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

• With an operating system in place
  – Multiple applications can 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
  – An operating system must bridge the gap between low-level hardware and high-level abstractions

• Everything is now connected by computer networks
  – An operating system must offer communication facilities
  – Distributed mechanisms (e.g., access to remote files) 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

• You will gain first-hand experience next week
REQUIRED BACKGROUND AND PREREQUISITES
Background Needed

- A few concepts from earlier courses
  - Integer arithmetic an bit-wise operators \textit{and}, \textit{or}, and \textit{not}
  - I/O: you should know the difference between standard library functions (e.g., \textit{fopen}, \textit{putc}, \textit{getc}, \textit{fread}, \textit{fwrite}) and system calls (e.g., \textit{open}, \textit{close}, \textit{read}, \textit{write})
  - File systems and hierarchical directories
  - Symbolic and hard links
  - File modes and protection
  - Key concepts from computer architecture, such as a bus
Background Needed
(continued)

- Data structures (e.g., linked lists)
- Binary and hex representation
- The run-time stack concept
- Local and global variable allocation
- Function calls, arguments, and calling conventions
- Concurrent programming experience: you should have written a program that uses fork or threads
Background Needed
(continued)

• An understanding of runtime storage
  – Segments (text, data, bss, and stack)
  – Basic heap storage management (e.g., malloc and free)

• C programming
  – At least one nontrivial program
  – Comfortable with low-level constructs (e.g., bit manipulation, pointers, and pointer arithmetic)
Background Needed  
(continued)

- Working knowledge of basic Unix tools (needed for programming assignments)
  - Text editor (e.g., emacs)
  - Compiler/linker/loader (i.e., gcc)
  - Tar archives
  - Make and Makefiles
- Desire to learn
Course Syllabus

See the handout or download a copy
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
A FEW THINGS TO THINK ABOUT
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)
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
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 A 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 the 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');
}
/* ---------------------------------------------------------------*/
/* sndB - Repeatedly emit 'B' on the console without terminating */
/* ---------------------------------------------------------------*/

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

Example Of Process Creation In Xinu (Part 2)
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 \textit{main})
  - Is executed by a single thread of control

- A process
  - Is an OS abstraction that is not visible in a programming language
  - Is created independent of code that is executed
  - Important idea: multiple processes can execute the same code concurrently

- In the following example, two processes execute function \textit{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

- 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 manager
- 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

- APPLICATION PROGRAMS
- FILE SYSTEM
- INTERMACHINE COMMUNICATION
- DEVICE MANAGER AND DEVICE DRIVERS
- REAL-TIME CLOCK MANAGER
- INTERPROCESS COMMUNICATION
- PROCESS COORDINATION
- PROCESS MANAGER
- MEMORY MANAGER
- HARDWARE
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
LISTS OF PROCESSES
Queues And Lists

• Keeping track of processes is fundamental throughout an operating system

• Various forms are needed
  – FIFO queues of processes
  – Lists of processes kept in priority order
  – Event lists ordered by the time of occurrence

• Operations required
  – Insert a process onto a list
  – Extract the “next” process from a list
  – Delete an arbitrary process
Lists And Queues In Xinu

- Important ideas
  - A process is known by an integer process ID
  - A list of processes really stores a set of process IDs
- A single data structure can be used to store many types of process 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 the 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 always ordered in descending order according to the key values.
- 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.
The example list contains two processes, 2 and 4
Process 4 has key 25
Process 2 has key 14
Pointers In An Empty List

- In an empty list, the head and tail nodes are linked
- Having a head and tail eliminates special cases for insertion and deletion
Reducing The List Size

- Pointers can mean a large memory footprint, especially on a 64-bit computer
- Important concept: a process can appear on at most one list at any time
- Xinu uses two clever techniques to reduce the size of lists
  - Relative pointers instead of memory addresses
  - An implicit data structure
Xinu List Optimizations

- Lists are stored in an array
  - Each item in the array stores one node of the list
  - Relative pointers: the array index is used to identify a node instead of an address
- Implicit data structure
  - Let $NPROC$ be the number of processes in the system
  - Assign process IDs 0 through $NPROC - 1$
  - Let $i^{th}$ element of the array correspond to process $i$, for $0 \leq i < NPROC$
  - Store heads and tails in same array at positions $NPROC$ and higher
An Illustration Of An Array Holding The Xinu List Structure

<table>
<thead>
<tr>
<th>KEY</th>
<th>PREV</th>
<th>NEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>NPROC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXKEY</td>
<td>–</td>
<td>4</td>
</tr>
<tr>
<td>MINKEY</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Each row corresponds to a single process.
- Conceptual boundary.
- Pairs of rows form the head and tail of a list.
Implementation

- A single array is used to hold all lists of processes
  - The array is global and available throughout the entire OS
  - The array is named `queuetab`
- Functions are available to manipulate lists
  - Include tests, such as `isempty`, as well as insertion and deletion operations
  - For efficiency, functions are implemented with inline macros when possible
- Example code shown after a discussion of types
A Question About Types In C

- K&R C defined *short*, *int*, and *long* to be machine-dependent
- ANSI C left *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?
  - *int32* to indicate the 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
- 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 follow
Definitions From queue.h (Part 1)

/* 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 {
  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[];
Definitions From queue.h (Part 2)

/* 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 isemptyy(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) < NPROC) || (int32)(x) >= NQENT-1)
Code For Insertion And Deletion From A FIFO Queue (Part 1)

/* queue.c - enqueue, dequeue */
#include <xinu.h>
struct qentry queuetab[NQENT]; /* Table of process queues */
/*------------------------------------------------------------------------*/
/* enqueue - Insert a process at the tail of a queue */
/*------------------------------------------------------------------------*/

pid32 enqueue(pid32 pid, qid16 q)
{
    pid32 prev, tail;
    if (isbadqid(q) || isbadpid(pid))
        return SYSERR;
    tail = queuetail(q);
    prev = queuetab[tail].qprev;
    queuetab[pid].qnext = tail;
    queuetab[pid].qprev = prev;
    queuetab[prev].qnext = pid;
    queuetab[tail].qprev = pid;
    return pid;
}

Copyright © 2022 by Douglas Comer and CRC Press, Inc. All rights reserved.
Code For Insertion And Deletion From A FIFO Queue (Part 2)

/*------------------------------------------------------------------------
 * dequeue - Remove and return the first process on a list
 *------------------------------------------------------------------------*/

pid32 dequeue(qid16 q /* ID queue to use */) {
  pid32 pid; /* ID of process removed */
  if (isbadqid(q)) {
    return SYSERR;
  } else if (isempty(q)) {
    return EMPTY;
  }
  pid = getfirst(q);
  queuetab[pid].qprev = EMPTY;
  queuetab[pid].qnext = EMPTY;
  return pid;
}
/* insert.c - insert */

#include <xinu.h>

/*------------------------------------------------------------------------
 * insert - Insert a process into a queue in descending key order
 *------------------------------------------------------------------------
 */

status insert(pid32 pid, /* ID of process to insert */
               qid16 q, /* ID of queue to use */
               int32 key /* Key for the inserted process */
) {

  qid16 curr; /* Runs through items in a queue*/
  qid16 prev; /* Holds previous node index */

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

  curr = firstid(q);
  while (queuetab[curr].qkey >= key) {
    curr = queuetab[curr].qnext;
  }

  return OK;
}
Code For Insertion In An Ordered List (Part 2)

/* Insert process between curr node and previous node */
prev = queuetab[curr].qprev;  /* Get index of previous node */
queuetab[pid].qnext = curr;
queuetab[pid].qprev = prev;
queuetab[pid].qkey = key;
queuetab[prev].qnext = pid;
queuetab[curr].qprev = pid;
return OK;
/* getitem.c - getfirst, getlast, getitem */

#include "xinu.h"

/*------------------------------------------------------------------------
* getfirst  -   Remove a process from the front of a queue
*------------------------------------------------------------------------*/

pid32 getfirst (qid16 q) /* ID of queue from which to */
                        /* Remove a process (assumed */
                        /* valid with no check) */
{
    pid32 head;

    if (isempty(q)) {
        return EMPTY;
    }

    head = queuehead(q);
    return getitem(queuetab[head].qnext);
}
/*------------------------------------------------------------------------
  *  getlast  - Remove a process from end of queue
  *------------------------------------------------------------------------*/

pid32 getlast(qid16 q)
{
    pid32 tail;

    if (isempty(q)) {
        return EMPTY;
    }

    tail = queuetail(q);
    return getitem(queue.tab[tail].qprev);
}
/*----------------------------- getitem - Remove a process from an arbitrary point in a queue -----------------------------*/

pid32 getitem(
    pid32 pid /* ID of process to remove */
)
{
    pid32 prev, next;

    next = queuetab[pid].qnext; /* Following node in list */
    prev = queuetab[pid].qprev; /* Previous node in list */
    queuetab[prev].qnext = next;
    queuetab[next].qprev = prev;
    return pid;
}
Allocating A New List

/* excerpt from newqueue.c */

qid16 newqueue(void)
{
    static qid16 nextqid=NPROC; /* Next list in queuetab to use */
    qid16 q; /* ID of allocated queue */

    q = nextqid;
    if (q > NQENT) { /* Check for table overflow */
        return SYSERR;
    }

    nextqid += 2; /* Increment index for next call*/

    /* Initialize head and tail nodes to form an empty queue */

    queuetab[queuehead(q)].qnext = queuetail(q);
    queuetab[queuehead(q)].qprev = EMPTY;
    queuetab[queuehead(q)].qkey = MAXKEY;
    queuetab[queuetail(q)].qnext = EMPTY;
    queuetab[queuetail(q)].qprev = queuehead(q);
    queuetab[queuetail(q)].qkey = MINKEY;
    return q;
}
Summary

• An operating system supplies a set of services
• System calls provide interface between OS and application
• Concurrency is fundamental concept
  – Between I/O devices and processor
  – Between multiple computations
• A process is OS abstraction for concurrency; it does not appear in the code
• A process differs from program or function
• You will learn how to design and implement system software that supports concurrent processing
Summary
(continued)

- An OS has well-understood internal components
- Complexity arises from interactions among components
- A multilevel approach helps organize system structure
- OS design involves inventing policies and mechanisms that enforce overall goals
- Xinu includes a compact list structure that uses relative pointers and an implicit data structure to reduce size
- Xinu type names specify both purpose and data size
Module 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 Aspects Of Code Pertinent To An OS

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

![Memory Segments Diagram]

• The stack grows downward (toward lower memory addresses)
• The heap grows upward
A compiler includes global variable names that specify segment addresses:
- Symbol `text` occupies the first byte of the text segment
- Symbol `etext` occupies the first byte beyond the text segment
- Symbol `edata` occupies the first byte beyond the data segment
- Symbol `end` occupies the first byte beyond the bss segment

A programmer can access the names by declaring them `extern`

```c
extern char text, etext, edata, end;
```

Only the addresses are significant; the values are irrelevant.

Note: some assembly languages prepend an underscore to, external names (e.g., `_end`)
Runtime Memory Segments For Xinu Processes

- The code (text segment) is shared
- Global variables (data and bss segments) are shared
- Stacks cannot be shared — each process must have its own stack
Runtime Memory Segments For Xinu Processes (continued)

- 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
Process Management
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 holds
  - 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, the OS must keep the following
  - A unique *process identifier*
  - The owner of the process (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
# Items In A Xinu Process Table

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>prstate</td>
<td>The current status of the process (e.g., whether the process is currently executing or waiting)</td>
</tr>
<tr>
<td>prprio</td>
<td>The scheduling priority of the process</td>
</tr>
<tr>
<td>prstkptr</td>
<td>The saved value of the process’s stack pointer when the process is not executing</td>
</tr>
<tr>
<td>prstkbase</td>
<td>The address of the base of the process’s stack</td>
</tr>
<tr>
<td>prstklen</td>
<td>A limit on the maximum size that the process’s stack can grow</td>
</tr>
<tr>
<td>prname</td>
<td>A name assigned to the process that humans use to identify the process’s purpose</td>
</tr>
</tbody>
</table>
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 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 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., can only happen when running operating system code)
A process is *eligible* if it is either ready to run but not running (i.e., its state is *ready*) or currently executing (i.e., its state is *current*)

To avoid searching the process table during scheduling:

- Keep all ready processes on linked list called a *ready list*
- Order the ready list in descending order by process priority
- Scheduling is efficient because selection of a highest-priority process can be performed in constant time merely by selecting the process at the head of the ready list
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 from the current process to the extracted process
- The current process will be moved back to the ready list if it remains eligible to run
Deferred Rescheduling

- The idea: temporarily delay rescheduling
- Temporarily delays enforcement of the scheduling invariant
  - A call to `resched_cntl(DEFER_START)` defers rescheduling
  - A call to `resched_cntl(DEFER_STOP)` resumes normal scheduling
- Main purpose: allow a device driver to make multiple processes ready before any of them run
- We will see an example later
- For now, just understand that the current process will not change during a deferred rescheduling period; later in the course we will see how deferred rescheduling is used
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.
- When the scheduler switches context, the first process on ready list will be selected.
- Note: the scheduler switches context if the first process on the ready list has priority equal to the current process.
- We will see how the implementation results in round-robin scheduling among equally high-priority processes without a special case in the code.
/* 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];
Xinu Scheduler Code (resched Part 2)

```c
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;
```
Xinu Scheduler Code (resched Part 3)

/*------------------------------------------------------------------------
 * resched_cntl - Control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------*/

status resched_cntl( /* Assumes interrupts are disabled */
    int32 defer /* Either DEFER_START or DEFER_STOP */
)
{
    switch (defer) {
        case DEFER_START: /* Handle a deferral request */
            if (Defer.ndefers++ == 0) {
                Defer.attempt = FALSE;
            }
            return OK;
        case DEFER_STOP: /* Handle end of deferral */
            if (Defer.ndefers <= 0) {
                return SYSERR;
            }
            if ( (--Defer.ndefers == 0) && Defer.attempt ) {
                resched();
            }
            return OK;
        default:
            return SYSERR;
    }
}

CS354 – module 2

Copyright © 2022 by Douglas Comer and CRC Press, Inc. All rights reserved.
Contents Of resched.h

/* resched.h */

/* Constants and variables related to deferred rescheduling */

#define DEFER_START 1  /* Start deferred rescheduling */
#define DEFER_STOP  2  /* Stop deferred rescheduling */

/* Structure that collects items related to deferred rescheduling */

struct defer {
    int32 ndefers;  /* Number of outstanding defers */
    bool8 attempt;  /* Was resched called during the */
                    /* deferral period? */
};

extern struct defer Defer;

• Note: Defer.ndefers is set to zero when the system boots
The Importance/Unimportance Of Process Scheduling

- Facts
  - At one time, process scheduling was the primary research topic in operating systems.
  - Extremely complex scheduling algorithms were created to keep processes proceeding.
  - By the 1990s, interest in scheduling algorithms had faded.
  - Now, almost no one uses complex scheduling algorithms.

- Why did the topic fade?

- Was the problem completely solved?

- Answer: processors became so fast that processing is no longer a scarce resource.
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
• 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
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)

...actually switch to the new process

- 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)
Xinu 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 */

movl %esp,(%eax) /* save the old process’s SP */

movl %esp,(%eax) /* Save old process’s SP */

movl 12(%ebp),%eax /* Get location from which to */

movl %esp,(%eax) /* restore new process’s SP */
Xinu 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 /* 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 */

popfl /* Restore interrupt mask */
add $4,%esp /* Skip saved value of ebp */
ret /* Return to new process */
Xinu Context Switch Code (ARM)

/* ctxsw.S - ctxsw (for ARM) */

.text
.globl ctxsw

/*------------------------------------------------------------------------
* ctxsw - ARM context switch; the call is ctxsw(&old_sp, &new_sp)
*------------------------------------------------------------------------
*/

ctxsw:
    push {r0-r11, lr} /* Push regs 0 - 11 and lr */
    push {lr} /* Push return address */
    mrs r2, cpsr /* Obtain status from coprocessor */
    push {r2} /* and push onto stack */
    str sp, [r0] /* Save old process’s SP */
    ldr sp, [r1] /* Pick up new process’s SP */
    pop {r0} /* Use status as argument and */
    bl restore /* call restore to restore it */
    pop {lr} /* Pick up the return address */
    pop {r0-r12} /* Restore other registers */
    mov pc, r12 /* Return to the new process */
Puzzle #1

- The Intel x86 is a CISC architecture with powerful instructions that may require many clock cycles to execute.
- ARM is a RISC architecture where each instruction performs one basic operation and only requires one clock cycle.
- Why is the Intel context switch code longer if instructions are more powerful?
Puzzle #2

- Our invariant says that at any time, a process must be executing
- The context switch code moves from one process to another
- Question: which process executes the context switch code?
Puzzle #3

- Our invariant says that at any time, one process must be executing
- Consider a situation in which all user processes are blocked (e.g., waiting for input)
- Which process executes?
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)
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 consumes power and 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
Module III

More Process Management:
Process Suspension/Resumption
And Inter-Process Communication
Process Manipulation

• An OS needs system calls that can be used to control processes

• Example operations
  – Suspend a process (keep it from running)
  – Resume a previously-suspended process
  – Block a process to receive a message from another process
  – Send a message to another process

• The OS uses the process state variable to record the status of the process
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 a 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 \textit{PR\_SUSP} to indicate that a process is in the suspended state
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: just call

\[
\text{suspend(getpid())}
\]

• When \textit{suspend} is asked to suspend the current process, it
  – Finds its entry in the process table, \textit{proctab[currpid]}
  – Sets the state in its process table entry to \textit{PR_SUSP}, indicating that it should be suspended
  – Calls \textit{resched} to reschedule to another process
A Note About System Calls

- An operating system contains many functions that 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
- Notes
  - Xinu does not prohibit applications from making direct calls to internal operating system functions or referencing operating system variables
  - However, good programming practice restricts applications to system calls (e.g., use getpid() instead of referencing currpid)
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
- Even trivial operations can cause problems when performed concurrently
The Classic Example Of A Concurrent Access Problem

• Consider incrementing an integer, \( x \)

• To increment \( x \), a programmer writes \( x++ \)

• On most hardware architectures, three instructions are required
  – Load variable \( x \) into a register
  – Add 1 to the register
  – Store the register into variable \( x \)

• An operating system can switch from one process to another between any two instructions

• Surprising consequence: if two processes attempt to increment a shared integer concurrently, errors can result
Illustration Of What Can Happen When Two Processes Attempt To Increment Integer x Concurrently

process 1

load x into register 1

incr register 1

at this point, the operating system switches to process 2

load x into register 2

incr register 2

store register 2 into x

interrupt occurs (context switch)

store register 1 into x
Preventing Concurrent Execution By Disabling Interrupts

- To prevent other processes from changing global data structures, a system call function can disable 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; the function returns a mask value that specifies whether interrupts were previously disabled or enabled.
  - Function `restore` takes as an argument a mask value that was previously obtained from `disable`, 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
*------------------------------------------------------------------------*/
systemcall 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;
    }

    /* Other code... */
};
// 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 == PRREADY) {
    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 the 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, \textit{ready} must re-establish the scheduling invariant.

- Surprisingly, \textit{ready} does not check the scheduling invariant explicitly, but instead simply calls \textit{resched}.

- We can now appreciate the design of \textit{resched}: if the newly ready process has a lower priority than the current process, \textit{resched} returns without switching context, and the current process remains running.
/* 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;
    }

/* Additional code */
The Resume System Call (Part 2)

```c
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., the scheduler needs to find the highest priority ready process)
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
Something To Think About

- Resume returns the priority of the resumed process
- The code
  - Extracts the priority from the process table entry
  - Makes the process ready
  - Returns the extracted priority to its caller
- Is the value returned guaranteed to be the priority of the process?
- Remember that in a concurrent environment, other processes can run at any time, and an arbitrary amount of time can pass between any two instructions
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*
Two Approaches To Message Passing

• Approach #1
  – Message passing is one of many services the operating system offers
  – Messages are basically data items sent from one process to another, and are independent of both normal I/O and process synchronization services
  – Message passing functions are implemented using lower-level mechanisms

• Approach #2
  – The entire operating system is message-based
  – Messages, not function calls, provide the fundamental building block
  – Messages are used to coordinate and control processes

• Note: a few research projects used approach #2, but most systems use approach #1
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
Why Message Passing Choices Are Difficult

- Message passing interacts with scheduling
  - Process $A$ sends a message to process $B$
  - Process $B$ does not check messages
  - Process $C$ sends a message to process $B$
  - Process $B$ eventually checks its messages
  - If process $C$ has higher priority than $A$, should $B$ receive the message from $C$ first?

- Message passing affects memory usage
  - If messages are stored with a receiver, senders can use up all the receiver’s memory by flooding the receiver with messages
  - If messages are stored with a sender, receivers can use up all the sender’s memory by not accepting messages
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
An Example Message Passing Facility
(continued)

- The system uses “first-message” semantics
  - The first message sent to a process is stored until it has been received
  - Subsequent attempts to send to the process fail
How To Use First-Message Semantics

• The idea: wait for one of several events to occur

• Example events
  – I/O completes
  – A user presses a key
  – Data arrives over a network
  – A hardware indicator signals a low battery

• To use message passing facility to wait for the first event
  – Create a process for each event
  – When the process detects its event, have it send a message
How To Use First-Message Semantics (continued)

• The idiom a receiver uses to identify the first event that occurs

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

• The above code returns first message that is sent, even if a higher priority process attempts to send later

• The receiver will block until a message arrives
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

Diagram:
- **READY**:
  - send to **RECEIVING**
  - resched to **CURRENT**
  - receive to **SUSPENDED**
  - suspend to **SUSPENDED**
- **RECEIVING**:
  - receive to **CURRENT**
  - send to **READY**
- **CURRENT**:
  - resched to **READY**
  - suspend to **SUSPENDED**
- **SUSPENDED**:
  - resume to **READY**
  - suspend to **SUSPENDED**
The Steps Taken To Receive A Message

- The current process calls *receive*
- *Receive* checks the current process’s entry in the process table
- If no message has arrived, *receive* moves the calling process to the *RECEIVING* state to block until a message arrives
- Once a message arrives, the process is moved to the *READY* state and execution of *receive* will eventually continue when resched chooses to run the process
- The code in *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* then returns the message to its caller
Blocking To Wait For A Message

- We have seen how the `suspend` function suspends the current process.
- Blocking the current process receive a message is almost the same.

**Receive**

- Finds the current process’s entry in the process table, `proctab[currid]`.
- Sets the state in the process table entry to `PR_RECV`, indicating that the process will be receiving.
- Calls `resched`.
/* 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;
    }
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;
/* 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[curpid];
    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.
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
Producer-Consumer Synchronization

- Typical scenario: a FIFO buffer shared by multiple processes
  - Processes that deposit items into the buffer are called *producers*
  - Processes that extract items from the buffer are called *consumers*
- The programmer must guarantee
  - When the buffer is full, a producer will block until space is available
  - When the buffer is empty, a consumer will block until an item has been deposited
- A given process may act as a consumer for one buffer and a producer for another
- Example: in Unix pipeline, a process may read input from one pipe and write output to another

```bash
cat employees | grep Name: | sort
```
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).
- Many data operations are non-atomic, which means a sequence of multiple operations are used to change a data item.
- Programmers must take steps to ensure that when one process executes a sequence of operations to change a data item, no other process can attempt to make changes concurrently.
Recall

- Even trivial changes to a shared variable (e.g., x++) can require a sequence of hardware operations
- Anyone working with concurrent processes must guard *every* access to shared data items
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 mechanisms that disable and restore interrupts
  – Hardware spin lock instructions
  – Semaphores (implemented in software)
Handling Mutual Exclusion With Spin Locks

- Used in multicore CPUs; does not work for a single processor
- A special hardware operation allows a core to test and/or set a special lock atomically
- The lock may consist of special hardware or may be a location in memory
- The hardware guarantees that only one core will be allowed to set the lock at any time
- The mechanism is known as a spin lock 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 was once known as test-and-set
An Example Of A Spin Lock (x86)

• An instruction performs an atomic compare and exchange (\textit{cmpxchg})

• Spin loop: repeat the following
  – Place an “unlocked” value (e.g., 0) in register \textit{eax}
  – Place a “locked” value (e.g., 1) in register \textit{ebx}
  – Place the address of a memory location to be used as a lock in register \textit{ecx}
  – Execute the \textit{cmpxchg} instruction
  – Register \textit{eax} will contain the value of the lock before the compare and exchange occurred
  – Continue the spin loop as long as \textit{eax} contains the “locked” value

• To release the lock, assign the “unlocked” value to the lock location in memory
Example Spin Lock Code For X86 (Part 1)

/* mutex.S - mutex_lock, mutex_unlock */

.text
.globl mutex_lock
.globl mutex_unlock

/*------------------------------------------------------------------------
* mutex_lock(uint32 *lock) -- Acquire a lock
*------------------------------------------------------------------------
*/
mutex_lock:

/* Save registers that will be modified */

pushl %eax
pushl %ebx
pushl %ecx

Copyright © 2022 by Douglas Comer and CRC Press, Inc. All rights reserved.
Example Spin Lock Code For X86 (Part 2)

```
spinloop:
    movl $0, %eax /* Place the "unlocked" value in eax */
    movl $1, %ebx /* Place the "locked" value in ebx */
    movl 16(%esp), %ecx /* Place the address of the lock in ecx */

    lock cmpxchg %ebx, (%ecx) /* Atomic compare-and-exchange: */
    /* Compare %eax with memory (%ecx) */
    /* if equal */
    /* load %ebx in memory (%ecx) */
    /* else */
    /* load %ebx in %eax */

    /* If eax is 1, the mutex was locked, so continue the spin loop */
    cmp $1, %eax
    je    spinloop

    /* We hold the lock now, so pop the saved registers and return */
    popl  %ecx
    popl  %ebx
    popl  %eax
    ret
```
Example Spin Lock Code For X86 (Part 3)

/*------------------------------------------------------------------------
 * mutex_unlock (uint32 *lock) - release a lock
 *------------------------------------------------------------------------
 */

mutex_unlock:

    /* Save register eax */
    pushl    %eax

    /* Load the address of lock onto eax */
    movl     8(%esp), %eax

    /* Store the "unlocked" value in the lock, thereby unlocking it */
    movl     $0, (%eax)

    /* Restore the saved register and return */
    popl     %eax
    ret
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
Mutual Exclusion Inside An Operating System

Several possible approaches have been used

Examples: allow only one process at a time to

- Run operating system code
- Run a given operating system function
- Access a given operating system component (a single component may comprise multiple functions)

Allowing more processes to execute concurrently increases performance

The general principle is:

to maximize performance, choose the smallest possible granularity for mutual exclusion
Low-Level Mutual Exclusion

• Mutual exclusion is needed in two places
  – In application processes
  – Inside the 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
Interrupt Mask

- A hardware mechanism that controls interrupts
- Implemented by an internal machine register, and may be part of *processor status word*
- On some hardware, a zero value means interrupts can occur; on other hardware, a non-zero value means interrupts can occur
- The OS can
  - Examine the current interrupt mask (find out whether interrupts are enabled)
  - Set the interrupt mask to prevent interrupts
  - Clear the interrupt mask to allow interrupts
Masking Interrupts

• Important principle:

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

• Technique used: a given function
  – Saves the current interrupt status
  – *Disables* interrupts
  – Proceeds through a critical section
  – *Restores* the interrupt status from the saved copy

• Key insight: save/restore allows nested calls
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 (and some device interrupts must be serviced within a specifies period)
  – It restricts execution to one process for the entire system

• Disabling interrupts can interfere with the scheduling invariant and lead to a priority inversion where a low-priority process prevents execution of 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

• When used, a programmer must minimize the amount of time interrupts remain disabled
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 compete for access to each critical section independent of other critical sections
  – Provide an access policy that specifies how waiting processes gain access
• Good news: a single mechanism, the counting semaphore, solves the problem
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)
  - \textit{Count} is an integer
  - \textit{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

\[
\begin{align*}
\text{semid} & = \text{semcreate}(\text{initial\_count}) & \text{Creates a semaphore and returns an ID} \\
\text{semdelete}(\text{semid}) & & \text{Deletes the specified semaphore} \\
\text{wait}(\text{semid}) & & \text{Waits on the specified semaphore} \\
\text{signal}(\text{semid}) & & \text{signals the specified semaphore}
\end{align*}
\]
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
  
  \[ \text{sid} = \text{semcreate}(1); \]
- Use: bracket each critical section in the code with calls to \textit{wait} and \textit{signal}
  
  \begin{verbatim}
  wait(sid);
  ...	ext{critical section to use the shared item}...
  signal(sid);
  \end{verbatim}
- All processes must agree to use semaphores (hence the term \textit{cooperative})
- Only one process will access the critical section at any time (others will be blocked)
A Potential Problem: Deadlock

- Consider two processes that use semaphores to protect two data items, x and y.
- The two semaphores are created:
  
  \[
  \text{semidx} = \text{semcreate}(1); \quad \text{semidy} = \text{semcreate}(1);
  \]
- Then the two processes take the following steps:

  /* Process 1 */
  
  ... 
  wait(semidx);
  start to modify x
  wait(semidy);
  modify y
  signal(semidy);
  finish modifying x
  signal(semidx);

  /* Process 2 */
  
  ... 
  wait(semidy);
  start to modify y
  wait(semidx);
  deadlock! → modify x
  signal(semidx);
  finish modifying y
  signal(semidy);
When Using Semaphores For Mutual Exclusion

- Good news: counting semaphores work well when a set of processes needs exclusive access to a single resource
- Bad news: using semaphores with multiple resources can be tricky
- To avoid trouble
  - Limit mutual exclusion to a single resource at any time, when possible
  - When processes must obtain exclusive access to multiple resources, insure that all processes access and release the resources in the same order
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);
  }
  ```
The consumer algorithm

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

- **READY**
  -suspend
  -resume

- **SUSPENDED**
  -create
  -suspend

- **CURRENT**
  -resched
  -suspend

- **WAITING**
  -wait
  -signal
  -resched

- **RECEIVING**
  -send
  -receive
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)

#include <xinu.h>

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 semaphore 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)

```c
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 */
}
```

• 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
Consequences Of A Semaphore Queuing Policy

- 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?
Choosing A Semaphore Queueing Policy

• The choice is difficult
• There is no single best answer
  – Fairness not easy to define
  – Scheduling and coordination interact in subtle ways
  – The choice may affect other OS policies
• The interactions of heuristic policies may produce unexpected results
The Semaphore Queuing Policy In Xinu

- Xinu uses first-come-first-served
- The approach has several advantages
  - Is straightforward to implement
  - Is extremely efficient
  - Works well for traditional uses of semaphores
  - Guarantees all contending processes will obtain access
- The FIFO approach has an interesting disadvantage: a low-priority process can obtain access to a resource while a high-priority process remains blocked
Implementation Of Xinu’s FIFO Semaphore Policy

- Recall: each semaphore has a list of processes
- For a FIFO policy, the list is treated as a queue
- When it needs to insert the current process on a list, *wait* enqueues the calling process at the tail of the queue
- When it chooses a waiting process to run, *signal* selects the process at the head of the queue
- The code for signal follows
/* signal.c - signal */
#include <xinu.h>

/*------------------------------------------------------------------------
* signal - Signal a semaphore, releasing a process if one is waiting
*------------------------------------------------------------------------*/
systemkner 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)

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

- Notice how little code is required to signal a semaphore
Possible Semaphore Creation Strategies

- **Static**
  - All semaphores are defined at compile time
  - The approach is more efficient, but less powerful

- **Dynamic**
  - Semaphores are created at runtime
  - The approach is more flexible

