OS organization

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Last lecture

- C and systems resources
  - Books, websites
  - Pre-requisites
  - Typical mistakes

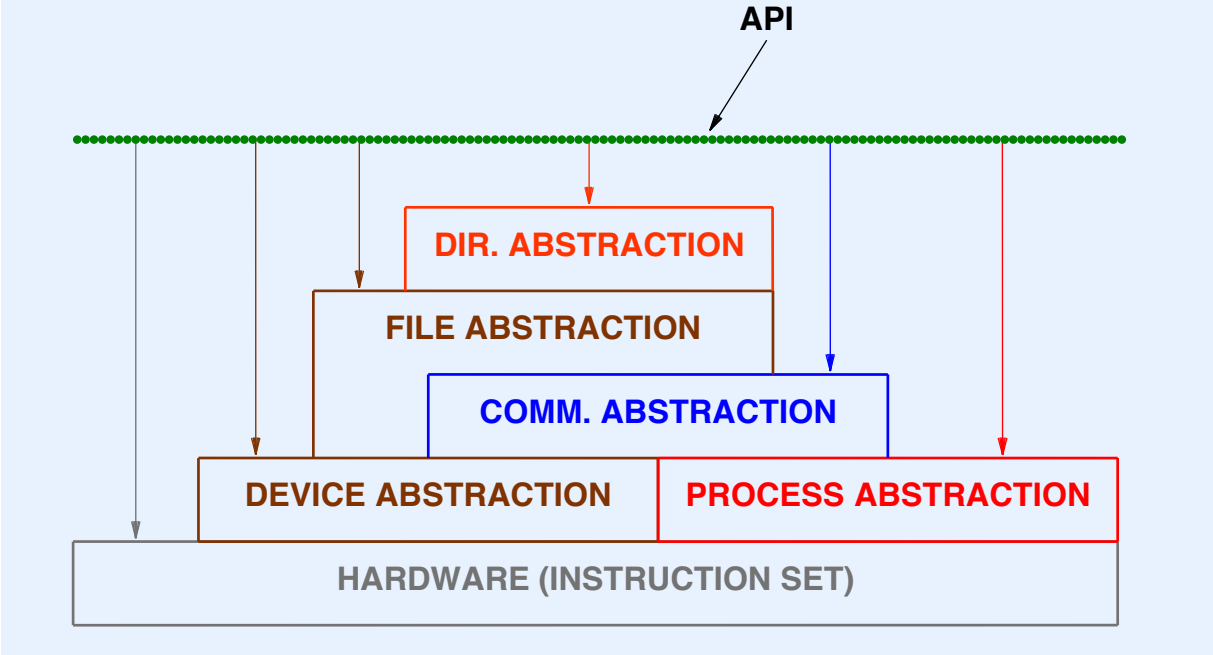
Last lecture

- OS interface:
  - Kernel/user mode
  - Booting process
  - Interrupts
  - System call implementation
  - OS services enabled by HW functions

The system interface

- Single copy of OS per computer
  - Hidden from users
  - Accessible only to application programs
- Application Program Interface (API)
  - Defines services the OS makes available
  - Defines parameters for those 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

An OS from outside

Example: System Call in Xinu

- Write a character on console

```
// ex1.c - main
// Write "hi" on the console.

#include <xinu.h>

void main(void) {
    putc(CONSOLE, 'h');
    putc(CONSOLE, 'i');
    putc(CONSOLE, '\n');
    ...
```

- 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 services includes functions to **suspend** and then **resume** a process
  - Socket API used for Internet communications 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 and **read** data
  - Calls **close** to terminate the use
- Internally, the set of I/O functions coordinate:
  - **Open** returns a descriptor **d**
  - **Read** and **write** operate on a descriptor **d**
- Q: Why do we need a descriptor?

