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 and 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, '
');
}

• 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');
}
Example Of Process Creation In Xinu (Part 2)

```c
/*---------------------------------------------
 * sndB  -  Repeatedly emit 'B' on the console without terminating
 *---------------------------------------------
 */
void sndB(void)
{
    while( 1 )
        putc(CONSOLE, 'B');
}
```
The Difference Between Function Call And Process Creation

- A normal function call
  - Only involves a single computation
  - Executes synchronously (caller waits until the call returns)

- The create system call
  - Starts a new process and returns
  - Both the old process and the new process proceed to run after the call
The Distinction Between A Program And A Process

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

/* ex3.c - main, sndch */

#include <xinu.h>

void sndch(char);

/*------------------------------------------------------------------------
 * main - Example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------*/

void main(void)
{
    resume( create(sndch, 1024, 20, "send A", 1, 'A') );
    resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch - Output a character on a serial device indefinitely
 *------------------------------------------------------------------------*/

void sndch(
    char ch    /* The character to emit continuously */
)
{
    while ( 1 )
    {
        putc(CONSOLE, ch);
    }
}
Storage Allocation When Multiple Processes Execute

- Various memory models exist for concurrent processes
- Each process requires its own storage for
  - A runtime stack of function calls
  - Local variables
  - Copies of arguments passed to functions
- A process *may* have private heap storage as well
Consequence For Programmers

A copy of function arguments and local variables is associated with each process executing a particular function, not with the code in which the variables and arguments are declared.
AN OPERATING SYSTEM
FROM THE INSIDE
Operating System Properties

- An OS contains well-understood subsystems
- An OS must handle dynamic situations (processes come and go)
- Unlike most applications, an OS uses a heuristic approach
  - A heuristic can have corner cases
  - Policies from one subsystem can conflict with policies from others
- Complexity arises from interactions among subsystems, and the side-effects can be
  - Unintended
  - Unanticipated, even by the OS designer
- We will see examples
Building An Operating System

- 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 \textit{key} as well as a process ID, even though the key is not used in a FIFO list.
- Each list has a \textit{head} and \textit{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>61</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NPROC-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>
</tbody>
</table>

- Each row corresponds to a single process.
- Pairs of rows form the head and tail of a list.
- Conceptual boundary.
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
/* queue.h - firstid, firstkey, isempty, lastkey, nonempty */
/* Queue structure declarations, constants, and inline functions */

/* Default # of queue entries: 1 per process plus 2 for ready list plus */
/* 2 for sleep list plus 2 per semaphore */
#ifndef NQENT
#define NQENT (NPROC + 4 + NSEM + NSEM)
#endif
#define EMPTY (-1) /* Null value for qnext or qprev index */
#define MAXKEY 0x7FFFFFFF /* Max key that can be stored in queue */
#define MINKEY 0x80000000 /* Min key that can be stored in queue */

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

extern struct qentry queuetab[];
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, /* ID of process to insert */
    qid16 q /* ID of queue to use */
)
{
    qid16 tail, prev; /* Tail & previous node indexes */
    if (isbadqid(q) || isbadpid(pid)) {
        return SYSERR;
    }
    tail = queuetail(q);
    prev = queuetab[tail].qprev;
    queuetab[pid].qnext = tail; /* Insert just before tail node */
    queuetab[pid].qprev = prev;
    queuetab[prev].qnext = pid;
    queuetab[tail].qprev = pid;
    return pid;
}
/* ________________________________________________________________ *
* dequeue - Remove and return the first process on a list              *
* ________________________________________________________________ *
*/
dep32 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;
    }
}
/* 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); 
}
Accessing An Item In A List (Part 2)

/*------------------------------------------------------------------------
* getlast - Remove a process from end of queue
*------------------------------------------------------------------------
*/

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

- The stack grows downward (toward lower memory addresses)
- The heap grows upward
Run-Time Aspects Of Code Pertinent To An OS
(continued)

• 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
  
  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
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 hold
  – Is called a process table
  – Usually part of OS address space that is not accessible to applications
Information Kept In A Process Table

- For each process, 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
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)
Implementation Of Scheduling

- 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
Xinu Scheduler Code (resched Part 1)

