Operating System Principles

操作系统原理（ v2 ）

Chapter　12

Operating System Design

操作系统的设计

LeeXudong

—Nankai Univ. SE.

Objectives

- The nature of the design problem
- Stages of OS Design
- Interface design
- Implementation
- Design Pattern
- Performance
- Project management
- Trends in operating system design

The nature of the design problem

- 1. Goals
- 2. Why is it hard to design an operating system

Goals

- Main items for general-purpose OS
  - 1.Define abstractions
  - 2.Provide primitive operations
  - 3.Ensure isolation
  - 4.Manage the hardware

Goals

- Tasks
  - 1.define the right abstractions
    - processes, file, thread, memory model, semaphore, I/O
  - 2.Each of the abstractions can be instantiated in the form of concrete data structures
  - 3.provide mechanisms to keep multiple logged users separated
  - 4.isolate failures
  - 5.manage the hardware

Why is it hard to design an operating system?

Moore's law says that computer hardware improves by a factor of 100 every decade.

Nobody has a law saying that operating systems improve by a factor of 100 every decade.

It's hard to design an OS

- 1. backward compatibility
- 2. the failure to adhere to good design principles
- 3. have to deal with concurrency
- 4. have to deal with potentially hostile users
- 5. share some of information and resources
- 6. os designers really do not have a good idea of how their systems will be used, so they need to provide for considerable generality
- 7. modern operating systems are generally designed to be portable, meaning they have to run on multiple hardware platforms

Stages of OS Design

- 三个阶段
  - 功能设计
    - 根据系统的设计目标和使用要求，确定所设计的操作系统应具备哪些功能；以及操作系统的类型
  - 算法设计
    - 根据计算机的性能和操作系统的功能，选择和设计满足系统功能的算法和策略，并分析和估算其效能
  - 结构设计
    - 按照系统的功能和特性要求，选择合适的结构，使用相应方法将系统逐步地分解、抽象和综合，使 OS 结构清晰、简明、可靠、易读、易改，而且使用方便、适应性强。
- 设计目标：
  - 可靠、高效、易维护、易移植、安全、可适应性、简明、使用方便
- 结构问题
  - 程序结构、软件结构、操作系统体系结构

Interface design

- Guiding Principles
- Paradigms 范型
- The System Call Interface

Interface design:Guiding Principles

- 1. Simplicity
  - Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away
  - KISS: Keep It Simple, Stupid.
- 2. Completeness
  - The interface must make it possible to do everything that the users need to do, that is, it must be complete
  - Everything should be as simple as possible, but no simpler.
  - MINIX: send, receive, sendrec
  - Amoeba: sendrec
- 3. Efficiency
  - If a feature or system call cannot be implemented efficiently, it is probably not worth having

Interface design:Paradigms

- architectural coherence
  - once the goals have been established, the design can begin
  - a good starting place is thinking about how the customers will view the system
  - one of the most important issues is how to make all the features of the system hang together well and present it
- Paradigms
  - User Interface Paradigms: GUI:WIMP, VCR
  - Execution Paradigms
    - Algorithmic paradigm*、 event-driven paradigm*
  - Data Paradigms:
    - UNIX: everything is a file
    - Windows2000: everything look like an object

Paradigms: Algorithmic paradigm

```
main( )
{
        int ... ;

        init( );
        do_something( );
        read(...);
        do_something_else( );
        write(...);
        keep_going( );
        exit(0);
}
```

Algorithmic code

Paradigms: event-driven paradigm

```
main( )
{
        mess_t msg;

        init( );
        while (get_message(&msg)) {
                switch (msg.type) {
                        case 1: ... ;
                        case 2: ... ;
                        case 3: ... ;
                }
        }
}
```

Event-driven code

The System Call Interface

- Simplicity:
  - Tanenbaum’s first law of software
    - Adding more code adds more bugs
  - UNIX System Call: exec(name,argp,envp)
    - API1: execl(name,arg0,arg1,…,argn,0)
    - APIm: execle(name,arg0,arg1,…,argn,envp)
- Hardware Power:
  - Lampson’s slogan(1984)
    - Don’t hide power
- Connection-oriented versus connectionless calls
- the system call interface is its visibility

Implementation

- System Structure*
- Mechanism versus Policy
- Orthogonality
- Naming*
- Binding Time
- Static versus Dynamic Structures*
- Top-Down versus Bottom-Up Implementation
- Useful Techniques

System Structure

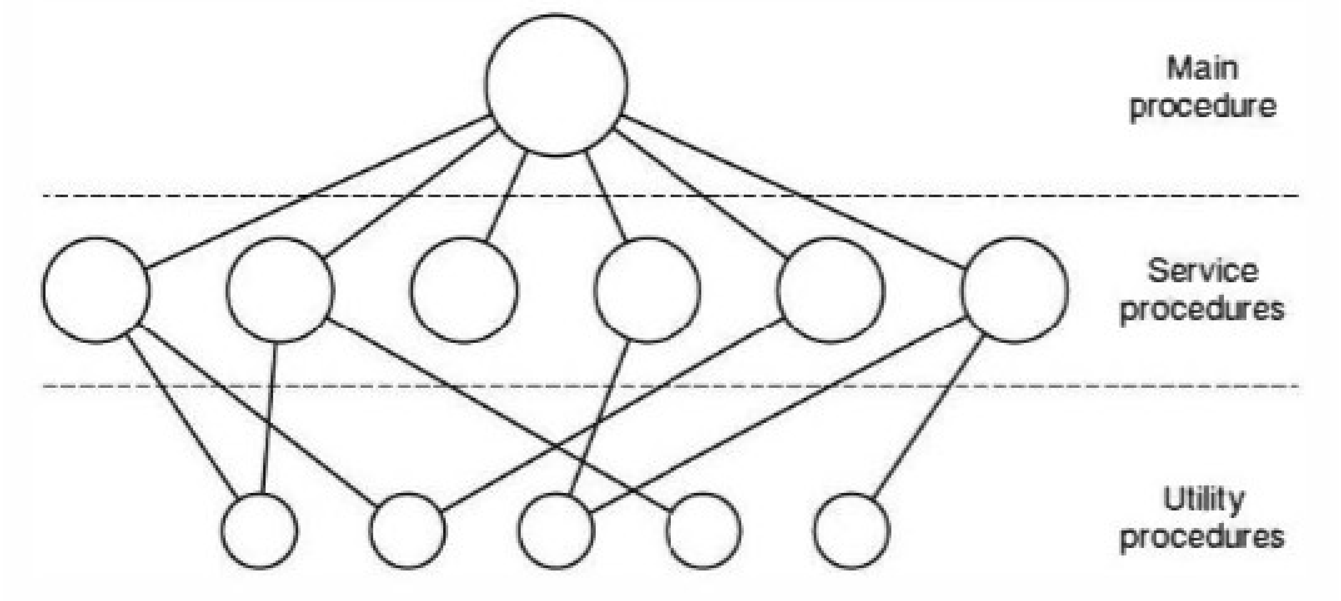
- 1) Runtime System Structure
- 2) System structure of Design and Implementation

Implementation: System Structure

- Monolithic Systems
- Layered Systems
- Exokernels
- Client-Server Systems
- Extensible Systems
- Kernel Threads

Monolithic Systems

- A main program that invokes the requested service procedure.
- A set of service procedures that carry out the system calls.
- A set of utility procedures that help the service procedures



Monolithic Systems

- 最常用的组织方式：整体式系统结构
  - 常被称为“大杂烩”，本质上无结构；
  - 也称为“无序模块结构、模块接口结构”
  - 整个操作系统是一堆过程的集合，每个过程都可以任意调用其它过程。使用这种技术时，系统中的每个过程都有一个完好定义的接口，即入口参数和返回值，而且相互间的调用不受约束
  - 使用此结构方式时，开发人员为了构造最终的目标操作系统系统程序，首先将一些独立的过程，或包含过程的文件进行编译，然后用链接程序将它们链接成一个单独的目标程序。每个过程都对其他过程可见。
  - 此结构也存在低度结构化：系统调用层。其存在三类程序：
    - 有处理服务过程请求的一个主要程序（主过程）
    - 有执行系统调用的一套服务过程
    - 有支持服务过程的一套使用程序

