CSCI 8530
Advanced Operating Systems

Part 2

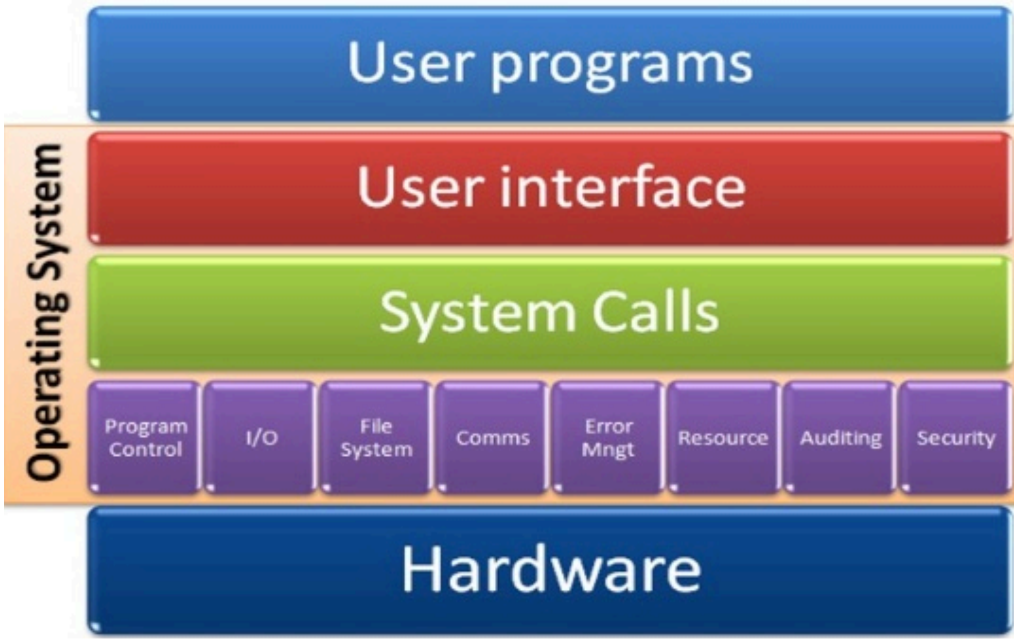
Organization of an Operating System

Review: What Is An Operating System?