- Xinu supports dynamic semaphore allocation, but to achieve efficiency preallocates a fixed-size array of possible semaphores
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)

```c
/*------------------------------------------------------------------------
* 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++; /* Next semaphore index to try */
        if (nextsem >= NSEM) nextsem = 0; /* Iterate through # entries */
        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 a semaphore is deleted
- We must choose how to dispose of each waiting process
- The Xinu disposition policy: if a process is waiting on a semaphore when the semaphore is deleted, the process becomes ready
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
* semdelete - Delete a semaphore by releasing its table entry
*------------------------------------------------------------------------*/
system semdelete(
    sid32 sem /* ID of semaphore to delete */
)
{
    intmask mask; /* Saved interrupt mask */
    struct sentry *semptr; /* Ptr to semaphore 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;
}
Deferred rescheduling allows all waiting processes to be made ready before any of them to run

Before it ends deferred rescheduling, semdelete ensures the semaphore data structure is ready for other processes to use
Do you understand semaphores?
Semaphore Behavior (A True Story)

- A process creates a semaphore
  
  ```
  mutex = semcreate(1);
  ```

- Three processes then execute the following code
  
  ```
  process convoy(char_to_print)
  do forever {
    think (i.e., use CPU);
    wait(mutex);
    print(char_to_print);
    signal(mutex);
  }
  ```

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

- The initial output is
  - 20 A’s, 20 B’s, 20 C’s, 20 A’s, etc.

- After tens of seconds, however, the output becomes $ABCABCABC...$

- Facts
  - Everything is correct
  - No other processes are executing
  - The output is nonblocking (i.e., it uses polled I/O)
The Convoy
(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 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
Module V

Low-Level Memory Management
Process Creation And Termination
Low-Level Memory Management
Location Of Low-Level Memory Management In The Hierarchy
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
  
  \[
  \text{addr} = \text{getstk}(\text{numbytes});
  \]
  
  \[
  \text{freestk}(\text{addr}, \text{numbytes});
  \]

- Two functions control allocation of heap storage
  
  \[
  \text{addr} = \text{getmem}(\text{numbytes});
  \]
  
  \[
  \text{freemem}(\text{addr}, \text{numbytes});
  \]

- Memory is allocated until none remains

- Only \textit{getmem/freemem} are intended for use by Xinu application processes; \textit{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 exits
  - 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 (a total of 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 allocated 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 a process stack
    * For I/O buffers
    * For pointers
  - Free memory blocks are kept on free list
  - One cannot allocate/free an individual byte of memory efficiently
- 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 (variable *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 `memblok`
- As the last node shows, the length includes the bytes used by the header
- 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
  - For getmem allocated the lowest addresses in the block
  - For getstk allocated the highest addresses in the block
When Searching The Free List

- Use two pointers that point to two successive nodes on the list
- An invariant controls the pointers during the search
  - Pointer $curr$ points to a node on the free list (or $NULL$, if at the end of the list)
  - Pointer $prev$ points to the previous node (or $memlist$, if at the beginning of the list)
- The invariant is established initially by making $prev$ point to $memlist$ and making $curr$ point to the item to which $memlist$ points
- The invariant must be maintained each time pointers move along the list
```c
/* 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 \textit{getmem} chooses a block that is larger then the requested size
- \textit{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 \textit{next} to point to a block on the free list, and \textit{prev} to point to the previous block (or \textit{memlist})

- Stop when the address of the block being freed lies between \textit{prev} and \textit{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 free list, combine the new and existing block into one larger block
- Note: the code must check for coalescing with
  - The preceding block only
  - The following block only
  - Both the preceding and following blocks
Xinu Freemem (Part 1)

/* freemem.c - freemem */

#include <xinu.h>

/*------------------------------------------------------------------------
* freemem - Free a memory block, returning the block to the free list
*------------------------------------------------------------------------
*/
systemc 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)

```c
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;
```
/* 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;
/* 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;

    fitsprev = NULL;
    while (curr){
        if (fits == NULL || fits->size < curr->size){
            fits = curr;
            fitsprev = prev;
        }
        prev = curr;
        curr = curr->mnext;
    }

    if (fitsprev == NULL){
        return (char *)fits->mem
    } else {
        return (char *)(fitsprev->mem + fitsprev->size - 1);
    }
}
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 for efficiency

Technique

− Convert address from the highest address in block being freed to the lowest address in the block

− Call freemem with the converted address
Process Creation
And Termination
Process Creation

- Process creation and termination use the memory manager
- Creation
  - Allocates a stack for the process being created
  - Fills in process table entry
  - Fills in the process’s stack to have a valid frame
- Two design decisions
  - Choose an initial state for the process
  - Choose an action for the case where a process “returns” from the top-level function
The Xinu Design

- The initial state of a new process
  - A process is created in the suspended state
  - Consequence: execution can only begin after the process is resumed
- Return from top-level function
  - Causes the process to exit (similar to Unix)
  - Implementation: place a “pseudo call” on the stack (make it appear that the top-level function in the process was called)
  - Initialize the return address in the pseudo call to \textit{INITRET}
- Note: \textit{INITRET} is defined to be function \textit{userret}
- Function \textit{userret} causes the current process to exit
Xinu Function Userret

/* userret.c - userret */

#include <xinu.h>

/*================================================================--------
   * userret - Called when a process returns from the top-level function
   *------------------------------------------------------------------------
   */

void userret(void)
{
    kill(getpid()); /* Force process to exit */
}
The Pseudo Call On An Initial Stack

- Seems straightforward
- Is actually extremely tricky
- The trick: arrange the stack as if the new process was stopped in a call to \textit{ctxsw}
- Several details make it difficult
  - \textit{Ctxsw} runs with interrupts disabled, but a new process should start with interrupts enabled
  - We must store arguments for the new process so that the top-level function receives them
- We will examine code for process creation after looking at process termination
Process Termination
Killing A Process

- Formally known as *process termination*
- The action taken depends on the state of the process
  - If a process is on a list, it must be removed
  - If a process is waiting on a semaphore, the semaphore count must be adjusted
- In Xinu, function *kill* implements process termination
Xinu Implementation Of Kill (Part 1)

/* kill.c - kill */

#include <xinu.h>

/*------------------------------------------------------------------------
  * kill - Kill a process and remove it from the system
  *------------------------------------------------------------------------
 */
syscall kill(
    pid32 pid  /* ID of process to kill */
)
{
    intmask mask;  /* Saved interrupt mask */
    struct procent *prptr;  /* Ptr to process’ table entry */
    int32 i;  /* Index into descriptors */

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

    if (--prcount <= 1) {  /* Last user process completes */
        xdone();
    }
}
Xinu Implementation Of Kill (Part 2)

```c
send(prptr->prparent, pid);
freestk(prptr->prstkbase, prptr->prstklen);

switch (prptr->prstate) {
    case PR_CURR:
        prptr->prstate = PR_FREE; /* Suicide */
        resched();
    case PR_SLEEP:
    case PR_RECTIM:
        unsleep(pid);
        prptr->prstate = PR_FREE;
        break;
    case PR_WAIT:
        semtab[prptr->prsem].scount++;
        /* Fall through */
    case PR_READY:
       getitem(pid); /* Remove from queue */
        /* Fall through */
    default:
        prptr->prstate = PR_FREE;
}
restore(mask);
return OK;
```
Killing The Current Process

- Look carefully at the code
  - Step 1: free the process’s stack
  - Step 2: perform other actions
- Consider what happens when a current process kills itself: the call to `resched` occurs after the process’s stack has been freed
- Why does it work?
- Answer: because in Xinu, even after stack has been freed, the memory is still available to the process
The Xdone Function

• Function xdone is called when the count of user processes reaches zero
• Nothing further will happen — only the null process remains running
• The function prints a warning message for the user

/* xdone.c - xdone */

#include <xinu.h>

/*------------------------------------------------------------------------
 * xdone - Print system completion message as last process exits
 *------------------------------------------------------------------------
 */
void xdone(void)
{
    kprintf("\n\nAll user processes have completed.\n\n");
    halt();  /* Halt the processor */
}
Process Creation
The Steps For Process Creation

- Allocate a process table entry
- Allocate a stack
- Place values on the stack as if the top-level function was called (pseudo-call)
- Arrange the saved state on the stack so context switch can switch to the process
- Details depend on
  - The hardware and calling conventions
  - The way context switch is written
- Consider example code for ARM and x86 processors
Process Creation On ARM (Part 1)

/* create.c - create, newpid */

#include <xinu.h>

local int newpid();

#define roundew(x) ( (x+3)& ~0x3)

/*------------------------------------------------------------------------
* create - create a process to start running a procedure
*------------------------------------------------------------------------*/

pid32 create(
    void *procaddr, /* procedure address */
    uint32 ssize, /* stack size in bytes */
    pri16 priority, /* process priority > 0 */
    char *name, /* name (for debugging) */
    uint32 nargs, /* number of args that follow */
    ...
)
{
    intmask mask; /* interrupt mask */
    pid32 pid; /* stores new process id */
    struct procent *prptr; /* pointer to proc. table entry */
    int32 i;
    uint32 *a; /* points to list of args */
    uint32 *saddr; /* stack address */
    ...
mask = disable();
if (ssize < MINSTK)
    ssize = MINSTK;
ssize = (uint32) roundew(ssize);
if (((saddr = (uint32 *)getstk(ssize)) ==
    (uint32 *)SYSERR) || (pid = newpid()) == SYSERR || priority < 1)
    {
    restore(mask);
    return SYSERR;
}
prcount++;
prptr = &proctab[pid];

/* initialize process table entry for new process */
prptr->prstate = PR_SUSP;  /* initial state is suspended */
prptr->prprio = priority;
prptr->prstkbase = (char *)saddr;
prptr->prstklen = ssize;
prptr->prname[PNMLEN-1] = NULLCH;
for (i = 0; i < PNMLEN-1 && (prptr->prname[i] = name[i]) != NULLCH; i++)
    {
    prptr->prsem = -1;
    prptr->prparent = (pid32)getpid();
    prptr->prhasmsg = FALSE;
}
/* set up initial device descriptors for the shell */
prptr->prdesc[0] = CONSOLE; /* stdin is CONSOLE device */
prptr->prdesc[1] = CONSOLE; /* stdout is CONSOLE device */
prptr->prdesc[2] = CONSOLE; /* stderr is CONSOLE device */

/* Initialize stack as if the process was called */
*saddr = STACKMAGIC;

/* push arguments */
a = (uint32 *)(nargs + 1); /* start of args */
a += nargs - 1; /* last argument */
for (; nargs > 4; nargs--) /* machine dependent; copy args */
  *--saddr = *(a--); /* onto created process's stack */
*--saddr = (long)procaddr;
for (i = 11; i >= 4; i--) {
  *--saddr = 0;
  for (i = 4; i > 0; i--) {
    if (i <= nargs)
      *--saddr = *(a--);
    else
      *--saddr = 0;
  }
}

*--saddr = (long)INITRET; /* push on return address */
*--saddr = (long)0x00000053; /* CPSR F bit set, */
/* Supervisor mode */

prptr->prstkptr = (char *)saddr;
restore(mask);
return pid;
/**-----------------------------------------**
* newpid    -  Obtain a new (free) process ID*
*-----------------------------------------**
*/

local pid32 newpid(void) {
    uint32 i; /* iterate through all processes*/
    static pid32 nextpid = 1; /* position in table to try or */
                   /* one beyond end of table */

    /* check all NPROC slots */
    for (i = 0; i < NPROC; i++) {
        nextpid %= NPROC; /* wrap around to beginning */
        if (proctab[nextpid].prstate == PR_FREE) {
            return nextpid++;
        } else {
            return (pid32) SYSERR;
        }
    }

    return (pid32) SYSERR;
}
Process Creation On X86 (Part 1)

/* create.c - create, newpid */

#include <xinu.h>

local int newpid();

/*------------------------------------------------------------------------
* create - Create a process to start running a function on x86
*------------------------------------------------------------------------*/

pid32 create(
    void *funcaddr, /* Address of the function */
    uint32 ssize, /* Stack size in bytes */
    pri16 priority, /* Process priority > 0 */
    char *name, /* Name (for debugging) */
    uint32 nargs, /* Number of args that follow */
    ...
)
{
    uint32 savsp, *pushsp;
    intmask mask; /* Interrupt mask */
    pid32 pid; /* Stores new process id */
    struct procent *prptr; /* Pointer to proc. table entry */
    int32 i;
    uint32 *a; /* Points to list of args */
    uint32 *saddr; /* Stack address */
mask = disable();
if (ssize < MINSTK)
    ssize = MINSTK;
ssize = (uint32) roundmb(ssize);
if ( (priority < 1) || ((pid=newpid()) == SYSERR) ||
    ((saddr = (uint32 *)getstk(ssize)) == (uint32 *)SYSERR) ) {
    restore(mask);
    return SYSERR;
}
prcount++;
prptr = &proctab[pid];

/* Initialize process table entry for new process */
prptr->prstate = PR_SUSP;    /* Initial state is suspended */
prptr->prprio = priority;
prptr->prstkbase = (char *)saddr;
prptr->prstklen = ssize;
prptr->prname[PNMLEN-1] = NULLCH;
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH; i++)
    ;
prptr->prsem = -1;
prptr->prparent = (pid32)getpid();
prptr->prhasmsg = FALSE;
Process Creation On X86 (Part 3)

/* Set up stdin, stdout, and stderr descriptors for the shell */
prptr->prdesc[0] = CONSOLE;
prptr->prdesc[1] = CONSOLE;
prptr->prdesc[2] = CONSOLE;

/* Initialize stack as if the process was called */

*saddr = STACKMAGIC;
savsp = (uint32)saddr;

/* Push arguments */
a = (uint32 *)(&nargs + 1); /* Start of args */
a += nargs -1; /* Last argument */
for ( ; nargs > 0 ; nargs--)
    *--saddr = *a--; /* Machine dependent; copy args */
    *--saddr = (long)INITRET; /* onto created process' stack*/
    *--saddr = (long)INITRET; /* Push on return address */
Process Creation On X86 (Part 4)

/* The following entries on the stack must match what ctxsw expects a saved process state to contain: ret address, ebp, interrupt mask, flags, registers, and an old SP */
*--saddr = (long)funcaddr; /* Make the stack look like it’s half-way through a call to ctxsw that "returns" to the new process */
*--saddr = savsp; /* This will be register ebp for process exit */
savsp = (uint32) saddr; /* Start of frame for ctxsw */
*--saddr = 0x00000200; /* New process runs with interrupts enabled */

/* Basically, the following emulates an x86 "pushal" instruction*/
*--saddr = 0; /* %eax */
*--saddr = 0; /* %ecx */
*--saddr = 0; /* %edx */
*--saddr = 0; /* %ebx */
*--saddr = 0; /* %esp; value filled in below */
pushsp = saddr; /* Remember this location */
*--saddr = savsp; /* %ebp (while finishing ctxsw) */
*--saddr = 0; /* %esi */
*--saddr = 0; /* %edi */
pushsp = (unsigned long) (prptr->prstkptr = (char *)saddr);
restore(mask);
return pid;
}
Process Creation On X86 (Part 5)

/*================================================================--------
* newpid - Obtain a new (free) process ID
*----------------------------------------------------------------------*/

local pid32 newpid(void)
{
    uint32 i; /* Iterate through all processes*/
    static pid32 nextpid = 1; /* Position in table to try or */
                               /* one beyond end of table */

    /* Check all NPROC slots */

    for (i = 0; i < NPROC; i++) {
        nextpid %= NPROC; /* Wrap around to beginning */
        if (proctab[nextpid].prstate == PR_FREE) {
            return nextpid++;
        } else {
            nextpid++;
        }
    }

    return (pid32) SYSERR;
}
An Assessment Of Process Creation

- Process creation code is among the most difficult pieces of code to understand.
- One must know:
  - The hardware architecture
  - The function calling conventions
  - The way `ctxsw` chooses to save state
  - How interrupts are handled
- As you struggle to understand it, imagine trying to write such code.
Summary

• To preserve a multi-level hierarchy, the memory manager is divided into two pieces
  – A low-level manager is used in kernel to allocate address spaces
  – A high-level manager is used to handle abstractions of virtual memory and paging within a process’s address space

• The Xinu low-level manager offers two types of allocation
  – Memory for a process stack
  – Memory from the heap

• Stack requests tend to repeat the same size
Summary
(continued)

• The Xinu low-level memory manager
  – Places all free memory on a single list
  – Rounds all requests to multiples of \texttt{struct memblk}
  – Uses first-fit allocation for heap requests and last-fit allocation for stack requests
• Process creation and termination use the memory manager to allocate and free process stacks
• \textit{Create} handcrafts an initial stack as if the top-level function had been called; the stack includes a return address given by constant \texttt{INITRET}
Module VI

High-Level
Memory Management
Location Of High-Level Memory Management In The Hierarchy
Our Approach To Memory Management (Review)

- Divide the memory manager into two pieces
- Low-level piece
  - A basic facility
  - Provides functions for stack and heap allocation
  - Treats memory as exhaustible resource
- High-level piece
  - Accommodates other memory uses
  - Assumes both operating system modules and sets of applications need dynamic memory allocation
  - Prevents exhaustion
Motivation For Memory Partitioning

- Competition exists for kernel memory
- Many subsystems in the operating system
  - Allocate blocks of memory
  - Have needs that change dynamically
- Examples
  - The disk subsystem allocates buffers for disk blocks
  - The network subsystem allocates packet buffers
- Interaction among subsystems can be subtle and complex
Managing Memory Demands

- Overall goals can conflict
  - Protect information
  - Share information
- Extremes
  - Xinu has much sharing and almost no protection
  - The original Unix™ system had much protection and almost no sharing
**The Concept Of Subsystem Isolation**

- An OS designer desires
  - Predictable behavior
  - Provable assertions (e.g., “network traffic will never deprive the disk driver of buffers”)

- The reality
  - Subsystems are designed independently; there is no global policy or guarantee about their memory use
  - If one subsystem allocates memory excessively, others can be deprived

- Conclusions
  - We must not treat memory as a single, global resource
  - We need a way to isolate subsystems from one another
Providing Abstract Memory Resources

Assertion: to be able to make guarantees about subsystem behavior, one must partition memory into abstract resources with each resource dedicated to one subsystem.
A Few Examples Of Abstract Resources

- Disk buffers
- Network buffers
- Message buffers
- A separate address space for each process as in Unix
- Inter-process communication buffers (e.g., Unix pipes)

Note that
- Each subsystem should operate safely and independently
- An operating system designer may choose to define finer granularity separations
  * A separate set of buffers for each network interface (Wi-Fi and Ethernet)
  * A separate set of buffers for each disk
The Xinu High-Level Memory Manager

- Partitions memory into groups of *buffer pools*
- Each pool is created once and persists until the system shuts down
- All buffers in a given pool are the same size
- At pool creation, the caller specifies the
  - The size of buffers in the pool
  - The number of buffers in the pool
- Once a pool has been created, buffer allocation and release is dynamic
- The system provides a completely synchronous interface
Xinu Buffer Pool Functions

- poolinit: Initialize the entire buffer pool mechanism
- mkbufpool: Create a pool
- getbuf: Allocate buffer from a pool
- freebuf: Return buffer to a pool

- Memory for a pool is allocated by \textit{mkbufpool} when the pool is formed

Although the buffer pool system allows callers to allocate a buffer from a pool and later release the buffer back to the pool, the pool itself cannot be deallocated, which means that the memory occupied by the pool can never be released.
The Traditional Approach To Identifying A Buffer

- Most systems use the address of lowest byte in the buffer as the buffer address
- Doing so means
  - Each buffer is guaranteed to have a unique ID
  - A buffer can be identified by a single pointer
- The scheme
  - Works well in C
  - Is convenient for programmers
Consequences Of Using A Single Pointer As An ID

• Consider function \textit{freebuf}
  – It must return a buffer to the correct pool
  – It takes the buffer identifier as argument
• Information about buffer pools must be kept in a table
• Given a buffer, \textit{freebuf} needs to find the pool from which the buffer was allocated
Finding The Pool To Which A Buffer Belongs

- Obvious possibilities
  - Search the table of buffer pools to find the correct pool
  - Use an external data structure to map a buffer address to the correct pool (e.g., keep a list of allocated buffers and the pool to which each belongs)

- An alternative
  - Have `getbuf` pass the caller two values: a pool ID and a buffer address
  - Have `freebuf` take two arguments: a pool ID and a buffer address

- Unfortunately, using two arguments
  - Is inconvenient for programmers
  - Does not work well in C
Solving The Single Pointer Problem

• Xinu uses a clever trick to avoid passing two values
  – Use the address of the lowest usable byte as a buffer identifier
  – Store a pool ID along with each buffer, but hide it from the user

• The implementation
  – When allocating a buffer, allocate enough extra bytes to hold the pool ID
  – Store the pool ID in the extra bytes
  – Place the extra bytes before the buffer
  – Return a pointer to the buffer, not the extra bytes

• A process can use a buffer without knowing that the extra bytes exist
Illustration Of A Pool ID Stored With A Buffer

- Xinu allocates four bytes more than the user specifies
- Conceptually, the additional bytes precede the buffer, and are used to store the ID of the buffer pool
- `getbuf` returns a single pointer to the data area of the buffer (beyond the extra bytes)
- `freebuf` expects the same pointer that `getbuf` returns to a caller
- The pool ID is transparent to applications using the buffer pool
Potential Downsides Of The Xinu Scheme

- Some device hardware requires a buffer to start on a page boundary, but adding four bytes to the size may ruin alignment.
- If the pool id is accidentally overwritten, the buffer will either be returned to the wrong pool or an error will occur because the pool ID is invalid.
Buffer Pool Operations

- Create a pool (**mkpool**)
  - Increase the requested buffer size by 4 to hold a pool ID
  - Use **getmem** to allocate memory for all the buffers that will be in the pool
  - Form a singly-linked list of the buffers (storing links in the buffers themselves)
  - Allocate a semaphore to count buffers
  - Return an ID of the allocated buffer pool

- Allocate a buffer from a pool (**getbuf**)
  - Take the pool ID as an argument, and use it locate the correct buffer pool
  - *Wait* on the semaphore associated with a pool (i.e., block until a buffer is available)
  - Extract a buffer from the free list, insert the ID, and return the buffer to the caller
Buffer Pool Operations (continued)

- Free (deallocation) a previously-allocated buffer (freebuf)
  - Extract the pool ID from the extra bytes that precede the buffer
  - Use the pool ID to locate the buffer pool
  - Insert the buffer at the head of the list for the pool
  - Signal the semaphore associated with the pool
Xinu Mkbufpool (Part 1)

/* mkbufpool.c - mkbufpool */

#include <xinu.h>

/*------------------------------------------------------------------------
* mkbufpool - Allocate memory for a buffer pool and link the buffers
*------------------------------------------------------------------------*/
bpid32 mkbufpool(
    int32 bufsiz, /* Size of a buffer in the pool */
    int32 numbufs /* Number of buffers in the pool*/
)
    
    intmask mask; /* Saved interrupt mask */
    bpid32 poolid; /* ID of pool that is created */
    struct bpentry *bpptr; /* Pointer to entry in buftab */
    char *buf; /* Pointer to memory for buffer */

    mask = disable();
    if (bufsiz<BP_MINB || bufsiz>BP_MAXB
        || numbufs<1 || numbufs>BP_MAXN
        || nbpools >= NBPOOLS) {
        restore(mask);
        return (bpid32)SYSERR;
    }

/* Round request to a multiple of 4 bytes */

bufsiz = ( (bufsiz + 3) & (~3) );

buf = (char *)getmem( numbufs * (bufsiz+sizeof(bpid32)) );
if ((int32)buf == SYSERR) {
    restore(mask);
    return (bpid32)SYSERR;
}

poolid = nbpools++;
bpptr = &buftab[poolid];
bpptr->bpnext = (struct bpentry *)buf;
bpptr->bpsize = bufsiz;
if ( (bpptr->bpsem = semcreate(numbufs)) == SYSERR) {
    freemem(buf, numbufs * (bufsiz+sizeof(bpid32)) );
    nbpools--;
    restore(mask);
    return (bpid32)SYSERR;
}

bufsiz+=sizeof(bpid32);
for (numbufs-- ; numbufs>0 ; numbufs-- ) {
    bpptr = (struct bpentry *)buf;
    buf += bufsiz;
    bpptr->bpnext = (struct bpentry *)buf;
}

bpptr = (struct bpentry *)buf;
bpptr->bpnext = (struct bpentry *)NULL;
restore(mask);
return poolid;
Xinu Getbuf (Part 1)

/* getbuf.c - getbuf */

#include <xinu.h>

/*------------------------------------------------------------------------
* getbuf - Get a buffer from a preestablished buffer pool
*----------------------------------------------------------------------
*/

char *getbuf(
    bpid32 poolid /* Index of pool in buftab */
)
{
    intmask mask; /* Saved interrupt mask */
    struct bpentry *bp.ptr; /* Pointer to entry in buftab */
    struct bpentry *bufptr; /* Pointer to a buffer */

    mask = disable();

    /* Check arguments */

    if ( (poolid < 0 || poolid >= nbpools) ) {
        restore(mask);
        return (char *)SYSERR;
    }

    bpptr = &buftab[poolid];
/* Wait for pool to have > 0 buffers and allocate a buffer */

wait(bpptr->bpsem);
bufptr = bpptr->bpnext;

/* Unlink buffer from pool */

bpptr->bpnext = bufptr->bpnext;

/* Record pool ID in first four bytes of buffer and skip */

*(bpid32 *)bufptr = poolid;
bufptr = (struct bpentry *)(sizeof(bpid32) + (char *)bufptr);
restore(mask);
return (char *)bufptr;
}
Xinu Freebuf (Part 1)

/* freebuf.c - freebuf */

#include <xinu.h>

/*------------------------------------------------------------------------
* freebuf - Free a buffer that was allocated from a pool by getbuf
*------------------------------------------------------------------------
*/
syntax freebuf(
    char *bufaddr /* Address of buffer to return */
)
{
    intmask mask; /* Saved interrupt mask */
    struct bptentry *bpptr; /* Pointer to entry in buftab */
    bpid32 poolid; /* ID of buffer’s pool */

    mask = disable();

    /* Extract pool ID from integer prior to buffer address */

    bufaddr -= sizeof(bpid32);
    poolid = *(bpid32 *)bufaddr;
    if (poolid < 0 || poolid >= nbpools) {
        restore(mask);
        return SYSERR;
    }
Xinu Freebuf (Part 2)

/* Get address of correct pool entry in table */
bpptr = &buftab[poolid];

/* Insert buffer into list and signal semaphore */
((struct bpentry *)bufaddr)->bpnext = bpptr->bpnext;
bpptr->bpnext = (struct bpentry *)bufaddr;
signal(bpptr->bpsem);
restore(mask);
return OK;
Virtual Memory
Definition Of Virtual Memory

- An abstraction of physical memory
- It separates a process’s view of memory from underlying hardware
- Primarily used with applications (user processes)
- Provides each application process with an address space that is independent of
  - Physical memory size
  - A position in physical memory
  - Isolated from other process’s address spaces
- 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 process’s address space (all code, data, and stack) to memory when selecting a process to run
  - Write the entire address space back to disk when switching to another process

- **Segmentation**
  - Divide the image into large “segments” (e.g., make the code and data for each function a segment)
  - Transfer a segment to memory as needed (e.g., when the function is called)

- **Paging**
  - Divide image into small, fixed-size pieces called *pages*
  - Transfer a page to memory when referenced
Approaches That Have Been Used
(continued)

- **Segmentation with paging**
  - Divide an image into very large segments (e.g., a module with multiple functions)
  - Further subdivide each segment into fixed-size pages

- **Notes**
  - The programming language community favor some form of segmentation
  - Hardware engineers favor paging
A Widely-Used Approach

- Paging has emerged as the most widely used approach for virtual memory because
  - Choosing a reasonable page size (e.g., 4K bytes) makes the paging overhead reasonable for most applications
  - Using a page size that is a power of two enables the hardware to be extremely efficient

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)
Virtual And Physical Addresses

- *Address translation* maps a virtual address to a physical address
- To make hardware efficient
  - Choose a page size that is a power of 2
  - Use the upper bits in a virtual address as a page number, \( P \)
  - Use the lower bits in a virtual address as an offset into the page, \( O \)
- To map an address
  - Extract the page number, \( P \)
  - Use \( P \) as an index into the page table array and find the frame where the page currently resides in memory
  - Add the offset, \( O \) to get the physical address of the byte being referenced
Each page table entry contains a physical frame address

Choosing a page size to be a power of 2 means hardware can perform translation without using multiplication, division, or modulus operations
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 (text, data, bss, and heap at the bottom, and stack at the top)
- 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
- Copy a page into memory dynamically when the page is referenced
- Copy a page from the secondary store to a frame in main memory on demand (when the page is referenced)
- When a frame is needed for a newly-referenced pages, move one of the pages currently in memory back to its 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 hardware that supports a memory copy instruction, one instruction can reference *multiple* pages
- The point: hardware support is needed to perform high-speed address translation for each of the above
Hardware Support For Address Mapping

- In addition to normal address translation, a special-purpose hardware unit further speeds page lookup and makes paging practical.

- The special hardware unit
  - Is called a *Translation Look-aside Buffer (TLB)*
  - Is implemented with T-CAM

- A TLB caches most recent address translations and returns translations quickly.

- 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.
Mappings In a TLB And Context Switch

• Facts
  – Each process has an address space that starts at zero
  – Each process has its own page zero
  – The location of page 0 in memory may differ among processes, and page 0 from some processes may not even be in memory

• Consequence: address translation must change when switching context from one process to another

• The point: the mappings cached in a TLB will not remain valid when switching context from one process to another
How An Operating System Manages A TLB

- When it switches context from one process to another, an operating system must ensure that the old mappings in the TLB are not used.

- On some hardware, the operating system flushes the TLB to remove all current entries.

- On other hardware, tags are used to distinguish among address spaces.

- Tags used in a TLB:
  - A unique tag is assigned to each process by the OS (typically, the process ID).
  - The operating system tells the VM hardware which tag to use.
  - When placing a mapping in the TLB, the hardware appends the current tag to the address.
  - When searching the TLB, the hardware appends the current tag to the address.
  - Advantage: the OS only needs to change the tag when switching context.
Can Page Tables Be Paged?

- On some hardware, yes
- Store all page tables in memory
- Lock the current page table to avoid paging it
- The current thinking about paging page tables
  - It introduces extra overhead
  - Lookup becomes less efficient
  - Large memory sizes make 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* exception when a referenced page is not resident
  - The operation system handles the exception
- The operating system
  - Allocates a frame in physical memory
  - Retrieves the needed page from secondary storage (allowing other processes to execute while page is being fetched)
  - Once the page arrives, marks the page table entry to indicate the page is now resident
  - Re却saws 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 space 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 should apply to a shared library?
A Critical Trade-off For Demand Paging

- For a given process, paging represents delay; from a system perspective, paging is merely overhead.

- Paging overhead and latency for a given process can be reduced by giving the process more physical memory (more frames).

- However, processor utilization and overall throughput can be increased by increasing the level of multiprogramming (i.e., by having more concurrent processes ready to run when one of them blocks to wait for I/O or some other reason).