Monolithic Systems 整体式系统结构

- 整体式结构的特点
  - 模块是以功能而不是以程序或数据的特点来划分的
  - 数据作为全程量使用
  - 不同模块间可不加限制地互相调用和转移，模块间信息传递方式可以随意约定
  - 优点
    - 结构紧密、接口简单、系统效率高
  - 缺点
    - 模块间独立性差，结构不清晰，不易解读，不易修改
    - 为了保证数据的完整性，往往采用全局封中断的方式，从而限制了系统的并发性
- ？哪些 OS 属于此类

Layered Systems

- THE 系统 (1968)：　(荷兰)E.W.Dijkstra 等 , Eindhoven 技术学院
  - 目标是减少各模块之间毫无规则地相互调用、相互依存关系；
  - 分层：所用功能模块按功能的调用次序分成若干层
  - 最终系统各部分仍被链接成了完整的单个目标程序。
- THE 层次式系统中 , 上层软件都基于下一层软件之上。

5	操作员
4	用户程序
3	输入 / 输出管理
2	操作员 - 进程通信
1	内存和磁盘管理
0	处理器分配和多道程序

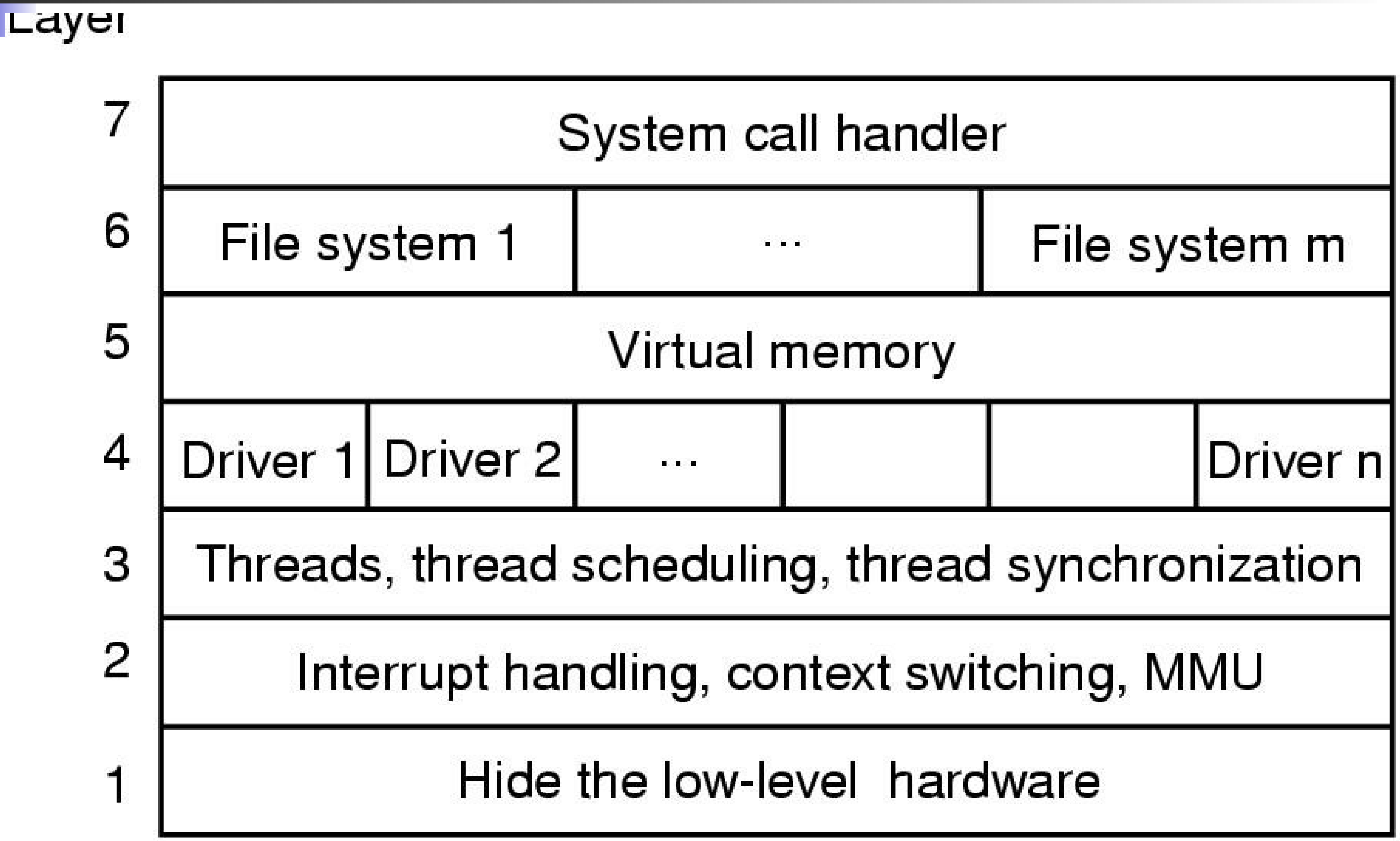
Layered Systems

- 分层结构
  - 将整个操作系统分解成若干个基本模块，并按照一定的原则将这些模块排列成若干层，各层之间只有单向依赖关系，即低层为高层服务，高层依赖于低层，各层之间不能构成循环。
- 全序结构
  - 在层次结构 OS 中，如果不仅层间是单向依赖关系，而且同一层的各模块间也相互独立、单向依赖和不构成循环的称为全序结构。
- 优点：
  - 将整体问题局部化，以保证各层模块的正确性
  - 层次结构的 OS 每层中各模块以统一的接口提供给上层模块调用，大大减少了接口量，使各层次间的调用更加清晰和规范
  - 对于以进程作为层次中模块基本单位的层次结构（即进程分层结构），能较好地体现 OS 的并发特征，能动态地描述系统的执行过程
  - 各层次间独立性强，灵活性高，易于维护、修改和移植
  - 系统结构清晰，易于阅读和理解
- 缺点：增加了系统的开销，信息传递效率比整体式结构低，由核心统一管理后，调度的负荷随之加重

Layered Systems

- Edsger W. Dijkstra (EWD)
  - Dijkstra 被称为“结构程序设计之父”和“先知先觉 -Oracle”
  - 1972 年 Dijkstra 获得 ACM 图灵奖
  - 解决编译系统难题：堆栈思路
  - ALGOL60 起草人之一
  - 同步进程的协调 (死锁 , 信号量) 和操作系统的结构
  - Go To 语句有害和结构程序设计 ,“ 一个程序的质量与程序中所含的 Go To 语句的数量成反比”
  - 名言：　“有效的程序员不应该浪费很多时间用于程序调试 , 他们应该一开始就不要把故障引入。”
  - “Do only what only you can do”
  - http://www.cs.utexas.edu/users/EWD/