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

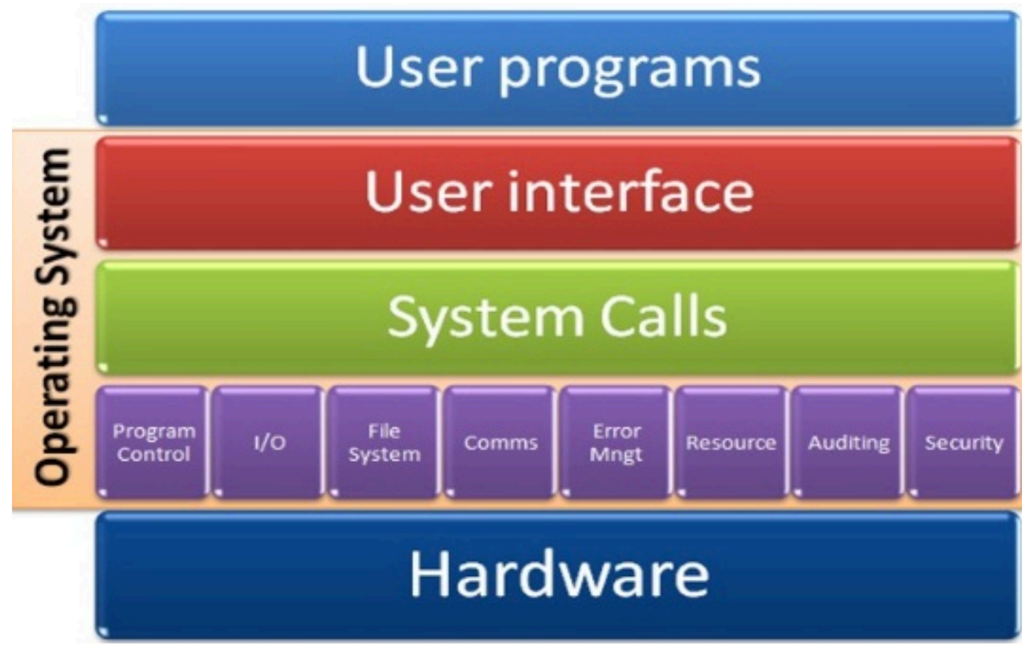
Review: Example Services an OS Supplies

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



Review: What an Operating System is NOT

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

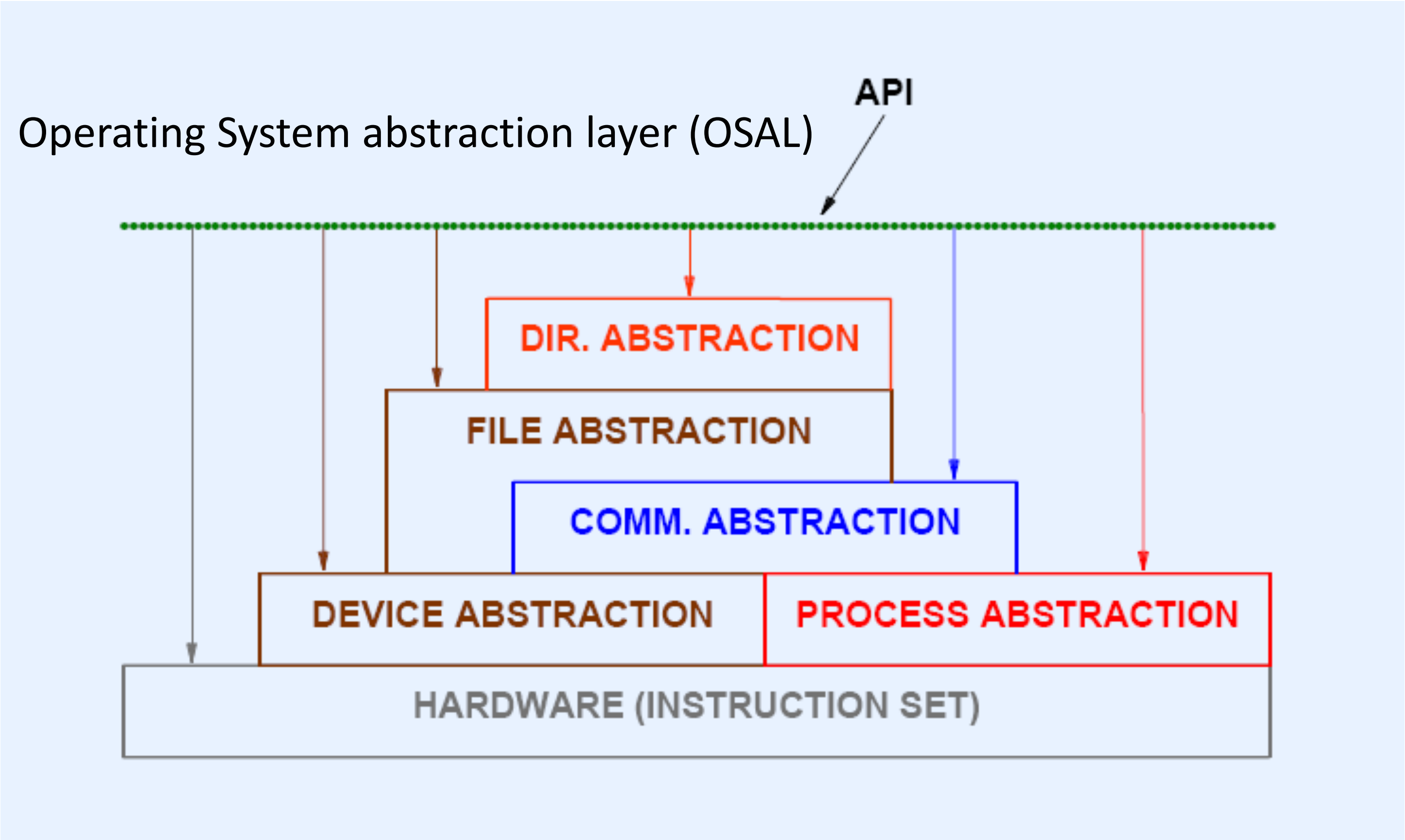


AN OPERATING SYSTEM FROM THE OUTSIDE

The System Interface

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

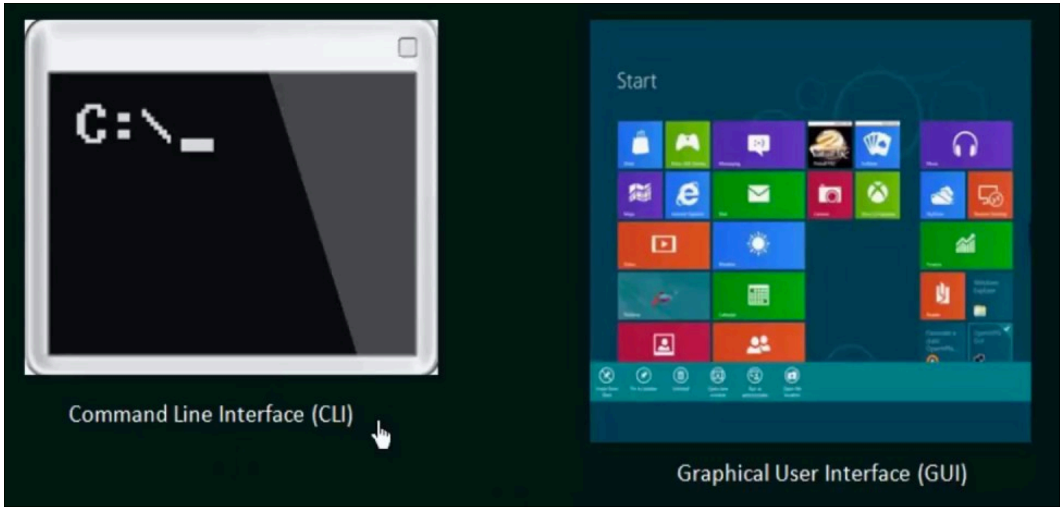
OS Abstractions and Application Interface



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

Interface to System Services

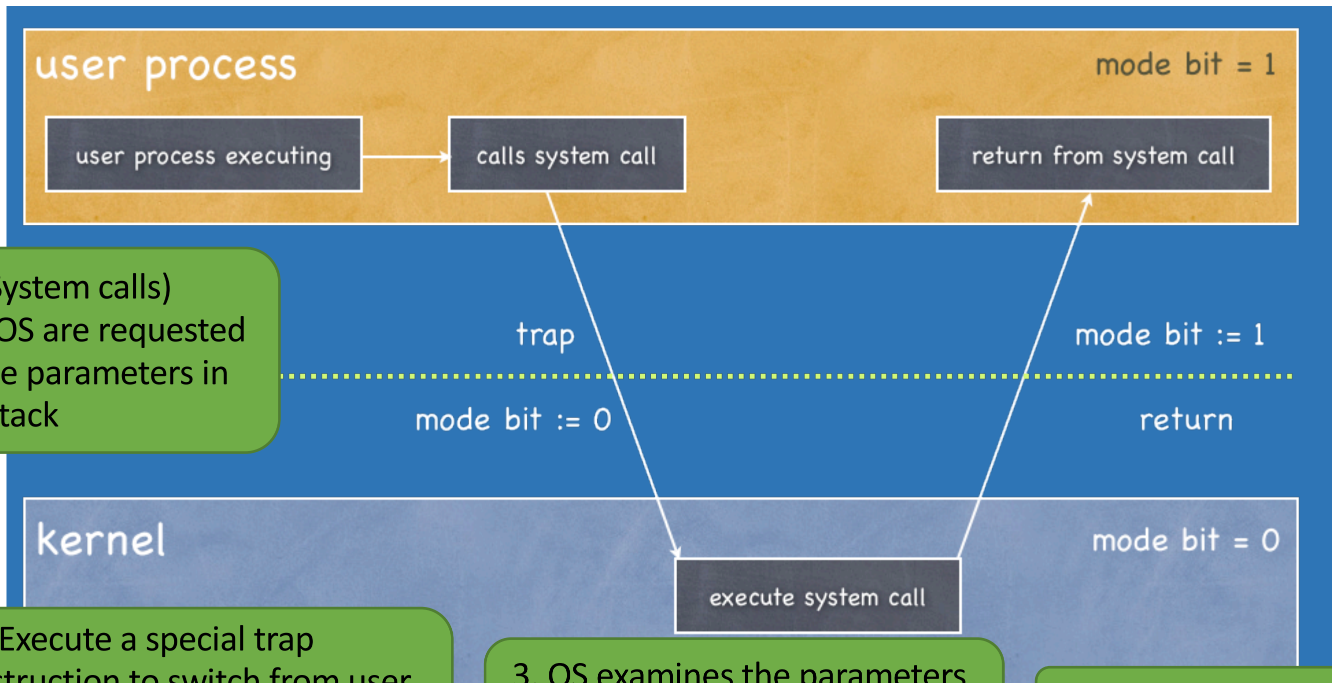
- Appears to operate like a function call mechanism
  - OS makes set of "functions" available to applications
  - Application supplies arguments using standard mechanism
  - Application "calls" one of the OS functions
- Control transfers to OS code that implements the function
- Control returns to caller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



Interface to System Services (continued)

- Requires special instruction to invoke OS function
  - Moves from application *address space* to OS
  - Changes from application *mode* or *privilege level* to OS
- Terminology used by various vendors
  - *System call*
  - *Trap/exception*
  - *Supervisor call*
- We will use the generic term *system call*
  - *A system call*
    - The programmatic way in which a computer program requests a service from the kernel of OS it is executed on
    - This way for programs to interact with OS

System Calls: A Closer Look



1. Services (System calls) provided by OS are requested by putting the parameters in registers or stack

2. Execute a special trap instruction to switch from user mode to kernel mode and transfer the control to OS

3. OS examines the parameters of the call to determine which system call to be carried out.

4. OS invoke the system call

5. System call finished, and control is given back to the user program

Example 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');
}
```

OS Services and System Calls

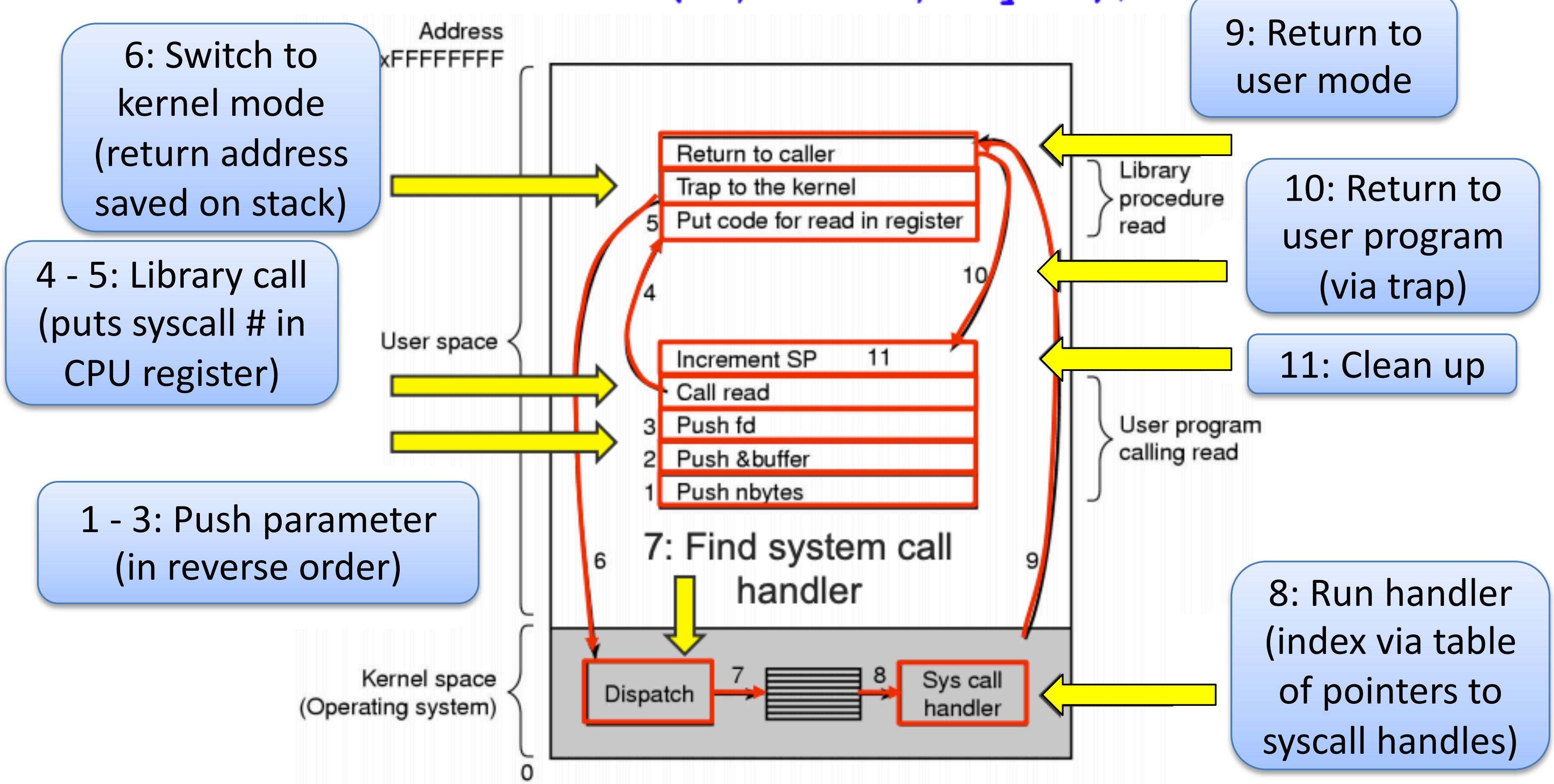
- OS provides services accessed through system call interface
- Most services employ a set of several system calls
- Examples
  - Process management service includes functions: *suspend* and then *resume* a process
  - Socket API used for Internet communication includes many functions: establish a socket on the *client* side
    - Create a socket with the *socket()* system call
    - Connect the socket to the address of the server using the *connect()* system call
    - Send and receive data (used with I/O)

System Calls Used With I/O

- An I/O subsystem comprises of I/O devices and their corresponding driver software
  - Drivers hide the peculiarities of specific hardware devices from the users.
- 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
    - *Read* and *write* operate on descriptor

Steps for Making a System Call (Example: read call)