- 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 a frame is currently unused, the selection is trivial — select the 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 to point to the frame.
- Question: which frame should be selected when all are in use?
Choosing A Frame

- Researchers have studied
  - Global competition: when choosing a frame, include resident pages from all processes in the selection
  - Local competition: when choosing a frame for process P, select from among the other pages that process P has resident
- Researchers 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 has actually changed since it was read from secondary storage (i.e., 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 disk I/O, so little computation can be performed
- We use the term *thrashing* to describe fetching a new page often
- 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

• 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 VII

High-Level Synchronous Message Passing
Location Of Synchronous Message Passing In The Hierarchy
A Review Of Xinu’s Low-Level Message Passing Facility

- A message is always sent from one process directly to another
- Each process has a one-message message buffer
- Transmission is asynchronous (non-blocking)
- Reception is synchronous (blocking)
- An asynchronous function can be used to clear the message buffer
Features Of The Xinu High-Level Message Passing Mechanism

- Defines a set of message storage facilities called *ports*
- The user specifies the number of messages a given port can hold
- The mechanism supports many-to-many communication
  - Allows an arbitrary process to send a message to a port
  - Allows an arbitrary process to receive a message from a port
- Uses a synchronous interface
  - Blocks a sender if a port is full
  - Blocks a receiver until a message arrives at a port
- Handles port deletion and reset
An Example Use Of Ports: A Concurrent Server

- Create a port, $P$
- Think of messages that are sent to the port as requests for some service
- Create a set of server processes that each repeatedly receive a request from $P$ and “handle” the request (supply the service)
- An arbitrary process can send a request to $P$; one of the server processes handles the request
- Because server processes run concurrently, a server process can receive a later request and start handling it while another process continues to handle a previous request
- The advantage: short requests can be serviced quickly
A Few Details

- When the port system is initialized, a global pool of messages is created
  - The maximum number of messages in all ports is specified
  - Memory is allocated for the pool, and messages are linked onto a free list
- An individual port can be created (and later deleted) dynamically
- Semaphores are used to
  - Block a sender if a port is full
  - Block a receiver if a port is empty
- When a port is created
  - An argument specifies the number of messages that can be stored in the port
  - The message count is used to initialize a semaphore
Functions That Operate On Ports

- **Ptinit**
  - Must be called once before ports can be used
  - Initializes the entire port system

- **Ptcreate**
  - Creates a new port
  - An argument specifies maximum number of messages

- **Ptsend**
  - Sends a message to a port

- **Ptrecv**
  - Retrieves a message from a port
Functions That Operate On Ports
(continued)

- **Ptreset**
  - Resets existing port
  - Disposes of existing messages
  - Allows waiting processes to continue

- **Ptdelete**
  - Deletes existing port
  - Disposes of existing messages
  - Allows blocked processes to continue
Programmer’s Responsibility

- A programmer must plan ahead
  - Specify the maximum number of messages when calling ptcreate
  - Avoid creating ports that can take more than the total messages
- Worst case: ptsend will panic if no message buffers appear on the free list
- Possible improvement: keep a global count of messages, and decrement it each time ptcreate is called and increment it each time ptdelete is called
Port Declarations

/* ports.h - isbadport */

#define NPORTS 30 /* Maximum number of ports */
#define PT_MSGS 100 /* Total messages in system */
#define PT_FREE 1 /* Port is free */
#define PT_LIMBO 2 /* Port is being deleted/reset */
#define PT_ALLOC 3 /* Port is allocated */

struct ptnode {
    /* Node on list of messages */
    uint32 ptmsg; /* A one-word message */
    struct ptnode *ptnext; /* Pointer to next node on list */
};

struct ptentry {
    /* Entry in the port table */
    sid32 ptssem; /* Sender semaphore */
    sid32 ptrsem; /* Receiver semaphore */
    uint16 ptstate; /* Port state (FREE/LIMBO/ALLOC) */
    uint16 ptxorcnt; /* Max messages to be queued */
    int32 ptseq; /* Sequence changed at creation */
    struct ptnode *pthead; /* List of message pointers */
    struct ptnode *pttail; /* Tail of message list */
};

extern struct ptnode *ptfree; /* List of free nodes */
extern struct ptentry porttab[]; /* Port table */
extern int32 ptnextid; /* Next port ID to try when looking for a free slot */

#define isbadport(portid) ( (portid)<0 || (portid)>=NPORTS )
Xinu Ptinit (Part 1)

/* ptinit.c - ptinit */

#include <xinu.h>

struct ptnode *ptfree; /* List of free message nodes */
struct ptentry porttab[NPORTS]; /* Port table */
int32 ptnextid; /* Next table entry to try */

/*------------------------------------------------------------------------

ptinit - Initialize all ports
*------------------------------------------------------------------------*/
syscall ptinit(
    int32 maxmsgs /* Total messages in all ports */
) {
    int32 i; /* Runs through the port table */
    struct ptnode *next, *curr; /* Used to build a free list */

    /* Allocate memory for all messages on all ports */
    ptfree = (struct ptnode *)getmem(maxmsgs*sizeof(struct ptnode));
    if (ptfree == (struct ptnode *)SYSERR) {
        panic("ptinit - insufficient memory");
    }
/ * Initialize all port table entries to free */

for (i=0 ; i<NPORTS ; i++) {
    porttab[i].ptstate = PT_FREE;
    porttab[i].ptseq = 0;
}

ptnextid = 0;

/* Create a free list of message nodes linked together */

for ( curr=next=ptfree ; --maxmsgs > 0 ; curr=next ) {
    curr->ptnext = ++next;
}

/* Set the pointer in the final node to NULL */

curr->ptnext = NULL;

return OK;
/* ptcreate.c - ptcreate */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ptcreate - Create a port that allows "count" outstanding messages
 *------------------------------------------------------------------------
 */
syscall ptcreate(
    int32 count /* Size of port */
)
{
    intmask mask; /* Saved interrupt mask */
    int32 i; /* Counts all possible ports */
    int32 ptnum; /* Candidate port number to try */
    struct ptentry *ptptr; /* Pointer to port table entry */

    mask = disable();
    if (count < 0) {
        restore(mask);
        return SYSERR;
    }
Xinu Ptcreate (Part 2)

for (i=0 ; i<NPORTS ; i++) { /* Count all table entries */
    ptnum = ptnextid; /* Get an entry to check */
    if (++ptnextid >= NPORTS) {
        ptnextid = 0; /* Reset for next iteration */
    }

    /* Check table entry that corresponds to ID ptnum */

    ptptr= &porttab[ptnum];
    if (ptptr->ptstate == PT_FREE) {
        ptptr->ptstate = PT_ALLOC;
        ptptr->ptssem = semcreate(count);
        ptptr->ptrsem = semcreate(0);
        ptptr->pthead = ptptr->pttail = NULL;
        ptptr->ptseq++; 
        ptptr->ptmaxcnt = count;
        restore(mask);
        return ptnum;
    }

    restore(mask);
    return SYSERR;
}
Xinu Ptsend (Part 1)

/* ptsend.c - ptsend */

#include <xinu.h>

/*------------------------------------------------------------------------
* ptsend - Send a message to a port by adding it to the queue
*------------------------------------------------------------------------
*/
syscall ptsend(
    int32 portid,  /* ID of port to use */
    umsg32 msg    /* Message to send */
)
{
    intmask mask;  /* Saved interrupt mask */
    struct ptentry *ptptr;  /* Pointer to table entry */
    int32 seq;  /* Local copy of sequence num. */
    struct ptnode *msgnode;  /* Allocated message node */
    struct ptnode *tailnode;  /* Last node in port or NULL */

    mask = disable();
    if ( isbadport(portid) ||
        (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
        restore(mask);
        return SYSERR;
    }

    /*...*/
Xinu Ptsend (Part 2)

/* Wait for space and verify port has not been reset */

seq = ptptr->ptseq;           /* Record original sequence */
if (wait(ptptr->ptssem) == SYSERR
    || ptptr->ptstate != PT_ALLOC
    || ptptr->ptseq != seq) {
    restore(mask);
    return SYSERR;
}
if (ptfree == NULL) {
    panic("Port system ran out of message nodes");
}

/* Obtain node from free list by unlinking */

msgnode = ptfree;            /* Point to first free node */
ptfree = msgnode->ptnext;    /* Unlink from the free list */
msgnode->ptnext = NULL;      /* Set fields in the node */
msgnode->ptmsg = msg;
/* Link into queue for the specified port */

tailnode = ptptr->pttail;
if (tailnode == NULL) {
    /* Queue for port was empty */
    ptptr->pttail = ptptr->pthead = msgnode;
} else {
    /* Insert new node at tail */
    tailnode->ptnext = msgnode;
    ptptr->pttail = msgnode;
}
signal(ptptr->ptrsem);
restore(mask);
return OK;
Xinu Ptrecv (Part 1)

/* ptrecv.c - ptrecv */

#include <xinu.h>

/*------------------------------------------------------------------------
* ptrecv - Receive a message from a port, blocking if port empty
*------------------------------------------------------------------------
*/

uint32 ptrecv(
    int32 portid /* ID of port to use */
)
{
    intmask mask; /* Saved interrupt mask */
    struct ptentry *ptptr; /* Pointer to table entry */
    int32 seq; /* Local copy of sequence num. */
    umsg32 msg; /* Message to return */
    struct ptnode *msgnode; /* First node on message list */

    mask = disable();
    if ( isbadport(portid) ||
        (ptptr = &porttab[portid])->ptstate != PT_ALLOC ) {
        restore(mask);
        return (uint32)SYSERR;
    }

    /* More code here to receive the message... */
Xinu Ptrecv (Part 2)

/* Wait for message and verify that the port is still allocated */
seq = ptptr->ptseq;          /* Record original sequence */
if (wait(ptptr->ptrsem) == SYSERR ||
    ptptr->ptstate != PT_ALLOC
    || ptptr->ptseq != seq) {
    restore(mask);
    return (uint32)SYSERR;
}

/* Dequeue first message that is waiting in the port */
msgnode = ptptr->pthead;
msg = msgnode->ptmsg;
if (ptptr->pthead == ptptr->pttail) /* Delete last item */
    ptptr->pthead = ptptr->pttail = NULL;
else
    ptptr->pthead = msgnode->ptnext;
msgnode->ptnext = ptfree;  /* Return to free list */
ptfree = msgnode;
signal(ptptr->ptssem);
restore(mask);
return msg;
Port Deletion And Reset

- Illustrate how difficult it can be to delete resources in a concurrent system
- Situations that must be handled
  - If the port is full, processes may be blocked waiting to send messages to the port
  - If the port is empty, processes may be blocked waiting to receive messages from the port
  - If the port contains messages, some processing may be needed for each message
- An example of message processing during deletion
  - Suppose an application allocates heap memory and uses a message to send a pointer to the block of memory
  - When deleting such a port, the appropriate action may be to free the block of memory associated with each message
Disposing Of Messages

- Message disposition is needed during both reset and deletion
- What action should the system take to dispose of a message?
- Key idea: only the applications using the port will know how to dispose of messages
- To accommodate disposition
  - Both \textit{ptreset} and \textit{ptdelete} include an extra argument that specifies a disposition function
  - When a message is removed from the port, the disposition function is called with the message as an argument
How Dynamic Deletion Complicates A Design

• If concurrent processes can create/use/delete a resource, they can interfere
• Consider what happens with ports if
  – Process A invokes `ptsend` to send a message to a port
  – The port is full, so process A is blocked
  – While process A is blocked, process B starts to delete the port
  – Once the semaphores are deleted, process A will become ready
• If process B has lower priority than process A, process A will run
• How will process A know that the port is being deleted?
• A similar situation occurs for senders
• Another surprise: suppose multiple processes attempt to delete and/or reset the port concurrently
Concurrency And Message Disposition

• The function used to dispose of messages during deletion or reset
  – Is specified by user
  – May reschedule allowing other processes to execute

• An example
  – Suppose each message contains a pointer to a buffer from a buffer pool
  – The user’s disposition function calls \texttt{freebuf} to free the buffer
  – \texttt{Freebuf} signals a semaphore, which calls \texttt{resched}

• Consequence: we need to handle attempts to use the port concurrently during reset or deletion
Three Possible Ways To Handle Reset/Deletion

• Mechanism 1: Accession Numbers
  – A sequence number is associated with each port
  – The sequence number is incremented when the port is created and when the port is deleted or reset
  – Functions \texttt{ptsend} and \texttt{ptrecv} record the sequence number when an operation begins and check the sequence number after \texttt{wait} returns
  – If the sequence number changed, the port was reset, so the operation must abort
Three Possible Ways To Handle Reset/Deletion (continued)

- Mechanism 2: A New State For The Port
  - Each port has a state variable
  - Many OS objects only need a bit to specify whether the object is in use or free
  - Use an additional state to handle deletion/reset
    * PTFREE if the entry for the port is not in use
    * PTALLOC if the port is in use
    * PTLIMBO if the port is being reset/deleted
  - Functions ptsend and ptrecv examine the state variable
  - If the state is PTLIMBO, the port is currently being reset or deleted and cannot be used
Three Possible Ways To Handle Reset/Deletion
(continued)

• Mechanism 3: Deferred Rescheduling
  – Is not included in the current code
  – The idea: temporarily postpone scheduling decisions during reset
  – To apply deferred rescheduling
    * Call `resched_cntl(DEFER_START)` at the start of reset or delete
    * Call `resched_cntl(DEFER_STOP)` after all operations are performed
  – Note that deferred rescheduling means that message disposition will not start other concurrent processes
Common Code For Reset and Deletion

- We will see that port reset and deletion perform many of the same actions
- To eliminate code duplication
  - Place common code in an internal function, \_ptclear
  - Have both ptreset and ptdelete call \_ptclear
- Note: the designation “internal” means that \_ptclear is not a system call — it must be called with interrupts disabled
Xinu Ptdelete

/* ptdelete.c - ptdelete */
#include <xinu.h>

/*------------------------------------------------------------------------
* ptdelete - Delete a port, freeing waiting processes and messages
*------------------------------------------------------------------------*/
systemcall ptdelete(
    int32 portid, /* ID of port to delete */
    int32 (*disp)(int32) /* Function to call to dispose */
) {

    intmask mask; /* Saved interrupt mask */
    struct ptentry *ptptr; /* Pointer to port table entry */

    mask = disable();
    if ( isbadport(portid) ||
        (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
        restore(mask);
        return SYSERR;
    }

    _ptclear(ptptr, PT_FREE, disp);
    ptnextid = portid;
    restore(mask);
    return OK;
}
Xinu Ptreset

/* ptreset.c - ptreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ptreset - Reset a port, freeing waiting processes and messages and
 * leaving the port ready for further use
 *------------------------------------------------------------------------
 */
syscall ptreset(
    int32 portid, /* ID of port to reset */
    int32 (*disp)(int32) /* Function to call to dispose */
)
{
    intmask mask; /* Saved interrupt mask */
    struct ptentry *ptptr; /* Pointer to port table entry */

    mask = disable();
    if (isbadport(portid) ||
        (ptptr = &porttab[portid])->ptstate != PT_ALLOC) {
        restore(mask);
        return SYSERR;
    }
    _ptclear(ptptr, PT_ALLOC, disp);
    restore(mask);
    return OK;
}
void _ptclear(
    struct ptnode *walk; /* Pointer to walk message list */
)

struct ptnode *walk; /* Pointer to walk message list */

/* Place port in limbo state while waiting processes are freed */

ptptr->ptstate = PT_LIMBO;

ptptr->ptseq++; /* Reset accession number */
walk = ptptr->pthead; /* First item on msg list */
Xinu _ptclear (Part 2)

if ( walk != NULL ) { /* If message list nonempty */

/* Walk message list and dispose of each message */

for( ; walk!=NULL ; walk=walk->ptnext) {
   (*dispose)( walk->ptmsg );
}

/* Link entire message list into the free list */

(ptptr->pttail)->ptnext = ptfree;
ptfree = ptptr->pthead;
}

if (newstate == PT_ALLOC) {
   ptptr->pttail = ptptr->pthead = NULL;
   semreset(ptptr->ptssem, ptptr->ptmaxcnt);
   semreset(ptptr->ptrsem, 0);
} else {
   semdelete(ptptr->ptssem);
   semdelete(ptptr->ptrsem);
}
ptptr->ptstate = newstate;
return;
Summary

- Xinu offers a high-level message passing mechanism
- The system uses ports for message storage
- A port can be created dynamically, can have arbitrary senders, and arbitrary receivers
- The interface is completely synchronous — a sender blocks if a port is full, and a receiver blocks if a port is empty
- Port reset/deletion is tricky because
  - Concurrent processes may attempt to use the port while reset or deletion is occurring
  - Senders and receivers must be able to tell that the port changed while they were blocked
Summary (continued)

- Three techniques can handle transition
  - A sequence number informs waiting processes whether the port was reset or deleted while they were blocked
  - A limbo state prevents new processes from using the port while it is being reset or deleted
  - Processes using ports can defer rescheduling during reset and deletion to guarantee that no other processes execute
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 currently executing process executes interrupt code**
Historic Interrupt Software

- A separate interrupt function was created for each device
  - Very low-level code
  - Interrupt code must handle 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
  - Handles low-level details, such as finding the interrupting device on the bus
  - 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 written in assembly language
- 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, which is written in C
Conceptual View Of Interrupt Dispatching

- Note: only the dispatcher is 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 array of *interrupt vectors*
- Each item in the array points to a handler
- 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 type of device
- The controller invokes the correct dispatcher
Interrupts On A BeagleBone Black (ARM)

- Uses a two-level scheme where the 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 hardware 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 from the interrupt

- **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 (an input interrupt occurs)
  - The device has finished sending outgoing data (an output interrupt occurs)
- 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 handler 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 handler 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

- As an example, suppose
  - An interrupt handler calls \textit{signal}
  - \textit{Signal} calls \textit{resched}
  - \textit{Resched} switches to a new process
  - The new process executes with interrupts enabled
- Will interrupts pile up indefinitely?
The Answer

• No, interrupts will not pile up indefinitely
• Reason:
  – Interrupt status is associated with each process, not with the hardware
  – After switching to a process that has interrupts enabled, that process can be interrupted
  – In the worst case, all processes can end up executing interrupt code with further interrupts disabled
  – If another context switch occurs, it will be to a process that has interrupts disabled, and the system must return from the interrupt to have interrupts enabled again
The Answer
(continued)

• As an example, let $T$ be the current process
• When an interrupt occurs, $T$ executes an interrupt handler with interrupts disabled
• If the handler that $T$ is executing calls $signal$
  – $Signal$ may call $resched$
  – A context switch may occur and process $S$ may run
  – $S$ may run with interrupts enabled
  – If a second interrupt occurs, $S$ may execute an interrupt handler with interrupts disabled
• Only $NPROC$ interrupts can occur before all processes are running with interrupts disabled
The Principle

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., disable/restore is used instead of 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

I/O operations available to applications

device drivers

device 1
device 2
device n

device driver upper-half (device 1)
device driver upper-half (device 2)
device driver upper-half (device n)
device driver lower-half (device 1)
device driver lower-half (device 2)
device driver lower-half (device n)

application processes
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 \textit{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

- **SLEEPING**
  - **wakeup**
  - **sleep**

- **RECEIVING**
  - **send**
  - **receive**

- **WAITING**
  - **signal**
  - **wait**

- **READY**
  - **resched**
  - **suspend**
  - **resume**

- **CURRENT**
  - **resched**

- **SUSPENDED**
  - **suspend**
  - **create**
/* sleep.c - sleep sleepms */

#include <xinu.h>

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

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

/* wakeup.c - wakeup */

#include <xinu.h>

/*------------------------------------------------------------------------
* wakeup - Called by clock interrupt handler to awaken processes
*------------------------------------------------------------------------
*/
void wakeup(void)
{
    /* Awaken all processes that have no more time to sleep */

    resched_cntl(DEFER_START);
    while (nonempty(sleepq) && (firstkey(sleepq) <= 0)) {
        ready(dequeue(sleepq));
    }

    resched_cntl(DEFER_STOP);
    return;
}

• Note that rescheduling is deferred until all processes are awakened
Timed Message Reception

- Many operating system components offer a “timeout” on operations
- Timeout is especially useful in building communication protocols
- A Xinu example: receive with timeout
  - Operates like `receive`, but includes a timeout argument
  - If a message arrives before the timer expires, the message is returned
  - If the timer expires before a message arrives, the value `TIMEOUT` is returned
  - Implemented with `recvtime`
- `recvtime` uses the same queue and wakeup mechanism as sleeping processes
/* recvtime.c - recvtime */

#include <xinu.h>

/*------------------------------------------------------------------------
  * recvtime - Wait specified time to receive a message and return
  *------------------------------------------------------------------------
*/
umsg32 recvtime(
    int32 maxwait /* Ticks to wait before timeout */
)
{
    intmask mask; /* Saved interrupt mask */
    struct procent *prptr; /* Tbl entry of current process */
    umsg32 msg; /* Message to return */

    if (maxwait < 0) {
        return SYSERR;
    }

    mask = disable();
/* Schedule wakeup and place process in timed-receive state */

prptr = &proctab[currpid];
if (prptr->prhasmsg == FALSE) {
    /* Delay if no message waiting */
    if (insertd(currpid,sleepq,maxwait) == SYSERR) {
        restore(mask);
        return SYSERR;
    }
    prptr->prstate = PR_RECTIM;
    resched();
}

/* Either message arrived or timer expired */

if (prptr->prhasmsg) {
    msg = prptr->prmsg; /* Retrieve message */
    prptr->prhasmsg = FALSE; /* Reset message indicator */
} else {
    msg = TIMEOUT;
}
restore(mask);
return msg;
Look Again At Send.c (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;
    }
}
Look Again At Send.c (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;
/* unsleep.c - unsleep */

#include <xinu.h>

/*------------------------------------------------------------------------
 * unsleep - Internal function to remove a process from the sleep queue prematurely. The caller must adjust the delay of successive processes.
 *------------------------------------------------------------------------*/

status unsleep(pid32 pid /* ID of process to remove */ )
{
    intmask mask; /* Saved interrupt mask */
    struct procent *prptr; /* Ptr to process’s table entry */
    pid32 pidnext; /* ID of process on sleep queue that follows the process which is being removed */

    mask = disable();
Unsleep - Remove A Sleeping Process (Part 2)

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

/* Verify that candidate process is on the sleep queue */

prptr = &proctab[pid];
if (((prptr->prstate!=PR_SLEEP) && (prptr->prstate!=PR_RECTIM)) {
    restore(mask);
    return SYSERR;
}

/* Increment delay of next process if such a process exists */

pidnext = queuetab[pid].qnext;
if (pidnext < NPROC) {
    queuetab[pidnext].qkey += queuetab[pid].qkey;
}

getitem(pid); /* Unlink process from queue */
restore(mask);
return OK;
The Clock Hardware Interface

- The clock interface follows the pattern used by all devices
- The system uses a memory-mapped interaction
  - Some high bus addresses correspond to the clock device, not memory
  - When the processor stores data to one of the special addresses, the data being stored goes to the clock device
  - When the processor fetches from the special addresses, the clock device answers the request and sends information to the processor
  - Typically, the processor sends commands to a device
- A device driver defines a structure that specifies the layout of special addresses and their meaning as well as constants used (usually called control and status registers)
/* clock.h */

extern uint32 clktime; /* current time in secs since boot */
extern qid16 sleepq; /* queue for sleeping processes */
extern int32 slnonempty; /* nonzero if sleepq is nonempty */
extern int32 *sltop; /* ptr to key in first item on sleepq */
extern uint32 preempt; /* preemption counter */

struct am335x_timer1ms {
    uint32 tidr; /* Identification register */
    uint32 resl[3]; /* Reserved */
    uint32 tiocp_cfg; /* OCP Interface register */
    uint32 tisstat; /* Status register */
    uint32 tisr; /* Interrupt status register */
    uint32 tier; /* Interrupt enable register */
    uint32 twer; /* Wakeup enable register */
    uint32 tclr; /* Optional features */
    uint32 tcrr; /* Internal counter value */
    uint32 tldr; /* Timer load value */
    uint32 ttgr; /* Trigger register */
    uint32 twps; /* Write posting register */
    uint32 tmar; /* Match register */
}
ARM Clock Definitions (Part 2)

uint32 tcar1; /* Capture register 1 */
uint32 tsicr; /* Synchronous interface control */
uint32 tcar2; /* Capture register 2 */
uint32 tpir; /* Positive increment register */
uint32 tnir; /* Negative increment register */
uint32 tcvr; /* 1ms control register */
uint32 tocr; /* Overflow mask register */
uint32 towr; /* no. of overflows */

};

#define AM335X_TIMER1MS_ADDR 0x44E31000
#define AM335X_TIMER1MS_IRQ 67
#define AM335X_TIMER1MS_TIOCP_CFG_SOFTRESET 0x00000002
#define AM335X_TIMER1MS_TISTAT_RESETDONE 0x00000001
#define AM335X_TIMER1MS_TISR_MAT_IT_FLAG 0x00000001
#define AM335X_TIMER1MS_TISR_OVF_IT_FLAG 0x00000002
#define AM335X_TIMER1MS_TISR_TCAR_IT_FLAG 0x00000004
#define AM335X_TIMER1MS_TIER_MAT_IT_ENA 0x00000001
#define AM335X_TIMER1MS_TIER_OVF_IT_ENA 0x00000002
#define AM335X_TIMER1MS_TIER_TCAR_IT_ENA 0x00000004
ARM Clock Definitions (Part 3)

#define AM335X_TIMER1MS_TCLR_ST 0x00000001
#define AM335X_TIMER1MS_TCLR_AR 0x00000002
#define AM335X_TIMER1MS_CLKCTRL_ADDR 0x44E004C4
#define AM335X_TIMER1MS_CLKCTRL_EN 0x00000002
Clock Interrupt Handler Code (Part 1)

/* clkhandler.c - clkhandler */

#include <xinu.h>

/*--------------------------------------------
* clkhandler - high level clock interrupt handler
*--------------------------------------------
*/

void clkhandler()
{
    static uint32 count1000 = 1000; /* variable to count 1000ms */
    volatile struct am335x_timer1ms *csrptr =
        (struct am335x_timer1ms *)0x44E31000;
    /* Set csrptr to address of timer CSR */

    /* If there is no interrupt, return */
    if((csrptr->tisr & AM335X_TIMER1MS_TISR_OVF_IT_FLAG) == 0) { 
        return;
    }
/* Acknowledge the interrupt */
csrptr->tisr = AM335X_TIMER1MS_TISR_OVF_IT_FLAG;

/* Decrement 1000ms counter */
count1000--;

/* After 1 sec, increment clktime */
if(count1000 == 0) {
    clktime++;
    count1000 = 1000;
}
/* check if sleep queue is empty */
if(!isempty(sleepq)) {
    /* sleepq nonempty, decrement the key of */
    /* topmost process on sleepq */
    if((--queuetab[firstid(sleepq)].qkey) == 0) {
        wakeup();
    }
}

/* Decrement the preemption counter */
/* Reschedule if necessary */

if((--preempt) == 0) {
    preempt = QUANTUM;
    resched();
}
Clock Initialization (Part 1)

```c
#include <xinu.h>

uint32 clktime; /* Seconds since boot */
uint32 ctr1000 = 0; /* Milliseconds since boot */
qid16 sleepq; /* Queue of sleeping processes */
uint32 preempt; /* Preemption counter */

void clkinit(void) {
    volatile struct am335x_timer1ms *csrptr =
        (volatile struct am335x_timer1ms *)AM335X_TIMER1MS_ADDR;
    volatile uint32 *clkctrl =
        (volatile uint32 *)AM335X_TIMER1MS_CLKCTRL_ADDR;

    *clkctrl = AM335X_TIMER1MS_CLKCTRL_EN;
    while((*clkctrl) != 0x2) /* Do nothing */ ;
}```
/* Reset the timer module */
csrptr->tiocp_cfg |= AM335X_TIMER1MS_TIOCP_CFG_SOFTRESET;

/* Wait until the reset is complete */
while((csrptr->tistat & AM335X_TIMER1MS_TISTAT_RESETDONE) == 0)
    /* Do nothing */ ;

/* Set interrupt vector for clock to invoke clkint */
set_evec(AM335X_TIMER1MS_IRQ, (uint32)clkhandler);

sleepq = newqueue(); /* Allocate a queue to hold the delta */
    /* list of sleeping processes */
preempt = QUANTUM; /* Set the preemption time */
clktime = 0; /* Start counting seconds */

/* The following values are calculated for a */
/* timer that generates 1ms tick rate */

csrptr->tpir = 1000000;
csrptr->tnir = 0;
csrptr->tldr = 0xFFFFFFFF - 26000;
/* Set the timer to auto reload */
csrptr->tclr = AM335X_TIMER1MS_TCLR_AR;

/* Start the timer */
csrptr->tclr |= AM335X_TIMER1MS_TCLR_ST;

/* Enable overflow interrupt which will generate */
/* an interrupt every 1 ms */
csrptr->tier = AM335X_TIMER1MS_TIER_OVF_IT_ENA;

/* Kickstart the timer */
csrptr->ttgr = 1;
return;
Notes About Device Hardware Interfaces

- Hardware is incredibly low level
- The interface to a hardware device can is tedious
- Hardware defines
  - Many registers that each have some special meaning
  - Special constants that must be used
- A programmer must deal with
  - Silly details
  - A lack of concepts and principles
  - Multiple commands to perform a simple task
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 specify a maximum time to wait for a message to arrive
Module IX

Device Management
Device-Independent I/O
And An Example Device Driver
Location Of Device Management In The Hierarchy
Goals For A Devices Interface

- Isolation from hardware: ensure that applications do not contain details related to device hardware
- Portability: allow applications to run on any brand or model of equivalent device unchanged
- Elegance: limit the interface to a minimal number of orthogonal functions
- Generality: use a common paradigm across all devices
- Integration: integrate the device manager with the process manager and other operating system facilities
Achieving The Goals

- Devise a small set of functions that applications use to
  - Obtain incoming data from a device
  - Transfer outgoing data to a device
  - Control the device
- Examples of controlling a device
  - Adjust the volume on headphones
  - Turn off character echo when reading a password
  - Eject a USB drive
- The approach is known as a *device-independent* interface
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 single character (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 or CONSOLE)

• 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

• Examples
  – When a process reads from the ETHER device, the device manager invokes `ethread`
  – When a process reads from the CONSOLE, the device manager invokes `ttyread`
Mapping A Generic I/O Function To A Device-Specific Function

- The mapping must be extremely efficient
- Solution: use a two-dimensional array known as a *device switch table*
- The device switch table
  - Is a kernel data structure that is initialized at compile time
  - Has one row for each device
  - Has one column for each possible I/O operation
- An entry in the table points to a function to be called to perform the operation on the device
- A device ID is chosen to be a index into rows of the table
Entries In The Device Switch Table

- A given device-independent operation may not make sense for some devices
  - *Seek* on a keyboard, network interface, or audio output device
  - *Close* on a mouse

- To avoid special cases in the code
  - Make each entry in the device switch table point to a valid function
  - Use 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 A Device Switch Table

<table>
<thead>
<tr>
<th>device</th>
<th>operation</th>
<th>operation</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSOLE</td>
<td>&amp;ttyopen</td>
<td>&amp;ttyread</td>
<td>&amp;ttywrite</td>
</tr>
<tr>
<td>SERIAL0</td>
<td>&amp;ionull</td>
<td>&amp;comread</td>
<td>&amp;comwrite</td>
</tr>
<tr>
<td>SERIAL1</td>
<td>&amp;ionull</td>
<td>&amp;comread</td>
<td>&amp;comwrite</td>
</tr>
<tr>
<td>ETHER</td>
<td>&amp;ethopen</td>
<td>&amp;ethread</td>
<td>&amp;ethwrite</td>
</tr>
</tbody>
</table>

- 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 disks
  – Two monitors

• Goal have one copy of device driver code for each type of device and use the same code with multiple devices
Parameterized Device Drivers

- A device driver must
  - Know which physical copy of a device type to use (e.g., which disk)
  - Keep information about each physical copy of a device separate from information for other physical devices (e.g., maintain separate information for each disk)

- To accommodate multiple copies of a device
  - Assign each instance a replicated device a unique number (0, 1, 2, ...) known as its *minor device number*
  - Store the minor device number in the device switch table
  - Example 1: for two disks of the same type, assign minor numbers 0 and 1
  - Example 2: for three NICs of the same type assign minor numbers 0, 1, and 2

- The point: minor numbers only distinguish among devices of the same type
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: it’s automatic — 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
Device Configuration

- We will see more details later; for now, it is sufficient to know that
  - The OS designer creates a file named *Configuration* that
    * Lists devices in the system and gives each a name (e.g., CONSOLE)
    * Specifies a *type* for each device
    * Specifies the driver functions to use for each operation on the device (open, close, read, write, putc, getc, etc)
  - The config program generates
    * A file named *conf.h* that contains declarations for data structures used in the device switch table
    * A file names *conf.c* that contains a definition of the device switch table with initial values specified, including minor numbers
Initializing The I/O Subsystem

- At system startup
  - Entries in the device switch table are already initialized
  - The interrupt vectors (and perhaps the bus) must be initialized
  - The *init* function is called for each device, which initializes both the device hardware and the driver (e.g., creates the semaphores the driver uses for coordination)

- In lab, you will create a driver and understand how an array can hold information for a set of replicated devices and how the minor number of the device corresponds to an index into the array
An Example Device Driver
Our Example

- Consider a console device that
  - Displays output in a text window on a user’s screen
  - Accepts input from the user’s keyboard
- The device is character-oriented
- Input consists of characters that come from the keyboard
- Output consists of characters sent to the screen
- Following the Unix convention, we used the term/tty to describe the type of device
Hardware For The Example Device

- The underlying hardware consists of a *Universal Asynchronous Receiver and Transmitter (UART)*

- A UART transfers a single character (bytes) at a time, but the hardware has on-board input and output buffers that hardware engineers call *FIFOs*

- When an input interrupt occurs
  - One or more characters are available in the input FIFO
  - The interrupt handler must extract all the characters

- Our driver also maintains its own buffers for input and output

- Our driver uses semaphores to synchronize upper and lower halves
General Idea

- When an application writes characters to a console device, the device driver
  - Places outgoing characters in a buffer
  - Starts the device
- Note: a process can generate characters faster than the hardware can send them, and having a buffer in the device driver allows a process to write one or more lines of text before being blocked to wait for the device
- The console device interrupts when it has finished sending a character
- During the Interrupt
  - The lower-half of the device driver repeatedly removes an outgoing character from the buffer and sends it to the device FIFO until the FIFO fills
  - In essence, the device driver keeps the device busy as long as output exists
### Tty Device Driver Functions

<table>
<thead>
<tr>
<th>Upper-Half</th>
<th>Lower-Half</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttyinit</td>
<td>ttyhandler (interrupt handler)</td>
</tr>
<tr>
<td>ttyopen</td>
<td>ttyhandle_in (input interrupt)</td>
</tr>
<tr>
<td>ttyclose</td>
<td>ttyhandle_out (output interrupt)</td>
</tr>
<tr>
<td>ttyread</td>
<td></td>
</tr>
<tr>
<td>ttywrite</td>
<td></td>
</tr>
<tr>
<td>ttyputc</td>
<td></td>
</tr>
<tr>
<td>ttygetc</td>
<td></td>
</tr>
<tr>
<td>ttycontrol</td>
<td></td>
</tr>
</tbody>
</table>
An output semaphore counts spaces in the device driver buffer

When the upper-half is given a character to send
  - It waits on the output semaphore to guarantee buffer space is available
  - It deposits the character in next buffer slot
  - It “kicks” the device, which causes the device to interrupt

The lower-half
  - Is invoked when the device interrupts
  - Extracts a character from next filled slot in the buffer, and stores the character in the device output FIFO
  - Signals the semaphore to indicate that the buffer now has one more empty slot
Tty Driver Complexity

- The hardware is fairly “dumb”
- The device driver provides *modes* similar to the modes Unix offers
- Raw mode sends and receives individual bytes with no processing at all
- Cooked mode echos input characters, allows a user to backspace or erase an entire line, handles flow control, and delivers an entire line of input at a time
- Cbreak mode handles some of the cooked mode facilities
In addition to a mode that sets many parameters, the driver allows many parameters to be controlled individually at any time:

- Whether CRLF mapping is in effect
- Whether input character echo is turned on
- Whether flow control (^S/^Q) is enabled
- Whether control characters are visualized (e.g., ^A for control-A)
- Whether backspacing over a character “erases” the character from the display
**Summary Of Tty Modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw</td>
<td>The driver delivers each incoming character as it arrives without echoing the character, buffering a line of text, performing translation, or controlling the output flow</td>
</tr>
<tr>
<td>cooked</td>
<td>The driver buffers input, echoes characters in a readable form, honors backspace and line kill, allows type-ahead, handles flow control, and delivers an entire line of text</td>
</tr>
<tr>
<td>cbreak</td>
<td>The driver handles character translation, echoing, and flow control, but instead of buffering an entire line of text, the driver delivers each incoming character as it arrives</td>
</tr>
</tbody>
</table>

- The mode determines how input characters are processed
- An application can change the mode at any time
Circular Input And Output Buffers

- The tty driver maintains three buffers
  - One for incoming characters
  - One for outgoing characters
  - One for echoed characters
- Conceptually, each buffer is circular
- The implementation uses an array where the head and tail are pointers to positions in the array, which wrap around when they go beyond the end of the array
The figure shows
(a) A circular buffer
(b) An implementation with an array using head and tail integers to indicate positions
Definitions Used By The Tty Driver (Part 1)

/* tty.h */

#define TY_OBMINSP 20 /* Min space in buffer before */
/* processes awakened to write*/
#define TY_EBUFLEN 20 /* Size of echo queue */

/* Size constants */

#ifndef Ntty
#define Ntty 1 /* Number of serial tty lines */
#endif

#ifndef TY_IBUFLEN
#define TY_IBUFLEN 128 /* Num. chars in input queue */
#endif

#ifndef TY_OBUFLEN
#define TY_OBUFLEN 64 /* Num. chars in output queue */
#endif

/* Mode constants for input and output modes */

#define TY_IMRAW 'R' /* Raw input mode => no edits */
#define TY_IMCOOKED 'C' /* Cooked mode => line editing */
#define TY_IMCBREAK 'K' /* Honor echo, etc, no line edit*/
#define TY_OMRAW 'R' /* Raw output mode => no edits */
Definitions Used By The Tty Driver (Part 2)