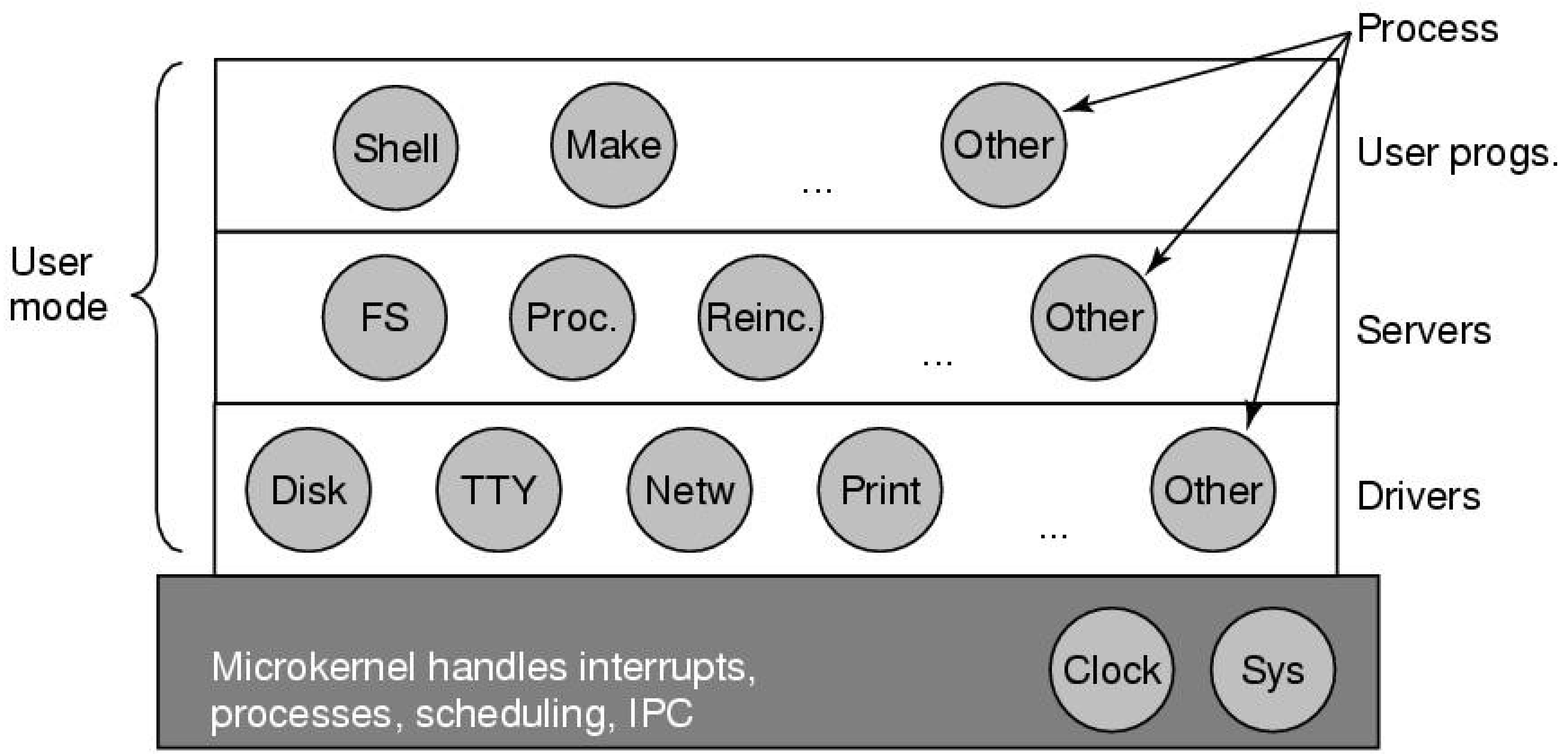
Layered Systems


Layer
7
System call handler
6
File system 1
...
File system m
5
Virtual memory
4
Driver 1
Driver 2
...
Driver n
3
Threads, thread scheduling, thread synchronization
2
Interrupt handling, context switching, MMU
1
Hide the low-level hardware

One possible design for a modern layered operating system

Microkernels 微内核

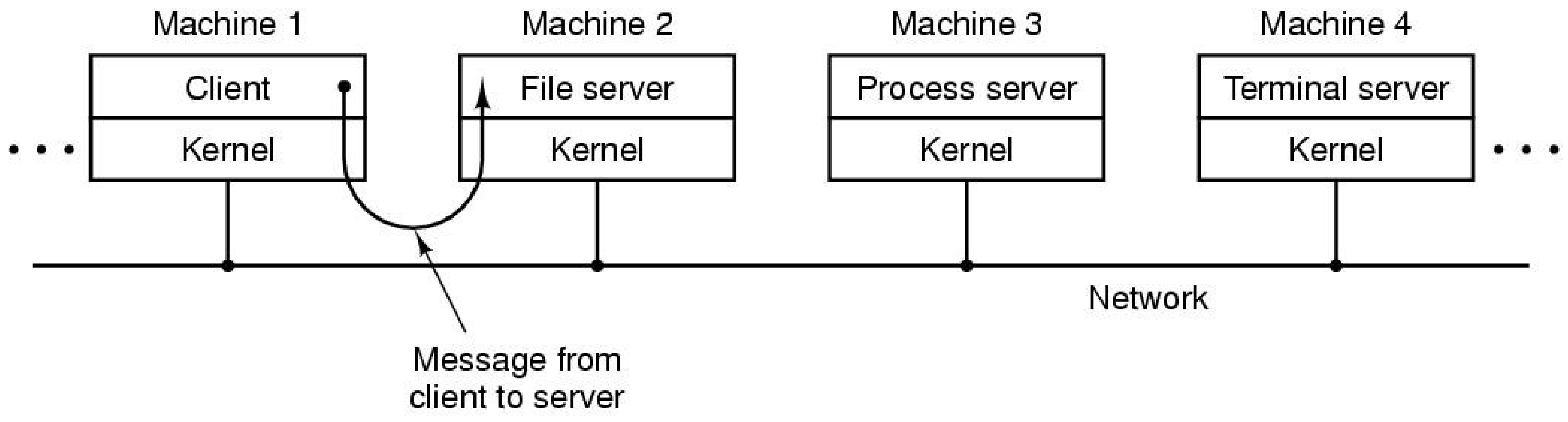
- Case: QNX, MINIX 3



Structure of the MINIX 3 system

Client-Server Model

- Client-Server Model
  - Communication between clients and servers is often by message passing



The client-server model over a network.

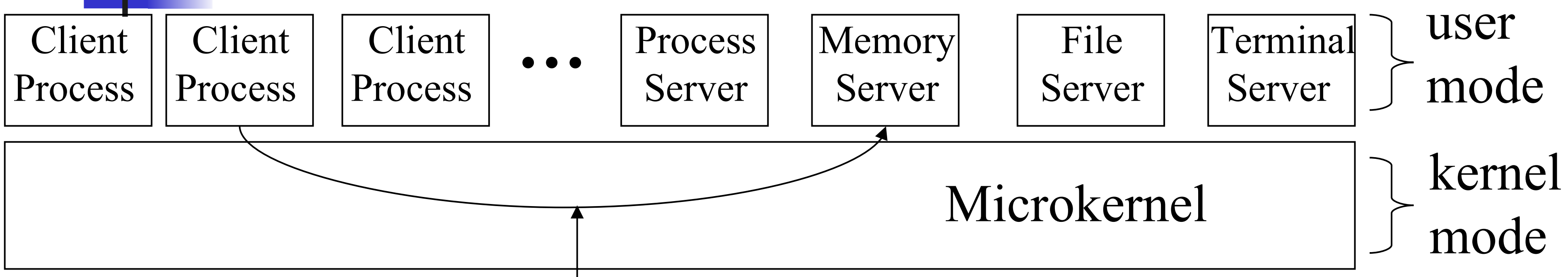
Client-Server Model

- 客户机 / 服务器系统结构
  - 又称微内核的操作系统体系结构
  - 主要思想 :(现代操作系统新理念)
    - 进一步发展这种将代码移到更高层次的思想 , 即尽可能多地从 OS 中去掉东西 , 只留下一个很小的内核 (kernel)
  - Carniegie Univ.:
    - Mach OS
    - GNU/Hurd/Mach
  - Windows NT 早期版本
- 主要特点
  - 运行在核心态的内核
  - 运行在用户态并以客户 / 服务器方式运行的进程层