```
count = read(fd, buffer, nbytes);
```



Implementing a System Call

- System calls are often implemented using traps
  - OS gains control through trap
  - Switches to supervisor mode
  - Performs the service
  - Switches back to user mode
  - Gives control back to user

```
movl $1, %eax
int $0x80
```

Which call?

1: exit
2: fork
3: read
4: write
5: open
6: close
…

Trap to the OS

System-call specific arguments are put in registers

Major System Calls (1/2)

Process Management	
`pid = fork( )`	Create a child process identical to the parent
`pid = waitpid(pid, &statloc, options)`	Wait for a child to terminate
`s = execve(name, argv, environp)`	Replace a process' core image
`exit(status)`	Terminate process execution and return status

File Management	
`fd = open(file, how, ...)`	Open a file for reading, writing or both
`s = close(fd)`	Close an open file
`n = read(fd, buffer, nbytes)`	Read data from a file into a buffer
`n = write(fd, buffer, nbytes)`	Write data from a buffer into a file
`position = lseek(fd, offset, whence)`	Move the file pointer
`s = stat(name, &buf)`	Get a file's status information

Major System Calls (2/2)

Directory and File System Management	
`s = mkdir(name, mode)`	Create a new directory
`s = rmdir(name)`	Remove an empty directory
`s = link(name, name)`	Create a new entry, name, pointing to name
`s = unlink(name)`	Remove a directory entry
`s = mount(special, name, flag)`	Mount a file system
`s = umount(special)`	Unmount a file system

Miscellaneous	
`s = chdir(dirname)`	Change the working directory
`s = chmod(name, mode)`	Change a file's protection bits
`s = kill(pid, signal)`	Send a signal to a process
`seconds = time(&seconds)`	Get the elapsed time since January 1, 1970

Question

- What kinds of system calls does an OS need?
  - Process creation and management
  - Main memory management
  - File Access, Directory and File system management
  - Device handling(I/O)
  - Protection
  - Networking, etc.

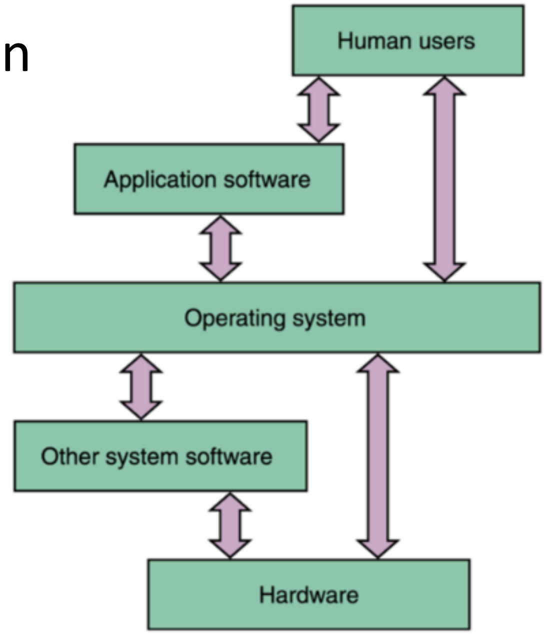
Question

Why does the OS control I/O?

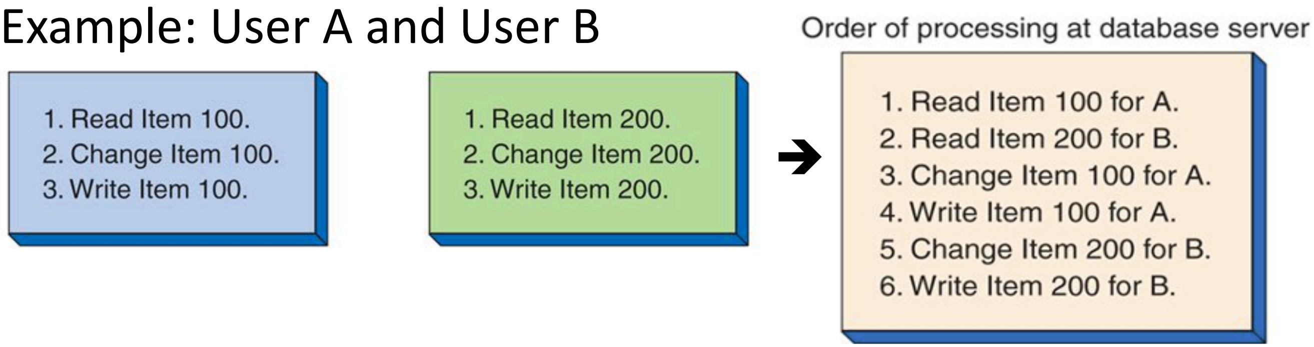
- Safety
  - The computer must ensure that if a program has a bug in it, then it doesn't crash or mess up
    - The system
    - Other programs that may be running at the same time or later
- Fairness
  - Make sure other programs have a fair use of device

Concurrent Processing

- Fundamental concept dominates OS design
- *Real concurrency* achieved by hardware
  - I/O devices operate at same time as processor
  - Multiple processors/cores each operate at the same time
- *Apparent concurrency* achieved with multitasking (multiprogramming)
  - Multiple programs appear to operate simultaneously
  - OS provides the illusion
  - Example: User A and User B

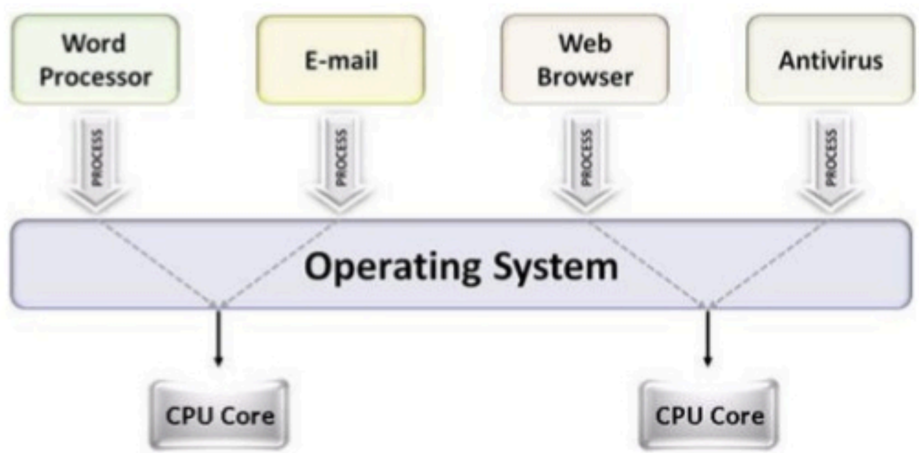


Fundamental concept



Multitasking

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

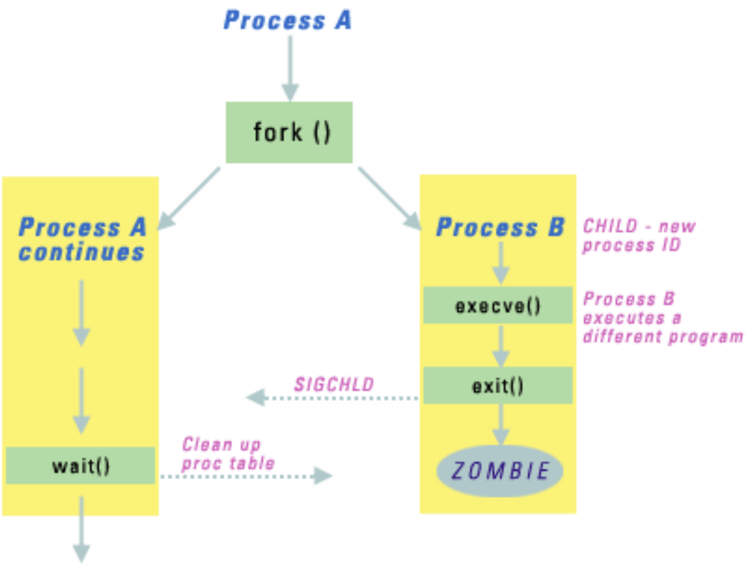


Terminology Used with Multitasking

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

Process

- A process is an instance of a running program
- Managed entirely by OS; unknown to hardware
- Process provides each program with two key OS abstractions
  - **Logical control flow**
    - Each program seems to have exclusive use of the CPU
  - **Private virtual address space**
    - Each program seems to have exclusive use of main memory
- How are these illusions maintained?
  - Process executions interleaved (multitasking) or run on separate cores
  - Address spaces managed by virtual memory system
- Operates *concurrently* with other processes



Example of Process Creation in Xinu