Concurrent processing

- Fundamental concept that dominates OS design
- *Real concurrency* achieved by hardware:
  - I/O device operates at the 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
  - Operating system provides the illusion

Multitasking

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

Terminology used with multitasking

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

Process

- OS abstraction
- Create by OS system call
- Managed entirely by OS; unknown to hardware
- Operates (i.e., runs) concurrently with other processes

Example of process creation in XINU

```
// ex2.c
// Example of creating processes in Xinu

#include <xinu.h>

void main(void) {
    resume( create(sndA, 1024, 20, "process 1", 0) );
    resume( create(sndB, 1024, 20, "process 2", 0) );
}

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

void sndB(void) {                   // Fixed: Lecture slide
    while(1) putc(CONSOLE, ‘B');    // had a type on the name
}
```

‵‵‵

Difference between function call and process creation

- Function call:
  - Synchronous execution
- Process creation (i.e., create() syscall):
  - Asynchronous execution
  - Two processes are running after create
  - …

Distinction between Program and Process

- Program:
  - Set of functions (code)
  - And initial data
- Process:
  - Computational abstraction
  - Not usually part of the programming language
  - Key idea: Multiple processes can execute the same program

Storage allocation when multiple processes execute

- Various models exist for multitasking environments
- Each process requires its own:
  - Runtime stack
  - Local variables
  - Copy of arguments

Example: Two processes sharing code

```
// Example of 2 processes executing the same code concurrently

#include <xinu.h>

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

// Output a character on a serial device indefinitely
char sndch(char ch) {
    while (1)     putc(CONSOLE, ch);
}
```

An OS from the inside

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 IO
- Philosophy: fairness
- Policy: FCFS resource access
- Mechanism: queue of requests (FIFO)
- Implementation: program written in C

Lists and queues

- Fundamental data structures for OS
- Various forms:
  - FIFOs
  - Priority lists
- Operation:
  - Insert item
  - Extract item
  - Delete arbitrary item

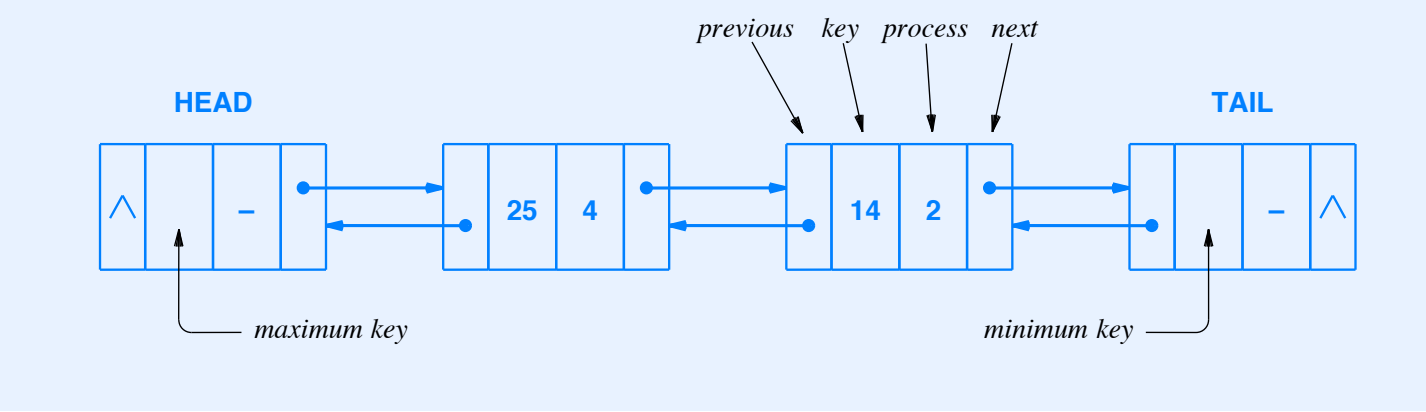
Lists and queues in XINU

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

Unified list storage in XINU

- Basic backing storage: **queuetab**
  - A single array used to implement doubly-linked lists
  - Array index represents **pid**
  - Each array element holds: **prev, next** and **key**
    - Conceptually each node also holds the process id
- Versatile uses:
  - FIFO (key is not used)
  - Priority list (key is the priority)
- Head and tail are stored elements store in a reserved space in queuetab:
  - Head holds the maximum integer as a key
  - Tail holds the minimum integer as a key

Unified list storage in XINU



- Conceptually nodes contain: **prev**, **next**, **pid**, **key**
- Each list has a **head** and **tail**
- Key is optional (list dependent)