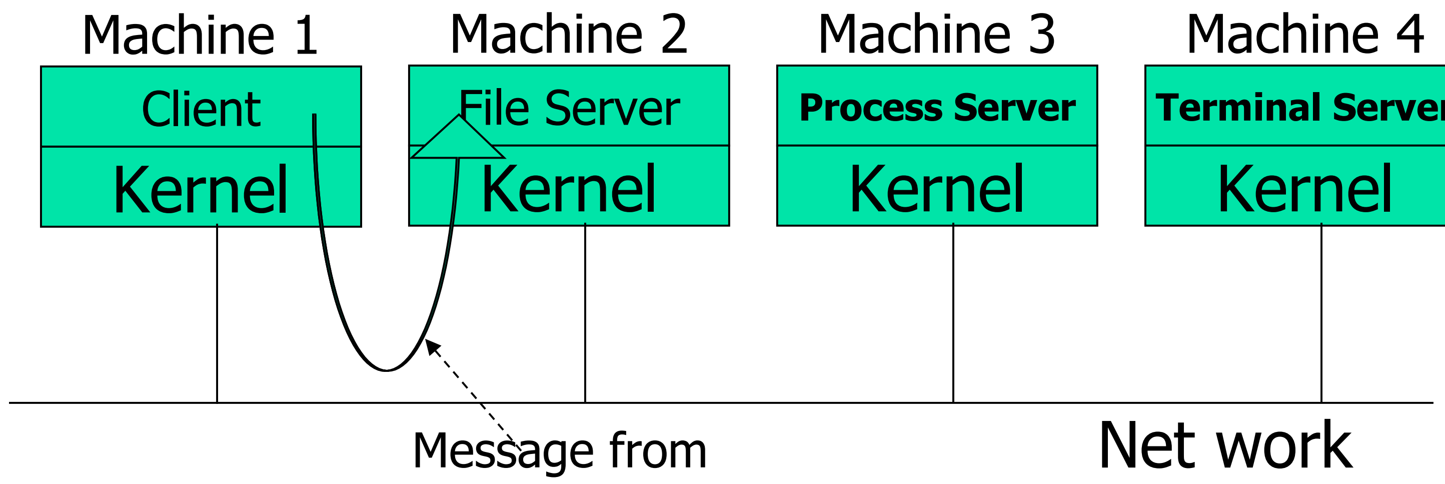
Client-Server Model

- 实现方法
  - 由用户进程来实现大多数操作系统的功能，为了得到某项服务，用户进程 client process 把请求发给服务器进程 server process, 随后服务器继承完成这个操作并返送回答信息
  - OS 内核的全部工作是处理客户机 / 服务器之间的通信。 OS 被分成了多个部分，每个部分仅仅处理一个方面的功能，如文件服务、进程服务、终端服务或存储器服务等，这样每个部分更小，更易于管理
  - 所有的服务都以用户进程的形式运行，它们不在核心态下运行，所以不直接访问硬件，因此一个服务器崩溃了不会导致整个系统的崩溃。
  - 另外，此模型适用于分布式系统，客户机通过消息传递与服务器通信，客户机不需知道这条消息是在本地机处理的还是通过网络送给了远程机器上的服务器。 内核只处理客户机与服务器之间的消息传递
- 当某些特殊的 OS 功能不能单靠用户空间程序时，解决方法:
  - 一种是建立一些运行与核心态的关键的服务进程（ I/O ），它们拥有访问所有硬件的绝对权利，但它们仍然使用通常的消息机制与其它的进程通信;
  - 另一种是在内核中建立起最小的机制 mechanism, 从而把策略 policy 留给在用户空间的服务进程

Client-Server Model



Client obtains service by sending message to server process



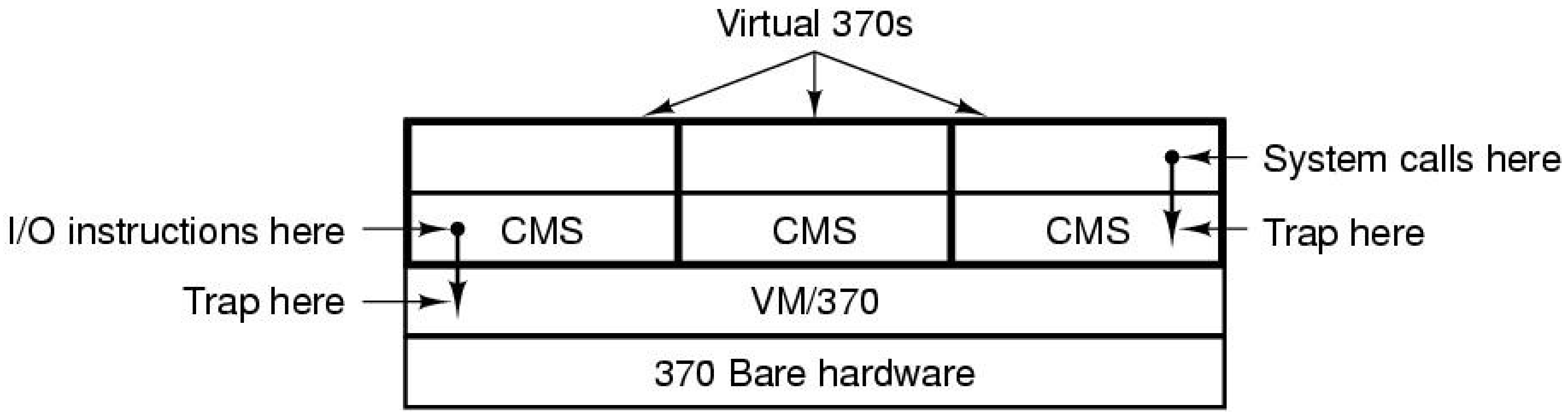
The client-server model in a distributed system

Client-Server Model

- 优点
  - 可靠：每个分支都是独立的和自包含的
  - 灵活：便于增加新的服务功能
  - 适于分布式处理的计算环境
- 缺点
  - 通信的效率问题
  - etc., 高性能的图形用户界面系统

Virtual Machines

- Case: IBM's VM370, 1979



The structure of VM/370 with CMS

Virtual Machines

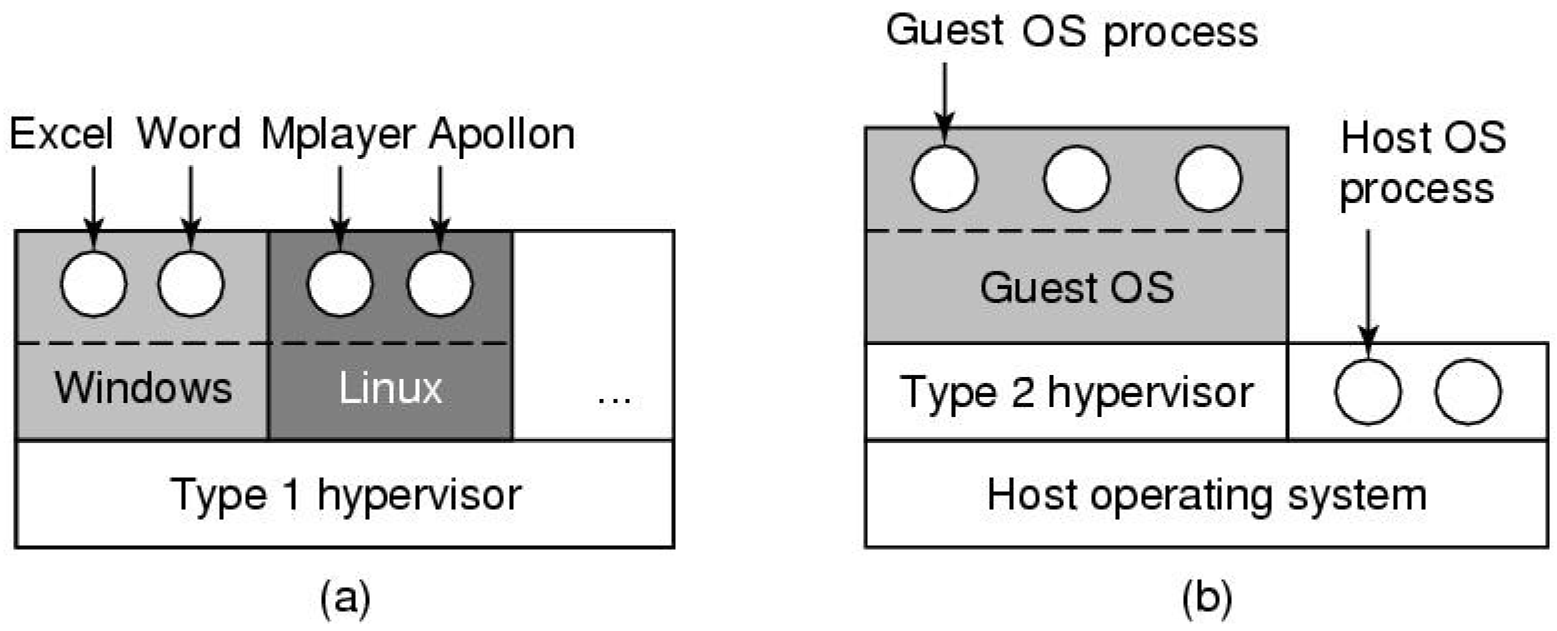
- 虚拟机由来
  - IBM OS/360 最早纯粹是批处理系统，许多用户希望使用分时系统。
  - CP/CMS 后改名为 VM/370(Seawright,Machkinnon, 1979), 其设计思想是:
    - 提供多道程序
    - 一个比裸机具有更方便的扩展界面的计算机
  - VM/370 的任务是将这两者彻底隔离开来；
  - VM/370 把大部分传统操作系统的代码（实现的扩展计算机）分离出来放在了更高的层上 CMS 上，但 VM/370 本身仍然非常复杂 (为了高效率)