```
/* 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');
}
```

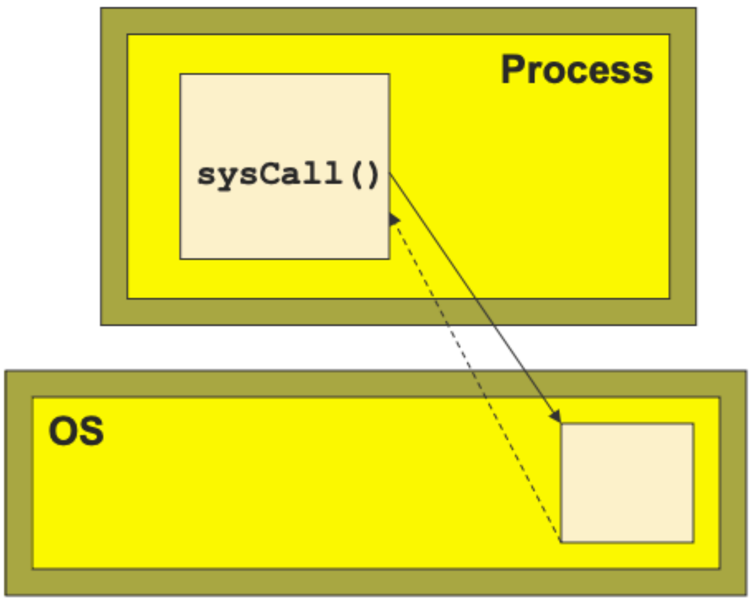
Example of Process Creation in Xinu

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

Question

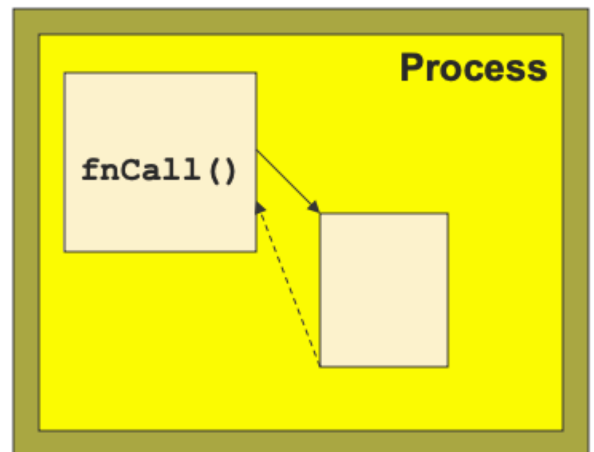
- System Calls V.S. Function Calls?

System Calls



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

Function Calls



Caller and callee are in the same Process
- Same user
- Same “domain of trust”

Xinu Difference Between Function Call and Process Creation

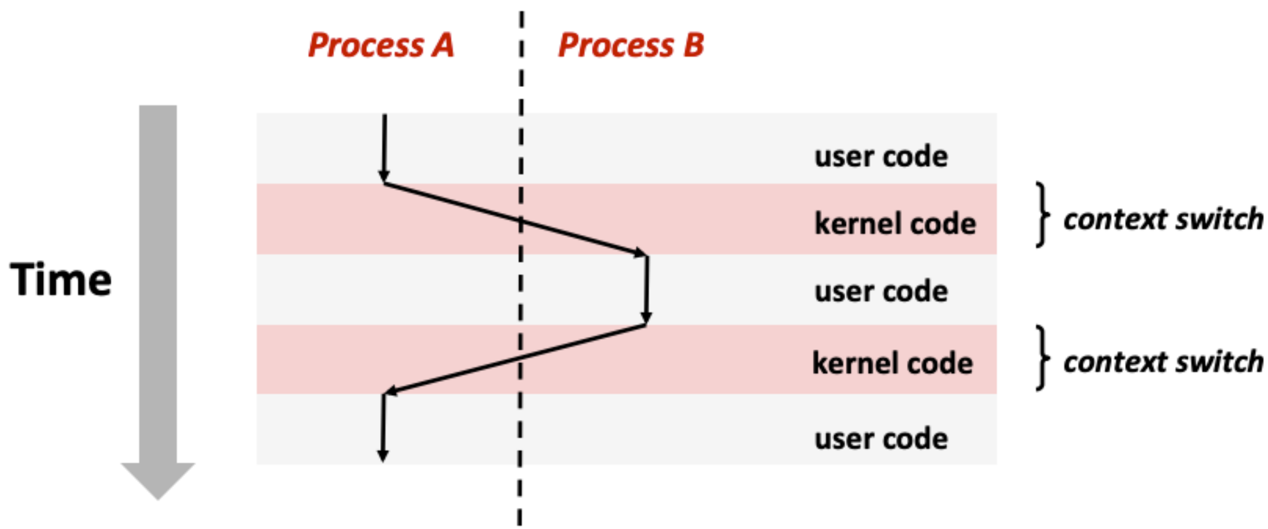
- Normal function call
  - Synchronous execution
  - Single computation
- The "*create*" system call that starts a new process
  - Asynchronous execution
  - Two processes proceed after the call

Distinction Between a Program and a Process

- Sequential program
  - Set of functions executed by a single thread of control
- Process
  - Computational abstraction not usually part of the programming language
  - Created independent of code that is executed
  - Key idea: multiple processes can execute the same code concurrently
- In the following example, two processes execute function *sndch* concurrently

Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a context switch

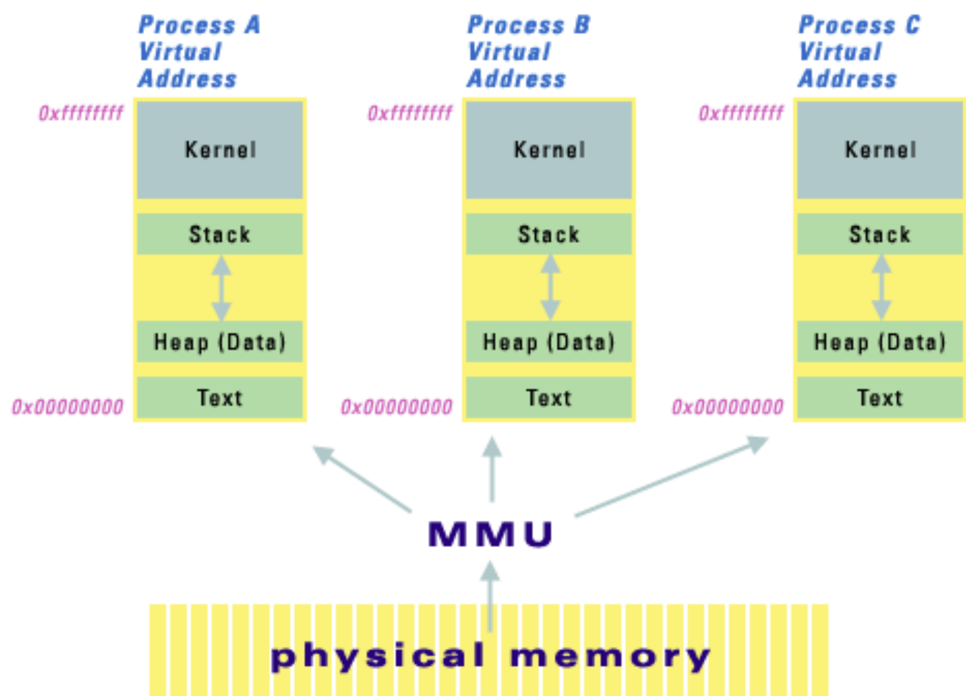


Example of Two Processes Sharing Code