```c
struct ttycblk {
    char *tyihead; /* Tty line control block */
    char *tyitaill; /* Next input char to read */
    char *tyibuff[TY_IBUFLEN]; /* Next slot for arriving char */
    char tyisem; /* Input buffer (holds one line) */
    char *tyohead; /* Input semaphore */
    char *tyotail; /* Next output char to xmit */
    char *tyobuff[TY_OBUFLEN]; /* Next slot for outgoing char */
    char tyosem; /* Output buffer */
    char *tyehead; /* Output semaphore */
    char *tyetail; /* Next echo char to xmit */
    char *tyebuff[TY_EBUFLEN]; /* Next slot to deposit echo ch */
    char tyimode; /* Echo buffer */
    bool8 tyiecho; /* Input mode raw/cbreak/cooked */
    bool8 tyieback; /* Is input echoed? */
    bool8 tyiverse; /* Do erasing backspace on echo? */
    bool8 tyecrlf; /* Echo control chars as ^X ? */
    bool8 tycrlf; /* Echo CR-LF for newline? */
    bool8 tyirasec; /* Map \r' to \n' on input? */
    bool8 tyirasec2; /* Honor \r' to \n' on input? */
    char *tyierasec; /* Alternate erase character */
    bool8 tyeosf; /* Honor EOF character? */
    bool8 tyoeofch; /* Honor line kill character? */
    char *tyoeofchc; /* EOF character (usually ^D) */
    char *tyikill; /* Line kill character */
    bool8 tyikillc; /* Line kill character */
}
```
Definitions Used By The Tty Driver (Part 3)

```c
int32  tyicursor; /* Current cursor position */
bool8  tyoflow;  /* Honor ostop/ostart? */
bool8  tyoheld;  /* Output currently being held? */
char   tyostop;  /* Character that stops output */
char   tyostart; /* Character that starts output */
bool8  tyocrlf;  /* Output CR/LF for LF? */
char   tyifullc; /* Char to send when input full */
};
extern struct ttycblk ttytab[];
/* Characters with meaning to the tty driver */

#define TY_BACKSP   '\b'  /* Backspace character */
#define TY_BACKSP2  '\177' /* Alternate backspace char. */
#define TY_BELL     '\07'  /* Character for audible beep */
#define TY_EOFCH    '\04'  /* Control-D is EOF on input */
#define TY_BLANK    '\0'   /* Blank */
#define TY_NEWLINE  '\n'   /* Newline == line feed */
#defineTY_RETURN   '\r'   /* Carriage return character */
#define TY_STOPCH   '\023' /* Control-S stops output */
#define TY_STRTCH   '\021' /* Control-Q restarts output */
#define TY_KILLCH   '\025' /* Control-U is line kill */
#define TY_UPARROW  '\^'   /* Used for control chars (^X) */
#define TY_FULLCH   TY_BELL /* Char to echo when buffer full */
```
Definitions Used By The Tty Driver (Part 4)

/* Tty control function codes */

#define TC_NEXTC 3   /* Look ahead 1 character */
#define TC_MODER 4   /* Set input mode to raw */
#define TC_MODEC 5   /* Set input mode to cooked */
#define TC_MODEK 6   /* Set input mode to cbreak */
#define TC_ICHARS 8  /* Return number of input chars */
#define TC_ECHO 9    /* Turn on echo */
#define TC_NOECHO 10 /* Turn off echo */
Driver Definitions

- Note the complexity of the definitions
- Conclusion: although a tty device seems straightforward, the parameters used to control character processing complicate the driver
- Now consider driver functions to transfer data, perform control functions, and handle interrupts
Xinu Ttputc (Part 1))

/* ttputc.c - ttyputc */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttyputc - Write one character to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
*/
devcall ttputc(
    struct dentry *devptr,         /* Entry in device switch table */
    char ch                         /* Character to write */
){
    struct ttycblk *typtr;         /* Pointer to tty control block */
    typtr = &ttytab[devptr->dvminor];

    /* Handle output CRLF by sending CR first */
    if ( ch==TY_NEWLINE && typtr->tyocrlf ) {
        ttyputc(devptr, TY_RETURN);
    }
}
wait(typtr->tyosem);    /* Wait for space in queue */
*typtr->tyotail++ = ch;

/* Wrap around to beginning of buffer, if needed */
if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
    typtr->tyotail = typtr->tyobuff;
}

/* Start output in case device is idle */
ttykickout((struct uart_csreg *)devptr->dvcsr);

return OK;
/* ttygetc.c - ttygetc */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttygetc - Read one character from a tty device (interrupts disabled)
*------------------------------------------------------------------------*/
devcall ttygetc(
    struct dentry *devptr /* Entry in device switch table */
) {
    char ch; /* Character to return */
    struct ttycblk *typtr; /* Pointer to ttytab entry */

    typtr = &ttytab[devptr->dvminor];

    /* Wait for a character in the buffer and extract one character */
    wait(typtr->tyisem);
    ch = *typtr->tyihead++;

    /* Wrap around to beginning of buffer, if needed */
    if (typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
        typtr->tyihead = typtr->tyibuff;
    }
}
/* In cooked mode, check for the EOF character */

if ( (typtr->tyimode == TY_IMCOOKED) && (typtr->tyeof) &&
    (ch == typtr->tyeofch) ) {
    return (devcall)EOF;
}

return (devcall)ch;
Xinu Ttywrite

/* ttywrite.c - ttywrite */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttywrite - Write character(s) to a tty device (interrupts disabled)
*------------------------------------------------------------------------
*/
devcall ttywrite(
    struct dentry *devptr, /* Entry in device switch table */
    char *buff, /* Buffer of characters */
    int32 count /* Count of character to write */
) {
    /* Handle negative and zero counts */
    if (count < 0) {
        return SYSERR;
    } else if (count == 0) {
        return OK;
    }

    /* Write count characters one at a time */
    for (; count > 0; count--) {
        ttyputc(devptr, *buff++);
    }
    return OK;
}

Copyright © 2022 by Douglas Comer and CRC Press, Inc. All rights reserved.
/* ttyread.c - ttyread */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttyread - Read character(s) from a tty device (interrupts disabled)
*------------------------------------------------------------------------*/
devcall ttyread(
    struct dentry *devptr, /* Entry in device switch table */
    char *buff, /* Buffer of characters */
    int32 count /* Count of character to read */
) {
    struct ttycblk *typtr; /* Pointer to tty control block */
    int32 avail; /* Characters available in buff.*/
    int32 nread; /* Number of characters read */
    int32 firstch; /* First input character on line*/
    char ch; /* Next input character */

    if (count < 0) {
        return SYSERR;
    }
Xinu Ttyread (Part 2)

typtr = &ttytab[devptr->dvminor];
if (typtr->tyimode != TY_IMCOOKED) {
    /* For count of zero, return all available characters */
    if (count == 0) {
        avail = semcount(typtr->tyisem);
        if (avail == 0) {
            return 0;
        } else {
            count = avail;
        }
    }
    for (nread = 0; nread < count; nread++) {
        *buff++ = (char) ttygetc(devptr);
    }
    return nread;
}
/* Block until input arrives */
firstch = ttygetc(devptr);
Xinu Ttyread (Part 3)

/* Check for End-Of-File */

if (firstch == EOF) {
    return EOF;
}

/* Read up to a line */

ch = (char) firstch;
*buff++ = ch;
nread = 1;
while ( (nread < count) && (ch != TY_NEWLINE) && (ch != TY_RETURN) ) {
    ch = ttygetc(devptr);
    *buff++ = ch;
    nread++;
}

return nread;
Xinu Ttycontrol (Part 1)

/* ttycontrol.c - ttycontrol */
#include <xinu.h>

/*------------------------------------------------------------------------
* ttycontrol - Control a tty device by setting modes
*------------------------------------------------------------------------*/
devcall ttycontrol(
    struct dentry *devptr, /* Entry in device switch table */
    int32 func, /* Function to perform */
    int32 arg1, /* Argument 1 for request */
    int32 arg2 /* Argument 2 for request */
)
{
    struct ttycblk *typtr; /* Pointer to tty control block */
    char ch; /* Character for lookahead */

    typtr = &ttytab[devptr->dvminor];
/ * Process the request */

switch (func) {

case TC_NEXTC:
    wait(typtr->tyisem);
    ch = *typtr->tyitail;
    signal(typtr->tyisem);
    return (devcall)ch;

case TC_MODER:
    typtr->tyimode = TY_IMRAW;
    return (devcall)OK;

case TC_MODEC:
    typtr->tyimode = TY_IMCOOKED;
    return (devcall)OK;

case TC_MODEK:
    typtr->tyimode = TY_IMCBREAK;
    return (devcall)OK;

case TC_ICHARS:
    return(semcount(typtr->tyisem));
Xinu Ttycontrol (Part 3)

case TC_ECHO:
    typtr->tyiecho = TRUE;
    return (devcall)OK;

case TC_NOECHO:
    typtr->tyiecho = FALSE;
    return (devcall)OK;

default:
    return (devcall)SYSERR;
}
Xinu Tty Handler (Part 1)

/* ttyhandler.c - ttyhandler */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttyhandler - Handle an interrupt for a tty (serial) device
 *------------------------------------------------------------------------
 */

void ttyhandler(void) {
    struct dentry *devptr; /* Address of device control blk*/
    struct ttycblk *typtr; /* Pointer to ttytab entry */
    struct uart_csreg *csrptr; /* Address of UART’s CSR */
    byte iir = 0; /* Interrupt identification */
    byte lsr = 0; /* Line status */

    /* Get CSR address of the device (assume console for now) */
    devptr = (struct dentry *) &devtab[CONSOLE];
    csrptr = (struct uart_csreg *) devptr->dvcsr;

    /* Obtain a pointer to the tty control block */
    typtr = &ttytab[ devptr->dvminor ];
}
/* Decode hardware interrupt request from UART device */
/* Check interrupt identification register */
iir = csrptr->iir;
if (iir & UART_IIR_IRQ) {
    return;
}

/* Decode the interrupt cause based upon the value extracted from the UART interrupt identification register. Clear the interrupt source and perform the appropriate handling to coordinate with the upper half of the driver */

/* Decode the interrupt cause */
iir &= UART_IIR_IDMASK; /* Mask off the interrupt ID */
switch (iir) {
    /* Receiver line status interrupt (error) */
    case UART_IIR_RLSI:
        return;
/* Receiver data available or timed out */

case UART_IIR_RDA:
case UART_IIR_RTO:
    resched_cntl(DEFER_START);

    /* While chars avail. in UART buffer, call ttyinter_in */
    while ( (csrptr->lsr & UART_LSR_DR) != 0) {
        ttyhandle_in(ty.ptr, csrptr);
    }

    resched_cntl(DEFER_STOP);

    return;
Xinu Tty Handler (Part 4)

/* Transmitter output FIFO is empty (i.e., ready for more) */

case UART_IIR_THRE:
    ttyhandle_out(typtr, csrptr);
    return;

/* Modem status change (simply ignore) */

case UART_IIR_MSC:
    return;
}
Input Interrupt Handling

- Recall that when an input interrupt occurs
  - One or more characters have arrived at the device
  - The driver must drain all characters from the device
- If multiple processes are waiting for input, the driver cannot let any of them proceed until all characters have been extracted from the device
- Technique used: defer rescheduling while extracting characters
/* ttyhandle_in.c - ttyhandle_in, erase1, eputc, echoch */

#include <xinu.h>

local void erase1(struct ttycblk *, struct uart_csreg *);
local void echoch(char, struct ttycblk *, struct uart_csreg *);
local void eputc(char, struct ttycblk *, struct uart_csreg *);

/*------------------------------------------------------------------------
* ttyhandle_in - Handle one arriving char (interrupts disabled)
 *------------------------------------------------------------------------*/

void ttyhandle_in (
        struct ttycblk *typtr, /* Pointer to ttytab entry */
        struct uart_csreg *csrptr /* Address of UART’s CSR */
    )
{
    char ch; /* Next char from device */
    int32 avail; /* Chars available in buffer */

    ch = csrptr->buffer;

    /* Compute chars available */

    avail = semcount(typtr->tyisem);
    if (avail < 0) {
        /* One or more processes waiting*/
        avail = 0;
    }
/* Handle raw mode */

if (typtr->tyimode == TY_IMRAW) {
    if (avail >= TY_IBUFLEN) { /* No space => ignore input */
        return;
    }

    /* Place char in buffer with no editing */
    *typtr->tyitail++ = ch;

    /* Wrap buffer pointer */
    if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
        typtr->tyitail = typtr->tyibuff;
    }

    /* Signal input semaphore and return */
    signal(typtr->tyisem);
    return;
}

/* Handle cooked and cbreak modes (common part) */

if ( (ch == TY_RETURN) && typtr->tyicrlf ) {
    ch = TY_NEWLINE;
}
/ * If flow control is in effect, handle ^S and ^Q */

if (typtr->tyoflow) {
    if (ch == typtr->tyostart) { /* ^Q starts output */
        typtr->tyoheld = FALSE;
        ttykickout(csrptr);
        return;
    } else if (ch == typtr->tyostop) { /* ^S stops output */
        typtr->tyoheld = TRUE;
        return;
    }
}

typtr->tyoheld = FALSE; /* Any other char starts output */
Xinu Ttyhandle_in (Part 4)

if (typtr->tyimode == TY_IMCBREAK) { /* Just cbreak mode */
    /* If input buffer is full, send bell to user */
    if (avail >= TY_IBUFLEN) {
        eputc(typtr->tyifullc, typtr, csrptr);
    } else { /* Input buffer has space for this char */
        *typtr->tyitail++ = ch;
        /* Wrap around buffer */
        if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
            typtr->tyitail = typtr->tyibuff;
        }
        if (typtr->tyiecho) { /* Are we echoing chars?*/
            echoch(ch, typtr, csrptr);
        }
    }
    return;
}
Xinu Ttyhandle_in (Part 5)

} else { /* Just cooked mode (see common code above) */

/* Line kill character arrives - kill entire line */

if (ch == typtr->tyikillc && typtr->tyikill) {
    typtr->tyitail -= typtr->tyicursor;
    if (typtr->tyitail < typtr->tyibuff) {
        typtr->tyitail += TY_IBUFLEN;
    }
    typtr->tyicursor = 0;
    eputc(TY_RETURN, typtr, csrptr);
    eputc(TY_NEWLINE, typtr, csrptr);
    return;
}

/* Erase (backspace) character */

if ((ch==typtr->tyierasec || (ch==typtr->tyierasec2))
    && typtr->tyierase) {
    if (typtr->tyicursor > 0) {
        typtr->tyicursor--;
        erase1(typtr, csrptr);
    }
    return;
}

Copyright © 2022 by Douglas Comer and CRC Press, Inc. All rights reserved.
Xinu Ttyhandle_in (Part 6)

/* End of line */
if ( (ch == TY_NEWLINE) || (ch == TY_RETURN) ) {
    if (typtr->tyiecho) {
        echoch(ch, typtr, csrptr);
    }
    *typtr->tyitail++ = ch;
    if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
        typtr->tyitail = typtr->tyibuff;
    }
    /* Make entire line (plus \n or \r) available */
    signaln(typtr->tyisem, typtr->tyicursor + 1);
    typtr->tyicursor = 0; /* Reset for next line */
    return;
}

/* Character to be placed in buffer - send bell if */
/* buffer has overflowed */

avail = semcount(typtr->tyisem);
if (avail < 0) {
    avail = 0;
}
if ((avail + typtr->tyicursor) >= TY_IBUFLEN-1) {
    eputc(typtr->tyifullc, typtr, csrptr);
    return;
}
Xinu Ttyhandle_in (Part 7)

/* EOF character: recognize at beginning of line, but */
/* print and ignore otherwise. */

if (ch == typtr->tyeofch && typtr->tyeof) {
    if (typtr->tyiecho) {
        echoch(ch, typtr, csrptr);
    }
    if (typtr->tyicursor != 0) {
        return;
    }
    *typtr->tyitail++ = ch;
    signal(typtr->tyisem);
    return;
}

/* Echo the character */
if (typtr->tyiecho) {
    echoch(ch, typtr, csrptr);
}

/* Insert in the input buffer */
typtr->tyicursor++;
*typtr->tyitail++ = ch;
Xinu Ttyhandle_in (Part 8)

/* Wrap around if needed */

if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
    typtr->tyitail = typtr->tyibuff;
} return;

/*------------------------------------------------------------------------
* erase1 - Erase one character honoring erasing backspace
*------------------------------------------------------------------------*/

local void erase1(
    struct ttycblk *typtr, /* Ptr to ttytab entry */
    struct uart_csreg *csrptr /* Address of UART’s CSRs */
)
{
    char ch; /* Character to erase */

    if ( (--typtr->tyitail) < typtr->tyibuff) {
        typtr->tyitail += TY_IBUFLEN;
    }
}
/* Pick up char to erase */

ch = *typtr->tyitail;
if (typtr->tyiecho) { /* Are we echoing? */
    if (ch < TY_BLANK || ch == 0177) { /* Nonprintable */
        if (typtr->tyevis) { /* Visual cntl chars */
            eputc(TY_BACKSP, typtr, csrptr);
            if (typtr->tyieback) { /* Erase char */
                eputc(TY_BLANK, typtr, csrptr);
                eputc(TY_BACKSP, typtr, csrptr);
            }
        }
        eputc(TY_BACKSP, typtr, csrptr);/* Bypass up arr*/
        if (typtr->tyieback) {
            eputc(TY_BLANK, typtr, csrptr);
            eputc(TY_BACKSP, typtr, csrptr);
        }
    }
    else { /* A normal character that is printable */
        eputc(TY_BACKSP, typtr, csrptr);
        if (typtr->tyieback) { /* erase the character */
            eputc(TY_BLANK, typtr, csrptr);
            eputc(TY_BACKSP, typtr, csrptr);
        }
    }
}

return;
/*-----------------------------------------------*/
/* echoch - Echo a character with visual and output crlf options */
/*-----------------------------------------------*/
void echoch(char ch, struct ttycblk *typtr, struct uart_csreg *csrptr)
{
    if ((ch==TY_NEWLINE || ch==TY_RETURN) && typtr->tyecrlf)
        eputc(TY_RETURN, typtr, csrptr);eputc(TY_NEWLINE, typtr, csrptr);
    else if ( (ch<TY_BLANK||ch==0177) && typtr->tyevis) {
        eputc(TY_UPARROW, typtr, csrptr); /* print ^x */
        eputc(ch+0100, typtr, csrptr); /* Make it printable */
    } else {
        eputc(ch, typtr, csrptr);
    }
}
Xinu Ttyhandle_in (Part 11)

```c
/*------------------------------------------
 * eputc - Put one character in the echo queue
 *------------------------------------------
 */

local void eputc(
    char ch, /* Character to echo */
    struct ttycblk *typtr, /* Ptr to ttytab entry */
    struct uart_csreg *csrptr /* Address of UART’s CSRs */
)
{
    *typtr->tyetail++ = ch; /* Wrap around buffer, if needed */

    if (typtr->tyetail >= &typtr->tyebuff[TY_EBUFLEN]) {
        ttykickout(csrptr);
        return;
    }

    ttykickout(csrptr);
    return;
}
```
Kicking A Device

- We said that kicking a device causes the device to interrupt
- The technique simplifies device driver software
- Key idea
  - If hardware is idle, kicking it forces an interrupt
  - If hardware is currently busy, kicking it has no effect (an interrupt will occur as usual when the operation completes)
- The point: kicking avoids a race condition because the processor does not ask the device whether it is idle before kicking it
Xinu Ttykickout

/* ttykickout.c - ttykickout */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttykickout - "Kick" the hardware for a tty device, causing it to
* generate an output interrupt (interrupts disabled)
*------------------------------------------------------------------------*/

void ttykickout(
    struct uart_csreg *csrptr /* Address of UART’s CSRs */
)
{
    /* Force the UART hardware to generate an output interrupt */
    csrptr->ier = UART_IER_ERBFI | UART_IER_ETBEI;

    return;
}
Xinu Ttyhandle_out (Part 1)

/* ttyhandle_out.c - ttyhandle_out */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttyhandle_out - Handle an output on a tty device by sending more
* characters to the device FIFO (interrupts disabled)
*------------------------------------------------------------------------*/

void ttyhandle_out(
    struct ttcblk *typtr, /* Ptr to ttytab entry */
    struct uart_csreg *csrptr /* Address of UART’s CSRs */
)
{
    int32 ochars; /* Number of output chars sent */
    /* to the UART */
    int32 avail; /* Available chars in output buf*/
    int32 uspace; /* Space left in onboard UART */
    /* output FIFO */
    byte ier = 0;

    /* If output is currently held, simply ignore the call */
    if (typtr->tyoheld) {
        return;
    }
Xinu Ttyhandle_out (Part 2)

/* If echo and output queues empty, turn off interrupts */
if ( (typtr->tyehead == typtr->tyetail) &&
     (semcount(tyosem) >= TY_OBUFLEN) ) {
    ier = csrptr->ier;
    csrptr->ier = ier & ~UART_IER_ETBEI;
    return;
}

/* Initialize uspace to the size of the transmit FIFO */
uspace = UART_FIFO_SIZE;

/* While onboard FIFO is not full and the echo queue is nonempty, xmit chars from the echo queue */
while ( (uspace>0) && typtr->tyehead != typtr->tyetail ) {
    csrptr->buffer = *typtr->tyehead++;
    if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
        typtr->tyehead = typtr->tyebuff;
    }
    uspace--;
}
Xinu Ttyhandle_out (Part 3)

/* While onboard FIFO is not full and the output queue is nonempty, transmit chars from the output queue */

chars = 0;
avail = TY_OBUFLEN - semcount(typtr->tyosem);
while ( (uspace>0) && (avail > 0) ) {
    csrptr->buffer = *typtr->tyohead++;
    if (typtr->tyohead >= &typtr->tyobuff[TY_OBUFLEN]) {
        typtr->tyohead = typtr->tyobuff;
    }
    avail--;
    uspace--;
    ochars++;
}
if (ochars > 0) {
    signaln(tyosem, ochars);
}
return;
Xinu Ttyinit (Part 1)

/* ttyinit.c - ttyinit */

#include <xinu.h>

struct ttycblk ttytab[Ntty];

/*------------------------------------------------------------------------
* ttyinit - Initialize buffers and modes for a tty line
*------------------------------------------------------------------------*/

devcall ttyinit(
    struct dentry *devptr /* Entry in device switch table */
){
    struct ttycblk *typtr; /* Pointer to ttytab entry */
    struct uart_csreg *uptr; /* Address of UART’s CSRs */

    typtr = &ttytab[ devptr->dvminor ];

    /* Initialize values in the tty control block */

    typtr->tyihead = typtr->tyitail = /* Set up input queue */
                        &typtr->tyibuff[0];    /* as empty */
    typtr->tyisem = semcreate(0); /* Input semaphore */
    typtr->tyohead = typtr->tyotail = /* Set up output queue */
                    &typtr->tyobuff[0];    /* as empty */
Xinu Ttyinit (Part 2)

typtr->tyosem = semcreate(TY_OBUFLEN); /* Output semaphore */
typtr->tyehead = typtr->tyetail = /* Set up echo queue */
   &typtr->tyebuff[0]; /* as empty */
typtr->tyimode = TY_IMCOOKED; /* Start in cooked mode */
typtr->tyiecho = TRUE; /* Echo console input */
typtr->tyieback = TRUE; /* Honor erasing bksp */
typtr->tyevvis = TRUE; /* Visual control chars */
typtr->tyecrlf = TRUE; /* Echo CRLF for NEWLINE */
typtr->tyicrlf = TRUE; /* Map CR to NEWLINE */
typtr->tyierase = TRUE; /* Do erasing backspace */
typtr->tyierasesec = TY_BACKSP; /* Primary erase char */
typtr->tyierasesec2 = TY_BACKSP2; /* Alternate erase char */
typtr->tyeof = TRUE; /* Honor eof on input */
typtr->tyeofch = TY_EOFCH; /* End-of-file character */
typtr->tyikill = TRUE; /* Allow line kill */
typtr->tyikillc = TY_KILLCH; /* Set line kill to ^U */
typtr->tyicursor = 0; /* Start of input line */
typtr->tyoflow = TRUE; /* Handle flow control */
typtr->tyoheld = FALSE; /* Output not held */
typtr->tyostop = TY_STOPCH; /* Stop char is ^S */
typtr->tyostart = TY_STRTCH; /* Start char is ^Q */
typtr->tyocrlf = TRUE; /* Send CRLF for NEWLINE */
typtr->tyifullc = TY_FULLCH; /* Send ^G when buffer */
   /* is full */

/* Initialize the UART */
uptr = (struct uart_csreg *)devptr->dvcsr;
/* Set baud rate */
uptr->lcr = UART_LCR_DLAB;
uptr->dlm = 0x00;
uptr->dll = 0x18;

uptr->lcr = UART_LCR_8N1;    /* 8 bit char, No Parity, 1 Stop*/
uptr->fcr = 0x00;            /* Disable FIFO for now */

/* Register the interrupt dispatcher for the tty device */
set_evec( devptr->dvirq, (uint32)devptr->dvintr );

/* Enable interrupts on the device: reset the transmit and receive FIFOS, and set the interrupt trigger level */
uptr->fcr = UART_FCR_EFIFO | UART_FCR_RRESET |
            UART_FCR_TRESET   | UART_FCR_TRIG2;

/* Start the device */
ttykickout(uptr);
return OK;
Perspective

- UART hardware is primitive
- The device driver software, not the hardware, displays characters on the screen as the user enters them on a keyboard
- Most of the features a user expects, such as erasing backspace, are handled entirely by software
- Unlike abstractions covered earlier (e.g., semaphores), a basic tty driver is incredibly complex
- A driver has many parameters that control its operation
- Small details complicate the code
Summary

- The *device manager* in an operating system provides an interface that applications use to request I/O.

- Device-independent I/O functions
  - Provide a uniform interface
  - Define generic operations that must be mapped to device-specific functions

- Xinu uses a device switch table to map a device-independent operation to the correct driver function.

- A device driver
  - Consists of functions that applications call to perform I/O on the device
  - Also provides an interrupt handler for the device

- Dynamic parameters and other details complicate a tty driver.
Module X

Networking And Protocol Implementation
Location Of Networking In The Hierarchy
Is The Hierarchical Level Correct?

- There are two possible approaches
  - Build a conventional operating system and add networking
  - Build networking code first and ensure all pieces of the operating system are distributed (e.g., a distributed process manager)
- Xinu places networking code at a high level of the hierarchy because most of the operating system is not distributed
A Fundamental Observation

One cannot undertake an operating system design without including network communication protocols, even in the embedded systems world.
Communication Systems

- A variety of network technologies have been devised
  - Wired (e.g., Ethernet)
  - Wireless (e.g., Wi-Fi and 5G)

- A computer can use
  - Local network communication: communicate directly over a network with other systems on the same network
  - Internet communication: communicate over a local network, but send packets through a router to an arbitrary computer on the Internet

- Internet communication has become the standard except for small, special-purpose embedded systems
Communication Protocols

- We use the term communication protocols to describe the standards that specify communication details such as
  - Message formats
  - Data representation (e.g., endianness)
  - Rules for message exchange
  - How to handle errors (e.g., lost packets)
- Protocols used in the Internet are known by the name TCP/IP protocols
Communication Protocols And This Course

• We will not discuss protocol details
• We will consider only a minimalistic subset of Internet protocols and focus on aspects pertinent to operating systems design
  – How applications use the communication system
  – The processes that are needed
  – The need for buffering
• To learn more
  – Read a leading text on TCP/IP
  – Take an internetworking course that uses an expert’s text
Synchronous Interface For Network Hardware

- As in most operating systems, Xinu has a device driver for each network interface
- For example, Xinu defines an *ETHER* device for an Ethernet interface
- The device driver for the device provides
  - Synchronous *read* that blocks until a packet arrives and then returns the packet
  - Synchronous *write* that blocks until a buffer is available and then accepts an outgoing packet
- Our example code assumes all communication uses an Ethernet
DMA Device Drivers

- Ethernet device hardware uses *Direct Memory Access (DMA)*
- The operating system
  - Allocates a set of input buffers and a set of output buffers in memory and give the device the addresses of the buffers
  - Marks the input buffers empty and starts device input
  - Places outgoing packets in the output buffers and starts device output
- The device hardware
  - Picks up outgoing packets directly from the output buffers
  - Delivers incoming packets directly to the input buffer
- See Chapter 16 in the text for explanation of how a DMA driver works
Network I/O

- Except for special-purpose embedded systems, application processes
  - Never *read* or *write* directly to a network device
  - Always invoke network protocol software functions to perform network communication

- Network protocol software in the operating system
  - Accepts requests from applications to contact a remote site, forms outgoing packets as needed, and sends them
  - Blocks applications that request network input until a message and/or data arrives
  - Uses a dedicated process to read incoming packets, process them, and deliver the results to waiting applications
Protocols In Our Example

- You do not need to understand protocols, but you will see the following names:
  
  **IP**  
  Internet Protocol – defines an Internet Protocol address (IP address) for each computer on the global Internet plus the format of Internet packets.

  **UDP**  
  User Datagram Protocol – defines *protocol port numbers* used to identify individual applications on a given computer and a message format used when UDP messages travel across the Internet.

  **ARP**  
  Address Resolution Protocol – allows a computer to find the Ethernet address of a computer on a local network given its IP address.

  **DHCP**  
  Dynamic Host Configuration Protocol – used by a computer at startup to obtain its IP address and related information.

  **ICMP**  
  Internet Control Message Protocol – in our implementation, only used by the *ping* program to see if a computer is alive.
Protocol Headers And Message Formats

- Each packet starts with a series of headers followed by data
- In our implementation, a packet being sent or received will have one of the following forms:

