Module I

Course Overview
And
Introduction To Operating Systems

COURSE MOTIVATION AND SCOPE

Scope

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

What We Will Cover

- Operating system fundamentals
- Functionality an operating system offers
- Major system components
- Interdependencies and system structure
- The key relationships between operating system abstractions and the underlying hardware (especially processes and interrupts)
- A few implementation details and examples

What You Will Learn

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

What We Will NOT Cover

- A comparison of large commercial and open source operating systems
- A description of features or instructions on how to use a particular commercial system
- A survey of research systems and alternative approaches that have been studied
- A set of techniques for building operating systems on unusual hardware

How Operating Systems Changed Programming

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

Why Operating Systems Are Difficult To Build

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

An Observation About Efficiency

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

The Once And Future Hot Topic

- In the 1970s and early 1980s, operating systems was one of the hottest topics in CS
- By the mid-1990s, OS research had stagnated
- Now things have heated up again, and new operating systems are being designed for
  - Smart phones
  - Multicore systems
  - Data centers
  - Large and small embedded devices (the Internet of Things)

XINU AND THE LAB

Motivation For Studying A Real Operating System

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

Can We Study Commercial Systems?

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

An Alternative: Xinu

- Small — can be read and understood in a semester
- Complete — includes all the major components
- Elegant — provides an excellent example of clean design
- Powerful — has dynamic process creation, dynamic memory management, flexible I/O, and basic Internet protocols
- Practical — has been used in real products

The Xinu Lab

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

How The Xinu Lab Works

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

REQUIRED BACKGROUND AND PREREQUISITES

Background Needed

- A few concepts from earlier courses
  - I/O: you should know the difference between standard library functions (e.g., *fopen*, *putc*, *getc*, *fread*, *fwrite*) and system calls (e.g., *open*, *close*, *read*, *write*)
  - File systems and hierarchical directories
  - Symbolic and hard links
  - File modes and protection
- Concurrent programming experience: you should have written a program that uses *fork* or *threads*

Background Needed
(continued)

- An understanding of runtime storage components
  - Segments (text, data, and bss) and their layout
  - Runtime stack used for function call; argument passing
  - Basic heap storage management (malloc and free)
- C programming
  - At least one nontrivial program
  - Comfortable with low-level constructs (e.g., bit manipulation and pointers)

Background Needed
(continued)

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

Course Syllabus

See the handout

or

download from blackboard

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

Questions?

A FEW THINGS
TO THINK ABOUT

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

– Antoine de Saint-Exupery

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)

Introduction To Operating Systems (Definitions And Functionality)

What Is An Operating System?

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

Example Services An OS Supplies

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

What An Operating System Is NOT

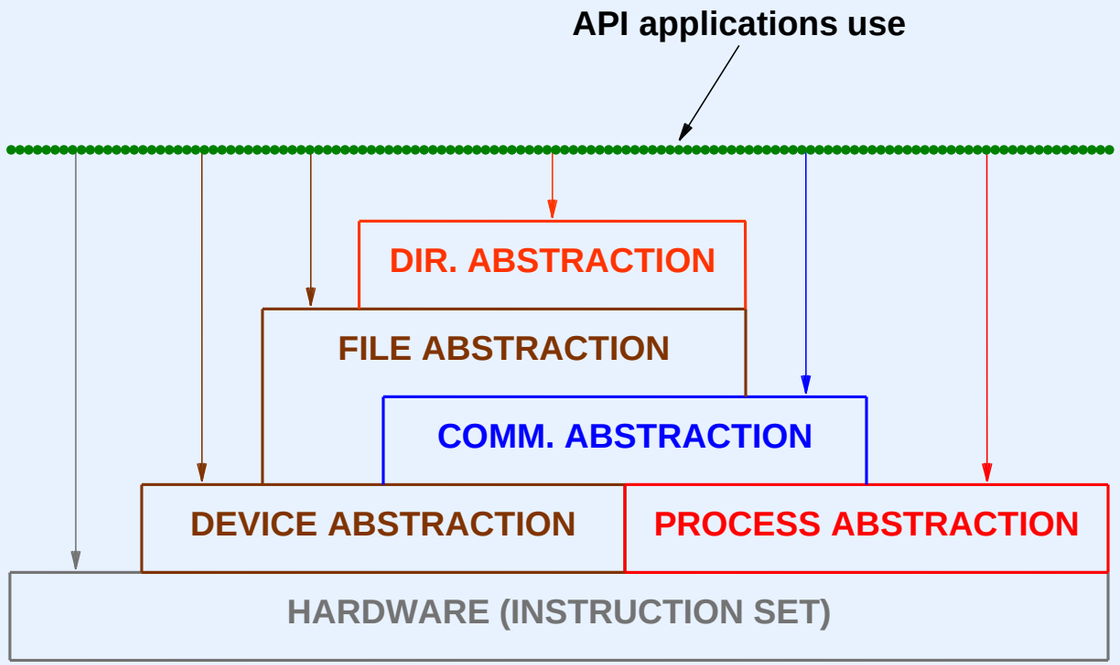
- A hardware mechanism
- A programming language
- A compiler
- A windowing system or a browser
- A command interpreter
- A library of utility functions
- A graphical desktop