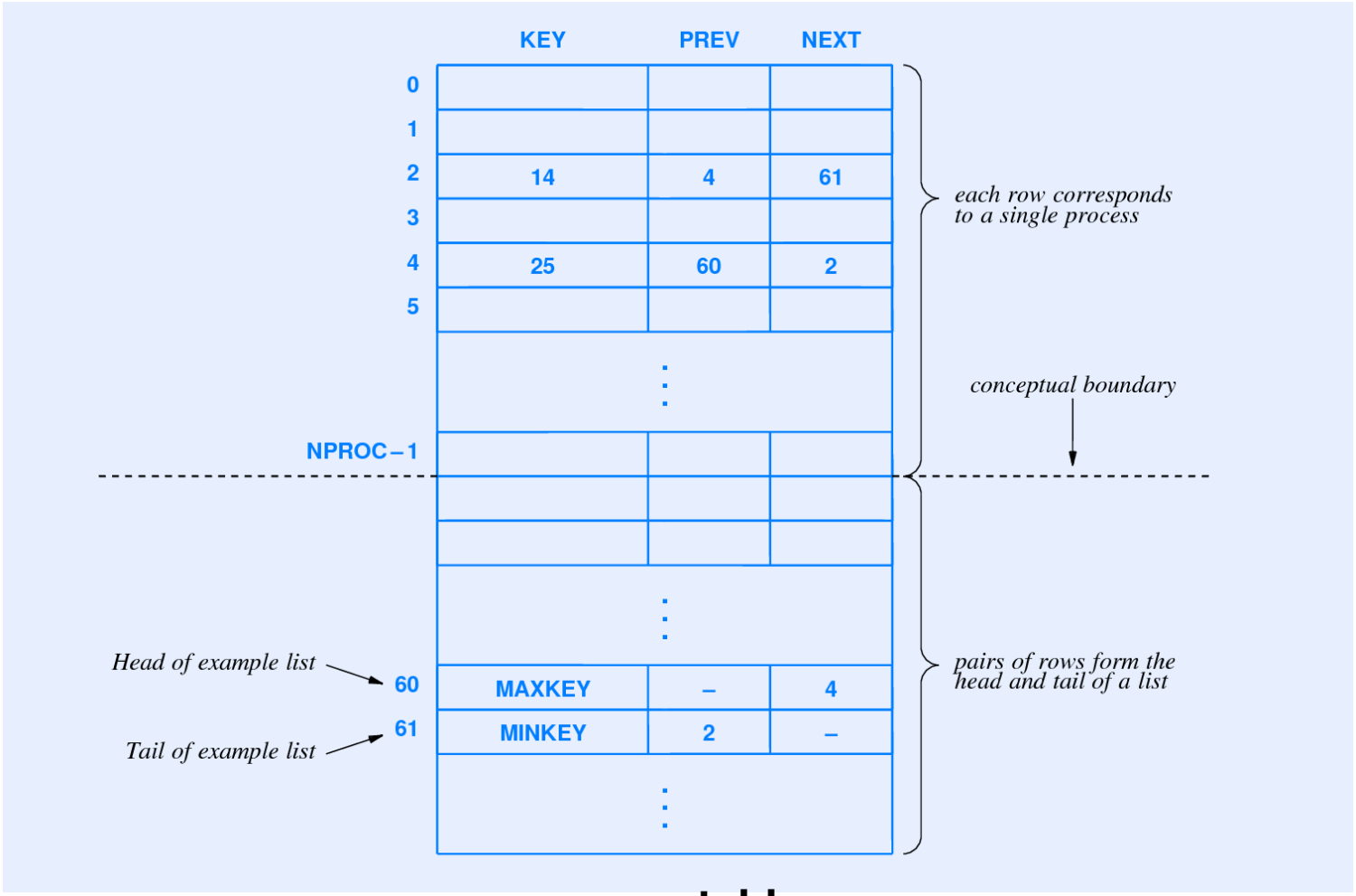
Empty list?

- Two nodes: head and tail node

XINU lists

- Important: Each process can only be in one list
  - This allows some simplifications in the implementation
- Number of processes and list is bounded and small
  - And known during compile time
- Memory usage is important
  - Specially for embedded systems

Unified list storage in XINU



queue table

A question about types in C

- K&R C defines **short**, **int** and **long** to be machine-dependent
- ANSI C leaves **int** as a machine-dependent type
- A programmer can define type names
- Q: 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 types encompass both purpose and size
- Example: consider a variable that holds an index of the 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

Summary

- Operating systems supply a set of services
- System calls provide interface between OS and application
- Concurrency is a fundamental concept
  - Between IO devices and processor
  - Between multiple computations
- Process an OS abstraction for concurrency
- Process differs from program and function
- Queues and lists