/* resched.c - resched */

#include <xinu.h>

struct defer Defer;

/*------------------------------------------------------------------------
* resched - Reschedule processor to highest priority eligible process
*------------------------------------------------------------------------*/

void resched(void) /* Assumes interrupts are disabled */{

    struct procent *ptold; /* Ptr to table entry for old process */
    struct procent *ptnew; /* Ptr to table entry for new process */

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

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

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

    ptold = &proctab[currpid];
}
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 */
ctxtsw(&ptold->prstkptr, &ptnew->prstkptr);

/* Old process returns here when resumed */

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

```c
/*------------------------------------------------------------------------
* 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;
    }
}
```
/* 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
Illustration Of Transitions Between The Current And Ready States

- Single function \((\text{resched})\) 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 */
               /* save the old process’s SP */
movl %esp,(%eax)  /* Save old process’s SP */
movl 12(%ebp),%eax /* Get location from which to */
               /* restore new process’s SP */
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 */
     /* interrupts */
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)
Code For The Null Process

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

  ```c
  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 $PR_{SUSP}$ to indicate that a process is in the suspended state
State Transitions For Suspension And Resumption

- As the diagram shows, only a current or ready process can be suspended
- Only a suspended process can be resumed
- System calls \textit{suspend} and \textit{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 \text{suspend} is asked to suspend the current process, it
  
  – Finds its entry in the process table, \text{proctab[currpid]}
  
  – Sets the state in its process table entry to \text{PR_SUSP}, indicating that it should be suspended
  
  – Calls \text{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

process₁
load x into register 1
incr register 1
at this point, the operating system switches to process 2
interrupt occurs (context switch)
store register 1 into x

process₂
load x into register 2
incr register 2
store register 2 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

```c
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;
    }
}
The Suspend System Call (Part 2)

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

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

if (prptr->prstate == PR_READY) {
    getitem(pid); /* Remove a ready process */
    /* from the ready list */
    prptr->prstate = PR_SUSP;
}
else {
    prptr->prstate = PR_SUSP; /* Mark the current process */
    resched(); /* suspended and resched. */
}

prio = prptr->prprio;
restore(mask);
return prio;
**Process Resumption**

- The idea: resume execution of previously suspended process
- A detail: `resume` returns the priority of the resumed process
- Method
  - Make the process eligible to use the processor again
  - Re-establish 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 \textit{ready} that makes a process ready
- \textit{Ready} takes a process ID as an argument, and makes the process ready
- The steps are
  - Change the process’s state to \textit{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.
The Resume System Call (Part 1)

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

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

    mask = disable();
    if (isbadpid(pid)) {
        restore(mask);
        return (pri16)SYSERR;
    }
}
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
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`. 
Xinu Code For Message Reception

/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 * receive - Wait for a message and return the message to the caller
 *------------------------------------------------------------------------
 */

umsg32 receive(void)
{
    intmask mask; /* Saved interrupt mask */
    struct procent *prptr; /* Ptr to process’ table entry */
    umsg32 msg; /* Message to return */

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

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

#include <xinu.h>

/*------------------------------------------------------------------------
* send - Pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------*/
systemcall 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[currpid];
    if (prptr->prhasmsg == TRUE) {
        msg = prptr->prmsg; /* Retrieve message */
        prptr->prhasmsg = FALSE; /* Reset message flag */
    } else {
        msg = OK;
    }
    restore(mask);
    return msg;
}
Summary Of Message Passing

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

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

\textit{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)
  - *Count* is an integer
  - *Set* is a set of processes that are waiting on the semaphore
Operations On Semaphores

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

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

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

- A set of processes use a semaphore to guard a shared item
- Initialize: create a mutex semaphore
  \[ \text{sid} = \text{semcreate}(1); \]
- Use: bracket each critical section in the code with calls to \textit{wait} and \textit{signal}
  \[
  \text{wait(sid);}
  \]
  \[
  \ldots \text{critical section to use the shared item...}
  \]
  \[
  \text{signal(sid);}
  \]
- 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
  