```
/* 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 multitasking environments
- Each process requires its own
  - Runtime stack for function calls
  - Storage for local variables
  - Copy of arguments
- A process *may* have private heap storage as well



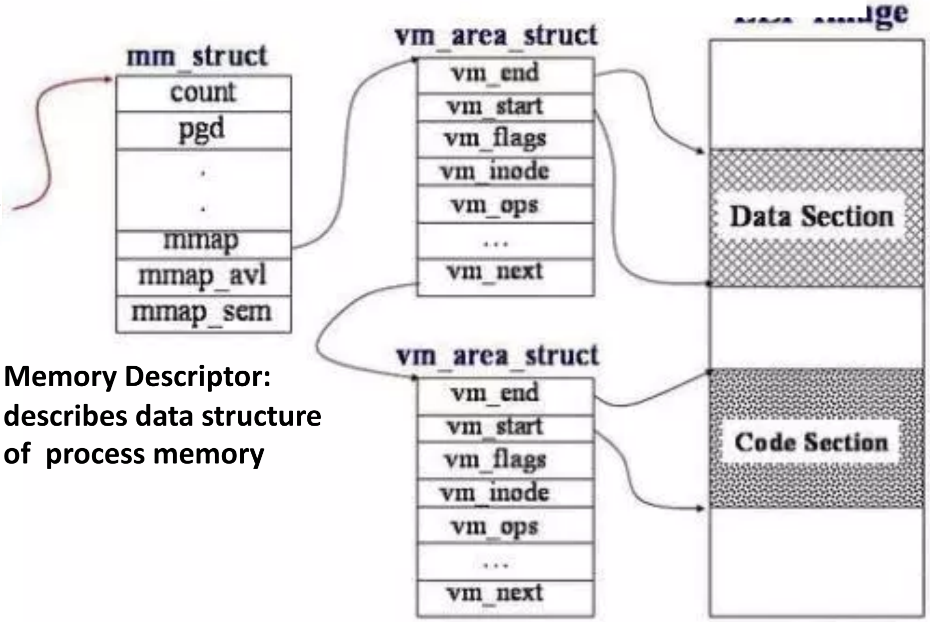
http://www.lynx.com/posix-processes-name-spaces-and-virtual-memory/mmu00a/

Consequence for Programmers

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

Process Virtual Address

Each process



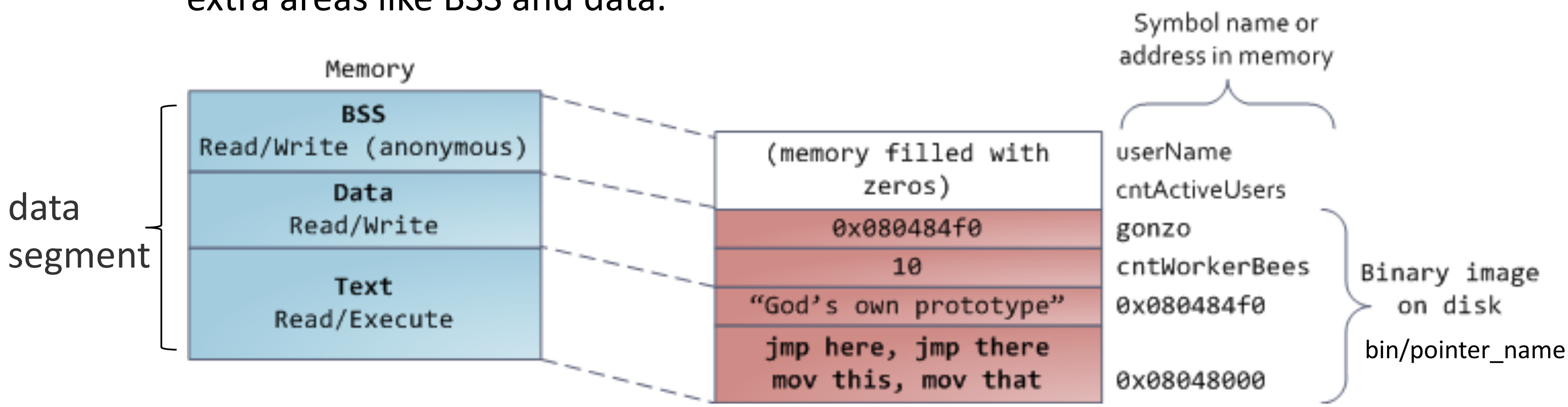
Memory Descriptor: describes data structure of process memory

OS allocates the memory of this process for the kernel

Linux: Management of Process Memory

Example: How Kernel Manages Memory (1/3)

- The contents of a *pointer* - a 4-byte memory address - live in the data segment
- The **text** segment: the actual string is read-only and stores all your code
  - Maps your binary file in memory
  - Writes to this area earn your program as "**Segmentation Fault**" (Help prevent pointer bugs)
- This diagram shows the segment and the example variables
  - A segment may contain many areas. For example, each memory mapped file normally has its own area in the mmap segment, and dynamic libraries have extra areas like BSS and data.



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

Example: How Kernel Manages Memory (2/3)

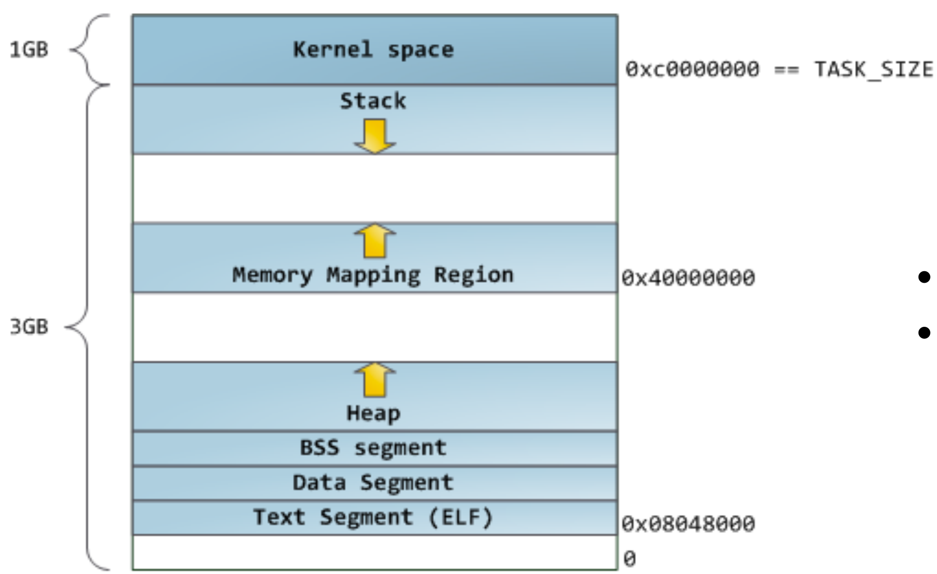
- Examine binary images in Ubnutu using the nm and objdump commands to display symbols, their addresses, segments, and so on.