```
Ether Hdr  ARP message
```

```
Ether Hdr  IP Header  UDP Hdr  UDP message (data sent by a local or remote application)
```

```
Ether Hdr  IP Header  UDP Hdr  DHCP Message (only used by the OS at startup)
```

```
Ether Hdr  IP Header  ICMP Hdr  ICMP Message (either a ping request or response)
```
Implementing Concatenated Headers

- Most systems build a packet dynamically, adding headers one at a time as needed.
- Xinu takes a shortcut: define two structures
  - One for an Ethernet header followed by an arp message
  - Another for the three cases of an Internet packet
    * Ethernet header, IP header, UDP header, UDP message
    * Ethernet header, IP header, UDP header, DHCP message
    * Ethernet header, IP header, ICMP header, ICMP message
- A further simplification: the only ICMP messages are *echo request* and *echo reply* (i.e., *ping* messages)
- The point is merely to illustrate protocols in an operating system.
Packet Format Declarations

- The struct `arppacket` defines the format of an Ethernet packet carrying ARP messages
- The struct `(netpacket)` defines two cases of an Internet packet
- The netpacket struct starts with an Ethernet packet header followed by an IP header, and then has a union to define
  - A UDP packet encapsulated in the IP packet
  - An ICMP echo request or reply packet encapsulated in the IP packet
- A separate struct defines a DHCP message (which has many fields)
Network Definitions In net.h (Part 1)

/* net.h */
#define NETSTK 8192 /* Stack size for network setup */
#define NETPRIO 500 /* Network startup priority */
#define NETBOOTFILE 128 /* Size of the netboot filename */

/* Constants used in the networking code */
#define ETH_ARP 0x0806 /* Ethernet type for ARP */
#define ETH_IP 0x0800 /* Ethernet type for IP */
#define ETH_IPv6 0x86DD /* Ethernet type for IPv6 */

/* Format of an Ethernet packet carrying IPv4 and UDP */
#pragma pack(2)
struct netpacket {
    byte net_ethdst[ETH_ADDR_LEN]; /* Ethernet dest. MAC address */
    byte net_ethsrc[ETH_ADDR_LEN]; /* Ethernet source MAC address */
    uint16 net_ethtype; /* Ethernet type field */
    byte net_ipv4; /* IP version and hdr length */
    byte net_iptos; /* IP type of service */
    uint16 net_ipleno; /* IP total packet length */
    uint16 net_ipid; /* IP datagram ID */
    uint16 net_ipfrag; /* IP flags & fragment offset */
    byte net_iptime; /* IP time-to-live */
    byte net_ipproto; /* IP protocol (actually type) */
    uint16 net_ipcheck; /* IP checksum */
    uint32 net_ipsrc; /* IP source address */
    uint32 net_ipdst; /* IP destination address */
};
Network Definitions In net.h (Part 2)

union {
  struct {
    uint16 net_udpsport; /* UDP source protocol port */
    uint16 net_udpdport; /* UDP destination protocol port */
    uint16 net_udplen; /* UDP total length */
    uint16 net_udpcksum; /* UDP checksum */
    byte net_udpdata[1500-28]; /* UDP payload (1500-above) */
  };
  struct {
    byte net_ictype; /* ICMP message type */
    byte net_iccode; /* ICMP code field (0 for ping) */
    uint16 net_iccksum; /* ICMP message checksum */
    uint16 net_icident; /* ICMP identifier */
    uint16 net_icseq; /* ICMP sequence number */
    byte net_icdata[1500-28]; /* ICMP payload (1500-above) */
  };
};

#pragma pack()

#define PACKLEN sizeof(struct netpacket)

extern bpid32 netbufpool; /* ID of net packet buffer pool */
Network Definitions In net.h (Part 3)

```c
struct network { /* Network information */
    uint32 ipucast; /* Computer’s IP unicast address*/
    uint32 ipbcast; /* IP broadcast address */
    uint32 ipmask; /* IP address mask */
    uint32 ipprefix; /* IP (network) prefix */
    uint32 iprouter; /* Default router address */
    uint32 bootserver; /* Boot server address */
    uint32 dnsserver; /* DNS server address */
    uint32 ntpserver; /* NTP (time) server address */
    bool8 ipvalid; /* nonzero => above are valid */
    byte ethucast[ETH_ADDR_LEN]; /* Ethernet multicast address */
    byte ethbcast[ETH_ADDR_LEN]; /* Ethernet broadcast address */
    char bootfile[NETBOOTFILE]; /* Name of boot file */
};

extern struct network NetData; /* Local Network Interface info */
```

- **Global variable** *NetData* holds network information obtained at startup, including
  - The computer’s IP address (needed for outgoing as well as incoming packets)
  - The address mask for the local network
  - The address of an Internet router to use (needed for outgoing packets)
  - The address of an NTP time server (used to obtain the time of day)
Services An Application Can Use

• In this version of Xinu, an application can either
  – Use UDP to exchange messages with another application running on a computer on the Internet
  – Use ICMP to send a *ping* packet and receive a reply from an arbitrary computer on the Internet
• The other protocols (ARP and DHCP) merely provide support; they are handled by the network code and invisible to an application
Identifying An Application

- UDP allows multiple applications on a given computer to communicate with other applications running on computers attached to the Internet.

- To identify a remote application, a sending application must specify two items:
  - The computer on which the remote application runs
  - An ID that identifies a specific application on the computer

- For the two items, UDP uses:
  - The 32-bit IP address of the remote computer
  - A 16-bit integer called a UDP protocol port number that identifies an application

- For this course, you do not need to know how IP addresses and port numbers are obtained; just understand that two items are needed to identify each application.
Features Of Networks Related To Operating Systems

• Three aspects of Internet software relate directly to the operating system
  – The interface that applications use to communicate over the Internet
  – The process structure used internally to implement protocols
  – The need for buffering

• We will consider all three
The Interface To Network Protocols
An Interface Used To Communicate Over The Internet

- Xinu follows the same approach as the well-known socket API
  - Before sending data, an application calls a function to register information about a remote destination (i.e., a specific application on a specific remote computer)
  - Network code in the operating system responds by allocating an internal data structure, placing the information into the data structure, and returning a small integer descriptor that the application uses for communication
  - Similar to other descriptors in Xinu, each descriptor used for network communication is an index into an array, and is called colloquially called a slot number
  - The application uses the descriptor to send and receive data (there’s no need to specify the remote application each time the application sends or receives data)
  - When finished, the application releases the descriptor
Xinu Interface Functions That Applications Use for UDP

- `udp_register` – called by an application to register endpoint information, a remote computer (IP address), remote UDP port, and a local UDP port
- `udp_send` – called by an application to send a UDP packet to a previously-registered endpoint
- `udp_recv` – called by an application to receive a UDP packet from a previously-registered remote endpoint
- `udp_recvaddr` – called by a server application to receive a UDP packet and record the sender’s address (allows an application to receive messages from an arbitrary application)
- `udp_release` – called by an application to release a previously-registered endpoint
- Note: the descriptor returned by `udp_register` must be passed to the other functions
Processing An Incoming UDP Packet

- When a packet arrives, the network code calls internal function `udp_in`
- `Udp_in` searches the table of registered endpoints
  - If the incoming packet matches a registered endpoint, the packet is enqueued on the entry, and the semaphore for the entry is signaled to allow a waiting process (if any) to become ready and read the message
  - Else no match is found in the table, and the incoming packet is ignored (silently dropped)
Timeout And Retransmission

- Retransmission of a packet is fundamental in networking
- Retransmission handles packet loss by sending a second copy if the original is lost
- The idea: repeat the following $K$ times
  - Send a request
  - Wait up to $N$ milliseconds for a reply
- If a reply arrives, process the reply immediately
- If no reply arrives after $K$ times declare failure
- Typically, ($K$ is a small number, such as 3)
Xinu Network Functions And Timeout

• The network interface functions allow an application to specify a maximum time to wait for a reply

• Example
  – When calling udp_recv, an application specifies a maximum time to wait
  – The call either returns a message that was received or \textit{TIMEOUT}

• The ICMP (ping) interface operates the same way as the UDP interface

• The network code for UDP and ICMP implements the timeout
Implementation Of Timeout

- Functions that read an incoming message uses `recvtime` to implement timeout (recall that `recvtime` was covered previously).
- When it calls `udp_recv`, a process specifies a maximum wait time.
- The code in `udp_recv` performs the following steps:
  - If no packet has arrived the slot, the code:
    * Places the current process ID in the data structure for the slot.
    * Calls `recvtime` to block the calling process (note: when a packet arrives for the slot, the network code uses `send` to send a message to the waiting process).
    * If a TIMEOUT occurs, returns TIMEOUT to the caller.
  - Copies the contents of the packet to the callers buffer.
An Example Of Using UDP: Time Of Day

- The first time a process requests the time of day, the `gettime` function
  - Calls `getutime` to contact an NTP time server and obtains the current time of day (seconds since January 1, 1900)
  - Converts the time to Xinu time (seconds since January 1, 1970)
  - Computes and stores the time of day when the system booted (i.e., subtracts `clktime` from the current time of day)
- Once the time of day at which the system booted has been stored, Xinu never needs to contact a time server again
- Instead, `getutime` merely adds `clktime` to the time of day at which the system booted
Obtaining The Time Of Day From A Server

- Communication with an NTP server uses UDP
- To send an NTP message, a sender must know
  - The Internet address of a computer running an NTP server
  - The UDP port protocol number that the NTP server uses
  - A local UDP protocol port number that can be used
- Either DHCP returns the IP address for an NTP server or the code uses the address given by constant \textit{TIMESERVER}
- The local and remote protocol port numbers to use are given by constants
  - Constant \textit{TIMELPORT} defines a local UDP protocol port number to use
  - Constant \textit{TIMERPORT} define the protocol port number for an NTP server (123)
Obtaining The Time Of Day From A Server
(continued)

- Steps taken to obtain the current time
  - Call `udp_register` to obtain a slot number
  - Form an NTP request message and use `udp_send` to send the request to the server
  - Call `udp_recv`, specifying a maximum wait time of TIMETIMEOUT milliseconds
  - Call `udp_release` to release the slot
  - If a timeout occurred, return SYSERR; otherwise, store the time the system booted and return the current time

- Note:
  - Function `getutime` always returns `OK` or `SYSERR`
  - An argument specifies where to store the time of day if successful
Getutime: A Function That Uses UDP (Part 1)

/* getutime.c - getutime */

#include <xinu.h>
#include <stdio.h>

/* getutime - Obtain time in seconds past Jan 1, 1970, UCT (GMT) */

status getutime(
    uint32 *timvar /* Location to store the result */
){
    uint32 now; /* Current time in xinu format */
    int32 retval; /* Return value from call */
    uid32 slot; /* Slot in UDP table */
    struct ntp { /* Format of an NTP message */
        byte livn; /* LI:2 VN:3 and mode:3 fields */
        byte strat; /* Stratum */
        byte poll; /* Poll interval */
        byte precision; /* Precision */
        uint32 rootdelay; /* Root delay */
        uint32 rootdisp; /* Root dispersion */
        uint32 refid; /* Reference identifier */
        uint32 reftimestamp[2]; /* Reference timestamp */
    }
}
Getutime: A Function That Uses UDP (Part 2)

```c
uint32 oritimestamp[2];/* Originate timestamp */
uint32 rectimestamp[2];/* Receive timestamp */
uint32 trntimestamp[2];/* Transmit timestamp */

ntpmsg;

if (Date.dt_bootvalid) { /* Return time from local info */
    *timvar = Date.dt_boot + clktime;
    return OK;
}

/* Verify that we have obtained an IP address */

if (getlocalip() == SYSERR) {
    return SYSERR;
}

/* If the DHCP response did not contain an NTP server address */
/* use the default server */

if (NetData.ntpserver == 0) {
    if (dnslookup(TIMESERVER, &NetData.ntpserver) == SYSERR) {
        return SYSERR;
    }
}
```
Getutime: A Function That Uses UDP (Part 3)

/* Contact the time server to get the date and time */
slot = udp_register(NetData.ntpserver, TIMERPORT, TIMELPORT);
if (slot == SYSERR) {
    fprintf(stderr,"getutime: cannot register a udp port %d\n",
            TIMERPORT);
    return SYSERR;
}

/* Send a request message to the NTP server */
memset((char *)&ntpmsg, 0x00, sizeof(ntpmsg));
ntpmsg.livn = 0x1b;  /* Client request, protocol version 3 */
retval = udp_send(slot, (char *)&ntpmsg, sizeof(ntpmsg));
if (retval == SYSERR) {
    fprintf(stderr,"getutime: cannot send to the server\n");
    udp_release(slot);
    return SYSERR;
}

/* Read the response from the NTP server */
retval = udp_recv(slot, (char *)&ntpmsg, sizeof(ntpmsg), TIMETIMEOUT);
if ( (retval == SYSERR) || (retval == TIMEOUT) ) {
    udp_release(slot);
    return SYSERR;
}

udp_release(slot);

/* Extract the seconds since Jan 1900 and convert */

now = ntim2xtim( ntohl(ntpmsg.trntimestamp[0]) );
Date.dt_boot = now - clktime;
Date.dt_bootvalid = TRUE;
*timvar = now;
return OK;

• Notes

– Only a few lines of code call the network functions

– The Internet protocols send integers in network byte order, and function ntohl converts from network byte order to host byte order for a long integer
The ICMP Interface (For Ping)

- Follows the same approach as UDP
- An application
  - Calls `icmp_register` to register the remote address and receive a descriptor
  - Generates an ICMP request packet and calls `icmp_send` to send the packet
  - Calls `icmp_recv` to receive a reply, specifying a timeout
  - Handles the reply, if a valid reply was received
  - Calls `icmp_release` to release the registered endpoint
  - Either reports success or an error, if the request timed out
The Process Model
For Network Code
DHCP, Network Processes, And Delayed Use

- A computer uses DHCP at startup to obtain an IP address and related information
  - DHCP is only used once (i.e., it is only run during startup)
  - A DHCP message is sent using UDP (i.e., DHCP uses the UDP interface)
  - Sending a UDP message normally requires the sender to know its IP address

- How can a computer send a DHCP message before the computer has an IP address?
- Answer: the computer
  - Uses an all-0s IP address as the sender’s address (0.0.0.0 in dotted decimal)
  - Sends its initial DHCP request to a special all-1s IP broadcast address (255.255.255.255 in dotted decimal)

- The resulting packet is broadcast across the local network and a DHCP server responds without needing the computer’s IP address
DHCP, Network Processes, And Delayed Use
(continued)

- An interesting process coordination problem arises with DHCP
  - To use DHCP, network processes must be running (explained later)
  - Network processes are not started until late in the bootstrap sequence

- Our solution: delay using DHCP until an application needs to use the Internet
  - Start the network processes at the end during system initialization
  - Wait until the first time an application calls `getlocalip` to obtain the local IP address, use the call to trigger sending a DHCP request, obtain the reply and store the IP address locally
  - Note: successive calls to `getlocalip` obtain the stored value locally

- In essence, DHCP runs as a side effect of requesting the local IP address
The Need For Network Processes

- Most operating system functions are merely called by application processes.
- Network code requires independent processes to ensure that an incoming packet is handled, even if no application is waiting for the packet.

Examples:
- When a ping request arrives from another computer, the receiver must generate and send a reply even if no application is running.
- When an ARP request arrives from another computer, the receiver must send a reply before Internet packets can arrive from the computer.

- The device driver for a network hardware device allows processes to read incoming packets and write outgoing packets, but does not interpret the packets or send replies.
- Consequence: a process must always be waiting to read and handle incoming packets.
Xinu’s Network Input Process

- To handle asynchronous packet arrivals, Xinu keeps a network input process running at all times.

- The network input process repeatedly:
  - Calls `read` on the ETHER device to block and wait for the next incoming packet.
  - Handles the packet (e.g., if the packet contains UDP, the network input process calls `udp_in`).

- Sending an ARP replay is trivial — the network input process calls a function that forms a reply and writes it to the ETHER device.

- Unfortunately, sending a ping reply causes a problem.
The Problem With Ping Replies

- Ping replies travel in an IP packet
- Sending an outgoing IP packet *may* require an ARP exchange with another computer
- The steps are
  - Start with an outgoing IP packet
  - While holding the outgoing packet, send an ARP request to find the receiver’s Ethernet address
  - Receive an ARP reply
  - Add the information in the ARP reply to the original IP packet and send it
- The problem: if the network input process blocks to wait until the needed information arrives, a deadlock will result because no process will be running to read the ARP reply packet from the ETHER device
Avoiding Deadlock

- To avoid deadlock

**The network input process must never call a function that blocks to wait for a reply.**

- To prevent the network input process from blocking, Xinu uses a separate IP output process, and arranges for the network input process to deposit outgoing IP packets on a queue for the output process to handle

- The IP output process can block waiting for an ARP reply because the network input process remains running
Communication Between The Network Input And IP Output Processes

- Uses a queue of packets and a semaphore for coordination
- The output process repeatedly
  - Waits on the semaphore until an outgoing IP packet is placed in the queue
  - Performs the ARP exchange if necessary, possibly blocking to wait until a reply has been received and the information extracted
  - Uses the information to send the IP packet
- Note: it is not important that you understand the protocol details, but it important that you realize that protocols dictate the process structure that is needed
Simplified Illustration Of The Xinu Network Process Model

- *Netin* handles incoming UDP and ARP packets

- *Netin* enqueues ping replies for *ipout*, thereby preventing *netin* from blocking
Buffering Incoming Packets
The Need For Packet Queues

• The *netin* process has a high priority

• An application process may have a low priority

• Consequences
  – An application that is waiting for a packet may not execute immediately after the packet arrives
  – A second packet may arrive for a given application before the first packet has been handled

• To accommodate delayed processing, Xinu uses packet queues to absorb a small burst of packets without discarding any

• Note: the above only applies to UDP and ICMP because ARP packets are processed immediately by the netin process
The ARP Cache And Cache Timeout Processing

- The ARP protocol specifies that the network code must keep a cache of recent address bindings.
- Entries in the cache should be removed after 10 minutes.
- Is an additional process needed to implement ARP cache timeout?
- Using an additional process has disadvantages:
  - More context switching overhead
  - Uses system resources, such as stack space, with little real value.
The Xinu Approach To Cache Timeout

- To avoid having an extra process handle cache timeout, Xinu uses a trick
- When storing an entry in the cache, Xinu stores the current time in a timestamp field in the entry
- Whenever searching the cache, the code examines the timestamp field in each entry, and removes the entry if the time has expired
- The approach works well for an ARP cache because the cache is only expected to contain a few entries, and the search proceeds sequentially
Choosing Process Priorities

- The easy part: network processes should run at higher priority than user processes.
- The hard part: deciding whether the input process or output process should have higher priority.
- The choice is not clear:
  - If the input process has higher priority, output may become a bottleneck.
  - If the output process has a higher priority, incoming packets may not be handled quickly.
Summary

- Networking is an essential part of any operating system, and three aspects are important to OS designers
- The interface to protocols (Xinu’s is similar to the socket interface)
  - Register to specify a remote endpoint and obtain a descriptor to use
  - Use the descriptor to send and receive data
  - Release the descriptor
- The process structure for network processes
  - Depends on protocols
  - An input and output process are needed
- Packet buffers (queues to hold packets)
  - Needed because packets arrive in bursts (typically, a small queue suffices)
Module XI

Remote Disk And Remote File Access Mechanisms
Location Of Remote Disk And Remote File Access In The Hierarchy
Distributed Operating Systems

- Distributing OS functionality is extremely difficult
- The extent of sharing is determined by level of network communication in design hierarchy
- There have been many attempts to build a truly distributed operating system; the attempts have met with little success
- A few examples follow
Examples Of Distributed Systems

- Apollo Domain
  - The model: computers on a network share a 96-bit address space
  - Communication is positioned at lowest level of the system
- Xerox Alto Environment
  - The model: each computer has a local process manager, and all share files
  - Communication is positioned between the process manager and the file system
- Unix with Internet protocols
  - The model: interconnected autonomous systems
  - The operating system supplies a communication service, but the operating system itself is not distributed
Examples Of Distributed Systems
(continued)

- Unix’s Network File System (NFS)
  - The model: shared files and file names
  - Allows cross-mounting of directories
  - Builds on the Internet protocols (TCP/IP)
  - Only works among computers with identical user IDs
Remote Storage

- Two aspects of distributed operating systems functionality have emerged as significant
  - Remote disk access
  - Remote file system access
- Industry uses the generic term *remote storage access* to encompass both
- Note: separation of storage from processors has become more popular with cloud computing
The History Of Remote Storage Access

- Remote disks appeared in the 1980s
  - Diskless workstations were invented
  - A remote server provided disk storage for multiple computers
  - When disk I/O was needed, the operating system sent packets over the network to the remote disk server

- Remote file access appeared in the 1980s
  - A remote server provided file systems for multiple computers
  - When an application stored data in a file or read data from a file, the operating system sent requests over the network to the remote file server
Which Is Better?

- Remote disk access
  - Advantage: the operating system can use whatever file system it chooses, guaranteeing that local files and remote files have the same semantics
  - Disadvantage: transferring entire disk blocks and metadata over a network is inefficient compared to transferring data from files

- Remote file access
  - Advantage: Only the actual data stored in a file needs to be transferred over the network
  - Disadvantage: the server defines the naming scheme and the set of operations that can be performed on files
Remote Storage Access In The Modern World

- Remote storage access is alive and well, and is used in many data centers
- For various economic and technical reasons, data centers can choose to
  - Keep large storage facilities separate from the computational servers
  - Use a high-speed network to connect the racks of servers with storage facilities
  - Configure operating systems for remote access
- Note: data center customers who use cloud services may be unaware of the separation between servers and storage
The Cost Of Storage Access

- Bad news: accessing storage over a network is extremely slow
  - Even with a directly-attached solid state disk, a processor can execute thousands of instructions in the time it takes to perform one I/O operation
  - For remote accesses, the time required to access storage is significantly higher
- Consequences
  - Even for local storage, performance becomes unacceptable if storage I/O is not optimized
  - For remote storage, optimizations are essential or the system becomes unusable
Storage Access Pattern

- Storage accesses follow an important pattern that permit optimization:

  Disk accesses are highly repetitive — once a given block of data has been accessed, there is high probability that the same block will be accessed again and again in the near future.

- We use the term *temporal locality of reference* to describe the phenomenon, and say that disk accesses have *high temporal locality of reference*
Using Locality Of Reference

- To reduce the delay incurred in accessing remote storage over a network, an operating system makes heavy use of *caching*.

- Two approaches have been used:
  - The cache contains copies of entire disk blocks.
  - The cache contains pieces of files.

- The operating system:
  - Allocates a set of buffers to provide a cache.
  - Places an item in a buffer when the item is referenced.

- Successive references extract data from the cached copy in a local buffer without going across a network to contact a storage server.
Output With A Storage Cache

- It is easy to see how a storage cache handles incoming data (i.e., read operations)
- A storage cache must also handle write operations
- For example, consider caching outgoing disk blocks
  - When an application or the file system changes data in a disk block, the change is made to the copy in the cache
  - The cached copy is marked dirty to indicate that it has been modified since it was fetched from the storage system
- At some later time, dirty blocks must be written back to disk
- Key idea: writing dirty blocks back to remote storage can continue in background
Cache Management

- A storage cache is much more than a copy of items in memory because the operating system performs active *write-back*

- Using write-back
  - The OS maintains a request queue
  - When a disk block is changed, an entry is added to the queue of storage requests
  - In the background, the operating system continuously removes the next item from the queue and performs the request

- The result: although processes do not delay while data is written to remote storage, changes are propagated to the storage system without significant delay
Caching In A Distributed World

- A problem arises with shared storage items and multiple caches
- Example
  - Two operating systems are sharing a remote disk, $D$
  - Each operating system obtains a copy of disk block $B$
  - The applications on the two systems make changes
  - When the copies are written back to the storage server, one copy will be written first and the second copy will overwrite it
- Consequence: the changes made by one operating system will be lost
Handling Distributed Caches

• Sharing presents a *significant* problem for remote storage

• Specifically
  – On the one hand, each operating system *must* cache data or performance becomes unacceptable
  – On the other hand, unless the systems that are sharing items coordinate, the contents of their cached copies will differ
  – We say that the copies cached on the two systems can become *incoherent*

• The problem of keeping cached copies the same is known as the *cache coherence problem*
Two Examples Of Distributed OS Functionality From Xinu

- We will examine two abstractions from Xinu that deal with remote storage
  - A remote disk access system
  - A remote file access system

- Simplifications
  - The Xinu remote disk system allows caching because only one instance of Xinu accesses a given remote disk (i.e., there is no sharing)
  - The remote file system can be shared because Xinu does not cache data locally (each read or write operation is sent to the remote server)
A Remote Disk System
Disk Hardware

- We use the term *disk* to refer to a solid-state disk (SSD) as well as to an older electro-mechanical disk
- Conceptually, a disk appears to consist of an array of fixed-size *blocks*
- The de facto block size is 512 bytes (even a solid state disk provides a 512-byte block interface)
- The blocks on a disk are numbered 0, 1, 2, ...
- Disk hardware only supports two operations
  - *Fetch* a copy of the $i^{th}$ block into a 512-byte buffer in memory
  - *Store* data from a 512-byte buffer in memory to the $i^{th}$ block on the disk
- The hardware always transfers a complete block between memory and disk
- Note: special-purpose hardware used in high-performance storage systems typically uses a larger block size (e.g., 4096 bytes)
The Remote Disk Paradigm

- The idea: allow an operating system to fetch or store blocks to a remote disk
- In terms of hardware
  - A computer on the network runs remote disk server software
  - The computer has one or more physical disks attached
- In terms of software
  - A client operating system contains software that can send messages over a network to the remote disk server
  - Each request either contains a block to be written to the remote disk or a request to read a block from the remote disk
- Note: the client OS can store arbitrary data in each disk block (e.g., a boot block or pieces of a file system)
A Remote Disk Server In Practice

- Powerful remote disk server hardware provides disks for multiple clients
- The server may
  - Have $N$ physical disks and dedicate each disk to one client
  - Dedicate a *virtualized* disk to each client
- *Virtualized disks*
  - Is a popular approach
  - Provides each client with the illusion of a separate physical disk (i.e., the client has its own set of blocks 0, 1, 2, ...)
  - Maps the client requests onto the set of local disks
  - Hides the details of the mapping from clients
Virtualized Disk Storage

• Several mappings have been used for virtualized disks

• Partitioning
  – The server has a large physical disk
  – The disk is partitioned (i.e., divided) into smaller regions, and each client is mapped onto one of the regions

• Files
  – The server has a local file system, and maps each client to a file
  – When a client accesses block \( k \), the server accesses data in the file at byte offset \( 512 \times k \)

• The point: in either case, a client remains completely unaware of virtualization at the server
The Xinu Remote Disk Interface

- Works exactly like a local disk, by allowing a caller to
  - Write a specified block to the disk
  - Read a specified block from the disk
- Uses the same two basic data structures as a conventional disk driver
  - A cache of recently-accessed disk blocks
  - A queue of pending requests to be sent to the remote disk
The Xinu Remote Disk Interface
(continued)

- Defines a Xinu device named \textit{RDISK} that corresponds to the remote disk
- Arranges driver software for the RDISK device to support \textit{read} and \textit{write} operations
- Hides all network communication
- Note: all Xinu disk drivers interpret the “length” field in \textit{read} and \textit{write} calls as a disk block number
- Example: if the remote disk device is named \textit{RDISK}, to read block 5 of the remote disk, a process calls \textit{read} with 5 as the length argument:

\begin{verbatim}
read(RDISK, &buffer, 5);
\end{verbatim}
The Structure Of Xinu Remote Disk Driver Software

• Like a conventional device driver
  – Has upper-half *read* and *write* functions called by processes
  – Has shared data structures

• Unlike a conventional driver
  – Uses a dedicated, high-priority communication process in place of a lower-half
  – Does not use interrupts to trigger the lower-half, but arranges instead for the communication process to wait on a semaphore until a request arrives

• The communication process handles *all* communication with the remote server
Illustration Of The Remote Disk Device Driver

upper-half functions

write  read

shared data structures

process that provides lower-half functionality

network communication with remote server
Disk Block Semantics

- Xinu’s remote disk system emulates a local disk and offers the same semantics
- Input is synchronous: a *read* blocks the calling process until the data has been fetched from the remote disk
- Output enforces last-write semantics (i.e., a *read* from block $i$ always returns the data most recently written to block $i$)
- Optimizations used to increase performance make enforcement of last-write semantics more difficult
  - Caching
  - A queue of pending requests
The Queue Of Pending Requests

- Operates exactly like the request queue used by a local disk driver
- Each item in the request queue specifies:
  - A disk block number
  - An operation (*read* or *write*)
  - A pointer to a buffer that either contains data to be written (for *write*) or to be filled (for *read*)
  - A process waiting for the request to be fulfilled
- Items in a request queue are ordered in FIFO order (also used for solid-state disks)
- Each item in the cache contains a block number and buffer that holds the data for one disk block
The Sync Operation

- Disk hardware only supports *fetch* and *store* operations
- However, the request queue supports three operations
  - *Read* (fetch a block from disk)
  - *Write* (store a block to disk)
  - *Sync* (synchronize requests)
- The *sync* operation
  - Blocks the caller until all previously-written blocks have been stored on disk
  - Is used by a file system to guarantee data is saved
  - Is invoked with a *control* function
  - May be used by an individual processes as well as a file system
How Sync Works

- A process invokes the “sync” control function
- The device driver
  - Adds a sync request for the process to the request queue
  -Suspends the calling process
- Once the sync request reaches head of queue (i.e., all previous requests have been satisfied), the communication process
  -Resumes the process that made the sync request
- Note that sync is handled locally — no message is sent to the remote server and no data is transferred
Structure Of A Request Queue Node (from rdisksys.h)

/* Operations for request queue */
#define RD_OP_READ 1 /* Read operation on req. list */
#define RD_OP_WRITE 2 /* Write operation on req. list */
#define RD_OP_SYNC 3 /* Sync operation on req. list */

/* Definition of a request queue node */

struct rdqnode {
    struct rdqnode *rd_next; /* Node in the request queue */
    struct rdqnode *rd_prev; /* Pointer to next node */
    int32 rd_op; /* Pointer to previous node */
    uint32 rd_blknum; /* Operation - read/write/sync */
    char *rd_callbuf; /* Disk block number requested */
    pid32 rd_pid; /* Address of caller’s buffer */
    /* Process that initiated the */
    /* request */
};
**Structure Of A Cache Node (from rdisksys.h)**