  ```
  semidx = semcreate(1);  semidy = 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! → wait(semidy);  
  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
  
  \[
  \text{psem} = \text{semcreate}(\text{buffer-size}); \\
  \text{csem} = \text{semcreate}(0);
  \]

- The producer algorithm

\[
\text{repeat forever} \{ \\
  \text{generate an item to be added to the buffer;} \\
  \text{wait(psem);} \\
  \text{fill_next_buffer_slot;} \\
  \text{signal(csem);} \\
\}
\]
Producer-Consumer Synchronization With Semaphores (continued)

- The consumer algorithm

```c
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
  - resched
  - suspend
  - resume
- SUSPENDED
  - suspend
  - create
- CURRENT
  - resched
  - suspend
- WAITING
  - wait
  - signal
- RECEIVING
  - receive
  - send
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 */
};

extern struct sentry semtab[];

#define isbadsem(s) ((int32)(s) < 0 || (s) >= NSEM)
Implementation Of Wait (Part 1)

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

/*------------------------------------------------------------------------
 * wait - Cause current process to wait on a semaphore
 *------------------------------------------------------------------------*/
systemcall 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;
    }
}
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, \textit{wait} enqueues the calling process at the tail of the queue
- When it chooses a waiting process to run, \textit{signal} selects the process at the head of the queue
- The code for \textit{signal} follows
/* signal.c - signal */
#include <xinu.h>

/*------------------------------------------------------------------------
 * signal - Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
*/
system signal(
    sid32 sem /* ID of semaphore to signal */
)
{
    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;
    }
Implementation Of Signal (Part 2)

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

- Notice how little code is required to signal a semaphore
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)

#include <xinu.h>

#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;
}
/*----------------------------------------* 
* newsem - Allocate an unused semaphore and return its index 
*----------------------------------------* 
*/ 
local sid32 newsem(void) 
{ 
    static sid32 nextsem = 0; /* Next semaphore index to try */ 
    sid32 sem; /* Semaphore ID to return */ 
    int32 i; /* Iterate through # entries */ 
    for (i=0 ; i<NSEM ; i++) 
    { 
        sem = nextsem++; 
        if (nextsem >= NSEM) 
            nextsem = 0; 
        if (semtab[sem].sstate == S_FREE) 
            { 
                semtab[sem].sstate = S_USED; 
                return sem; 
            } 
    } 
    return SYSERR; 
}
Semaphore Deletion

- Wrinkle: one or more processes may be waiting when 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
Xinu Semdelete (Part 1)

/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------*/
* semdelete - Delete a semaphore by releasing its table entry
*------------------------------------------------------------------------*/
syscall semdelete(
    sid32 sem /* ID of semaphore to delete */
)
{
    intmask mask; /* Saved interrupt mask */
    struct sentry *semptr; /* Ptr to 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
  \(ABCABCABC\ldots\)

- 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
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/freeze 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 \((\text{getmem})\)
  - Last free block that is large enough \((\text{getstk})\)
- 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 \(\text{getmem}\) allocated the lowest addresses in the block
  - For \(\text{getstk}\) 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
Xinu Getmem (Part 1)

/* getmem.c - getmem */

#include <xinu.h>

/*-----------------------------------------------
 * getmem  - Allocate heap storage, returning lowest word address
 *-----------------------------------------------
 */

char *getmem(
    uint32 nbytes /* Size of memory requested */
)
{
    intmask mask; /* Saved interrupt mask */
    struct memblk *prev, *curr, *leftover;

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

    nbytes = (uint32) roundmb(nbytes); /* Use memblk multiples */
Xinu Getmem (Part 2)

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

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

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

    else { /* Move to next block */
        prev = curr;
        curr = curr->mnext;
    }
}

restore(mask);
return (char *)SYSERR;
Splitting A Block

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

- Round up the specified size to a multiple of memory blocks (allows the user to specify the same value during deallocation that was used during allocation)

- Walk the free list, using next to point to a block on the free list, and prev to point to the previous block (or memlist)

- Stop when the address of the block being freed lies between prev and next

- Either: insert the block into the list or handle coalescing
Coalescing Blocks

- The term *coalescing* refers to the opposite of splitting
- Coalescing occurs when a block being freed is adjacent to an existing free block
- Technique: instead of adding the new block to the 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
   ----------------------------------------------
*/
syscall freemem(
    char *blkaddr, /* Pointer to memory block */
    uint32 nbytes /* Size of block in bytes */
)
{
  intmask mask; /* Saved interrupt mask */
  struct memblk *next, *prev, *block;
  uint32 top;