Data structure for process descriptor

```
struct task_struct {
                 ...
                 struct mm_struct
 *mm;  /*Memory Descriptor data struct*/
                 ...
 }
```

struct mm_struct declared

```
struct mm_struct {
                  ...
                  struct vm_area_struct *mmap;
  /* list of VMAs */
                  ...
  }
```



- The virtual address layout
- A process in Linux is called a task

Example: How Kernel Manages Memory (3/3)

- Within memory descriptor for managing program memory: the set of virtual memory areas and the page tables
- Data structure of Virtual Address Space (VMA)

```
struct vm_area_struct{
    struct mm_struct * vm_mm;

    /* record VMA area start address*/
    unsigned long vm_start;
    /* record VMA area end address*/
    unsigned long vm_end;
    /* points to the next VMA area data structure*/
    struct vm_area_struct *vm_next;
     ...
    }
```

- Goal: VMA is more efficient to manage memory (paging)
- Memory mapping from Virtual to Physical memory: implement by using red-black trees instead of linked list

AN OPERATING SYSTEM FROM THE INSIDE

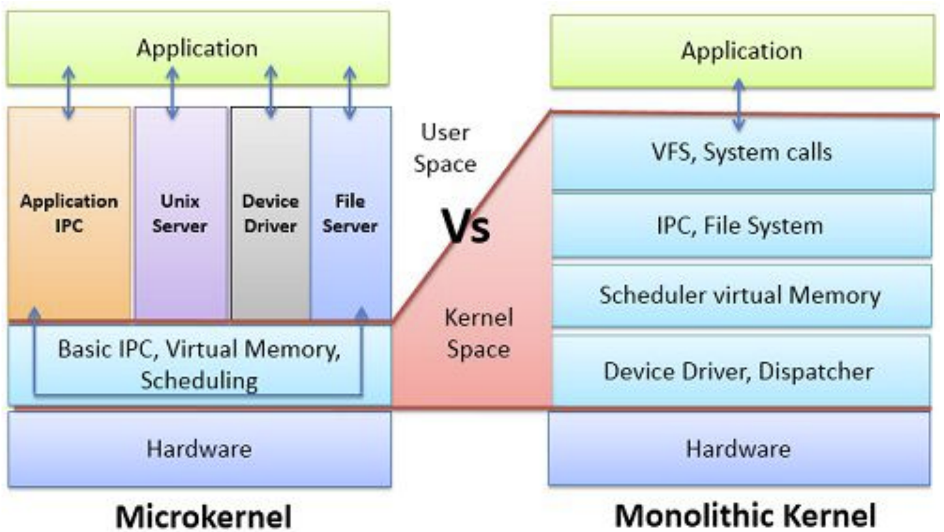
Operating System

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

Building an Operating System

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

1. The kernel is broken down into separate processes
2. Run in kernel/user
3. All servers in different address spaces
4. Servers invoke "services" by sending messages



1. Monolithic kernel is a single large process
2. Single static binary file
3. All services in kernel space
4. Invoke functions directly

Xinu: Major OS Components

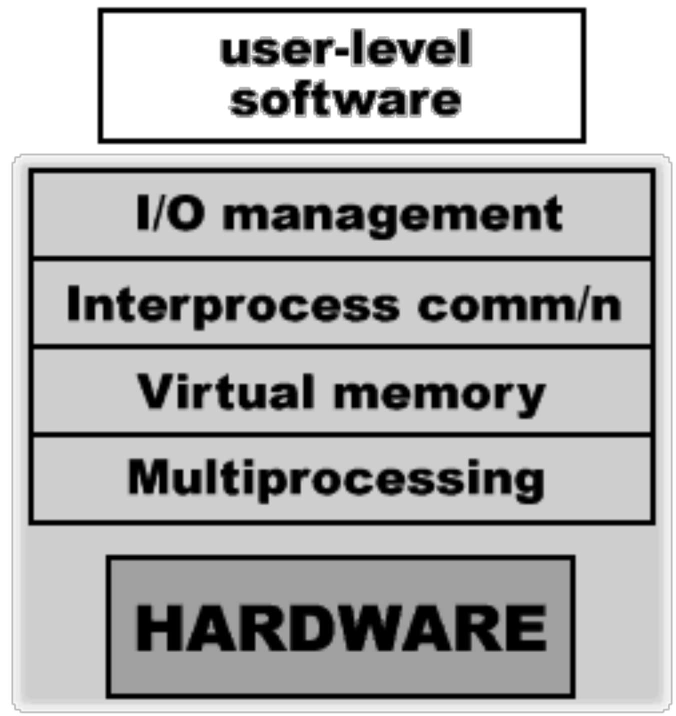
- **Process manager:** decide how to allocate CPU, keep track of status of each process
- **Memory manager:** in charge of main memory, such as checking the validity of each request for memory space
- **Device manger:** monitor every device, channel, and control unit
- Clock (time) manager
- **File manager:** track every file, including data files, assembler, compilers and application programs
- **Interprocess communication**: allow process to communicate with each other
- **Intermachine communication**
- **Assessment and accounting:** user/system, statistics

Xinu: Multilevel Structure

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

Multilevel vs. Multilayered Organization (1/2)

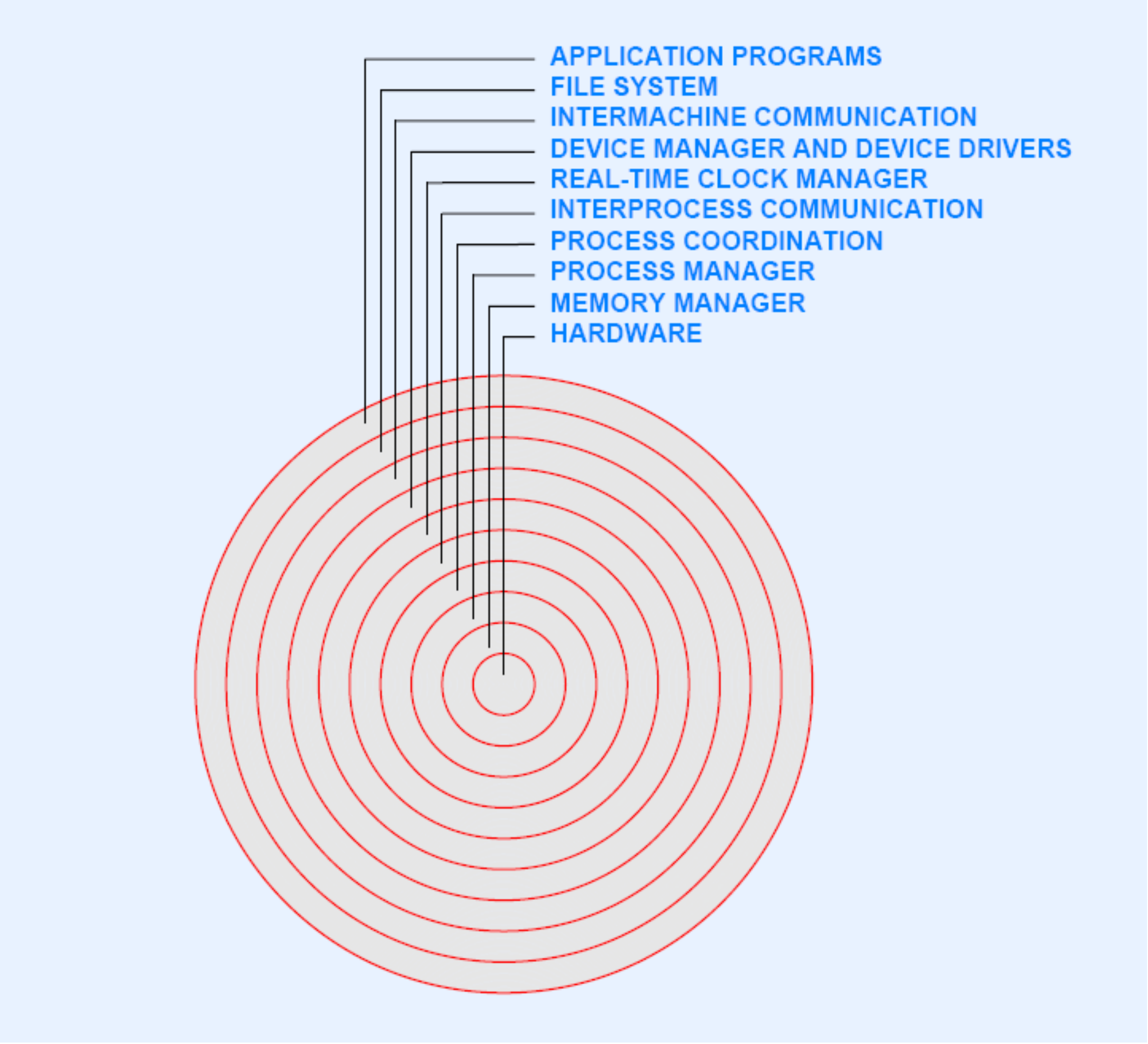
- Multilayer structure
  - Visible to user as well as designer
  - Each layer uses layer directly beneath
  - Involves protection as well as data abstraction
  - Examples
    - Internet protocol layering
    - MULTICS layered security structure
  - Can be inefficient



Multilevel vs. Multilayered Organization (2/2)

