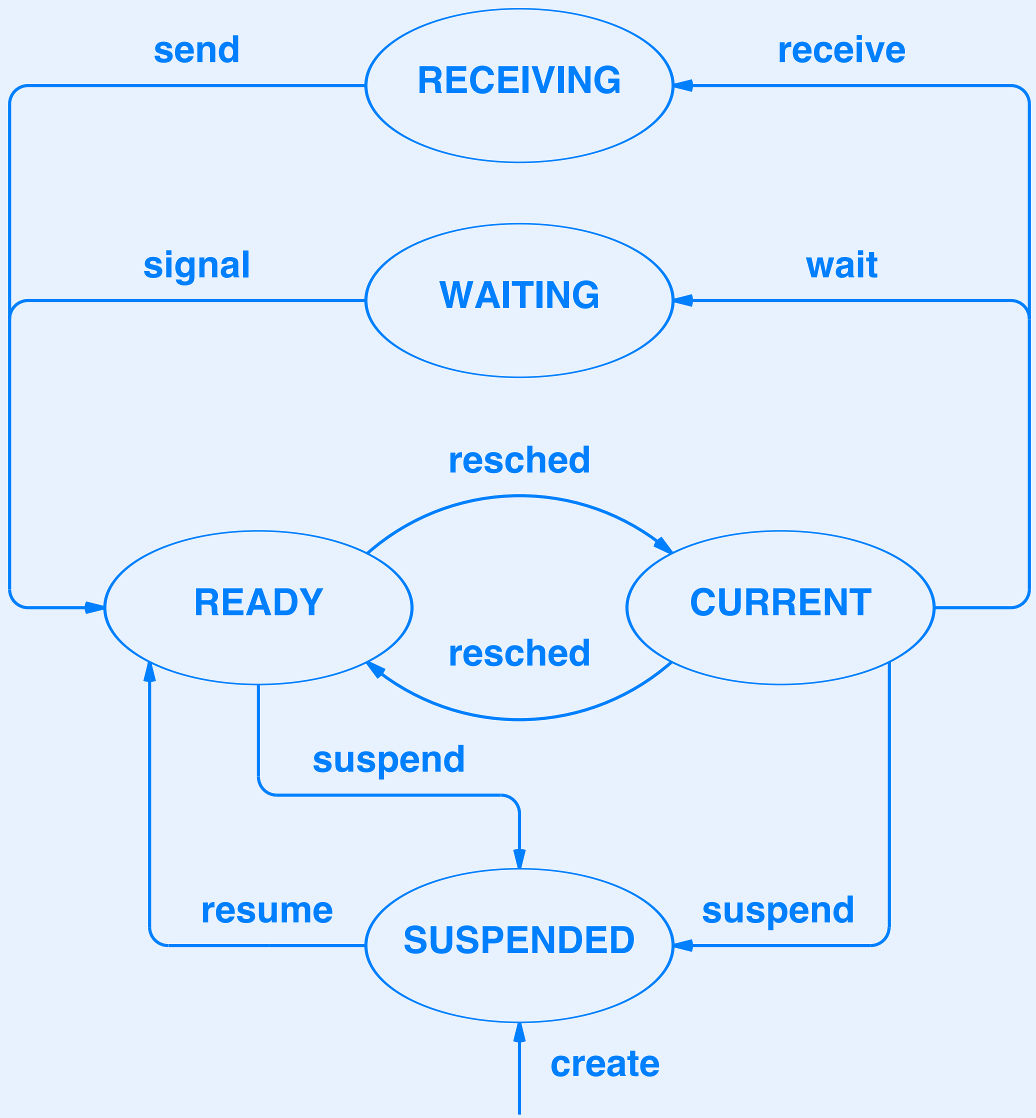
```
recvclr();  /* prepare to receive a message */
... /* allow other processes to send messages */
msg = receive();
```

- Above code returns first message that was sent

Process State For Message Reception

- While receiving a message, a process is not
  - Executing
  - Ready
  - Suspended
  - Waiting on a semaphore
- Therefore, a new state is needed for message passing
- Named *RECEIVING*
- Entered when *receive* called

State Transitions With Message Passing



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;
}
```

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

- Note: we will discuss receive-with-timeout later

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

- Inter-process communication
  - Implemented by message passing
  - Can be synchronous or asynchronous
- Synchronous interface is the simplest
- Xinu uses synchronous reception and asynchronous transmission

Questions?