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

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

Xinu Freemem (Part 2)

prev = &memlist;            /* Walk along free list */
next = memlist.mnext;
while ((next != NULL) && (next < block)) {
    prev = next;
    next = next->mnext;
}
if (prev == &memlist) {    /* Compute top of previous block*/
    top = (uint32) NULL;
} else {
    top = (uint32) prev + prev->mlength;
}
/* Ensure new block does not overlap previous or next blocks   */
if (((prev != &memlist) && (uint32) block < top)
    || ((next != NULL) && (uint32) block+nbytes>(uint32)next)) {
    restore(mask);
    return SYSERR;
}
memlist.mlength += nbytes;
/* 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;
    curr = curr->mnext;
    while (fits == NULL)
        if (curr) { /* Walk through memory list */
            fitsprev = fits; /* Record block that fits */
            fits = curr;
            if (fits->msize >= nbytes) { /* Record block that fits */
                fitsprev = curr;
            }
            curr = curr->mnext;
        }
    return (char *)fitsprev;
}
while (curr != NULL) {
    /* Scan entire list */
    if (curr->mlength >= nbytes) { /* Record block address */
        /* when request fits */
        fits = curr;
        fitsprev = prev;
        fitsprev->mnext = fits->mnext;
        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 $ctxsw$
- Several details make it difficult
  - $ctxsw$ 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
*------------------------------------------------------------------------*/
system call 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)

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

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

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 \textit{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 \textit{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.

```c
/* 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
/* 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 */
    uint32 i;
    uint32 *a;      /* points to list of args */
    uint32 *saddr;  /* stack address */
    ...
Process Creation On ARM (Part 2)

```c
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;
Process Creation On ARM (Part 4)

```c
/*------------------------------------------------------------------------
* 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 */

    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;
}
```

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

intmask mask; /* Interrupt mask */
struct procent *prptr; /* Pointer to proc. table entry */
int32 i;
uint32 *a; /* Points to list of args */
uint32 *saddr; /* Stack address */

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->prpriority = 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--) /* Machine dependent; copy args */
    *--saddr = *(a--); /* 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
The Xinu low-level memory manager

- Places all free memory on a single list
- Rounds all requests to multiples of \textit{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 \textit{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 `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, `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 \((\text{freebuf})\)
  - 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
/* 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 || 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 = &buptab[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;
/* 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 *bpptr; /* 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      
  *------------------------------------------------------------------------
*/
systemcall freebuf(
        char        *bufaddr /* Address of buffer to return */
)
{
    intmask mask; /* Saved interrupt mask */
    struct bpentry *bp.ptr; /* 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
• 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
Illustration Of Address Translation

- 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
  - Restarts the process that caused the page fault
Researchers Have Studied Many Aspects Of Paging