Virtual Machines

- 虚拟机的核心
  - 虚拟机监控程序 virtual machine monitor, 其在裸机上运行并且具备了多道程序功能。
  - 该系统向上提供了若干台虚拟机，但它不同于其它 OS 系统的是：这些虚拟机不是那种具有文件等优良特征的扩展计算机，而仅仅是精确复制的裸机硬件，它包括：核心态/用户态、 I/O 功能、中断、及其它真实硬件所具有的全部内容。
  - 由于每台虚拟机都与裸机相同，所以每台虚拟机可以运行一台裸机所能够运行的任何类型的操作系统。不同的虚拟机可以运行不同的操作系统。
- IBM 在此基础上提供了 CMS(Conversational Monitor System) 会话监控系统，一个单用户交互式系统

Virtual Machines

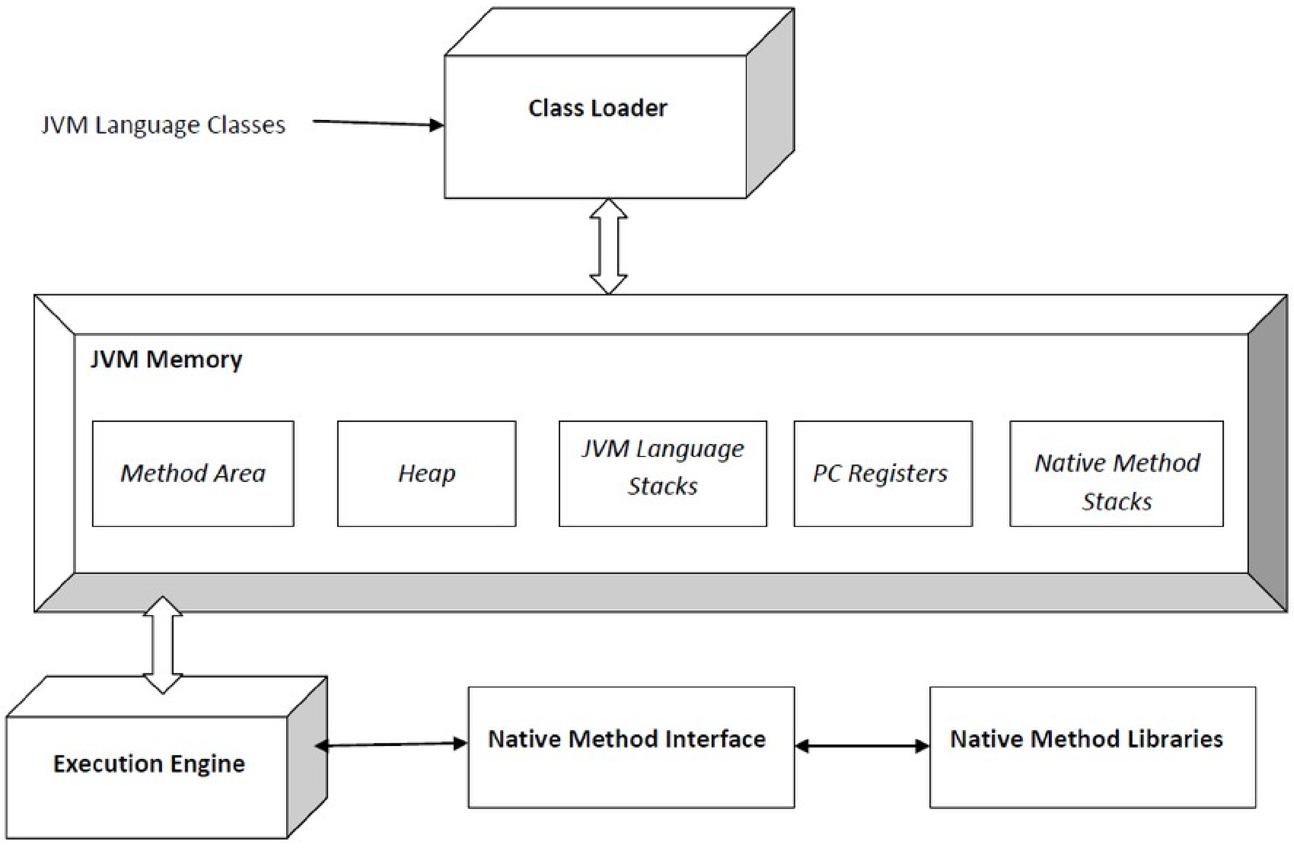
- Virtual Machines
  - Hypervisor I, Hypervisor II



(a) A type 1 hypervisor. (b) A type 2 hypervisor

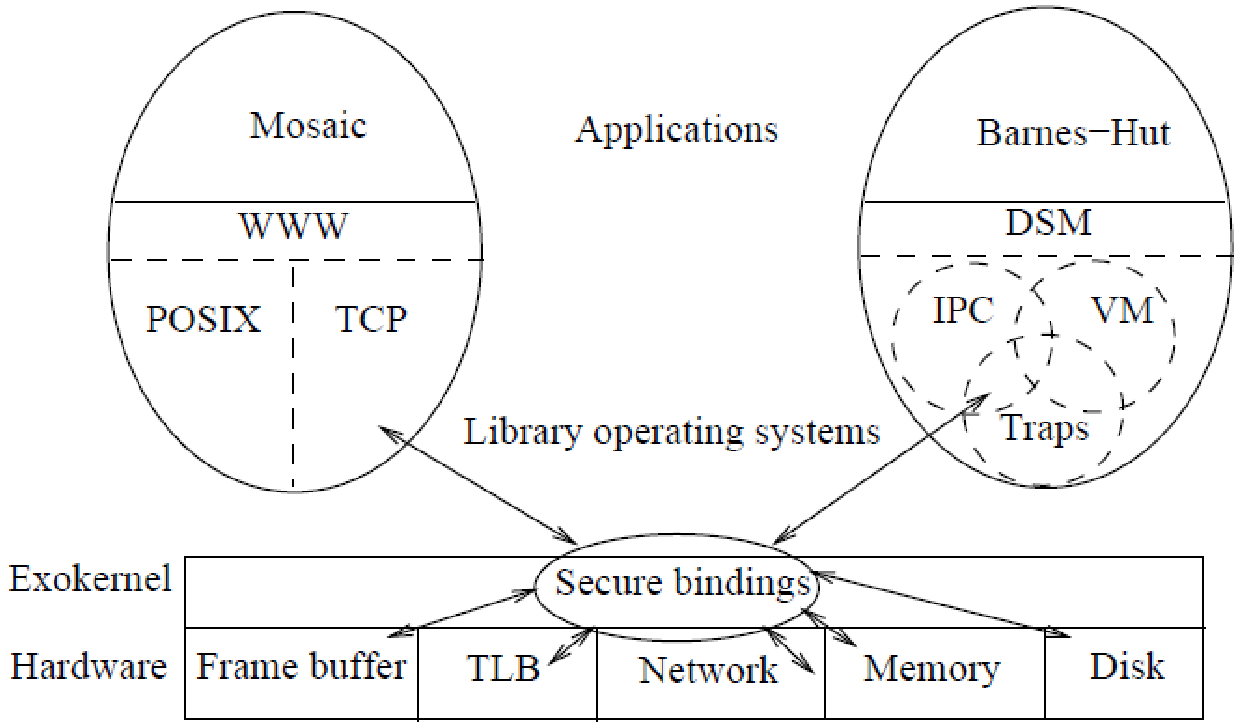
Virtual Machines

- The Java Virtual Machine
- Windows .Net Platform



Exokernels 外核

- Partitioning the actual machine, rather than cloning the actual machine
- developed by the MIT Parallel and Distributed Operating Systems group
- Hardware
- ExoKernel
- Library OSes
- Applications