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

Multilevel Structure of Xinu



How to Build an OS

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

Design Example

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

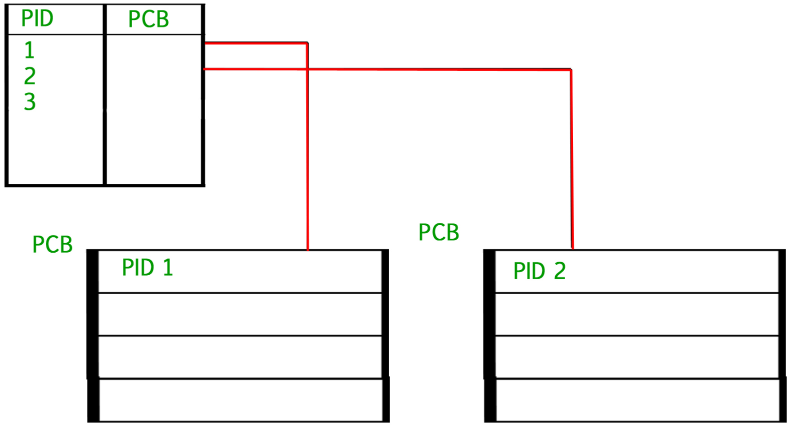
DATA STRUCTURE IN XINU: LIST MANIPULATION

Queues and Lists

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

Lists and Queues in Xinu

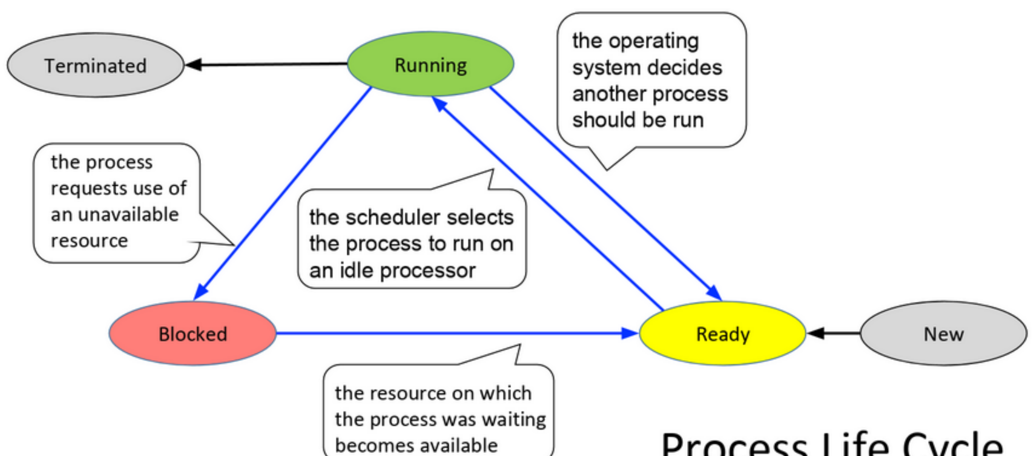
- Important ideas
  - Many lists store processes
  - A process is known by an integer *process ID*
  - A list stores a set of process IDs
- A single data structure can be used to store many types of lists



Process table and process control block

Linked Lists in the OS

- Manipulating lists of processes is an important operation in the OS
  - A process's lifecycle consists of moving between, and in, queues and lists
- Xinu implements a unified approach to list management
  - All list management uses this common infrastructure
  - Common functions to create a new list, insert an element at the end of the list, insert or remove from the middle, remove an item from the front



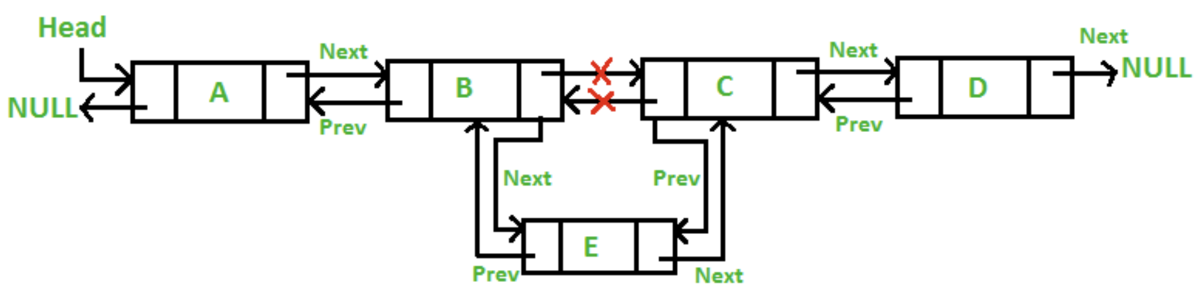
Process Life Cycle

Unified List Storage in Xinu

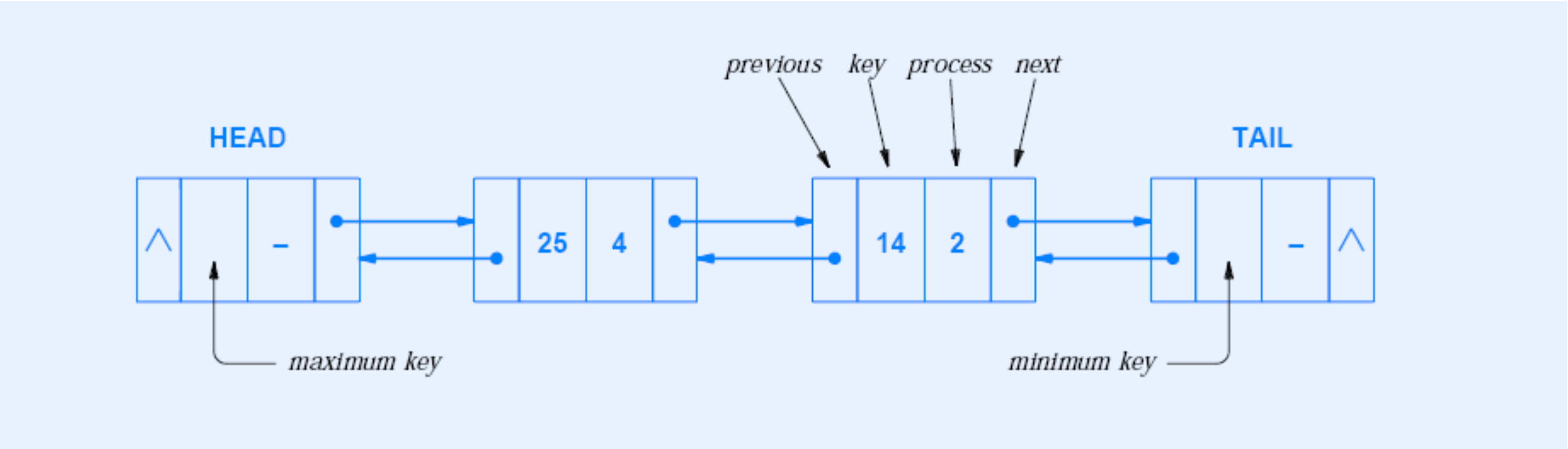
- The process manager handles processes
  - A process moves among the lists frequently
  - At any time, a process is only in one list
- Rather than store all the information about a process, the process manager is free to store only the *process ID (PID)* or *Thread ID (TID)* in a list
  - So when we refer to to putting a process on a list, it really means *putting the PID* there – the process control block need not move
- Unified implementation means that not every subsystem uses all the list features

List Properties

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

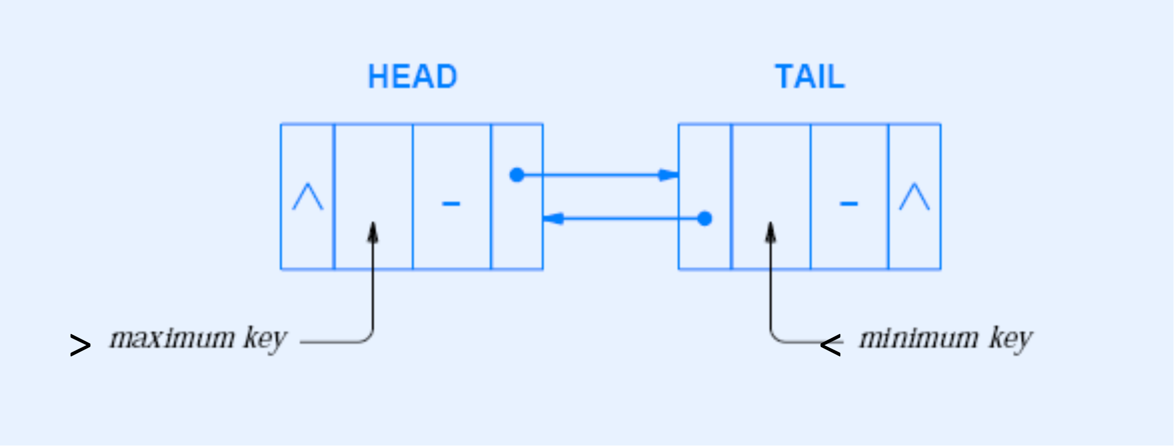


Conceptual List Structure



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

Pointers in an Empty List



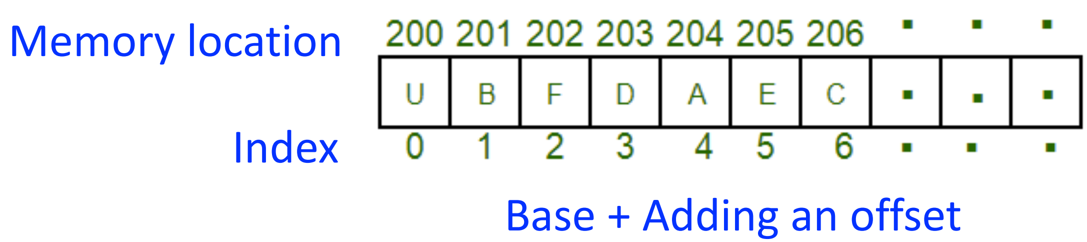
- Head and tail linked
- Eliminates special cases for insertion or deletion