- Which replacement policies are most effective?
- Which pages from a given address 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

```c
/* 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 pmaxcnt; /* Max messages to be queued */
    int32 ptseg; /* 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
 *-------------------------------------------------------------
 */
systemcall 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("pinit - 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
 *------------------------------------------------------------------------*/
system call 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;
    }


/* 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;
/* 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;
    }
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 `ptreset` and `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
Concurrenty 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 `freebuf` to free the buffer
  - `Freebuf` signals a semaphore, which calls `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;
}
/* 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;
}
Xinu _ptclear (Part 1)

/* ptclear.c - _ptclear */

#include <xinu.h>

/*------------------------------------------------------------------------
* _ptclear - Used by ptdelete and ptreset to clear or reset a port
* (internal function assumes interrupts disabled and
* arguments have been checked for validity)
*------------------------------------------------------------------------*/

void _ptclear(
    struct ptentry *ptptr,  /* Table entry to clear */
    uint16    newstate,    /* New state for port */
    int32    (*dispose)(int32)/* Disposal function to call */
)
{
    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
Module VIII

Device Management
Interrupts, Device Drivers,
Clocks. And Clock Management
Location Of Device Management In The Hierarchy
Ancient History

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

• A device manager is part of an operating system
• The operating system presents applications with a uniform interface to all devices (as much as possible)
• All I/O is interrupt-driven
A Device Manager In An Operating System

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

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

• The processor
  – Starts a device
  – Enables interrupts and continues with other computation

• The device
  – Performs the requested operation
  – Raises an interrupt on the bus

• Processor hardware
  – Checks for interrupts after each instruction is executed, and invokes an interrupt function if an interrupt is pending
  – Has a special instruction used to return from interrupt mode and resume normal processing
Processes And Interrupts

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

![Diagram of interrupt vector organization]
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 *signal*
  − *Signal* calls *resched*
  − *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

application processes

device drivers

device 1
device driver upper-half (device 1)
device driver lower-half (device 1)
device 2
device driver upper-half (device 2)
device driver lower-half (device 2)
device 3
device driver upper-half (device 3)
device driver lower-half (device 3)

... device n
device driver upper-half (device n)
device driver lower-half (device n)

device 1
device 2
device 3
device n
Synchronous Interface I/O

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

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

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

- Rescheduling during interrupt is safe provided
  - Global data structures valid
  - No process explicitly enables interrupts
- Device driver functions
  - Are divided into upper-half and lower-half
  - Can use existing primitives to block and unblock processes
Clocks And
Clock Management
Location Of Clock Management In The Hierarchy
Various Types Of Clock Hardware Exist

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

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

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

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

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

- Assume events for processes A through D will occur 6, 12, 27, and 50 ticks from now
- The delta keys are 6, 6, 15, and 23
Real-time Clock Processing In Xinu

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

- Processes on \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 \textit{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**: Can transition to **CURRENT** via `resched`, and to **SUSPENDED** via `suspend`.
- **CURRENT**: Can transition to **SLEEPING** via `resched`, **WAITING** via `send`, and **RECEIVING** via `signal`.
- **WAITING**: Can transition to **SLEEPING** via `resched`, **RECEIVING** via `send`, and **RECEIVING** via `signal`.
- **RECEIVING**: Can transition to **WAITING** via `receive`.
- **SUSPENDED**: Can transition to **SLEEPING** via `resched`, **RECEIVING** via `send`, and **RECEIVING** via `signal`.
/* sleep.c - sleep sleepms */

#include <xinu.h>

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

/*------------------------------------------------------------------------
 * sleep - Delay the calling process n seconds
 *------------------------------------------------------------------------*/
systemcall 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
Xinu Insertd (Part 1)

/* insertd.c - insertd */

#include <xinu.h>

/*-----------------------------------------------------------------------------
  * insertd - Insert a process in delta list using delay as the key
  *---------------------------------------------------------------------------*/

status insertd( /* Assumes interrupts disabled */
  pid32 pid, /* ID of process to insert */
  qid16 q, /* ID of queue to use */
  int32 key /* Delay from "now" (in ms.) */
)
{

  int32 next; /* Runs through the delta list */
  int32 prev; /* Follows next through the list*/

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

  return OK;
}
Xinu Insertd (Part 2)

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

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

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

return OK;
The Invariant Used During Sleepq Insertion

At any time during the search, both key and queuetab[next].qkey specify a delay relative to the time at which the predecessor of the “next” process awakens.
A Clock Interrupt Handler

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

- When sleeping processes awaken
  - More than one process may awaken at a given time
  - The processes may not have the same priority
  - If the clock interrupt handler starts a process running immediately, a higher priority process may remain on the sleep queue, even if its time has expired

- Solution: *wakeup* awakens *all* processes that have zero time remaining before allowing any of them to run
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
Xinu Recvtime (Part 1)

/* 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)

#include <xinu.h>

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;
Unsleep - Remove A Sleeping Process (Part 1)

/* 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)