AN OPERATING SYSTEM FROM THE OUTSIDE

The System Interface

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

OS Abstractions And The Application Interface



- Modules in the OS offer services to applications
- Internally, some services build on others

Interface To System Services

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

Interface To System Services
(continued)

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

An Example Of System Call In Xinu: Write A Character On The Console

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

#include <xinu.h>

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

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

OS Services And System Calls

- Each OS service accessed through system call interface
- Most services employ a set of several system calls
- Examples
  - Process management service includes functions to *suspend* and then *resume* a process
  - *Socket API* used for Internet communication includes many functions

System Calls Used With I/O

- Open-close-read-write paradigm
- Application
  - Uses *open* to connect to a file or device
  - Calls functions to *write* data or *read* data
  - Calls *close* to terminate use
- Internally, the set of I/O functions coordinate
  - *Open* returns a descriptor, *d*
  - *Read* and *write* operate on descriptor *d*

Concurrent Processing

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

How Multitasking Works

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

Terminology

- A *program* consists of static code and data
- A *function* is a unit of application program code
- A *process* (also called a *thread of execution*) is an active computation (i.e., the execution or “running” of a program)

A Process

- Is an OS abstraction
- Can be created when needed (an OS system call allows a running process to create a new process)
- Is managed entirely by the OS and is unknown to the hardware
- Operates concurrently with other processes

Example Of Process Creation In Xinu (Part 1)

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

#include <xinu.h>

void    sndA(void), sndB(void);

/*------------------------------------------------------------------------
 * main  -  Example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndA, 1024, 20, "process 1", 0) );
        resume( create(sndB, 1024, 20, "process 2", 0) );
}

/*------------------------------------------------------------------------
 * sndA  -  Repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}
```

Example Of Process Creation In Xinu (Part 2)

```
/*------------------------------------------------------------------------
 * sndB  -  Repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

The Difference Between Function Call And Process Creation

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

The Distinction Between A Program And A Process

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

Example Of Two Processes Running The Same Code

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

#include <xinu.h>

void    sndch(char);

/*------------------------------------------------------------------------
 * main  -  Example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 1024, 20, "send A", 1, 'A') );
        resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  -  Output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(
          char  ch              /* The character to emit continuously   */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

Storage Allocation When Multiple Processes Execute

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

Consequence For Programmers

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

AN OPERATING SYSTEM FROM THE INSIDE

Operating System Properties

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

Building An Operating System

- The intellectual challenge comes from the design of a “system” rather than from the design of individual pieces
- Structured design is needed
- It can be difficult to understand the consequences of individual choices
- We will study a hierarchical microkernel design that helps control complexity and provides a unifying architecture