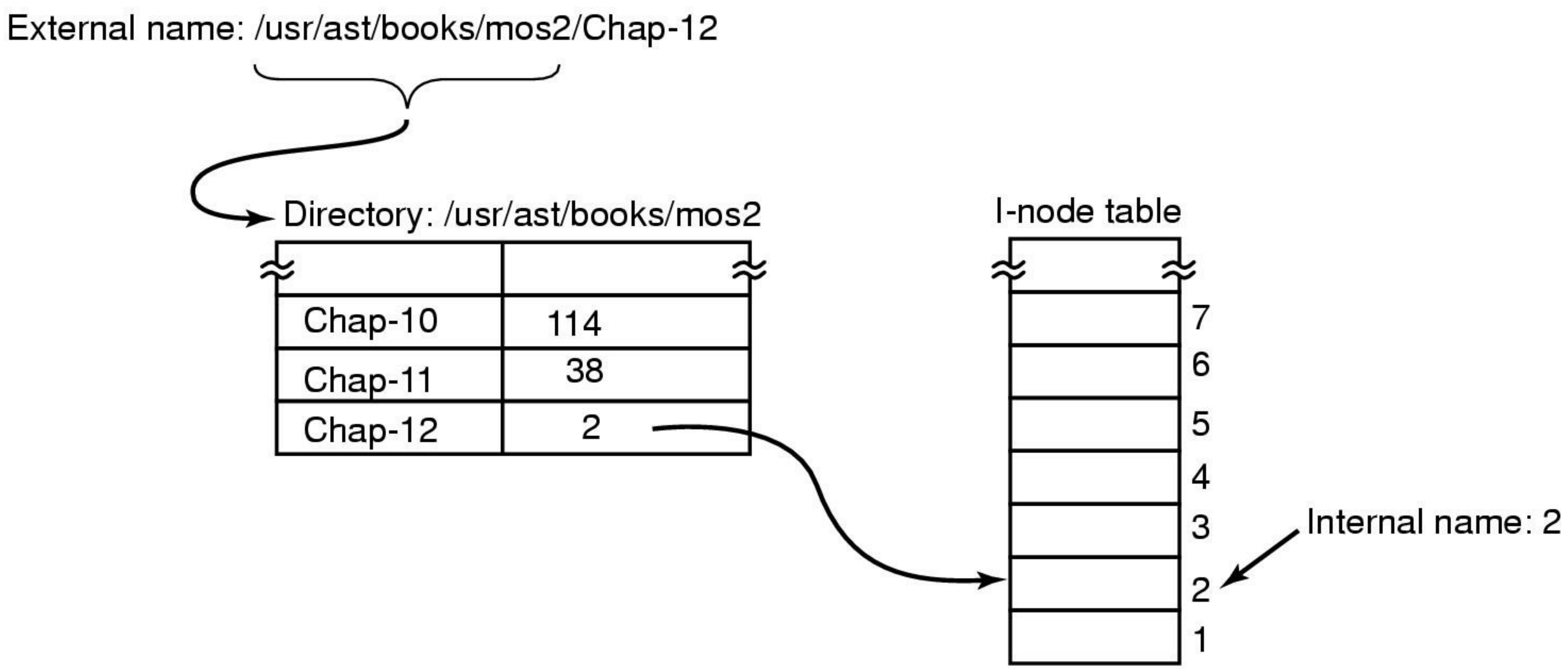


Engler D R , Kaashoek F M , Jr J O . Exokernel: an operating system architecture for application-level resource management[J]. 1995.

Runtime System Structure

- 传统的操作系统运行模式
  - 宏内核
  - 微内核
  - 管程结构 Monitor
  - 虚拟机模式
- 现代的操作系统运行模型
  - 客户 / 服务器模式 *
  - 对象模式
  - 多处理机模式

Naming



External name: /usr/ast/books/mos2/Chap-12
Directory: /usr/ast/books/mos2
Chap-10
114
Chap-11
38
Chap-12
2
I-node table
7
6
5
4
3
2
1
Internal name: 2

Directories are used to map external names onto internal names

Static Versus Dynamic Structures

Code for searching the process table for a given PID

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
        if (p->proc_pid == pid) {
                found = 1;
                break;
        }
}
```

Searching a static table for a pid

Implementation:Useful Techniques

- Hiding the Hardware*
- Indirection
- Reusability
- Reentrancy
- Brute Force
- Check for Errors First

Hiding the Hardware

- Example
  - 1. Interrupt
    - be turned into a pop-up thread instantly
    - convert an interrupt into an unlock operation on a mutex that the corresponding driver is waiting on
    - convert an interrupt into a message to some thread
  - 2. multiple hardware platforms*
    - CPU chip, MMU, word length, RAM size and other features that cannot easily be masked by the HAL or equivalent

Hiding the Hardware 1/2

```
#include "config.h"
init( )
{
#if (CPU == PENTIUM)
/* Pentium initialization here. */
#endif

#if (CPU == ULTRASPARC)
/* UltraSPARC initialization here. */
#endif
```

CPU-dependent conditional compilation

Hiding the Hardware 2/2

```
#include "config.h"
#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

Word-length dependent conditional compilation

Performance

All things being equal

- Why Are Operating Systems Slow?
- What Should Be Optimized?
- Space-Time Trade-offs
- Caching
- Hints
- Exploiting Locality
- Optimize the Common Case

Why Are Operating Systems Slow?

- 1. The slowness of many operating systems is to a large extent self-inflicted
  - MS-DOS, UNIX Version 7
  - Modern UNIX, Windows 2000
  - **Plug and play**
- 2.Scrap plug-and-play
  - An icon on the screen: Install new hardware
- 3.Adding new features and selective
  - Will the users like this?
  - code size, speed, complexity, reliability
- 4.Product marketing
  - A new version has all the features that are actually useful have been included and most of the people who need the product already have it

What Should Be Optimized?

- As a general rule, the first version of the system should be as straightforward as possible.
- The only optimizations should be things that are so obviously going to be a problem that they are unavoidable.
- A slogan
  - Good enough is good enough
  - once performance has achieved a reasonable level, it is probably not worth the effort and complexity to squeeze out the last few percent

Performance: Space-Time Trade-offs

- trade off time versus space
- A choice:
  - an algorithm that uses little memory but is slow
  - an algorithm that uses much more memory but is faster
- Example:
  - bit count
  - JPEG image compression
    - GIF
    - PostScript

Space-Time Trade-offs:bit count 1/2

A procedure for counting bits in a byte

```
#define BYTE_SIZE 8                          /* A byte contains 8 bits */
int bit_count(int byte)
{                                            /* Count the bits in a byte. */
    int i, count = 0;
    for (i = 0; i < BYTE_SIZE; i++)          /* loop over the bits in a byte */
        if ((byte >> i) & 1) count++;        /* if this bit is a 1, add to count */
    return(count);                           /* return sum */
}
```

(a)

A procedure to count the 1 bits in a byte

Space-Time Trade-offs:bit count 2/2

- A macro to count the bits
- A macro to access bit count in a table

```
/*Macro to add up the bits in a byte and return the sum. */
#define bit_count(b) (b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
                     ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1)
```

(b)

```
/*Macro to look up the bit count in a table. */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define bit_count(b) (int) bits[b]
```

(c)