```c
struct rdcnode {
    struct rdcnode *rd_next; /* Pointer to next node */
    struct rdcnode *rd_prev; /* Pointer to previous node */
    uint32 rd_blknum; /* Number of this disk block */
    byte rd_data[RD_BLKSIZ]; /* Data for the disk block */
};
```

- A given block only appears once in the cache (the latest copy)
- A block is not placed in the cache until it has been written to disk (i.e., a block number is not duplicated in both the request queue and cache)
/* Constants for remote disk device control block */

#define RD_IDLEN 64  /* Size of a remote disk ID */
#define RD_STACK 16384 /* Stack size for comm. process */
#define RD_PRIO 200  /* Priority of comm. process */
/* (Must be higher than any */
/* process that reads/writes */

/* Constants for state of the device */

#define RD_CLOSED 0     /* Device is not in use */
#define RD_OPEN 1       /* Device is open */
#define RD_PEND 2       /* Open is pending */
/* Device control block for a remote disk */

struct rdscblk {
    /* Remote disk control block */
    int32 rd_state;   /* State of device */
    char rd_id[RD_IDLEN]; /* Disk ID currently being used */
    int32 rd_seq;  /* Next sequence number to use */
    struct rdcnode *rd_chead; /* Head of cache */
    struct rdcnode *rd_ctail; /* Tail of cache */
    struct rdcnode *rd_cfree; /* Free list of cache nodes */
    struct rdqnode *rd_qhead; /* Head of request queue */
    struct rdqnode *rd_qtail; /* Tail of request queue */
    struct rdqnode *rd_qfree; /* Free list of request nodes */
    pid32 rd_comproc; /* Process ID of comm. process */
    bool8 rd_comruns; /* Has comm. process started? */
    sid32 rd_comsem; /* Semaphore ID for com process */
    uint32 rd_ser_ip; /* Server IP address */
    uint16 rd_ser_port; /* Server UDP port */
    uint16 rd_loc_port; /* Local (client) UPD port */
    bool8 rd_registered; /* Has UDP port been registered? */
    int32 rd_udpslot; /* Registered UDP slot */
};

extern struct rdscblk rdstab[]; /* Remote disk control block */
Messages Exchanged With The Remote Disk Server

- The remote disk system uses five message types when communicating between the local operating system and the remote disk server:
  
  $Open$ – Prepare the remote disk for use and specify a name
  
  $Close$ – Discontinue use of the remote disk
  
  $Read$ – Read a block from the remote disk
  
  $Write$ – Write a block to the remote disk
  
  $Delete$ – Remove the entire remote disk from the remote server
Names For Remote Disks

- A remote disk server
  - Retains disk contents across server reboots
  - Maintains multiple virtual disks
  - Can handle requests from multiple clients
- To prevent interference, each disk is given a unique name
- A disk name must be passed to the server in each request
- Possibilities
  - Students in a class could each use their login ID as a unique disk name
  - The IP address of a Xinu back-end computer (converted to a text string) could be used as a disk name
Message Formats

• The remote disk software in an operating system and the server software must agree on the format of messages and values used in the messages.

• One possible approach
  – Write the definitions in a document
  – Have software engineers who build pieces of the software follow the document

• A better approach
  – Place the definitions in an include (.h) file, and use the same file in both client and server software
  – Instead of defining individual hex values for each possible request and response, define a “response” bit and use it in the definition of message types

• Xinu uses the latter approach
Message Formats
(continued)

- For each operation, two message formats must be defined, such as
  - *Open* request and reply
  - *Read* request and reply
  - *Write* request and reply
- Note that the format of a reply often differs from the format of a request
- Example
  - A *read request* merely specifies the block number to fetch
  - A *read reply* contains actual data in addition to the block number
Declarations For Message Types (from rdisksys.h)

/************************************************************************
/* Definition of messages exchanged with the remote disk server */
/************************************************************************
/* Values for the type field in messages */

#define RD_MSG_RESPONSE 0x0100 /* Bit that indicates response */
#define RD_MSG_RREQ 0x0010 /* Read request and response */
#define RD_MSG_RRES (RD_MSG_RREQ | RD_MSG_RESPONSE)
#define RD_MSG_WREQ 0x0020 /* Write request and response */
#define RD_MSG_WRES (RD_MSG_WREQ | RD_MSG_RESPONSE)
#define RD_MSG_OREQ 0x0030 /* Open request and response */
#define RD_MSG_ORES (RD_MSG_OREQ | RD_MSG_RESPONSE)
#define RD_MSG_CREQ 0x0040 /* Close request and response */
#define RD_MSG_CRES (RD_MSG_CREQ | RD_MSG_RESPONSE)
#define RD_MSG_DREQ 0x0050 /* Delete request and response */
#define RD_MSG_DRES (RD_MSG_DREQ | RD_MSG_RESPONSE)
#define RD_MIN_REQ RD_MSG_RREQ /* Minimum request type */
#define RD_MAX_REQ RD_MSG_DREQ /* Maximum request type */
/* Message header fields present in each message */

#define RD_MSG_HDR /* Common message fields */
    uint16 rd_type; /* Message type */
    uint16 rd_status; /* 0 in req, status in response */
    uint32 rd_seq; /* Message sequence number */
    char rd_id[RD_IDLEN]; /* Null-terminated disk ID */

/*********************************************/
/* The standard header present in all messages with no extra fields */
#pragma pack(2)
struct rd_msg_hdr { /* Header fields present in each*/
    RD_MSG_HDR /* remote disk system message */
};
#pragma pack()
Message Formats (from rdisksys.h)

/**************************************************************************/
/*Read*/
/**************************************************************************/
#pragma pack(2)
struct rd_msg_rreq { /* Remote disk read request */
    RD_MSG_HDR /* Header fields */
    uint32 rd_blk; /* Block number to read */
};
#pragma pack()
#pragma pack(2)
struct rd_msg_rres { /* Remote disk read reply */
    RD_MSG_HDR /* Header fields */
    uint32 rd_blk; /* Block number that was read */
    char rd_data[RD_BLKSIZ]; /* Array containing one block */
};
#pragma pack()
Message Formats (from rdisksys.h)

/*************************************************************************/
/*Write*/
/*************************************************************************/
#pragma pack(2)
struct rd_msg_wreq { /* Remote disk write request */
   RD_MSG_HDR /* Header fields */
   uint32 rd_blk; /* Block number to write */
   char rd_data[RD_BLKSIZ]; /* Array containing one block */
};
#pragma pack()

#pragma pack(2)
struct rd_msg_wres { /* Remote disk write response */
   RD_MSG_HDR /* Header fields */
   uint32 rd_blk; /* Block number that was written*/
};
#pragma pack()
A Review Of Disk Semantics And The Remote Disk Implementation

- The remote disk implements the same semantics as a local disk

- **Write**
  - Remove any previous copy of the block from the cache
  - Place a request in the request queue

- **Read**
  - See if the request can be satisfied from the cache or a pending write for the block in request queue
  - Place a request in the request queue

- When adding a request to the queue: instead of kicking a hardware device, signal the communication process semaphore
The Lower-half Software

- The communication process, named \textit{rdsprocess}, repeatedly
  - Extracts next item from request queue and builds a request message
  - Calls function \textit{rdscomm} to send message to the server and wait for a reply

- Function \textit{rdscomm}
  - Performs all communication with remote server
  - Adds a sequence number to each outgoing message
  - Handles the details of timeout and retransmission, if needed

- Having a separate function for communication allows extra messages to be sent that are not in the request queue (e.g., a \textit{control} message to delete a disk)
The Importance Of Caching Disk Blocks

- Disk I/O is much more expensive than memory accesses
- Communication to a remote disk server makes disk access *extremely* expensive
- Conclusion: caching is an essential optimization for a remote disk
- Good news: disk accesses exhibit temporal locality in which a given block is accessed repeatedly
- An item can be kept in a cache in memory until the node it occupies is needed for a new item
- Note: most disk drivers (including drivers for local disks) cache blocks as long as possible
Requests And The Effect On A Cache

- It may be possible to handle a *read* request from the cache or from a previous *write* that is still on the request queue.

- When a *write* occurs for block $K$ and the cache contains a node for block $K$, the cached copy must be replaced.

- Question: if the request queue contains a write for block $K$, can the request be eliminated?
Next Steps

- Look through the files for the remote disk driver (in directory device/rds) to see how
  - A call to rdsread works
  - A call to rdswrite works
  - What happens when a process calls

\[
\text{control(RDISK, RD_CTL_SYNC, 0);} \\
\]

- Either ask questions now or come to the next class with questions

P.S. Be sure to look at the latest Xinu code and not the code in the textbook because the code has been updated.
A Remote File System
Remote File Access

- Involves two software components that
  - Operate on two separate computers
  - Communicate over a network or the Internet
- The *remote file server* component
  - Runs on a computer that has a local file system
  - Accepts requests to perform file operations
- The *remote file client* component
  - Is part of an operating system
  - Sends requests to a server and obtains replies
Operations A Remote File System Supports

- A remote file system usually supports typical file operations
  - Open or close a file
  - Read data from an open file
  - Write data to an open file
  - Move to an arbitrary position in an open file
  - Create, delete, or rename files
  - Change a file’s metadata, such as the ownership and access privileges


Design Questions

- Can multiple clients access a given server?
- Can a client access files on more than one server at the same time?
- Must user IDs on the client computer agree with user IDs on the server?
- Exactly what file semantics does a remote file system support?
  - Precisely the same file operations and semantics as a local file system?
  - A subset of the operations and semantics supported by the local file system?
  - A superset of the operations and semantics supported by the local file system?
The Xinu Remote File Paradigm

- Clients on multiple Xinu machines are allowed to send read and write requests to a given server concurrently.

- Allowing multiple clients to access a server introduces the possibility of interference (e.g., two clients may attempt to write to the same byte of a file at the same time).

- The Xinu solution:
  - A server serializes all incoming requests (i.e., enqueues them and handles one at a time).
  - A subsequent `read` always returns the last value written, independent of which client wrote it.
  - If additional coordination is needed among applications using a file, it is the programmer’s responsibility.
Server Operation

- A server maintains a set of currently open files
- When a client sends an *open* request, the server
  - Checks to see if the file is already open, and does nothing if it is
  - Otherwise opens the file and records it in the set of open files
- Read and write operations from all clients refer to the same open file (a client does not have its own open file on the server)
- When all clients close a file, the server closes the file
The Remote File Interface On A Xinu Client System

- Follow pattern to defining a master device \((RFILESYS)\) and a set of pseudo-devices
- To open a remote file, a process calls \textit{open} on the master device

\[
d = \text{open}(RFILESYS, \ "file\", \text{mode});
\]

- The \textit{open} call
  - Allocates one of the file pseudo devices
  - Returns, \(d\), the descriptor of the open device
- The caller uses descriptor \(d\) to \textit{read} or \textit{write} data to the file
- When it finishes using the file, the process calls \textit{close} on descriptor \(d\)
- Note: the RFILESYS device is also used for control operations (e.g., delete a file)
The Structure Of The Remote File System Code

- The remote file system client code differs from the remote disk client code.
- Unlike a remote disk client, a remote file client does not maintain a queue of requests, and does not need a communication process.
- Instead, remote server access uses a synchronous approach:
  - Each operation causes a request-response exchange with the remote server.
  - Only one request can be outstanding at a time.
- An upper-half function:
  - Forms a request message and calls function \texttt{rfscomm} to send the request to the server and obtain a response.
  - Waits for a response, and returns the response to its caller.
The Cost Of Remote File Operations

- The Xinu design has a downside: high latency
- Except for *seek*, each upper-half function performs an exchange with server
- Sending a request over a network and obtaining a response introduces significant latency
- The communication overhead is highest when only a small amount of data is transferred per request
- Example: sending 1000 bytes of data in a single request instead of one byte reduces the number of packets transferred by a factor of 1000!
- Consequence: programmers are discouraged from using *putc* or *getc* to access a remote file because using *read* and *write* to transfer large blocks of data incurs much less overhead
The Question Of File System Semantics

The Xinu remote file server runs on a Unix system (Linux) that has

- Hierarchical directories
- File modes and timestamps
- Hard and symbolic links

Further, Xinu defines an "o" mode used when opening a file (the file must exist), but Linux does not have an exact equivalent

There are two possibilities

- Arrange the remote file server to emulate (when possible) the Xinu file semantics
- Allow applications running on Xinu to use the Linux file system functionality and semantics
Our Design

Our remote file server implements Xinu file system semantics whenever doing so is both feasible and efficient.

- Example: to emulate Xinu "o" and "n" modes, the remote file server checks whether the file exists before opening it.

The system provides Xinu applications with access to additional Linux file system functionality via the control function, allowing a Xinu process to:

- Create or remove a directory
- Truncate a file
- Obtain the current size of a file (which allows a Xinu application to move to the end and append new data)
Operations For The Xinu Remote File System

- **Open** – open a file
- **Close** – terminate use of file
- **Read** or **getc** – obtain data from a file
- **Write** or **putc** – deposit data in a file
- **Size** – obtain the current file size
- **Delete** – remove a file
- **Truncate** – discard any existing contents
- **Mkdir** – make a directory
- **Rmdir** – remove a directory
- **Seek** – move to specified position (handled locally; no message sent to server)
File Position Information

- Operations like *read* and *write* assume the file system maintains a current position for each open file.

- For example, when an application calls *read*, the application receives bytes starting at the current file position (and the position is updated).

- Question: should the file position be maintained at the server or the client?

- Note the location where the position is stored affects sharing:
  - If the position is kept at the server and multiple clients share an open file, the position changes whenever any of the clients *read* or *write*.
  - Keeping the position information at the client allows multiple clients to each maintain their own file position (and works as long as they coordinate to avoid overwriting parts of the file that others are reading).
Xinu File Position And Seek

- Each call to open is assigned a new pseudo device
- The pseudo-device maintains a file position
- Consequence: two processes can open the same file and maintain their own file position
- The point: Xinu stores the position at the client
  - Every request sent to the server must specify a position explicitly
  - If multiple clients access a file simultaneously, they do not interfere with each other’s position information, but they cannot share the position
Xinu File Position And Seek
(continued)

- The *Seek* operation allows an application to move to a specific byte offset within the file.
- The Xinu design means *seek* is extremely efficient because the operation can be performed locally; no exchange with the server is needed.
Definitions For The Remote File System (Part 1)

/* rfilesys.h - definitions for remote file system pseudo-devices */

#ifndef Nrfl
#define Nrfl 10
#endif

#define RF_NAMLEN 128 /* Maximum length of file name */
#define RF_DATALEN 1024 /* Maximum data in read or write*/
#define RF_MODE_R F_MODE_R /* Bit to grant read access */
#define RF_MODE_W F_MODE_W /* Bit to grant write access */
#define RF_MODE_RW F_MODE_RW /* Mask for read and write bits */
#define RF_MODE_N F_MODE_N /* Bit for "new" mode */
#define RF_MODE_O F_MODE_O /* Bit for "old" mode */
#define RF_MODE_NO F_MODE_NO /* Mask for "n" and "o" bits */

/* Global data for the remote server */

#ifndef RF_SERVER
#define RF_SERVER "example.com"
#endif

#ifndef RF_SERVER_PORT
#define RF_SERVER_PORT 53224
#endif

#ifndef RF_LOC_PORT
#define RF_LOC_PORT 53224
#endif
Definitions For The Remote File System (Part 2)

/* Global data for remote file server access */

struct rfdata {
    int32 rf_seq; /* Next sequence number to use */
    uint32 rf_ser_ip; /* Server IP address */
    uint16 rf_ser_port; /* Server UDP port */
    uint16 rf_loc_port; /* Local (client) UDP port */
    sid32 rf_mutex; /* Mutual exclusion for access */
    bool8 rf_registered; /* Has UDP port been registered? */
};

extern struct rfdata Rf_data;

/* Definition of the control block for a remote file pseudo-device */

#define RF_FREE 0 /* Entry is currently unused */
#define RF_USED 1 /* Entry is currently in use */

struct rflcblk {
    int32 rfstate; /* Entry is free or used */
    int32 rfdev; /* Device number of this dev. */
    char rfname[RF_NAMLEN]; /* Name of the file */
    uint32 rfpos; /* Current file position */
    uint32 rffilemode; /* Mode: read access, write access or both */
};

extern struct rflcblk rfltab[]; /* Remote file control blocks */
Message Formats Used With The Remote File System

- Use the same approach as the remote disk system
- Define a request and reply message for each operation
- Note
  - File rfilesys.h is shared between client and server software
  - Key concept: the message formats and constants are only defined once
Definitions For The Remote File System (Part 3)

/* Definitions of parameters used when accessing a remote server */
define RF_RETRIES 3 /* Time to retry sending a msg */
define RF_TIMEOUT 1000 /* Wait one second for a reply */

/* Control functions for a remote file pseudo device */
define RFS_CTL_DEL F_CTL_DEL /* Delete a file */
define RFS_CTL_TRUNC F_CTL_TRUNC /* Truncate a file */
define RFS_CTL_MKDIR F_CTL_MKDIR /* Make a directory */
define RFS_CTL_RMDIR F_CTL_RMDIR /* Remove a directory */
define RFS_CTL_SIZE F_CTL_SIZE /* Obtain the size of a file */

/**************************** *
/ * Definition of messages exchanged with the remote server *
/ *
/ ****************************

/* Values for the type field in messages */
define RF_MSG_RESPONSE 0x0100 /* Bit that indicates response */
define RF_MSG_RREQ 0x0001 /* Read Request and response */
define RF_MSG_RRES (RF_MSG_RREQ | RF_MSG_RESPONSE)
define RF_MSG_WREQ 0x0002 /* Write Request and response */
define RF_MSG_WRES (RF_MSG_WREQ | RF_MSG_RESPONSE)
Definitions For The Remote File System (Part 4)

/* Message header fields present in each message */

#define RF_MSG_HDR /* Common message fields */
uint16 rf_type; /* Message type */
uint16 rf_status; /* 0 in req, status in response */
uint32 rf_seq; /* Message sequence number */
char rf_name[RF_NAMLEN]; /* Null-terminated file name */

/* The standard header present in all messages with no extra fields */

/*************************************************************/
/* */
/* Header */
/* */
/*************************************************************/

#pragma pack(2)
struct rf_msg_hdr { /* Header fields present in each*/
    RF_MSG_HDR /* remote file system message */
};
#pragma pack()
Definitions For The Remote File System (Part 5)

/*************************************************************************/
/**
/* Read
/**
 /*************************************************************************/

#pragma pack(2)
struct rf_msg_rreq { /* Remote file read request */
  RF_MSG_HDR /* Header fields */
  uint32 rf_pos; /* Position in file to read */
  uint32 rf_len; /* Number of bytes to read */
  /* (between 1 and 1024) */
};
#pragma pack()
#pragma pack(2)
struct rf_msg_rres { /* Remote file read reply */
  RF_MSG_HDR /* Header fields */
  uint32 rf_pos; /* Position in file */
  uint32 rf_len; /* Number of bytes that follow */
  /* (0 for EOF) */
  char rf_data[RF_DATALEN]; /* Array containing data from */
  /* the file */
};
#pragma pack()
Definitions For The Remote File System (Part 6)

```c
#pragma pack(2)
struct rf_msg_wreq { /* Remote file write request */
    RF_MSG_HDR /* Header fields */
    uint32 rf_pos; /* Position in file */
    uint32 rf_len; /* Number of valid bytes in */
    char rf_data[RF_DATALEN]; /* Array containing data to be */
    /* written to the file */
};
#pragma pack()