Major OS Components

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

Our Multilevel Structure

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

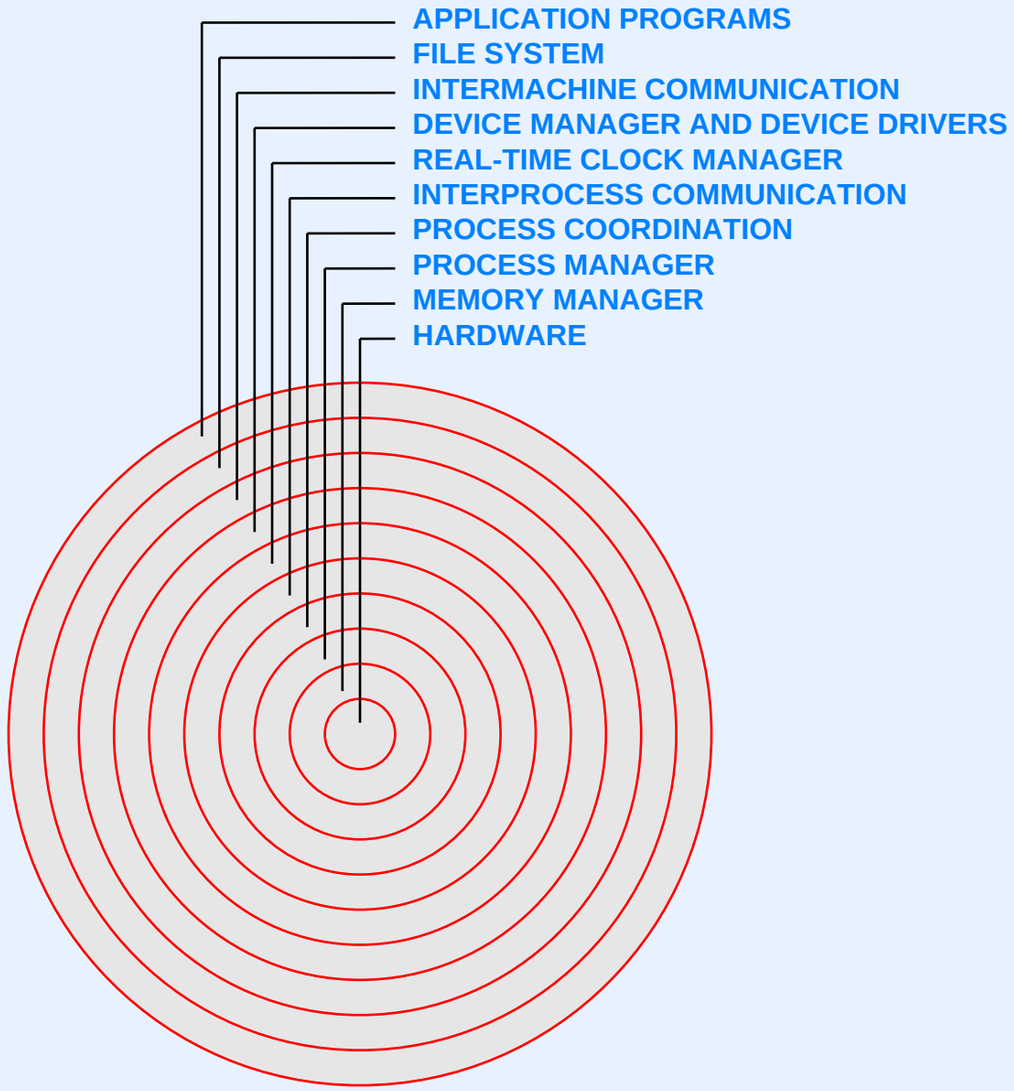
Multilevel Vs. Multilayered Organization

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

Multilevel Vs. Multilayered Organization (continued)

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

Multilevel Structure Of Xinu



How To Understand An OS

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

A Design Example

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

LISTS OF PROCESSES

Queues And Lists

- Keeping track of processes is fundamental throughout an operating system
- Various forms are needed
  - FIFO queues of processes
  - Lists of processes kept in priority order
  - Event lists ordered by the time of occurrence
- Operations required
  - Insert a process onto a list
  - Extract the “next” process from a list
  - Delete an arbitrary process