(b) Macro to count the bytes
(c) Macro to look up the count

Space-Time Trade-offs:Compression

24 Bits

3,8,13	3,8,13	26,4,9	90,2,6
3,8,13	3,8,13	4,19,20	4,6,9
4,6,9	10,30,8	5,8,1	22,2,0
10,11,5	4,2,17	88,4,3	66,4,43

(a)

8 Bits

7	7	2	6
7	7	3	4
4	5	10	0
8	9	2	11

(b)

24 Bits

11	66,4,43
10	5,8,1
9	4,2,17
8	10,11,5
7	3,8,13
6	90,2,6
5	10,30,8
4	4,6,9
3	4,19,20
2	88,4,3
1	26,4,9
0	22,2,0

(c)

(a) Part of an uncompressed image with 24 bits per pixel

(b) Same part compressed with GIF, 8 bits per pixel

(c) The color palate

Performance:Caching

Path	I-node number
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

Part of an i-node cache

Performance: Hints

- A cache search may fail
- It is convenient to have a table of hints
- Examples:
  - 1. URLs embedded on Web pages
  - 2. Connection with remote files

Performance: Exploiting Locality

- Processes and programs do not act at random.
  - They exhibit a fair amount of locality in time and space, and this information can be exploited in various ways to improve performance.
- Example:
  - 1.Berkeley Fast File System
  - 2.Thread scheduling in multiprocessors
    - run each thread on the CPU it last used, in hopes memory cache

Performance: Optimize the Common Case

- It is a frequently a good idea to distinguish between the most common case and the worst possible case and treat them differently.
- Example:
  - 1. Entering a critical region
  - 2. Setting an alarm(using signals in UNIX)

操作系统的性能评价

- 系统效率
  - 体现系统效率的指标包括系统处理能力（吞吐量）、各种资源的使用效率以及响应时间等。
- 系统的可靠性 R 、可用性 A 和可维护性 S
  - MTBF 平均故障时间
  - MTTR 平均故障修复时间
  - R=MTBF
  - A=MTBF/(MTBF+MTTR)
  - S=MTTR
- 方便性
  - 提供易学、易使用、易开发的接口
- 可移植性
  - OS 能很好地适应不同的机器系列，即在硬件发生变化时，操作系统只需要做最小的改变就能在新的机器上运行。

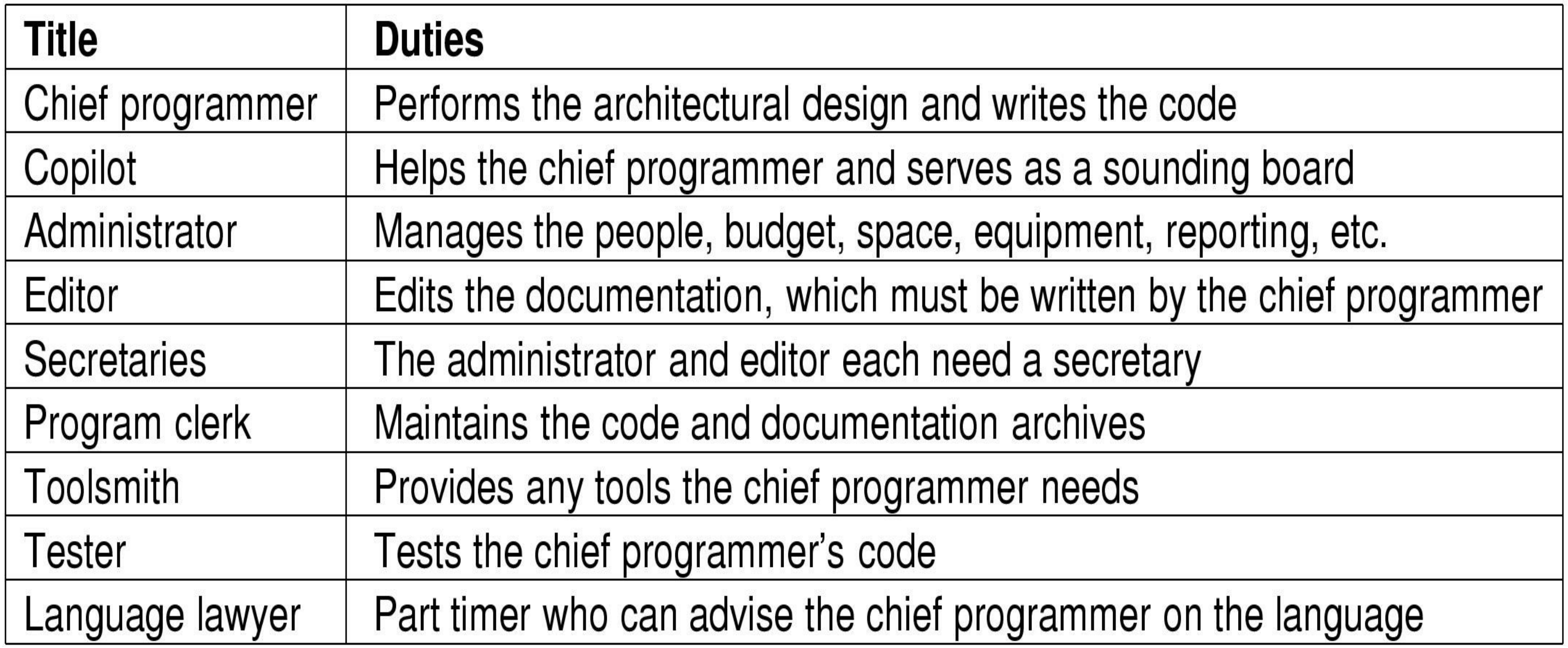
Project management

- The Mythical Man Month
- Team Structure
- The Role of Experience
- No silver Bullet

Project: The Mythical Man Month

- Brooks,1975,1995
- the work
  - 1/3 Planning 、 1/6 Coding
  - ¼ Module testing 、 ¼ System testing
- no such unit as a man-month
  - 15 people and 2 years
  - 360 people and one month
  - 1.the work cannot be fully parallelized
  - 2.the trade-off between people and months is far from linear on large projects (every module may potentially interact with every other module)
  - 3.Debugging is highly sequential