```c
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;
}

gtelem(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 tistat; /* 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)

```c
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 tower; /* 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
/* 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 *)0x44031000;
    /* 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();
}
/* clkinit.c - clkinit (BeagleBone Black) */

#include <xinu.h>

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

uint32 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 */ ;
Clock Initialization (Part 2)

/* 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->tclrn |= 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 to a device
  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., \textit{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 \textit{ethread}
  - When a process reads from the CONSOLE, the device manager invokes \textit{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></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>open</td>
<td>read</td>
<td>write</td>
<td></td>
</tr>
<tr>
<td>CONSOLE</td>
<td>&amp;ttyopen</td>
<td>&amp;ttyread</td>
<td>&amp;ttywrite</td>
<td></td>
</tr>
<tr>
<td>SERIAL0</td>
<td>&amp;ionull</td>
<td>&amp;comread</td>
<td>&amp;comwrite</td>
<td></td>
</tr>
<tr>
<td>SERIAL1</td>
<td>&amp;ionull</td>
<td>&amp;comread</td>
<td>&amp;comwrite</td>
<td></td>
</tr>
<tr>
<td>ETHER</td>
<td>&amp;ethopen</td>
<td>&amp;ethread</td>
<td>&amp;ethwrite</td>
<td></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 have 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>
Actions Taken For Character Output

- 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
Tty Driver Complexity  
(continued)

- 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
A Circular Buffer Implemented With An 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 */
#endif
#endif
#define Ntty 1 /* Number of serial tty lines */
#define TY_IMCBREAK 'K' /* Honor echo, etc, no line edit */
#define TY_OMRAW 'R' /* Raw output mode => no edits */
/* 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 { /* Tty line control block */
    char *tyihead; /* Next input char to read */
    char *tyitail; /* Next slot for arriving char */
    char tyibuff[TY_IBUFLEN]; /* Input buffer (holds one line) */
    sid32 tyisem; /* Input semaphore */
    char *tyohead; /* Next output char to xmit */
    char *tyotail; /* Next slot for outgoing char */
    char tyobuff[TY_OBUFLEN]; /* Output buffer */
    sid32 tyosem; /* Output semaphore */
    char *tyehead; /* Next echo char to xmit */
    char *tyetail; /* Next slot to deposit echo ch */
    char tyebuff[TY_EBUFLEN]; /* Echo buffer */
    char tyimode; /* Input mode raw/cbreak/cooked */
    bool8 tyecho; /* Is input echoed? */
    bool8 tyieback; /* Do erasing backspace on echo? */
    bool8 tyevis; /* Echo control chars as ^X ? */
    bool8 tyecrlf; /* Echo CR-LF for newline? */
    bool8 tyicrlf; /* Map \r to \n on input? */
    bool8 tyierase; /* Honor erase character? */
    char tyierasec; /* Primary erase character */
    char tyierasec2; /* Alternate erase character */
    bool8 tyeof; /* Honor EOF character? */
    char tyeofch; /* EOF character (usually ^D) */
    bool8 tyikill; /* Honor line kill character? */
    char 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 ' ' /* Blank */
#define TY_NEWLINE '\n' /* Newline == line feed */
#define TY_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 Ttyputc (Part 1))

/* ttyputc.c - ttyputc */

#include <xinu.h>

/*------------------------------------------------------------------------
* ttyputc - Write one character to a tty device (interrupts disabled)
*------------------------------------------------------------------------
*/
devcall ttyputc(
    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);
    }
}
Xinu Ttyputc (Part 2)

```c
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 ttycbblk *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;
}
/* 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 *tptr; /* 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(tyistr->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);
/* 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;
}
/* 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));
case TC_ECHO:
    typtr->tyiecho = TRUE;
    return (devcall)OK;

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

default:
    return (devcall)SYSERR;
}
/* 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 */
int 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(typtr, csrptr);
    }
    resched_cntl(DEFER_STOP);
    return;
/* Transmitter output FIFO is empty (i.e., ready for more) */

case UART_IIR_THRE:
    ttyhandle_out(ty.ptr, csr.ptr);
    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
Xinu Ttyhandle_in (Part 1)

/* 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;
}
Xinu Ttyhandle_in (Part 3)

/* 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 */
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--;
        erasel(typtr, csrptr);
    }
    return;
}
/* 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;
}
/* 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)

```c
/* Wrap around if needed */

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