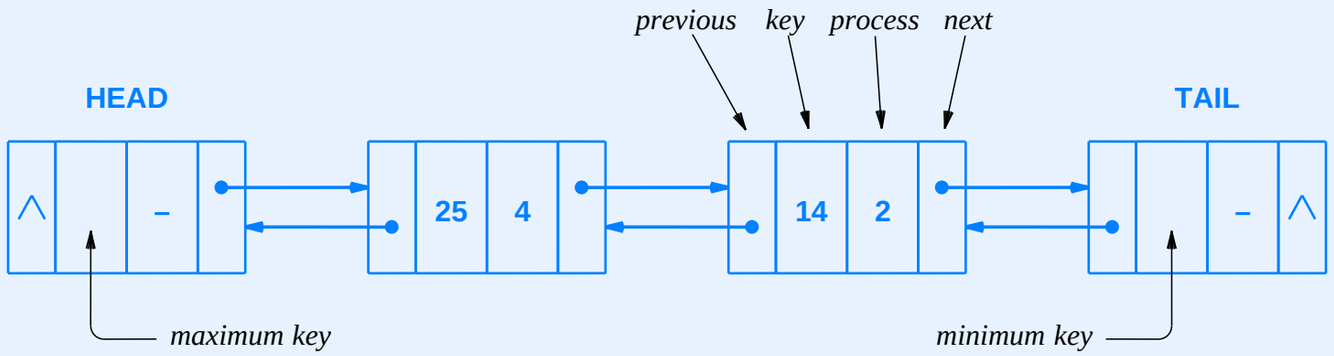
Lists And Queues In Xinu

- Important ideas
  - A process is known by an integer *process ID*
  - A list of processes really stores a set of process IDs
- A single data structure can be used to store many types of process lists

Unified List Storage in Xinu

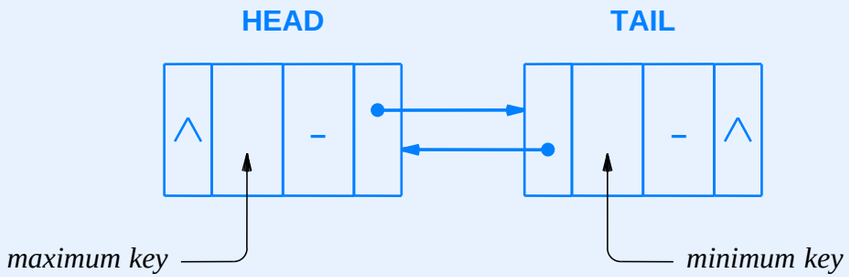
- All lists are doubly-linked, which means a node points to its predecessor and successor
- Each node stores a *key* as well as a process ID, even though the key is not used in a FIFO list
- Each list has a *head* and *tail*; the head and tail nodes have the same shape as other nodes
- Non-FIFO lists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