Project: Software team Structure



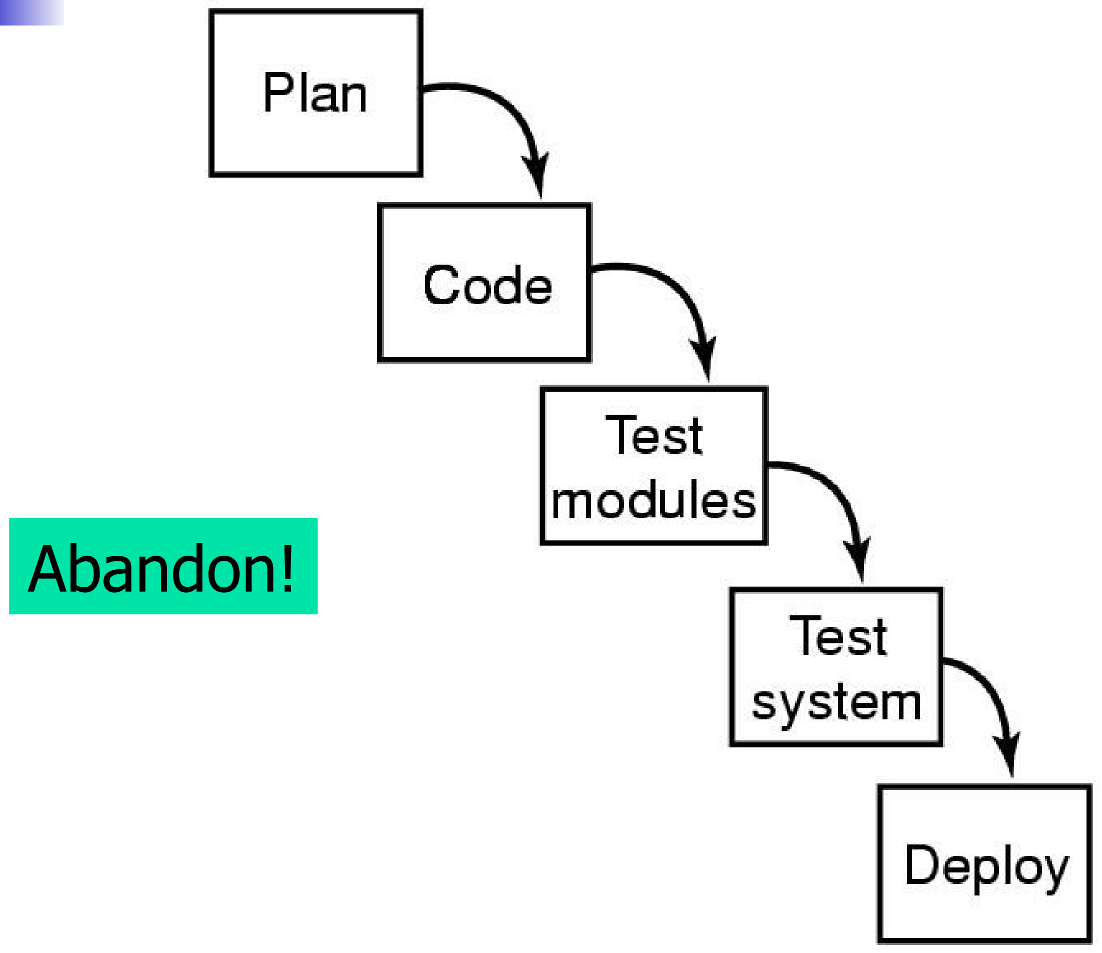
Title	Duties
Chief programmer	Performs the architectural design and writes the code
Copilot	Helps the chief programmer and serves as a sounding board
Administrator	Manages the people, budget, space, equipment, reporting, etc.
Editor	Edits the documentation, which must be written by the chief programmer
Secretaries	The administrator and editor each need a secretary
Program clerk	Maintains the code and documentation archives
Toolsmith	Provides any tools the chief programmer needs
Tester	Tests the chief programmer’s code
Language lawyer	Part timer who can advise the chief programmer on the language

1972, Mills’ proposal for populating a 10-person **chief programmer team**

Project: The Role of Experience

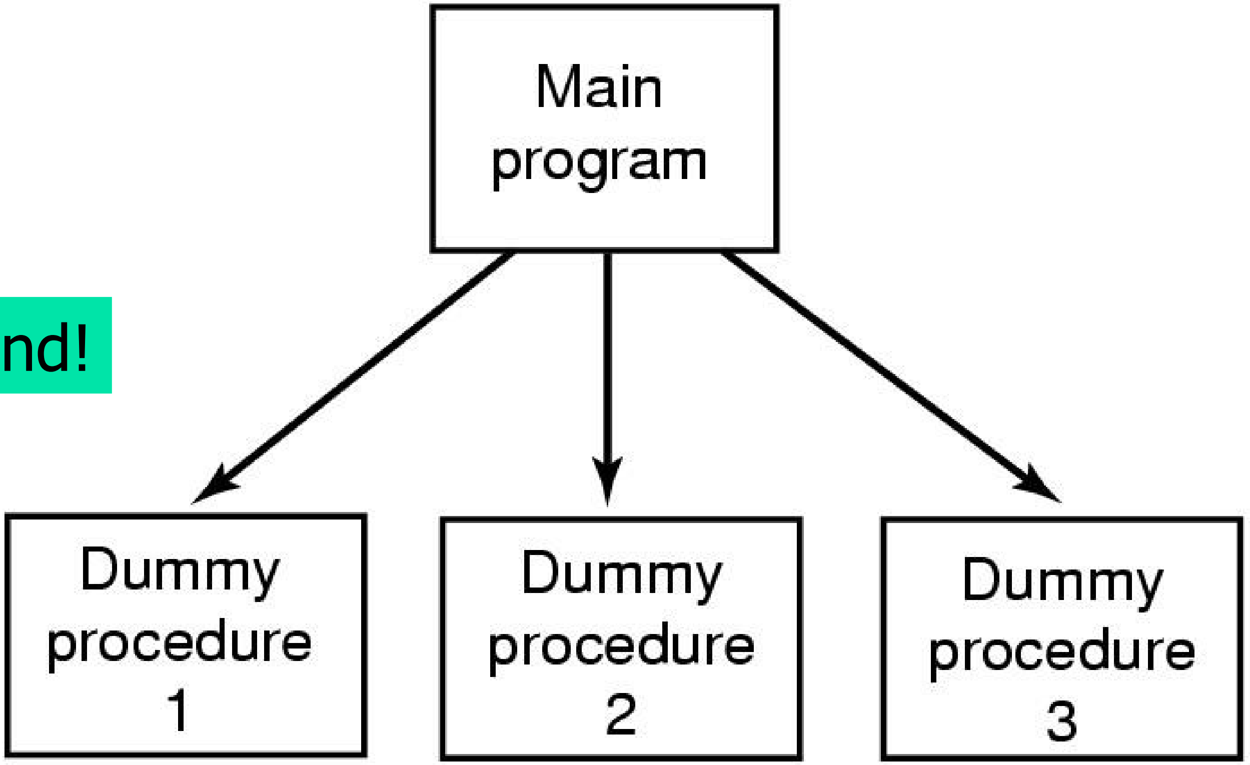
- Having experienced designers is critical to an operating systems project.
- Most of the errors are not in the code, but in the design.
- Brooks: the second system effect
    - CTSS -> MULTICS

Project: The Role of Experience



Traditional software design progresses in stages

Project: The Role of Experience



Recommend!

- Alternative design produces a working system
  - that does nothing starting on day 1

Project: No silver Bullet

- Brooks, 1987
  - None of the many nostrums being hawked by various people at the time was going to generate an order-of-magnitude improvement in software productivity within a decade.

操作系统的标准

- 没有标准化
  - 各个操作系统之间不能兼容
  - 大量应用软件无法在不同的操作系统平台上通用
  - 后果
    - 投资增加、开发周期加长、用户掌握困难、影响推广应用
- OS 国际化标准
  - POSIX 标准（ 1003 ）
    - ISO/IEC 9945-1:1990
    - Information Technology—Portable Operating System Interface
  - 同时被 X/Open 接纳为操作系统标准
    - 其中 9945-4 （即 POSIX 1003.4 ）为实时部分

操作系统理念的回归？

- 技术变化导致某些思想过时并迅速消失
- 技术的另一种变化还可能使它们复活
  - 磁盘上文件分配——连续文件
    - CD-ROM 文件系统
  - 动态链接（MULTICS 首先提出）
  - 计算服务（MULTICS, 以大量的、附有相对简单用户机器的、集中式 Internet 服务器形式回归）
  - IBM S/390 大机的复活
    - 三十年的改进，IBM S/390 已成为有高可靠性、可扩展性、及安全可用性的现代大型计算机系统
    - 采用面向对象程序设计、并行处理、分布式处理以及客户机/服务器技术，具有较强的互操作性、可移植性与可扩展性

Trends in OS design

Making predictions is always difficult, especially about the future.

Trends in OS design

- Large Address Space OS*
- Networking
- Parallel and Distributed Systems
- Multimedia
- Battery-Powered Computers
- Embedded Systems

Trends: Large Address Space OS

- 32-bit address spaces
- 64-bit address spaces
  - $2^{64}=2*10^{19}$
  - 3GB chunk/person
- what could we do with an address space of $2*10^{19}$ bytes?
  - eliminate the file system concept
  - a persistent object store
  - multiple processes run in the same address space at the same time, to share the objects
  - rethought virtual memory

References

- Books
  - Operating Systems: A Design-Oriented Approach, Crowley,1997
- Papers
  - Hints for Computer System Design, Lampson, 1984
  - On Building Systems that Will Fail, Corbato, 1991
  - End-to-End Arguments in System Design,Saltzer, 1984

Summary

- The nature of the design problem
- Stages of OS Design
- Interface design
- Implementation
- Design Pattern
- Performance
- Project management
- Trends in operating system design

Q&A

Thanks!