#pragma pack(2)
struct rf_msg_wres { /* Remote file write response */
    RF_MSG_HDR /* Header fields */
    uint32 rf_pos; /* Original position in file */
    uint32 rf_len; /* Number of bytes written */
};
#pragma pack()
```
Communication With The Remote File Server

- All communication goes through a single function, `rfscomm`
- Each upper-half function (except `seek`) uses `rfscomm`
- `Rfscomm` handles
  - Registering a UDP port
  - The assignment of a sequence number to each outgoing message
  - The transmission of a request
  - Timeout and retry
  - The reception of a reply
  - Validation of reply (to ensure it matches the request)
Remote File Communication (Part 1)

/* rfscomm.c - rfscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfscomm - Handle communication with RFS server (send request and
 * receive a reply, including sequencing and retries)
 *------------------------------------------------------------------------*/

int32 rfscomm (struct rf_msg_hdr *msg, /* Message to send */
int32 mlen, /* Message length */
struct rf_msg_hdr *reply, /* Buffer for reply */
int32 rlen /* Size of reply buffer */) {

int32 i; /* Counts retries */
int32 retval; /* Return value */
int32 seq; /* Sequence for this exchange */
int16 rtype; /* Reply type in host byte order */
int32 slot; /* UDP slot */

/* Code here */
Remote File Communication (Part 2)

/* For the first time after reboot, register the server port */

if ( ! Rf_data.rf_registered ) {
    if ( (retval = udp_register(Rf_data.rf_ser_ip,
                              Rf_data.rf_ser_port,
                              Rf_data.rf_loc_port)) == SYSERR) {
        return SYSERR;
    }
    Rf_data.rf_udp_slot = retval;
    Rf_data.rf_registered = TRUE;
}

/* Assign message next sequence number */

seq = Rf_data.rf_seq++;
msg->rf_seq = htonl(seq);

/* Repeat RF_RETRIES times: send message and receive reply */

for (i=0; i<RF_RETRIES; i++) {

Remote File Communication (Part 3)

/* Send a copy of the message */

retval = udp_send(Rf_data.rf_udp_slot, (char *)msg, mlen);
if (retval == SYSERR) {
    kprintf("Cannot send to remote file server\n");
    return SYSERR;
}

/* Receive a reply */

retval = udp_recv(Rf_data.rf_udp_slot, (char *)reply, rlen, RF_TIMEOUT);
if (retval == TIMEOUT) {
    continue;
} else if (retval == SYSERR) {
    kprintf("Error reading remote file reply\n");
    return SYSERR;
}
Remote File Communication (Part 4)

/* Verify that sequence in reply matches request */
if (ntohl(reply->rf_seq) != seq) {
    continue;
}

/* Verify the type in the reply matches the request */
rtype = ntohs(reply->rf_type);
if (rtype != ( ntohs(msg->rf_type) | RF_MSG_RESPONSE) ) {
    continue;
}

return retval; /* Return length to caller */

/* Retries exhausted without success */
kprintf("Timeout on exchange with remote file server\n");
return TIMEOUT;
An Example Remote File Operation (read)

- An application calls `read`, and control is passed to the upper-half read function, `rflread`

- `Rflread`
  - Forms a `read request` message and calls `rfscomm` to send the message
  - `Rfscomm` blocks the calling process to await a reply
  - When the reply arrives, `rfscomm`
    * Extracts the data from the message and copies it to the caller’s buffer
    * Updates the file position in the control block for the pseudo device
    * Returns to the caller, allowing the `read` to complete

- Note: each open file has a mutual exclusion semaphore to prevent other processes from using the file while an operation proceeds
Remote File Read (Part 1)

/* rflread.c - rflread */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflread - Read data from a remote file
 *------------------------------------------------------------------------
 */
devcall rflread (struct dentry *devptr, /* Entry in device switch table */char *buff, /* Buffer of bytes */int32 count /* Count of bytes to read */)
{
    struct rflcblk *rfptr; /* Pointer to control block */
    int32 retval; /* Return value */
    struct rf_msg_rreq msg; /* Request message to send */
    struct rf_msg_rres resp; /* Buffer for response */
    int32 i; /* Counts bytes copied */
    char *from, *to; /* Used during name copy */
    int32 len; /* Length of name */

    /* Wait for exclusive access */

    wait(Rf_data.rf_mutex);

Remote File Read (Part 2)

/* Verify count is legitimate */
if ( (count <= 0) || (count > RF_DATALEN) ) {
    signal(Rf_data.rf_mutex);
    return SYSERR;
}

/* Verify pseudo-device is in use */
rfptr = &rfltab[devptr->dvminor];

/* If device not currently in use, report an error */
if (rfptr->rfstate == RF_FREE) {
    signal(Rf_data.rf_mutex);
    return SYSERR;
}

/* Verify pseudo-device allows reading */
if ((rfptr->rfmode & RF_MODE_R) == 0) {
    signal(Rf_data.rf_mutex);
    return SYSERR;
}
Remote File Read (Part 3)

/* Form read request */

msg.rf_type = htons(RF_MSG_RREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;  /* Rfscomm will set sequence */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMLEN);  /* Start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) ) { /* Copy name to request */
    if (++len >= RF_NAMLEN) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
    }
}
msg.rf_pos = htonl(rfptr->rfpos);  /* Set file position */
msg.rf_len = htonl(count);  /* Set count of bytes to read */

/* Send message and receive response */

retval = rfscomm((struct rf_msg_hdr *)&msg,
                 sizeof(struct rf_msg_rreq),
                 (struct rf_msg_hdr *)&resp,
                 sizeof(struct rf_msg_rres) );
/* Check response */

if (retval == SYSERR) {
    signal(Rf_data.rf_mutex);
    return SYSERR;
} else if (retval == TIMEOUT) {
    kprintf("Timeout during remote file read\n");
    signal(Rf_data.rf_mutex);
    return SYSERR;
} else if (ntohs(resp.rf_status) != 0) {
    signal(Rf_data.rf_mutex);
    return SYSERR;
}

/* Copy data to application buffer and update file position */

for (i=0; i<ntohl(resp.rf_len); i++) {
    *buff++ = resp.rf_data[i];
}
rfptr->rfpos += ntohl(resp.rf_len);

signal(Rf_data.rf_mutex);
return ntohl(resp.rf_len);
The Remote File Server

- Is launched in a directory on a Unix system
- Accepts requests from client(s)
- Opens each file in the directory where it was started
- Prevents clients from accessing files in higher-level directories
- Examples: the server forbids access to files with names

/users/xxx/y/z
./../././bbb/qq
Summary

- Remote storage access mechanisms are a popular part of many operating systems
- In the Xinu remote disk subsystem, a device corresponds to a remote disk
- The remote disk device driver relies on a process to perform lower-half functions
- Caching is an important optimization for disk systems
- The Xinu remote disk system maintains a cache of recently-used disk blocks as well as a queue of requests
- The Xinu remote file access mechanism uses a synchronous approach that requires a message exchange for each operation, which means the remote file system access code does not need a separate process
- The Xinu remote file system implements Xinu file semantics whenever possible, and uses control to allow applications to access Linux file operations that are not normally available
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
General Observations About Sharing

- One of the most difficult aspects of file sharing revolves 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?
Questions About 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, changing 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.
Multiple File System Partitions (AKA Volumes)

- The idea: divide a physical disk into multiple areas, and place a separate, independent file system in each area
- Add a way to like all partitions together into a single unified hierarchy (e.g., by using Unix’s \textit{mount})
- Advantages
  - Higher reliability: fewer files tend to be lost in a crash
  - Lower maintenance cost: a smaller file system is much faster to check or repair
  - Faster performance on an electromechanical disk (index information kept closer to data files)
- Disadvantage: the partition sizes must be selected when a disk is formatted
Examples Of Multiple Partitions

- Traditional Unix systems had at least two partitions (/ and /usr)
  - The root partition (/)
    * Was used at startup
    * Only contained enough commands to boot the OS and check the file system in the other partition
  - The user file system (/usr)
    * Contained all the user directories
    * A program, `fsck`, checked the usr file system before it was mounted
- Apple recently moved to multiple partitions, even for external disk drives
File System Internals
Each level adds functionality

An implementation may integrate multiple levels for increased efficiency

We will examine each level
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 (as 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 – provide miscellaneous operations on files, such as changing modes or forming links
File Allocation Choices

- How should files be allocated?
- Static allocation
  - The early approach
  - Space is allocated before the file is used
  - The file size cannot grow beyond the limit
  - 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 on a disk)
The Desired Cost Of File Operations

- Read/write
  - The most common operations performed
  - Provide sequential data transfer
  - The desired cost is $O(t)$, where $t$ is size of transfer

- Seek (move to an arbitrary position in the file)
  - Needed for random access
  - Infrequently used
  - The desired cost is $O(\log n)$, 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, especially the latency required to open a file and move to the first byte
- Clever data structures are needed to achieve efficient access
- The data structures are on disk, not in memory
- Good news
  - SSD hardware is much faster than old electromechanical disks
  - Large memories allow file systems to cache many disk blocks
The Underlying Hardware

- Most file systems assume a traditional disk
  - The disk has fixed-size blocks (sectors) that are numbered 0, 1, 2, ...
  - The standard block size is 512 bytes, but cloud storage is moving to 4K blocks

- The disk interface
  - The hardware can only transfer (read or write) a complete block
  - The hardware provides random access by block number

- An important point, especially for metadata

  Disk hardware cannot perform partial-block transfers.
An Example
File System
The Xinu File System

- For now, assume one file system per disk
- Views the underlying disk as an array of 512-bytes blocks
- Takes a simplistic approach by dividing the disk into three areas
  - Directory area (only one block)
  - File index area (a small number of blocks)
  - Data area (the rest of the disk)

- The size of the index and data areas is chosen when the disk is formatted
The Data Area

- The file system treats the entire data area as an array of *data blocks*
  - Inside the file system, data blocks are numbered from 0 to $D - 1$, where $D$ is the total number of data blocks
  - Each data block is 512 bytes long, and occupies one physical disk block
  - Data block $j$ is not located at disk block $j$ because data blocks start beyond the directory and index blocks
  - Blocks in the data area only store file contents
  - Currently unused data blocks are linked on a free list on the disk
The Index Area

- The file system treats the index area as an array of *index blocks* (*i*-blocks)
  - Inside the file system, index blocks are numbered from 0 to \( I - 1 \), where \( I \) is the total number of index blocks
  - An index block is smaller than a physical disk block
  - Because an index block is smaller than 512 bytes, multiple index blocks occupy a given disk block
  - Each index block stores
    * Pointers to data blocks that make up a file
    * The offset in file of first data byte indexed by the index block
  - Currently unused index blocks are linked on free list on the disk
The Directory Area

- The file system treats the directory as an array of pairs:
  (file name, first index block for the file)

- Conceptually
  - A file consists of a list of index blocks with pointers to data blocks that contain the bytes of the file
  - A directory entry provides a mapping from a name to the index block list for the file

- In Xinu, the entire directory occupies the first physical disk block on the disk
- The directory is limited, but has sufficient size for a small embedded system
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).
Free Lists

- A Xinu file system contains two free lists
  - All the index blocks that are not currently used for any file are linked onto a free list of index blocks (on disk)
  - All the data blocks that are not currently used are linked onto a free list of data blocks (on disk)
- The directory contains “pointers” to the two free lists
- Important note: although we use the term *pointers*, the values are really the number of the first index block on the free list and the number of the first data block on the free list
A Few Index Block Details

- Think of the diagram, and imagine a linked list of index blocks for each file.

- An index block contains:
  - A pointer to the next index block
  - The byte offset of the first byte in the file indexed by this index block
  - Pointers to 16 data blocks of the file indexed by this index block

- Remember that a pointer is merely an integer that specifies one of the data blocks for the file.

- Although the diagram looks like a linked list in memory, each list is actually on disk.

- To find the data block for a given offset in the file, the file system must walk along the linked list of index blocks, which means reading items into memory.

- Xinu defines null values for both an index block pointer and a disk block pointer.
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 devices” defined when system configured
  - A single pseudo device, $LFILESYS$, is used to open files on the local file system
  - A set of $K$ additional file 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 $LFILESYS$ device only implements open and control (e.g., to delete or truncate a file)
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

• To access the file, the application calls read, write, and (possibly) seek, passing desc as the device descriptor on each call

• 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 the disk, if 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, $b$
    * Set the byte pointer to first byte in buffer $b$
The Xinu File Access Paradigm
(continued)

• When the application reads or writes data
  – If the file position has moved to a new data block, fetch the data block from disk
  – Read or write data from/to the data in memory, incrementing the position for each byte

• Note: even if all data in a given data block has been consumed, the file system does not fetch the “next” data block until it is referenced

• Key points
  – A copy of one index block and one data block are kept in memory
  – A pointer (lfbyte) gives the address of the next byte to read from the in-memory buffer that holds a copy of the current data block of the next byte to be read
The File System Pseudo-device Control Block

/* excerpt from lfilesys.h */

struct lflcblk { /* Local file control block */
    byte  lfstate; /* Is entry free or used */
    did32 lfdev; /* device ID of this device */
    sid32 lfmutex; /* Mutex for this file */
    struct ldentry *lfdirptr; /* Ptr to file’s entry in the */
        /* in-memory directory */
    int32  lfmode; /* mode (read/write/both) */
    uint32 lfp_pos; /* Byte position of next byte */
        /* to read or write */
    char  lfname[LF_NAME_LEN]; /* Name of the file */
    ibid32 lfinum; /* ID of current index block in */
        /* lfiblock or LF_INULL */
    struct lfiblk lfiblock; /* In-mem copy of current index */
        /* block */
    dbid32 lfdnum; /* Number of current data block */
        /* in lfdblock or LF_DNULL */
    char  lfdblock[LF_BLKSIZ]; /* in-mem copy of current data */
        /* block */
    char  *lfbyte; /* Ptr to byte in lfdblock if */
        /* pos is inside current block */
    bool8 lfibdirty; /* Has lfiblock changed? */
    bool8 lfdbdirty; /* Has lfdblock changed? */
};
Example File Access: lflgetc.c (Part 1)

/* lflgetc.c - lflgetc */

#include <xinu.h>

/*------------------------------------------------------------------------
* lflgetc - Read the next byte from an open local file
*------------------------------------------------------------------------
*/
devcall lflgetc (  
    struct dentry *devptr /* Entry in device switch table */
)
{
    struct lflcblk *lfptr; /* Ptr to open file table entry */
    struct ldentry *ldptr; /* Ptr to file’s entry in the */
    /* in-memory directory */
    int32 onebyte; /* Next data byte in the file */

    /* Obtain exclusive use of the file */

    lfptr = &lfltab[devptr->dvminor];
    wait(lfptr->lfmutex);

    /* If file is not open, return an error */

    if (lfptr->lfstate != LF_USED) {
        signal(lfptr->lfmutex);
        return SYSERR;
    }
Example File Access: lflgetc.c (Part 2)

/* Return EOF for any attempt to read beyond the end-of-file */

ldptr = lfptr->lfdirptr;
if (lfptr->lfpos >= ldptr->ld_size) {
    signal(lfptr->lfmutex);
    return EOF;
}

/* If byte pointer is beyond the current data block, set up */
/* a new data block */

if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {
    lfsetup(lfptr);
}

/* Extract the next byte from block, update file position, and */
/* return the byte to the caller */

onebyte = 0xff & *lfptr->lfbyte++;
lfptr->lfpos++;
signal(lfptr->lfmutex);
return onebyte;
Concurrent Access To A Shared File

- The chief design difficulty: shared file position
- Ambiguity can arise 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’s local file system 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 disk block
  – An index block is smaller than a disk block

• To store index block number $i$
  – Map $i$ to a physical disk 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)
Xinu stores seven i-blocks in each disk block

To compute the disk block number in which i-block $k$ resides, divide $k$ by 7 (integer arithmetic) and add 1 (because the index blocks start at disk block 1)

To compute the byte position of i-block $k$ within a disk block, calculate $r$, the remainder of dividing $k$ by 7, and multiply $r$ times the size of an i-block
Xinu I-block Definition

/* excerpt from lfilesys.h */

#define LF_AREA_IB 1 /* First sector of i-blocks */
#define LF_INULL (ibid32) -1 /* Index block null pointer */
#define LF_IBLEN 16 /* Data block ptrs per i-block */
#define LF_IMASK 0x00001fff /* Mask for the data indexed by */
    /* one index block (i.e., bytes 0 through 8191). */
#define LF_IDATA 8192 /* Bytes of data indexed by a single index block */

/* Structure of an index block on disk */

struct lfiblk { /* Format of index block */
    ibid32 ib_next; /* Address of next index block */
    uint32 ib_offset; /* First data byte of the file */
    dbid32 ib_dba[LF_IBLEN]; /* Ptrs to data blocks indexed */
};

#define ib2sect(ib) (((ib)/7)+LF_AREA_IB) /* Conversion between index block number and disk sector number */
#define ib2disp(ib) (((ib)%7)*sizeof(struct lfiblk)) /* Conversion between index block number and the relative offset within a disk sector */
Xinu Function To Read An I-block

/* excerpt from lfibget.c */

/* lfibget -- get an index block from disk given its number */

void lfibget(
    did32 diskdev, /* Device ID of disk to use */
    ibid32 inum,  /* ID of index block to fetch */
    struct lfiblk *ibuff /* Buffer to hold index block */
)
{
    char *from, *to; /* Pointers used in copying */
    int32 i; /* Loop index used during copy */
    char dbuff[LF_BLKSIZ]; /* Buffer to hold disk block */

    /* Read disk block that contains the specified index block */

    read(diskdev, dbuff, ib2sect(inum));

    /* Copy specified index block to caller’s ibuff */

    from = dbuff + ib2disp(inum);
    to = (char *)ibuff;
    for (i=0 ; i<sizeof(struct lfiblk) ; i++)
        *to++ = *from++;
    return;
}
Xinu Function To Write An I-block (Part 1)

/* lfibput.c - lfibput */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfibput - Write an index block to disk given its ID (assumes
 *          mutex is held)
 *------------------------------------------------------------------------*/

status lfibput(
    did32 diskdev, /* ID of disk device */
    ibid32 inum, /* ID of index block to write */
    struct lfiblk *ibuff /* Buffer holding the index blk */
)
{
    dbid32 diskblock; /* ID of disk sector (block) */
    char *from, *to; /* Pointers used in copying */
    int32 i; /* Loop index used during copy */
    char dbuff[LF_BLKSIZE]; /* Temp. buffer to hold d-block */

    /* Compute disk block number and offset of index block */

    diskblock = ib2sect(inum);
    to = dbuff + ib2disp(inum);
    from = (char *)ibuff;

/* Read disk block */
if (read(diskdev, dbuff, diskblock) == SYSERR) {
    return SYSERR;
}

/* Copy index block into place */
for (i=0 ; i<sizeof(struct lfiblk) ; i++) {
    *to++ = *from++;
}

/* Write the block back to disk */
write(diskdev, dbuff, diskblock);
return OK;
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 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 (e.g., to /dev/console)
  - 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 stored in 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 disk block that contains *128* pointers to blocks that contain indirect pointers
- One *triply indirect* pointer points to a disk block that contains *128* pointers to blocks that contain 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

doubly indirect

i-node

other info

to triply indirect
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)
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
Module XIII

File Names And A Syntactic Namespace
Location Of The Namespace In The Hierarchy
Review

- We said that a file system has three conceptual layers

  - DISK HARDWARE
  - DISK DEVICE DRIVER
  - FILE ACCESS
  - DIRECTORY ACCESS
  - FILE NAMING SCHEME

- We have already considered directory and file access mechanisms
- What about naming?
Identifiers

• Many modules in an operating system use the term identifier (ID) to designate an object identifier

• Processes use the IDs to identify objects when operating on them

• Examples
  – Each semaphore is given an ID that processes use when they call wait and signal
  – Each process has an ID that is used when invoking process management functions, such as suspend, resume, send, ready, and kill
  – Each device has an ID used in device manager functions, such as read and write
Identifier Choice And Efficient Object Identification

• We have seen that
  – The identifiers used in an operating system consist of integers
  – Choosing values 0, 1, 2, ... for identifiers means that the mapping from an identifier to an actual object is extremely efficient

• To achieve high efficiency, early programming languages required programmers to use numerical identifiers (e.g., FORTRAN specified that when a programmer called write, identifier 6 designated a printer)

• However

  Although using numerical identifiers makes mapping efficient, humans find such identifiers difficult to understand and remember.
Solving The Problem

- Tension exists between humans and machines
  - Computers work best with numeric values
  - Humans prefer identifiers that convey meaning
- How can we resolve the tension?
- A general approach is used to provide the advantages of each
  - Allow humans to use meaningful symbolic identifiers
  - Perform early binding to convert the symbolic identifiers into an internal numeric form (because early binding increases efficiency)
  - Once the binding has been done, use the numeric form
An Example Of Binding A Name To An Integer Identifier

• recall that in a Unix file system
  – The file system uses an inode number to identify a file
  – Humans use file names

• The directory system provides a binding from names to files

• A process uses a path name when opening a file:

  desc = open("path", omode, mode);

• The open function searches the directory system, and maps the name to an i-node number

• The file system used the i-node number internally
Identifiers, Mappings, And Vulnerability

- Facts
  - Symbolic identifiers work best for humans
  - Operating systems contain mechanisms that map symbolic identifiers into efficient, internal identifiers
- Unfortunately, revealing the mapping to users has a potential security downside: malevolent users (or malware) may be able to guess how internal identifiers are assigned, and then use the information to access other system resources
- An example:
  - The Internet standards specify protocol port numbers used for each application
  - An attacker can use the protocol port numbers in attempts to access services on a computer, even if the owner does not advertise the services
The Principle Of Transparency

- To prevent outsiders from misusing information about the internal representation, operating system designers follow a rule known as the *principle of transparency*:

  *Whenever possible, applications should remain unaware of implementation details such as the location of an object or its representation.*
Protecting Against Attack

- It may seem that because it reveals a mapping between identifiers and values used internally, early binding always introduces a vulnerability.
- However, knowing the internal value may not give others access.
- As an example, consider file descriptors in Unix:
  - A descriptor is only meaningful within one process.
  - Even if process 20 learns that process 27 is using descriptor 4 to access file X, process 20 will not be able to access the file because when process 20 references descriptor 4, the reference will be interpreted with respect to process 20’s descriptor table.
- Safety: it is safe to use early binding and to reveal how identifiers are mapped to internal values provided that additional protections are employed to prevent the knowledge from being exploited.
Transparency And Functionality

• At first glance, the principle of transparency seems both reasonable and innocuous

• However ... true transparency has consequences for functionality

• Consider an operating system that offers access to both a local file system and a remote file system
  – Suppose the functionality of the local and remote file systems differ (a common situation)
  – To keep local and remote access file completely transparent, the operating system must keep the interface identical
  – The effect: transparency means the set of operations that applications can use are limited to the intersection of the operations available on the two file systems

• Another consequence: if the interface is truly transparent, an application will not be able to find the actual location of an object, even if doing so is important
File Namespace

- We use the term *file namespace* to refer to the set of all valid file names.
- Note that a namespace includes all possible names, not just the names of existing files.
- Items in a namespace are bound by both syntactic and semantic restrictions.
- Examples:
  - Most file systems place a bound on the maximum length of a file name.
  - Some file systems prohibit unprintable characters in file names or prohibit “separator” characters (e.g., Unix prohibits the slash character from appearing in files names).
A Namespace For Hierarchical Directories

- Systems such as Multics and Unix provide a hierarchical directory structure in which a directory can contain files and other directories.
- A namespace for such a system usually:
  - Includes names for directories as well as files.
  - Gives uniform names for all files.
  - Two forms are used: absolute names and relative names.
An Example Of Absolute And Relative Names (Unix)

• In Unix, an *absolute name* begins with a slash, and gives a path downward from the root of the file system

  Examples of absolute names
  – /usr/bin/awk
  – /var/lib/vim/addons
  – /dev/null

• In Unix, a *relative name* gives a path starting from the current directory

  Examples
  – myfile
  – bin/awk
  – ../../lib/java/runtime
Heterogeneous File Names

• A variety of file naming schemes have been created
  – MS-DOS \(\text{Device: file}\)
  – V-System \([\text{context}]\) \(\text{name}\)
  – BSD Unix \(\text{machine: path}\)

• Unfortunately
  – The Internet means that when referring to a file on a remote computer, the form of the file name may differ completely from the form of the file name used on the user’s local computer
  – No single naming scheme is best, which means that it is unlikely a single scheme will ever be adopted by all systems
Gluing Together Many File Systems

- Can an operating system hide differences in names and provide users with a single, uniform view?

- One approach consists of building a single, large file system out of multiple pieces by inserting an extra level on top of the file system software hierarchy.

- The extra level is arranged to
  - Present a uniform interface to users and application programmers
  - Hide the details of the underlying file systems
  - Map unified file system names to names for specific underlying file systems
  - Map generic file operations to appropriate operations on the underlying file systems
An Example

- Suppose a computer has two disks and two separate Unix file systems, \textit{F1} and \textit{F2}
- To unite them into a single giant file system
  - Create a new root directory that is “above” the two file systems
  - Add two entries to the new root, one for \textit{F1} and one for \textit{F2}
- In the new system, a name of the form \textit{/F1/path\_1} will refer to a file in file system \textit{F1}, and a name of the form \textit{/F2/path\_2} will refer to a file in file system \textit{F2}
- Examples
  - The name \textit{/F1/var/mail/smith} will be interpreted as a reference to \textit{/var/mail/smith} in file system \textit{F1}
  - The name \textit{/F2/usr/bin/awk} will be interpreted as a reference to \textit{/usr/bin/awk} in file system \textit{F2}
Unix provides a unification mechanism similar to the one described above.

The idea is straightforward: make a file system appear to be one of the directories in the main file system.

The procedure is:

- Start with one file system as the root.
- Create an empty directory at any point in the directory, call it X hierarchy.
- Use the `mount` command to specify that another file system should attach in place of directory X.

Once the new file system is mounted, its root directory appears in place of directory X.

Note: mounting only affects the cached copy of an i-node in memory; the two file systems remain independent on disk.
Using Names
Compound Names And Their Parts

• Consider the Unix file name

/var/lib/vim/addons

• Because we know the meaning of items in a Unix file name, we tend to think of the name as three directories (*var*, *lib*, and *vim*) plus one file (*addons*)

• Syntactically, we think of the name as the four items separated by slash characters
Hierarchies, Strings, And Prefixes

• Instead of thinking of a file name as specifying a sequence of directories, think of it merely as a string of characters

• For example, think of /var/lib/vim/addons as a string of nineteen characters

• Observe that
  – The one-character prefix / specifies what we think of as the top level directory
  – The four-character prefix /var specifies what we think of as a second-level directory
  – The eight-character prefix /var/lib specifies what we think of as a third-level directory, and so on

• The point is that some prefixes of the string specify directories in the directory hierarchy
The Prefix Property

- We use the term *prefix property* to describe the relationship between prefixes and the directory hierarchy.
- Longer prefixes refer to items further down the directory hierarchy.
- Of course, an arbitrary length prefix may not correspond to a directory.
- In the example string `/var/lib/vim/addons`
  - The two-character prefix `/v` does not name a directory.
  - The seven-character prefix `/var/li` does not name a directory.
  - The seventeen-character prefix `/var/lib/vim/addo` does not name a directory.
- Conclusion: a prefix length must be chosen carefully or the prefix will not correspond to a directory or a file.
Using A Prefix To Identify A File System

- As an example of how we can use the prefix property to build a namespace, imagine a computer with two file systems, a local file system that uses names of the form C:file and a remote file system that supports Unix names of the form file or relative_path.

- Now imagine creating a file namespace that includes both file systems.

- We could choose the prefix /local/ for the local file system and /remote/ for the remote file system.

- If a user specifies the name /remote/a/b/c, the system will remove the prefix, and use the name a/b/c as a path name on the remote file system.

- The local file system is slightly more complex because the prefix must be replaced instead of removed: if a user specifies the name /local/X, the system must replace the prefix /local/ with the string C:, and then use the name C:X as the name of a file on the local file system.
Generalized Prefix Mapping

- Instead of building code that uses `if` statements to check for `/local/` and `/remote/`, consider a generalized system.
- The generalized system will use a table of prefixes.
- The *prefix table* will contain one entry for each possible prefix.
- Each entry in the table will be a 3-tuple that specifies:
  - A prefix to be matched
  - A replacement string (e.g., the prefix “/local/” might be replaced by “C:”)
  - A file system to be used once the file name has been modified
Using A Prefix Table

• When a user specifies a file name, the prefix mapping code searches the prefix table

• When it finds a prefix in the table that matches the prefix of the name the user specified, the code
  – Modifies the user’s file name by substituting the replacement string in the entry in place of the prefix
  – Uses the modified file name as the file name for the file system specified in the entry

• A prefix table has several advantages over using conditional statements in the code
  – Nothing is hard-wired
  – The mappings can be changed at any time, even while the operating system continues to run
A Prefix Table For Our Example

- Consider the local and remote file systems described on previous slides.
- Assume the local file system is \textit{local\_fs} and the remote file system is \textit{remote\_fs}.
- A prefix table that captures the needed mappings contains two entries:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Replacement</th>
<th>File System</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/local/&quot;</td>
<td>&quot;C:&quot;</td>
<td>\textit{local_fs}</td>
</tr>
<tr>
<td>&quot;/remote/&quot;</td>
<td>&quot;&quot;&quot;</td>
<td>\textit{remote_fs}</td>
</tr>
</tbody>
</table>

- Because the entry for the remote file system has a replacement set to the null string, the prefix \textit{/remote/} will be removed from the name, and no further modifications will occur.
Names, Prefixes, And Subdirectories

• Treating each name as a string has several implications

• No meaning: if a name is merely a string, no special meaning is assigned to any character
  – Therefore, “x/y/z” is merely a string of five characters
  – Humans might think of y as a subdirectory of x, but code that operates on strings does not

• Consequence: treating names as character strings means
  – The segments in a name may not match the levels in the directory hierarchy
  – Applications cannot infer semantic meaning
Possible Hierarchies

- Using prefixes allows us to create a unified abstract namespace that includes multiple file systems.

- We have seen, for example, that an abstract namespace can include both local and remote file systems.

- Interestingly, various arrangements of a hierarchy are possible: local and remote can be located at the same level, remote can be a subdirectory of the local file system, or the local file system can be a subdirectory of the remote file system, as illustrated.
A Xinu Namespace
The Xinu Syntactic Namespace

- Xinu uses a single namespace to unify multiple file systems into a single abstract naming scheme

- The characteristics are
  - Syntactic: the Xinu namespace uses prefix mapping
  - Optional: applications can choose to bypass the namespace and access a specific file system directly
  - Dynamic: the namespace mappings can be changed at run time
Xinu Namespace Details

- In Xinu, everything is a device; so the namespace is a device (actually a pseudo-device because there is no real hardware)
- The namespace pseudo-device is configured to the name \textit{NAMESPACE}
- An application
  - Calls \textit{open} on the \textit{NAMESPACE} pseudo-device
  - Supplies a file name
- The namespace \textit{open} function
  - Uses a prefix table to find an entry in the prefix table where the prefix matches the file name
  - Modifies the file name by replacing the prefix with the specified replacement string
  - Calls \textit{open} on the file system device in the entry
An Example Of Passing An Open Call To A File System

• Consider a Xinu namespace that has been configured to have an entry in the prefix table with the following values

  ("/remote/", ",", RFILESYS)

• Suppose a process calls

  open(NAMESPACE, "/remote/xyz", "r");

• The namespace driver will
  – Map "/remote/xyz" into "xyz"
  – Call open on the RFILESYS device with the mode argument the user specified

  open(RFILESYS, "xyz", "r");
Summary Of The Xinu Syntactic Namespace Operation

- The Xinu namespace uses a table to hold a set of 3-tuples
  \[(\text{prefix}, \text{replacement}, \text{device})\]
  
  where \text{prefix} and \text{replacement} are strings of characters

- Given a file name, the namespace
  - Checks the name against each entry in the table
  - If a prefix in the table matches the beginning of the name
    * Rewrite the file name by substituting the replacement string for the prefix
    * \textit{Open} the resulting file name on the device specified in the table entry, and return the result to the caller

- If none of the prefixes in the table match the name, return SYSERR to caller
A Default Prefix Mapping

- We said that if no prefix in the table matches a name, open returns SYSERR.
- To prevent errors, one can install a default entry in the table (i.e., an entry that will be used if none of the other entries match).
- Prefix matching means a default entry can be added without modifying the lookup code and without any special cases.
- To insert a default entry, specify a prefix of the null string:
  - The null string is considered to be a prefix of all other strings.
  - Therefore, an entry in which the prefix is null will always match.
- We will see examples of how a default entry can be used.
Adding Entries To The Prefix Table

- Xinu uses the same approach as Unix, a *mount* function
- In Xinu, *mount* merely adds an entry to the prefix table
- As expected, Xinu’s *mount* function takes three arguments
  - A string that specifies a prefix
  - A string that specifies a replacement
  - A device descriptor for a file system pseudo-device
- Example:

  ```
  mount ("/remote/", ",", RFILESYS);
  ```
Namespace Initialization

- The driver for the \textit{NAMESPACE} pseudo-device includes initialization function \textit{naminit}
- After Xinu boots, it calls the \textit{init} function for each device, which means it calls \textit{naminit} for the \textit{NAMESPACE} device
- \textit{Naminit}
  - Fills in initial values in the prefix table
  - Automatically creates an entry for each device
  - Uses a name of the form /dev/x for a device named X
- Example
  - \textit{Naminit} creates an entry /dev/console for the \textit{CONSOLE} device
  - An \textit{open} on /dev/console will be mapped to an \textit{open} on \textit{CONSOLE}
- To create additional entries, calls to \textit{mount} can be added to \textit{naminit}
An Excerpt From Naminit

• Here is an example of `mount` calls used to build a namespace

    Excerpt from naminit.c

    mount("/dev/null", "", NULLDEV);
mount("/remote/", "remote:", RFILESYS);
mount("/local/", NULLSTR, LFILESYS);
mount("/dev/", NULLSTR, SYSERR);
mount("~/", NULLSTR, LFILESYS);
mount("/", "root:", RFILESYS);
mount("", "", LFILESYS);

• Observe
  – The last entry uses a null prefix to provide a default (the local file system)
  – File names that start with ~/ are mapped to the local file system
  – Names that begin with a slash are mapped to the remote file system, and the prefix root: is added to the name
Declarations For The Namespace

/* name.h */

/* Constants that define the namespace mapping table sizes */

#define NM_PRELEN 64    /* Max size of a prefix string */
#define NM_REPLLEN 96   /* Maximum size of a replacement*/
#define NM_MAXLEN 256   /* Maximum size of a file name */
#define NNAMES 40       /* Number of prefix definitions */

/* Definition of the name prefix table that defines all name mappings */

struct nmentry {
    char nprefix[NM_PRELEN];   /* Null-terminated prefix */
    char nreplace[NM_REPLLEN]; /* Null-terminated replacement */
    did32 ndevice;             /* Device descriptor for prefix */
};

extern struct nmentry nametab[];    /* Table of name mappings */
extern int32 nnames;                 /* Number of entries allocated */
Open Function For The Namespace

/* namopen.c - namopen */

#include <xinu.h>

/*------------------------------------------------------------------------
* namopen - Open a file or device based on the name
*------------------------------------------------------------------------*/
devcall namopen(
    struct dentry *devptr, /* Entry in device switch table */
    char *name, /* Name to open */
    char *mode /* Mode argument */
    )
{
    char newname[NM_MAXLEN]; /* Name with prefix replaced */
    did32 newdev; /* Device ID after mapping */

    /* Use namespace to map name to a new name and new descriptor */
    newdev = nammap(name, newname, devptr->dvnum);

    if (newdev == SYSERR) {
        return SYSERR;
    }

    /* Open underlying device and return status */
    return open(newdev, newname, mode);
}
/* nammap.c - nammap, namrepl, namcpy */

#include <xinu.h>

status namcpy(char *, char *, int32);
did32 namrepl(char *, char[]);

/*------------------------------------------------------------------------
* nammap - Using namespace, map name to new name and new device
*------------------------------------------------------------------------*/
devcall nammap(
       char  *name, /* The name to map */
       char  newname[NM_MAXLEN], /* Buffer for mapped name */
       did32 namdev /* ID of the namespace device */
   )
{
    did32 newdev; /* Device descriptor to return */
    char  tmpname[NM_MAXLEN]; /* Temporary buffer for name */
    int32 iter; /* Number of iterations */

    /* Place original name in temporary buffer and null terminate */
    if (namcpy(tmpname, name, NM_MAXLEN) == SYSERR) {
        return SYSERR;
    }
}
/* Repeatedly substitute the name prefix until a non-namespace */
/* device is reached or an iteration limit is exceeded */

for (iter=0; iter<nnames ; iter++) {
    newdev = namrepl(tmpname, newname);
    if (newdev != namdev) {
        namcpy(tmpname, newname, NM_MAXLEN);
        return newdev; /* Either valid ID or SYSERR */
    }
}

return SYSERR;
Mapping Function For The Namespace (Part 3)

 /*------------------------------------------------------------------------
 * namrepl - Use the name table to perform prefix substitution
 *------------------------------------------------------------------------*/

did32 namrepl(
    char *name, /* Original name */
    char newname[NM_MAXLEN] /* Buffer for mapped name */
){
    int32 i; /* Iterate through name table */
    char *pptr; /* Walks through a prefix */
    char *rptr; /* Walks through a replacement */
    char *optr; /* Walks through original name */
    char *nptr; /* Walks through new name */
    char olen; /* Length of original name */
    /* including the NULL byte */
    int32 plen; /* Length of a prefix string */
    /* *not* including NULL byte */
    int32 rlen; /* Length of replacement string */
    int32 remain; /* Bytes in name beyond prefix */
    struct nmentry *namptr; /* Pointer to a table entry */
/ * Search name table for first prefix that matches *

for (i=0; i<nnames; i++) {
    namptr = &nametab[i];
    optr = name;       /* Start at beginning of name */
    pptr = namptr->nprefix; /* Start at beginning of prefix */

    /* Compare prefix to string and count prefix size */

    for (plen=0; *pptr != NULLCH ; plen++) {
        if (*pptr != *optr) {
            break;
        }
        pptr++;
        optr++;
    }
    if (*pptr != NULLCH) { /* Prefix does not match */
        continue;
    }

    /* Found a match - check that replacement string plus */
    /* bytes remaining at the end of the original name will */
    /* fit into new name buffer. Ignore null on replacement*/
    /* string, but keep null on remainder of name. */

olen = namlen(name,NM_MAXLEN);
rlen = namlen(namptr->nreplace,NM_MAXLEN) - 1;
remain = olen - plen;
if ( (rlen + remain) > NM_MAXLEN) {
    return (did32)SYSERR;
}
/* Place replacement string followed by remainder of original name (and null) into the new name buffer */

nptr = newname;
rptr = namptr->nreplace;
for (; rlen>0 ; rlen--) {
    *nptr++ = *rptr++;
}
for (; remain>0 ; remain--) {
    *nptr++ = *optr++;
}
return namptr->ndevice;
}
return (did32)SYSERR;
Problems With A Syntactic Approach

- Infinite name expansion
- Infinite recursion
- A short prefix hides a longer one
Problem #1: Infinite Name Expansion

• Suppose the namespace contains

```c
mount("a", "this_a", NAMESPACE);
mount("t", "and_t", NAMESPACE);
```

• Consider

```c
open(NAMESPACE, "and_that", "r");
```

• Repeated replacement keeps building a longer and longer name

```c
this_and_that
and_this_and_that
this_and_this_and_that
and_this_and_this_and_that
{and so on...}
```

• Solution: check the size of the expanded name, stop and return (SYSERR) if the size exceeds the maximum name length
Problem #2: Infinite Recursion

- Suppose the namespace contains
  
  mount("cs_", "ece_", NAMESPACE);
  mount("ece_", "cs_", NAMESPACE);

- Consider
  
  open(NAMESPACE, "cs_is_best", "r");

- Repeated substitution goes on forever, alternating between *cs_is_best* and *ece_is_best*

- Solution: limit the number of repeated substitutions to *nnames*
Problem #3: A Short Prefix Hides A Longer One

- Suppose the namespace contains two entries in the following order

  ```
  mount("l", "l", RFILESYS);
  mount("local/", ",", LFILESYS);
  ```

- Consider

  ```
  open(NAMESPACE, "local/x", "r");
  ```

- The first entry prevents the second from ever getting used, so the open always goes to `RFILESYS` even though the second entry appears to direct it to `LFILESYS`

- Solution: order entries in the table with longest prefix first (and prohibit duplicate prefixes)
Summary

- It is possible to build a naming hierarchy separate from the underlying file systems.
- When the naming hierarchy is viewed syntactically, prefixes define each piece of the hierarchy.
- A prefix table that includes replacement can be used to create a fairly general hierarchy.
- In Xinu, the syntactic namespace is implemented as the `NAMESPACE` pseudo-device.
- Opening the `NAMESPACE` device causes a file name to be mapped according to the prefix table and then passed to `open` on a specific file system.
- Using the null string as a prefix creates a default entry in the prefix table that is guaranteed to match any file name.
Module XIV

User Interface
Location Of The User Interface In The Hierarchy
The Two Operating System Interfaces

- Operating systems provide two ways to access services
- An interface for applications
  - Generically called an API (Application Program Interface)
  - Consists of a set of system calls
  - We have seen examples
- An interface for human users
  - Usually interactive
  - Can be a Command Line Interface (CLI) or Graphical User Interface (GUI)
  - Gives the system a “personality”
Characteristics Of User Interfaces

- **GUI**
  - Allows users to launch applications
  - May include *copy-and-paste* and *drag-and-drop* mechanisms
  - Relies on applications to handle most tasks

- **Command Line Interface**
  - Parses textual commands
  - Arguments passed to commands can allow the user to specify an arbitrary level of detail
A Command Interpreter (CLI)

- Software that accepts commands entered by users and performs the specified action
- Two implementations of command interpreters have been used
  - Early systems and some small, embedded systems: the command interpreter is integrated into the operating system
  - Multics/Unix and later systems: the command interpreter consists of an application that is separate from the operating system
A Command Interpreter Built Into OS

• Advantage: because the interpreter understands command syntax and semantics, it can
  – Offer command completion capability
  – Prompt for required arguments
  – Check arguments for correctness
  – Warn users about meaningless or dangerous requests

• Disadvantages
  – A user is limited to exactly the commands the OS provides
  – A user cannot select a non-standard command interpreter
  – Adding new commands is difficult and requires recompiling the OS
Command Interpreter Implemented By An Application

- Introduced by MULTICS; popularized by Unix
- The interpreter only handles basic command syntax
- Individual programs must check and interpret arguments
- Advantage
  - Each user can choose their own interpreter
  - New commands can be added at any time
- Disadvantages
  - Nonuniformity among commands and arguments
  - No built-in semantic checks (argument errors are reported after a command starts running)
Example Of A Separate Interpreter: The Unix Shell

- Runs as a standard application process (no special privilege is required)
- Provides per-line processing
- Interprets each line as a command
- Uses the same syntax for scripts as for interactive input
- Offers basic programming language constructs
  - Variables
  - Sequence of statements
  - Definite and indefinite iteration
  - Conditional execution
Shell Variables

- Have you used shell variables?
- Do you really understand how they work?
- The basics (from Korn shell)

```bash
X="hello"
echo $X
```

produces a line of output containing the word `hello`

- Given the above, the command

```bash
gcc $X.c
```

compiles file `hello.c`
Shell Binding Times

- Now consider a more complex example
- Suppose the current directory contains files
  
  ```
  aaa  bbb  ccc
  ```

- What do the following lines mean if typed into a shell?

```bash
BEES="b*"  # Define variable BEES
ls -l $BEES # This will list file bbb
touch bb   # Add another file that starts with b
ls -l $BEES # Will this line list just the file
             # named bbb, or both bb and bbb?
```
Another Example Of Binding Times

- The shell
  - Has both local a global variables
  - Uses the term *environment* for the set of global variables
- Environment variables
  - Imports variable definitions from the user’s environment
  - Allows a user to *export* specific variables to the environment
  - The environment is passed to each child process that the shell executes
- Note: programs such as *make* allow environment variables to be accessed
Environment Variable Binding Times

- Suppose a user defines an environment variable QQQ

```bash
QQQ=CS354  # Define variable QQQ
export QQQ  # Export QQQ to the environment
myscript    # Run a shell script as a command
echo $QQQ  # Print the value of QQQ
```

- What will the output be if `myscript` contains the following lines?