Reducing List Size

- Pointers can mean a large memory footprint
- Important concept: Compact memory usage
  - A process can appear on at most one list at any time
- Techniques used to reduce the size of Xinu lists
  - Relative pointers
  - Implicit data structure
- Most OS place a fixed upper bound on the number of processes
  - In Xinu, constant NPROC specifies the maximum number of processes each user can create, and process identifiers range from 0 through NPROC – 1.

Relative Pointers

- Store list elements in an array (contiguous memory locations)
  - Each item in array is one node
  - Use *array index* instead of address to identify a node
- Relative pointers
  - Given that there is some (small) fixed number of processes (NPROC)
  - One might use a pointer in this situation, which is 4 bytes (32-bit architecture)
    - Ex: NPROCS < 62, we only need 6 bits
  - Allocate the nodes in a contiguous array and use the array index as a "pointer"

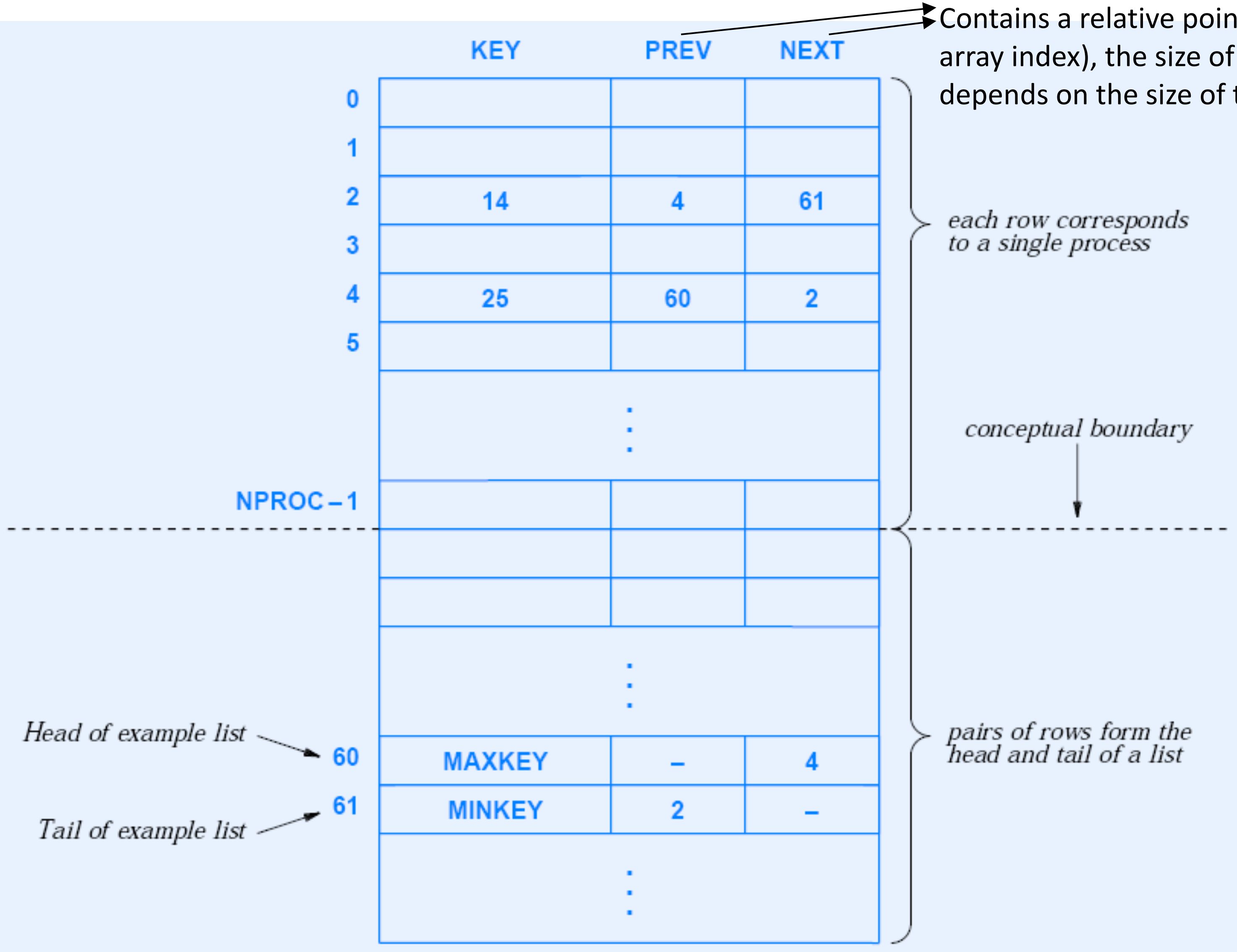


Implicit data structure

- Omit the process ID field from all nodes
- Because a process can only be in one list, we can use the list position to indicate the ID
- To omit the PID, use an array and use the i^{th} element of the array for process *i*
- 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 array correspond to process *i*, for $0 \leq i < NPROC$
  - Store heads and tails in same array at positions *NPROC* and higher

Illustration of Xinu List Structure

Contains a relative pointer (i.e., an array index), the size of the field depends on the size of the array



Queue Table Array

Implementation

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

A Question About Types in C

- K&R [1] C defines *short*, *int*, and *long* to be machine-dependent
- ANSI C leaves *int* as a machine-dependent type
- A programmer can define type names
- Question: should a type specify
  - The purpose of an item?
  - The size of an item?
- Example: should a process ID type be named
  - *processid_t* to indicate the purpose?
  - *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

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[];
```

allocates enough space for each process

Definitions from queue.h (part 2)

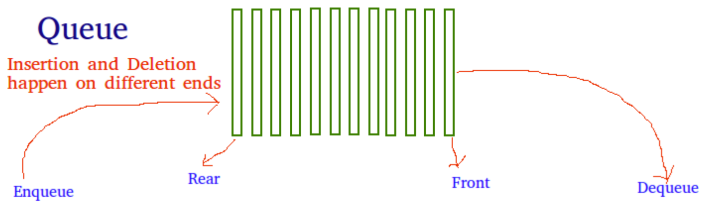
```
/* Inline queue manipulation functions */

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

List manipulation functions

- Look at the list manipulation functions
  - *enqueue* and *dequeue* (in *queue.c*)
  - *getfirst, getlast,* and *getitem* (in *getitem.c*)
  - *insert* (in *insert.c*)
  - *newqueue* (in *newqueue.c*)

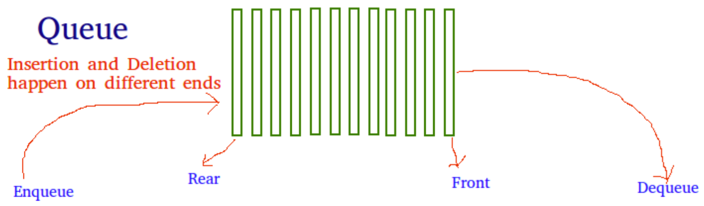
Code for Insertion and Deletion from a Queue (part 1)

```
/* 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 */
)
{
        int 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 */
)
{
        int16 curr;                         /* Runs through items in a queue*/
        int16 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

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

Summary
(continued)

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

References

1. Brian Kernighan and Dennis Ritchie published the first edition of *The C Programming Language* in 1978, known to C programmers as "K&R", as an informal specification of the language.
2. Introduction to Programming Systems: Systems Calls and Standard by Professor Jennifer Rexford from Princeton University.