```c
/*------------------------------------------------------------------------
* 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;
Xinu Ttyhandle_in (Part 10)

```c
/**-----------------------------------------------
 * echoch - Echo a character with visual and output crlf options
 *-----------------------------------------------
 */

local void echoch(
    char ch, /* Character to echo */
    struct ttycblk *tptr, /* Ptr to ttytab entry */
    struct uart_csreg *csrptr /* Address of UART’s CSRs */
){
    if ((ch==TY_NEWLINE || ch==TY_RETURN) && tptr->tyecrlf) {
        eputc(TY_RETURN, tptr, csrptr);
        eputc(TY_NEWLINE, tptr, csrptr);
    } else if ( (ch<TY_BLANK||ch==0177) && tptr->tyevis) {
        eputc(TY_UPARROW, tptr, csrptr); /* print ^x */
        eputc(ch+0100, tptr, csrptr); /* Make it printable */
    } else {
        eputc(ch, tptr, csrptr);
    }
}
```
/*---------------------------------------------* - 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])) {
        typtr->tyetail = typtr->tyebuff;
    }

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
/* 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;
}
/* 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 ttycblk *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;
    }
/* If echo and output queues empty, turn off interrupts */
if ( (typtr->tyehead == typtr->tyetail) &&
    (semcount(typtr->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--;
}
/* While onboard FIFO is not full and the output queue is nonempty, transmit chars from the output queue */

ochars = 0;
avail = TY_OBUFLEN - semcount(typt->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(typt->tyosem, ochars);
}
return;
/* 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 = &typtr->tyebuff[0]; /* Set up echo queue */
    /* as empty */
typtr->tyimode = TY_IMCOOKED; /* Start in cooked mode */
typtr->tyiecho = TRUE; /* Echo console input */
typtr->tyieback = TRUE; /* Honor erasing bksp */
typtr->tyevis = 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->tyierasec = TY_BACKSP; /* Primary erase char */
typtr->tyierasec2 = 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_event( devptr->dirq, (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
  - The interface that processes use to send over the Internet
  - The process model
  - 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 `arpacket` 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_iplen; /* IP total packet length */
    uint16 net_ipid; /* IP datagram ID */
    uint16 net_ipfrag; /* IP flags & fragment offset */
    byte net_ipttl; /* IP time-to-live */
    byte net_ipproto; /* IP protocol (actually type) */
    uint16 net_ipcksum; /* IP checksum */
    uint32 net_ipsrc; /* IP source address */
    uint32 net_ipdst; /* IP destination address */
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 {
    uint32 ipucast;
    uint32 ipbcast;
    uint32 ipmask;
    uint32 ipprefix;
    uint32 iprouter;
    uint32 bootserver;
    bool8  ipvalid;
    byte   ethucast[ETH_ADDR_LEN];
    byte   ethbcast[ETH_ADDR_LEN];
    char   bootfile[NETBOOTFILE];
};

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

- 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
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 int he 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 an 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 a semaphore is signaled to allow a waiting process (if any) to become ready and read the message
- If no match is found in the table, 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 **TIMEOUT**

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

- The network code handles the timeout
Implementation Of Timeout

- The network code 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
  - Contacts 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., subtract `clkt ime` 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, `gettime` merely adds `clkt ime` 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 `TIMESERVER`
- The local and remote protocol port numbers to use are given by constants
  - `TIMELPORT` specifies a local UDP protocol port number to use
  - `TIMERPORT` specifies the standard protocol port number an NTP server uses
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 (dot2ip(TIMESERVER, &NetData.ntpserver) == SYSERR) {
        return SYSERR;
    }
}
/* 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
Use Of DHCP At Startup

- 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
Delayed Use Of DHCP

- An interesting process coordination problem arises with DHCP
  - When first starting the operating system, the network processes are not yet running
  - Consequence: the startup code cannot use DHCP
- Our solution: delay using DHCP until an application needs to use the Internet
  - Start the network processes during system initialization, but do not attempt to use DHCP
  - The first time an application calls `getlocalip` to obtain the local IP address, the code sends a DHCP request to obtain the address and then stores it locally
  - Successive calls to `getlocalip` find the stored value
- In essence, DHCP runs as a side effect of requesting the local IP address
The Process Model
For Network Code
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.
  - 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 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 for an ARP reply, 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 for a reply
  - Sends 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
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)