```bash
echo $QQQ  # Print the current value of QQQ
QQQ=CS503  # Redefine QQQ
```
Environment Variable Binding Times (continued)

- The answer
  - A copy of the environment is kept for each process
  - A process inherits a copy from its parent when the process starts
  - Changes only affect the local copy (and processes that are created)

- In the example, the output is

  CS354
  CS354
The Basic Unix Shell Evaluation Algorithm

A shell repeats the following steps:

A. Read and parse the next command, dividing it into tokens

B. Perform macro substitution: replace $X$ with value of string $X$

C. Perform file name matching (e.g., eliminate “*”)

D. Perform variable assignment ($var=string$)

E. Search the user’s $PATH$ for the command named by first token

F. Invoke the command, passing remaining tokens as arguments
Data Types In The Unix Shell Language

- The shell supports one data type: string
- Builtin commands handle
  - Iteration (*while* and *for*)
  - Conditional execution (*if-then-else*)
- Quotes prevent substitution (delay binding)
  - Single quotes inhibit interpretation within the string
  - Double quotes allow variable substitution within the string
- Each command is executed by a separate process
- A command *pipeline* connects the output from one process to the input of another
Unix Shell Parsing In Practice

- The shell acts like a compiler
- Compound statements (while, for, if-then-else) can span multiple lines of input
- A long pipeline can span multiple lines as well
- The shell must also handle file redirection
- Consequence: a shell must check for balanced delimiters (e.g., if → fi)
Unix Shell Data Conversions

- Output from a command can be converted to a string
  ```
  `command`
  ```
- The contents of file can be assigned to a string
  ```
  `cat file`
  ```
- The contents of a variable can be converted to command input
  ```
  echo $string | command
  ```
- Literal text can be converted to command input
  ```
  command <<!
  ...
literal text goes here
  !
  ```
The Unix Shell: Paths And Command Invocation

- The shell maintains “search path”
  - The path specifies a list of directories
  - The shell uses the path during command lookup
- To find a file to execute, the shell searches the path one directory at a time
  - It prepends the next directory to the command name
  - It checks to see if the result is a file
  - It stops if the file exists
- Once a file has been found, the shell checks to see that the file is executable
- The current directory (denoted “.”) works like any other directory name in a path
Efficiency And Path Binding

- Two forms of binding have been used: late and early
- Late binding (used in the original Borne shell)
  - The shell searches the path each time a user enters a command
  - Is less efficient, but handles new commands easily
- Early binding (introduced in BSD Unix)
  - The shell presearches the path and caches the names of files in each directory
  - Is more efficient, but cannot detect changes in directory contents automatically
  - A user enters a `rehash` command to perform a new binding
**Automatic Rehash**

- A technique that allows a shell to find changes in directories
- The idea:
  - A user types a command
  - The command is not found
  - Before reporting an error, the shell automatically invokes *rehash* and retries the command
  - If the command is not found after the rehash, report an error to the user
Automatic Rehash (continued)

• Does automatic rehash work?
• Answer: it depends
• Consider a case where
  – A user adds an executable program to a directory on the path
  – The program has a unique name (not found in any other directory along the path)
  – Automatic rehash finds the new programs
• Now consider a case where
  – A user adds an executable program to the first directory on the path
  – An executable program with the same name appears in a later directory on the path
  – What happens?
I/O Redirection

• A user can redirect input or output
• The shell provides separate redirection for
  – Standard input
  – Standard output
  – Standard error
• Syntax is
  – Output:  > file
  – Input:    < file
• The syntax for standard error redirection depends on the shell
Synchronization Of Processes

• Background processing
  – The shell always creates a process to execute a command; background execution allows the shell to continue processing concurrently
  – The syntax is “&”

• Pipeline
  – The output from one process is fed to the input of another
  – An arbitrary pipeline is allowed
  – The “pipe” between two processes consists of a finite buffer
  – The operating system handles process synchronization
Shell Script

- The name given to a file that contains a set of shell commands
- The file must be executable
- A shell script uses the same syntax as an interactive shell (earlier operating systems used a special syntax for scripts)
- BSD Unix introduced the use of a two-byte *magic number* consisting of the ASCII characters `#!`
- If a file name follows the magic number, the file is taken to be the program to run with the script as input

```
#!/users/me/bin/my_program
```
Shell Input And A Challenge

- One can invoke a shell script, $X$, by:

  \[ \text{ksh} < X \]

  or by naming $X$ as an argument to the shell:

  \[ \text{ksh} X \]

- Challenge: create a shell script that behaves differently when invoked in the two ways shown above

- Note: feel free to use whatever shell you prefer (e.g., \textit{bash})
Design And Implementation Of An Example Shell
The Xinu Shell

- We will use the Xinu shell to illustrate the basics
- The Xinu shell
  - Is not fancy
  - Has a fixed set of commands compiled into shell itself
  - Uses a familiar command syntax:

    \[ \text{command\_name} \ \text{arg*} \ [\text{redirection}] \ [\text{background}] \]

    where \text{redirection} includes both input (< file) and output (> file), and \text{background} (&) allows a user to run a command in background
  - The notation \text{arg*} means “zero or more args”, and \text{[ x ]} means “x is optional”
  - The syntax and basic approach is similar to Unix
A Warning About Xinu Shell

- The Xinu shell is written ad hoc
- The data structures and algorithms are unusual
- The idea is to
  - Show a minimal implementation
  - Illustrate shell organization without unnecessary complexity
Lexical Tokens In The Xinu Shell

<table>
<thead>
<tr>
<th>Token</th>
<th>Type</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH_TOK_AMPER</td>
<td>0</td>
<td>&amp;</td>
<td>ampersand</td>
</tr>
<tr>
<td>SH_TOK_LESS</td>
<td>1</td>
<td>&lt;</td>
<td>less-than symbol</td>
</tr>
<tr>
<td>SH_TOK_GREATER</td>
<td>2</td>
<td>&gt;</td>
<td>greater-than symbol</td>
</tr>
<tr>
<td>SH_TOK_OTHER</td>
<td>3</td>
<td>‘…’</td>
<td>quoted string (single quotes)</td>
</tr>
<tr>
<td>SH_TOK_OTHER</td>
<td>3</td>
<td>&quot;…&quot;</td>
<td>quoted string (double-quotes)</td>
</tr>
<tr>
<td>SH_TOK_OTHER</td>
<td>3</td>
<td>other</td>
<td>sequence of non-whitespace</td>
</tr>
</tbody>
</table>

• Only six lexical tokens are needed and only four types
• A string that starts with one type of quote can contain the other type of quote

"Don’t blink!"
Organization Of Xinu Shell

- The shell is organized like an on-line compiler (i.e., an interpreter) with two main parts
  - A *lexical analyzer*
    - Divides an input line into a series of tokens
    - Stores each token (the characters that make up the token) along with the type
  - A *parser*
    - Checks to ensure the sequence of tokens is valid
    - Turns the tokens into a command with arguments
    - Executes the command
Lexical Analysis

- Because the Xinu shell only handles one line at a time, the shell
  - Reads an entire line
  - Calls a lexical analyzer to divide the line into tokens
- The lexical analyzer
  - Eliminates whitespace (i.e., blanks and tabs)
  - Checks for invalid tokens (e.g., file< )
  - Returns the number of tokens found
Token Storage

- Is unusual
- Uses two *parallel arrays* plus an array of characters (*tokbuf*)
- Each array has *ntok* entries for an input line with *ntok* tokens
- One array (*toktyp*) tells the type of the token
- Another array (*tok*) gives the index in tokbuf where the token begins
- Note: each token in tokbuf ends with the null character
Token Storage Example

- Given the input line: `date > file &`
- The shell stores the tokens by storing an index and a type in two arrays
Parsing And Execution

- The first token must be the name of a command
- The final tokens may specify
  - Background processing
  - I/O redirection (input or output)
- The remaining tokens are taken to be arguments to the command
Approach To Parsing

• Recall: the lexical analyzer divides the entire line into tokens before the parser runs
• The parser
  – Checks that the first token is a name of a command, and not other tokens, such as >, <, &
  – Works backward from the last tokens to check for
  – Indirection
  – File redirection (i.e., < filename or > filename)
• Uses the remaining tokens, if any, as arguments to the command
Arguments Passed To A Command

- Like Unix, Xinu only passes two arguments to a command
  - An integer count (nargs)
  - An array of pointers to argument strings
- Also like Unix, the first argument is the command name
An Example Of Command Arguments

- Consider the input line `date -f illegal`
- When a process runs the data command, the process receives two arguments
  - An integer count
  - An array of pointers to strings
- Illustration of the arguments
Internal Commands (Executed By The Shell Process)

- Unlike Unix
  - The set of command names is compiled into Xinu
  - The functions that implement commands are compiled into Xinu
- Why compile commands into the OS?
  - Xinu can run a shell even if there is no external storage
  - There is no overhead in running a command (important on slow, embedded processors)
Xinu Command Execution

- Two types are available
- Normal command
  - Is executed by a separate child process (i.e., the shell creates a process to run the command)
  - Can run in background and have I/O redirection
- *Builtin command*
  - Is executed by the shell with a function call
  - Cannot run in background or have I/O redirection
- Unix shells also support *builtin* commands
Examples Of Built-in Commands

- **Exit** — causes the shell to exit
- **Kill** — kill a process
- **Clear** — send the “screen clear” sequence to standard output
Executing A Non-Builtin Command

- The shell
  - Creates a process to run the command
  - Constructs an argument list that will be passed to the command process
  - Redirects input and/or output, if specified
  - Resumes the new process
- If the user did not specify background processing, the shell waits for the command to complete
- Recall than when a process exits, Xinu sends a message to the parent with value equal to the process ID of the exiting process
Implementation Of
The Xinu Shell
/* lexan.c - lexan */

#include <xinu.h>

/*------------------------------------------------------------------------
* lexan - Ad hoc lexical analyzer to divide command line into tokens
*------------------------------------------------------------------------*/

int32 lexan (char *line, /* Input line terminated with */
             int32 len, /* NEWLINE or NULLCH */
             char *tokbuf, /* Length of the input line, */
                     /* including NEWLINE */
             int32 *tlen, /* Buffer into which tokens are */
                       /* stored with a null */
                     /* following each token */
             int32 *tlen, /* Place to store number of */
                       /* chars in tokbuf */
             int32 tok[], /* Array of pointers to the */
                       /* start of each token */
             int32 toktyp[] /* Array that gives the type */
                    /* of each token */
      )
{
}
Lexan.c (Part 2)

```c
char quote; /* Character for quoted string */
uint32 ntok; /* Number of tokens found */
char *p; /* Pointer that walks along the */
/* input line */
int32 tbindex; /* Index into tokbuf */
char ch; /* Next char from input line */

/* Start at the beginning of the line with no tokens */

ntok = 0;
p = line;
tbindex = 0;

/* While not yet at end of line, get next token */
while ( (*p != NULLCH) && (*p != SH_NEWLINE) ) {
    /* If too many tokens, return error */
    if (ntok >= SHELL_MAXTOK) {
        return SYSERR;
    }

    /* Skip whitespace before token */
    while ( (*p == SH_BLANK) || (*p == SH_TAB) ) {
        p++;
    }
```

Copyright © 2022 by Douglas Comer and CRC Press, Inc. All rights reserved.
/ * Stop parsing at end of line (or end of string) */

ch = *p;
if ( (ch==SH_NEWLINE) || (ch==NULLCH) ) {
   *tlen = tbindex;
   return ntok;
}

/* Set next entry in tok array to be an index to the */
/* current location in the token buffer */
tok[ntok] = tbindex; /* the start of the token */

/* Set the token type */
switch (ch) {
   
case SH_AMPER: toktyp[ntok] = SH_TOK_AMPER;
   tokbuf[tbindex++] = ch;
   tokbuf[tbindex++] = NULLCH;
   ntok++; p++; continue;
case SH_LESS:      toktyp[ntok] = SH_TOK_LESS;
                   tokbuf[tbindex++] = ch;
                   tokbuf[tbindex++] = NULLCH;
                   ntok++;
                   p++;
                   continue;

case SH_GREATER:   toktyp[ntok] = SH_TOK_GREATER;
                   tokbuf[tbindex++] = ch;
                   tokbuf[tbindex++] = NULLCH;
                   ntok++;
                   p++;
                   continue;

default:           toktyp[ntok] = SH_TOK_OTHER;
/* Handle quoted string (single or double quote) */

if ( (ch==SH_SQUOTE) || (ch==SH_DQUOTE) ) {
    quote = ch; /* remember opening quote */

    /* Copy quoted string to arg area */
    p++; /* Move past starting quote */

    while ( ((ch=*p++) != quote) && (ch != SH_NEWLINE) && (ch != NULLCH) ) {
        tokbuf[tbindex++] = ch;
    }
    if (ch != quote) { /* string missing end quote */
        return SYSERR;
    }

    /* Finished string - count token and go on */
    tokbuf[tbindex++] = NULLCH; /* terminate token */
    ntok++; /* count string as one token */
    continue; /* go to next token */
}
A Few Shell Declarations (Part 1)

/* excerpt from shell.h */

/* Size constants */

#define SHELL_BUFLEN TY_IBUFLEN+1 /* Length of input buffer */
#define SHELL_MAXTOK 32 /* Maximum tokens per line */
#define SHELL_CMDSTK 8192 /* Size of stack for process that executes command */
#define SHELL_ARGLEN (SHELL_BUFLEN+SHELL_MAXTOK) /* Argument area */
#define SHELL_CMDPRIO 20 /* Process priority for command */

/* Constants used for lexical analysis */

#define SH_NEWLINE '\n' /* New line character */
#define SH_EOF '\04' /* Control-D is EOF */
#define SH_AMPER '&' /* Ampersand character */
#define SH_BLANK ' ' /* Blank character */
#define SH_TAB '\t' /* Tab character */
#define SH_SQUOTE '"' /* Single quote character */
#define SH_DQUOTE '"' /* Double quote character */
#define SH_LESS '<' /* Less-than character */
#define SH_GREATER '>' /* Greater-than character */
A Few Shell Declarations (Part 2)

/* Token types */
#define SH_TOK_AMPER 0    /* Ampersand token */
#define SH_TOK_LESS 1     /* Less-than token */
#define SH_TOK_GREATER 2  /* Greater-than token */
#define SH_TOK_OTHER 3    /* Token other than those */
    /* listed above (e.g., an */
    /* alphanumeric string) */

/* Shell return constants */
#define SHELL_OK 0
#define SHELL_ERROR 1
#define SHELL_EXIT -3

/* Structure of an entry in the table of shell commands */
struct cmdent {
    /* Entry in command table */
    char *cname;
    /* Name of command */
    bool8 cbuiltin;
    /* Is this a builtin command? */
    int32 (*cfunc)(int32,char*[]); /* Function for command */
};

extern uint32 ncmd;
extern const struct cmdent cmdtab[];
Command Arguments (Review)

- Recall that
  - A newly-created Xinu process can be passed arguments
  - The arguments are placed in a pseudo call on the new process stack
  - The create function takes a number of arguments, each of which is one word (integer, pointer, etc)

- Also recall
  - A command process receives two arguments, a count and an array of pointers to strings
A Question

- When creating a process to run a command, the shell must pass two values to create
  - The count of command-line arguments
  - The address of an array of pointers to strings
- Question: where should the strings be stored?
Background Processing And Command Storage

- Background processing complicates argument passing
  - Multiple commands can execute concurrently
  - Each command can have a different set of arguments
  - The shell cannot use a single variable to store the argument strings for all commands
- Important idea: storage for command-line arguments should be released when the process executing the command exits
Storing Command Arguments

- One possibility
  - Modify create
  - Extend the pseudo-call
  - Push command-line arguments on the new process’s stack
- Another possibility
  - Keep create unmodified
  - Place command-line arguments somewhere else
  - Ensure storage for the arguments is freed when the process exits
- Note: only the stack will be freed when a process exits
A Trick Used To Store Command-Line Arguments

- Create a process to run a command
- Before resuming the process, insert the arguments into bottom of the command’s process stack
  - Store the \textit{args} array plus the actual arguments
  - Placing arguments in the stack area means they will be released along with the stack when the process exits
  - A single contiguous array can be used
  - Added advantage: if a process accidentally overwrites its arguments, no other process will be affected
The Xinu Argument Layout At Runtime

- The shell
  - Creates a process to run a command, but does not resume the process
  - Computes the size needed for arguments, and copies them into the bottom of the process’s stack
  - Resumes the process

`args array starts at lowest byte of user’s stack or next multiple of 4 bytes beyond it`
/* addargs.c - addargs */

#include <xinu.h>
#include "shprototypes.h"

/*-----------------------------------------------
* addargs - Add local copy of argv-style arguments to the stack of
* a command process that has been created by the shell
*--------------------------------------------------
*/

status addargs(
    pid32 pid, /* ID of process to use */
    int32 ntok, /* Count of arguments */
    int32 tok[], /* Index of tokens in tokbuf */
    int32 tlen, /* Length of data in tokbuf */
    char *tokbuf, /* Array of null-term. tokens */
    void *dummy /* Dummy argument that was */
    /* used at creation and must */
    /* be replaced by a pointer */
    /* to an argument vector */

    )
{
    intmask mask; /* Saved interrupt mask */
    struct procent *prptr; /* Ptr to process' table entry */
    uint32 alloc; /* Argument location in process */
    /* stack as an integer */

    }
Addargs.c (Part 2)

```c
uint32 *argloc; /* Location in process’s stack */
    /* to place args vector */
char *argstr;  /* Location in process’s stack */
    /* to place arg strings */
uint32 *search; /* pointer that searches for */
    /* dummy argument on stack */
uint32 *aptr;  /* Walks through args array */
int32 i;      /* Index into tok array */

mask = disable();

/* Check argument count and data length */

if ( (ntok <= 0) || (tlen < 0) ) {
    restore(mask);
    return SYSERR;
}

prptr = &proctab[pid];

/* Compute lowest location in the process stack where the */
/* args array will be stored followed by the argument */
/* strings */

aloc = (uint32) (prptr->prstkbase
    - prptr->prstklen + sizeof(uint32));
argloc = (uint32*) ((aloc + 3) & ~0x3); /* round multiple of 4 */
```
/* Compute the first location beyond args array for the strings */
argstr = (char *) (argloc + (ntok+1)); /* +1 for a null ptr */

/* Set each location in the args vector to be the address of */
/* string area plus the offset of this argument */
for (aptr=argloc, i=0; i < ntok; i++) {
    *aptr++ = (uint32) (argstr + tok[i]);
}

/* Add a null pointer to the args array */
*aptr++ = (uint32)NULL;
/* Copy the argument strings from tokbuf into process’s stack */
/* just beyond the args vector */
memcpy(aptr, tokbuf, tlen);

/* Find the second argument in process’s stack */
for (search = (uint32 *)prptr->prstkptr;
     search < (uint32 *)prptr->prstkbase; search++) {
    /* If found, replace with the address of the args vector*/
    if (*search == (uint32)dummy) {
        *search = (uint32)argloc;
        restore(mask);
        return OK;
    }
}

/* Argument value not found on the stack - report an error */
restore(mask);
return SYSERR;
Shell.c (Part 1)

/* shell.c — shell */

#include <xinu.h>
#include <stdio.h>
#include "shprototypes.h"

/************************************************************************
/* Table of Xinu shell commands and the function associated with each */
/************************************************************************
const struct cmdent cmdtab[] = {
    {"argecho", TRUE, xsh_argecho},
    {"arp", FALSE, xsh_arp},
    {"cat", FALSE, xsh_cat},
    {"clear", TRUE, xsh_clear},
    {"date", FALSE, xsh_date},
    {"devdump", FALSE, xsh_devdump},
    {"echo", FALSE, xsh_echo},
    {"exit", TRUE, xsh_exit},
    {"help", FALSE, xsh_help},
    {"ipaddr", FALSE, xsh_ipaddr},
    {"kill", TRUE, xsh_kill},
    {"memdump", FALSE, xsh_memdump},
    {"memstat", FALSE, xsh_memstat},
    {"ping", FALSE, xsh_ping},
    {"ps", FALSE, xsh_ps},
};
uint32 ncmd = sizeof(cmdtab) / sizeof(struct cmdent);

/**********************************************************/
/* shell  - Provide an interactive user interface that executes */
/* commands. Each command begins with a command name, has */
/* a set of optional arguments, has optional input or */
/* output redirection, and an optional specification for */
/* background execution (ampersand). The syntax is: */
/* */
/* command_name [args*] [redirection] [&] */
/* */
/* Redirection is either or both of: */
/* */
/* < input_file */
/* or */
/* > output_file */
/**********************************************************/
Shell.c (Part 3)

```c
process shell (did32 dev /* ID of tty device from which */
    ) /* to accept commands */
{
    char buf[SHELL_BUFSIZE]; /* Input line (large enough for */
        /* one line from a tty device */
    int32 len; /* Length of line read */
    char tokbuf[SHELL_BUFSIZE + SHELL_MAXTOK]; /* Buffer to hold a set of */
        /* Contiguous null-terminated */
        /* Strings of tokens */
    int32 tlen; /* Current length of all data */
        /* in array tokbuf */
    int32 tok[SHELL_MAXTOK]; /* Index of each token in */
        /* array tokbuf */
    int32 toktyp[SHELL_MAXTOK]; /* Type of each token in tokbuf */
    int32 ntok; /* Number of tokens on line */
    pid32 child; /* Process ID of spawned child */
    bool8 backgnd; /* Run command in background? */
    char *outname, *inname; /* Pointers to strings for file */
        /* names that follow > and < */
    did32 stdinput, stdoutput; /* Descriptors for redirected */
        /* input and output */
    int32 i; /* Index into array of tokens */
    int32 j; /* Index into array of commands */
    int32 msg; /* Message from receive() for */
        /* child termination */
```
Shell.c (Part 4)

int32 tmparg; /* Address of this var is used */
/* when first creating child */
/* process, but is replaced */
char *src, *cmp; /* Pointers used during name */
/* comparison */
bool8 diff; /* Was difference found during */
/* comparison */
char *args[SHELL_MAXTOK]; /* Argument vector passed to */
/* builtin commands */

/* Print shell banner and startup message */

fprintf(dev, "\n\n%%s%%s%%s%%s%%s%%s%%s%%s%%s\n", 
SHELL_BAN0,SHELL_BAN1,SHELL_BAN2,SHELL_BAN3,SHELL_BAN4, 
SHELL_BAN5,SHELL_BAN6,SHELL_BAN7,SHELL_BAN8,SHELL_BAN9);

fprintf(dev, "%s\n\n", SHELL_STRTMSG);
Shell.c (Part 5)

/* Continually prompt the user, read input, and execute command */
while (TRUE) {
    /* Display prompt */
    fprintf(dev, SHELL_PROMPT);
    /* Read a command */
    len = read(dev, buf, sizeof(buf));
    /* Exit gracefully on end-of-file */
    if (len == EOF) {
        break;
    }
    /* If line contains only NEWLINE, go to next line */
    if (len <= 1) {
        continue;
    }
    buf[len] = SH_NEWLINE; /* terminate line */
/* Parse input line and divide into tokens */
ntok = lexan(buf, len, tokbuf, &tlen, tok, toktyp);

/* Handle parsing error */
if (ntok == SYSERR) {
    fprintf(dev, "%s\n", SHELL_SYNERRMSG);
    continue;
}

/* If line is empty, go to next input line */
if (ntok == 0) {
    fprintf(dev, "\n");
    continue;
}

/* If last token is '&', set background */
if (toktyp[ntok-1] == SH_TOK_AMPER) {
    ntok--;
    tlen-= 2;
    backgnd = TRUE;
} else {
    backgnd = FALSE;
}
/* Check for input/output redirection (default is none) */

outname = inname = NULL;
if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
    ||(toktyp[ntok-2] == SH_TOK_GREATER))){
    if (toktyp[ntok-1] != SH_TOK_OTHER) {
        fprintf(dev, "%s\n", SHELL_SYNERRMSG);
        continue;
    }
    if (toktyp[ntok-2] == SH_TOK_LESS) {
        inname = &tokbuf[tok[ntok-1]];
    } else {
        outname = &tokbuf[tok[ntok-1]];
    }
    ntok -= 2;
    tlen = tok[ntok];
}

if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
    ||(toktyp[ntok-2] == SH_TOK_GREATER))){
    if (toktyp[ntok-1] != SH_TOK_OTHER) {
        fprintf(dev, "%s\n", SHELL_SYNERRMSG);
        continue;
    }
}
Shell.c (Part 8)

```c
if (toktyp[ntok-2] == SH_TOK_LESS) {
    if (inname != NULL) {
        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
        continue;
    }
    inname = &tokbuf[tok[ntok-1]];
} else {
    if (outname != NULL) {
        fprintf(dev,"%s\n", SHELL_SYNERRMSG);
        continue;
    }
    outname = &tokbuf[tok[ntok-1]];
}
ntok -= 2;
tlen = tok[ntok];

/* Verify remaining tokens are type "other" */
for (i=0; i<ntok; i++) {
    if (toktyp[i] != SH_TOK_OTHER) {
        break;
    }
}
```
Shell.c (Part 9)

```c
if ((ntok == 0) || (i < ntok)) {
    fprintf(dev, SHELL_SYNERRMSG);
    continue;
}

stdin = stdout = dev;

/* Lookup first token in the command table */

for (j = 0; j < ncmd; j++) {
    src = cmdtab[j].cname;
    cmp = tokbuf;
    diff = FALSE;
    while (*src != NULLCH) {
        if (*cmp != *src) {
            diff = TRUE;
            break;
        }
        src++;
        cmp++;
    }
    if (diff || (*cmp != NULLCH)) {
        continue;
    } else {
        break;
    }
}
```
/* Handle command not found */
if (j >= ncmd) {
    fprintf(dev, "command %s not found\n", tokbuf);
    continue;
}

/* Handle built-in command */
if (cmdtab[j].cbuiltin) {
    /* No background or redirect. */
    if (inname != NULL || outname != NULL || backgnd) {
        fprintf(dev, SHELL_BGERRMSG);
        continue;
    } else {
        /* Set up arg vector for call */
        for (i=0; i<ntok; i++) {
            args[i] = &tokbuf[tok[i]];
        }

        /* Call builtin shell function */
        if ((cmdtab[j].cfunc)(ntok, args) == SHELL_EXIT) {
            break;
        }
    }
}
continue;
Shell.c (Part 11)

/* Open files and redirect I/O if specified */

if (inname != NULL) {
    stdin = open(NAMESPACE, inname, "ro");
    if (stdin == SYSERR) {
        fprintf(dev, SHELL_INERRMSG, inname);
        continue;
    }
}

if (outname != NULL) {
    stdout = open(NAMESPACE, outname, "w");
    if (stdout == SYSERR) {
        fprintf(dev, SHELL_OUTERRMSG, outname);
        continue;
    } else {
        control(stdout, F_CTL_TRUNC, 0, 0);
    }
}

/* Spawn child thread for non-built-in commands */

child = create(cmdtab[j].cfunc,
        SHELL_CMDSTK, SHELL_CMDPRIO,
        cmdtab[j].cname, 2, ntok, &tmparg);
Shell.c (Part 12)

/* If creation or argument copy fails, report error */
if ((child == SYSERR) ||
    (addargs(child, ntok, tok, tlen, tokbuf, &tmparg)
     == SYSERR)) {
    fprintf(dev, SHELL_CREATMSG);
    continue;
}

/* Set stdin and stdout in child to redirect I/O */
proctab[child].prdesc[0] = stdin;
proctab[child].prdesc[1] = stdout;

msg = recvclr();
resume(child);
if (! backgnd) {
    msg = receive();
    while (msg != child) {
        msg = receive();
    }
}

Shell.c (Part 13)

/* Terminate the shell process by returning from the top level */

fprintf(dev,SHELL_EXITMSG);
return OK;
}
Before The Next Class

• Read Chapter 26 on the Xinu shell
• Look at
  – The lexical analysis function, *lexan.c*
  – The parser, *shell.c*
  – The function to handle arguments, *addargs.c*
• Think about how to add pipes
• Come back with questions
Mice And Windowing Systems
A Mouse (Or Trackpad)

- A pointing device invented as a companion to a bit-mapped display
- Operates as an I/O device
- The hardware
  - Detects movement
  - Reports motion in 2-dimensions
- The hardware interface is surprising
Mouse Hardware

- A mouse contains
  - Two motion detectors
    * Arranged at right angles
    * Labeled $X$ and $Y$
  - Two A-to-D converters
  - 1 to 3 buttons
  - Scroll wheel(s) or touch controls for scrolling
Communication Between A Mouse And A Computer

- Can use a variety of physical hardware connections
  - Traditional serial port (RS232)
  - Serial communication over USB
  - Serial communication over Bluetooth
- Uses the same interface as a keyboard
  - Individual characters
  - Sends or receives one character at a time
Two-Way Mouse Communication

• Is unexpectedly sophisticated
• A mouse uses two-way interaction
• A mouse can
  – Accept commands from the computer
  – Respond to queries from the computer
  – Transmit data to the computer asynchronously
Mouse Modes

- Two basic modes are used
  - Polling mode
  - Streaming mode
- A given system may support both modes and allow the windowing system and/or individual applications to choose
Polling Mode

- A processor sends a request for information
- The mouse responds by reporting
  - Motion since last request
  - The status of the buttons
- Typically polling is only used
  - By low-end embedded systems
  - To reset after communication has been temporarily lost
Streaming Mode

• A processor
  – Specifies the resolution and scaling to be used for motion detection
  – Sends one request to start the stream
• The mouse transmits new information
  – When movement exceeds a predetermined threshold
  – When a button is pressed or released
  – When the thumbwheel moves
The Mouse Communications Interface

- Communication with a mouse is
  - Asynchronous (can occur at any time)
  - Full duplex (both directions operate independently)
  - Performed with 8-bit bytes, not just printable characters
  - The RS232 standard can be used, just like a keyboard
- Consequence: a mouse may send a report at the same time a computer sends a command to the mouse
The Mouse Communications Interface
(continued)

- Each message occupies multiple 8-bit bytes
- Example: each report sent by a mouse
  - Consists of one message with multiple fields
  - Each field in the message is a fixed length
  - The format is known as a *mouse packet*

- Notes
  - Although we use the term *packet*, the serial hardware only transfers individual bytes
  - The values in fields are interpreted according to current parameter settings
An Example Mouse Packet (2-Button Mouse)

<table>
<thead>
<tr>
<th>YOVF</th>
<th>XOVF</th>
<th>YNEG</th>
<th>XNEG</th>
<th>reserved</th>
<th>BUT₂</th>
<th>BUT₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X motion since last report</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y motion since last report</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Z motion (wheel) since last report</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Reserved bits must be zero
- *XOVF* and *YOVF* report overflow
- *XNEG* and *YNEG* are sign bits
- *BUT₁* and *BUT₂* give button status
Mouse Parameters

- Are set by the computer
- Determine
  - How motion is measured and scaled
  - How the mouse uses thresholds to decide when to send reports
- Users may be able to specify preferences
Typical Mouse Parameters

- Samples per second
  - Selectable from 10, 20, 30,... 200 samples per second
  - The standard is 100 samples per second
- Tracking can be
  - Linear (e.g., 2:1)
  - Non-linear (e.g., exponential)
- Resolution (the precision with which to measure)
  - Example: 4 counts per millimeter
- The point: a seemingly simple device, a mouse, is quite complex
Memory Mapped Display Screens

- A screen is divided into pixels
- Most displays are *memory mapped*, which means
  - A portion of the memory address space is reserved for each display
  - The operating system writes values to the display memory to change pixels on the screen
  - The display hardware repeatedly scans the display memory and updates the screen accordingly
- A modern graphics card keeps the display memory on the card and includes hardware that scans the display memory and updates the display faster than possible with normal DRAM
Display Hardware

• Early screens were black-and-white
  – One bit in the display memory corresponded to one pixel on the screen
  – The early displays were known as *bit-mapped* screens

• Current screens display color
  – One or more bytes of display memory correspond to a pixel on the screen
  – Color displays are sometimes called *byte-mapped*, but the term *bit-mapped* persists
Color Specification

- The hardware uses three primary colors: red, green, blue (RGB)
- Originally, the colors came from phosphors illuminated by an electron beam; modern displays use LEDs to produce the colors
- Think back to elementary school art class
  - Everyone learns that the primary colors are red, blue, and yellow
  - Other colors can be made by mixing primary colors
  - Example: yellow plus blue produces green
- Questions
  - Was the art teacher incorrect?
  - Why don’t computer displays use the same primary colors
Answer:
- Art classes use *reflective colors*
- Computer screens use *generated color*

A major difference
- Artists draw on white paper, so the absence of color is white (less color results in a lighter shade)
- On a computer, the absence of color is black (less color results in a darker shade)
Combining Colors

- On a display, all colors result from a combination of 0% to 100% of red, green, and blue.

- RGB colors add in interesting and unexpected ways:
  - Example: 100% red + 100% green + 0% blue gives yellow.
  - Recall that adding less of a color makes a darker shade (closer to black), so starting with full blue and reducing the amount will make a darker blue.

- In practice:
  - An integer value is used for each color instead of a floating point percentage.
  - Typically, the value for each color occupies one byte and ranges from 0 to 255.
  - A total of 16,777,216 colors are possible.
Extending Colors

- Additional colors can be provided two ways
  - Increase the number of bits (or bytes) per pixel
  - Use *color map* technology

- Color map technology
  - Observe that a given image or set of images do not contain many colors
  - Build mapping hardware that stores an array of multi-byte colors
  - Use an RGB value as an index into the array
  - Change the array when changing to a set of images with new colors
Windowing Systems

- Windows are an operating system abstraction handled by software
- Most window systems allow windows to be
  - Created/destroyed at any time
  - Moved/resized/iconified
- A typical implementation
  - Each window is a rectangular region on screen (but other shapes have been tried)
  - A window must be created before it can be used
  - The OS presents an application with a separate coordinate space for each window, where (0,0) is a corner of the window
Window Display Parameters

- Many details are involved
  - Background/foreground colors
  - A title for the window
  - The location of scroll bars
  - Borders and labels
- The details are controlled by a piece of software known as a *window manager*
- In Unix systems, each user can choose their own window manager
- Allowing users to choose a window manager means each user can see windows displayed in their preferred style, but means users may not all see the same display
Cursor Movement

• The goal
  – A cursor should appear on the screen at all times
  – The cursor on the screen should track the mouse/touchpad movement

• Unfortunately
  – Mouse hardware is not directly linked to video hardware

• Consequence: software must
  – Update the cursor when the mouse moves (i.e., a mouse packet arrives)
  – Map the new position to the correct window so mouse clicks can be forwarded to the process that owns the window
The Cursor Update Algorithm

- When the cursor moves, the window manager must
  - Undo the cursor at old position by repainting the original display values
  - Determine the new position for the cursor, $P$
  - Save video memory at position $P$ for a later “undo” operation
  - Use cursor color to paint cursor at position $P$
Optimizing Cursor Update

- A cursor update is required at each mouse interrupt
- Switching context to a window manager process introduces significant delay
- Delay makes cursor motion jerky
- To avoid long delays and achieve smooth cursor motion, the system must perform cursor update in the lower-half of device driver as part of interrupt processing
- The downside: if the processor becomes overloaded, interrupt processing can be delayed or missed, which means the cursor on the screen may lag mouse movements or some mouse movements may be missed
- Good news: unlike early computers, modern computers have fast, multicore processors that make lost interrupts unlikely
Summary

- Operating system support two styles of user interface
  - Graphical User Interface
  - Command-line interpreter
- The Unix shell
  - Runs as a separate application
  - Provides a miniature programming language
  - Supports concurrency and data pipelining
  - Limits variables to strings
  - Uses quotes to delay binding
  - Provides conversions between strings and command input/output
Summary (continued)

• A mouse
  – Is surprisingly complex
  – Uses a two-way communication system and delivers “packets”

• A computer display
  – Has primary colors red, green, and blue that add to generate all possible colors rather than the red, yellow, and blue primary colors used in art class to reflect color
  – Uses a memory-mapped approach where a processor writes bytes in a display memory and display hardware constantly scans the memory and updates the display
  – For a color display, the hardware display memory stores three values for each pixel that correspond to red, green, and blue
Summary (continued)

- Windowing systems
  - Are a software abstraction
  - Window manager handles details
- Cursor update
  - Is performed by software
  - Is often handled at interrupt time