Conceptual List Structure



- 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

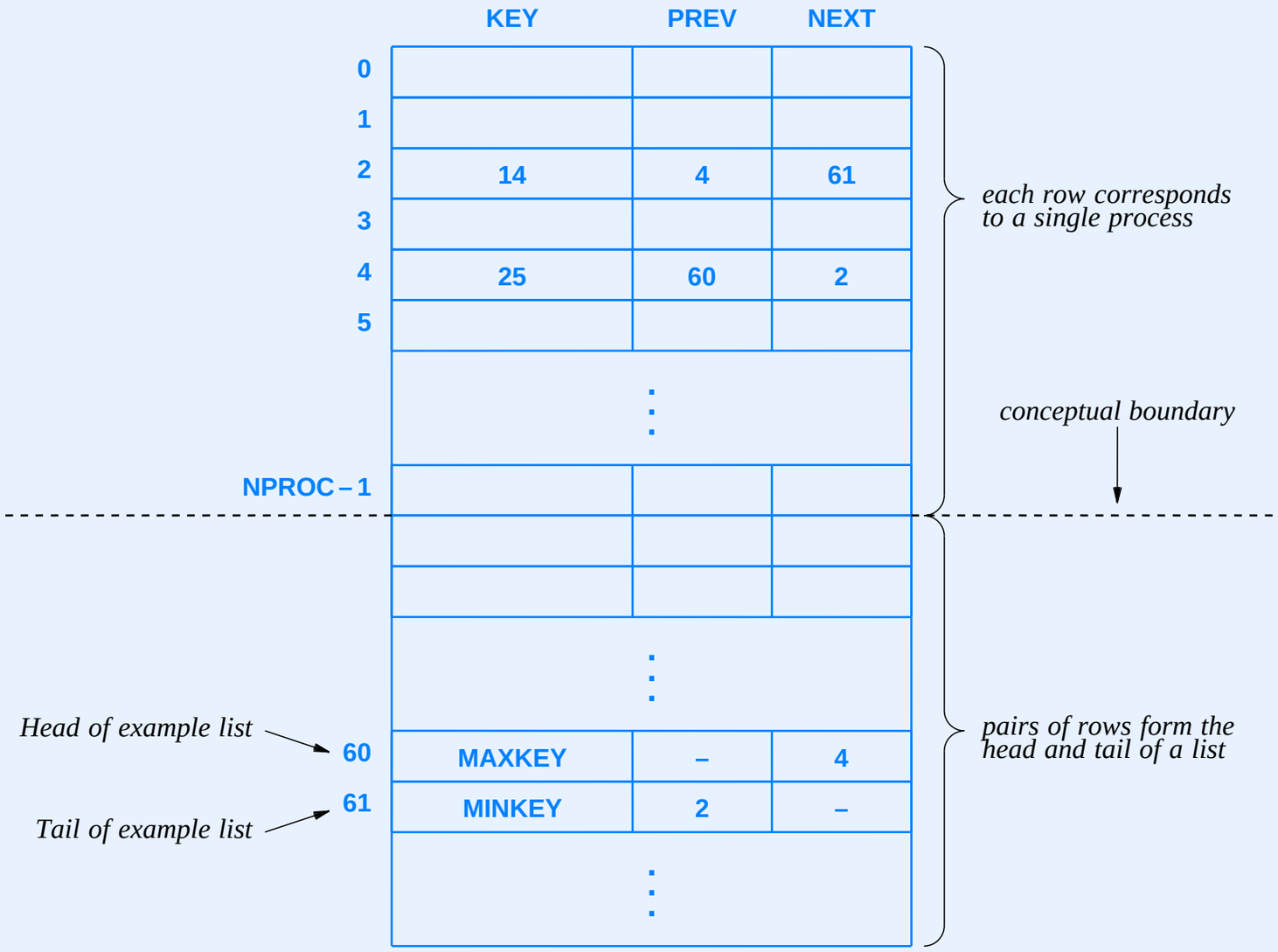
Reducing The List Size

- 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 is one node
  - 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



Implementation

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

A Question About Types In C

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

Type Names Used In Xinu

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

Definitions From queue.h (Part 1)

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

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

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

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

struct  qentry  {               /* 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 isempty(q)      (firstid(q) >= NPROC)
#define nonempty(q)     (firstid(q) <  NPROC)
#define firstkey(q)     (queuetab[firstid(q)].qkey)
#define lastkey(q)      (queuetab[ lastid(q)].qkey)

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

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

Code For Insertion And Deletion From A 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;
}
```

Code For Insertion And Deletion From A Queue (Part 2)

```
/*------------------------------------------------------------------------
 *  dequeue  -  Remove and return the first process on a list
 *------------------------------------------------------------------------
 */
pid32   dequeue(
          qid16         q               /* ID queue to use              */
        )
{
        pid32   pid;                    /* ID of process removed        */

        if (isbadqid(q)) {
                return SYSERR;
        } else if (isempty(q)) {
                return EMPTY;
        }

        pid = getfirst(q);
        queuetab[pid].qprev = EMPTY;
        queuetab[pid].qnext = EMPTY;
        return pid;
}
```

Code For Insertion In An Ordered List (Part 1)

```
/* 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;
        }
```

Code For Insertion In An Ordered List (Part 2)

```
        /* Insert process between curr node and previous node */

        prev = queuetab[curr].qprev;    /* Get index of previous node   */
        queuetab[pid].qnext = curr;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[curr].qprev = pid;
        return OK;
}
```

Accessing An Item In A List (Part 1)

```
/* 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(queuetab[tail].qprev);
}
```

Accessing An Item In A List (Part 3)

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

Questions?