Contents

I Introduction

1. Overview

1.1 What Operating Systems Do

操作系统是一个用户与计算机硬件之间的中介程序. 目的: 运行程序, 简化解决问题. 同时重复利用硬件资源.

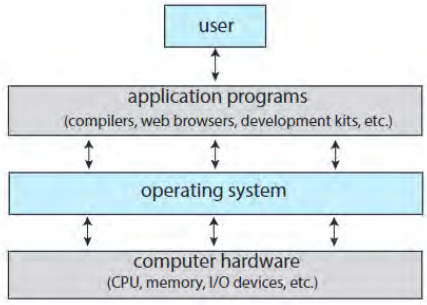


Figure I.1: Four Components of a Computer System

Operating System Definition OS 是资源管理器. 管理资源, 处理冲突以达到资源高效公平的利用. OS 是控制程序. 控制程序运行来防止错误与不适当的电脑使用.

kernel (内核), 始终正在运行的程序. 其他的要么是系统程序 (OS 提供的) 要么是应用程序.

Computer Startup 引导程序 (bootstrap program) 在重启或启动时加载. 其一般存储在 ROM or EPROM (固件, firmware).

1.2 Computer System Organization

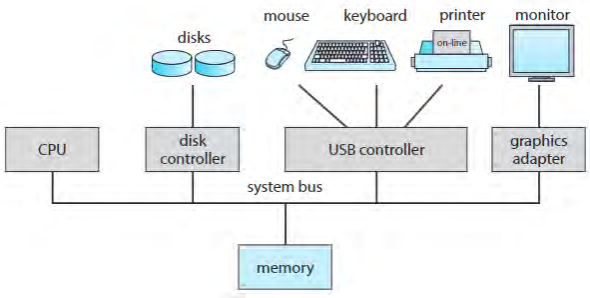


Figure I.2: A typical PC computer system

Computer-System Operation

1) I/O devices and the CPU can execute concurrently.

2) Each device controller is in charge of a particular device type.

3) Each device controller has a local buffer.

4) CPU moves data from/to main memory to/from local buffers

5) I/O is from the device to local buffer of controller.

6) Device controller informs CPU that it has finished its operation by causing an interrupt (via system bus).

concurrently 与 parallel 都是同时运行但是:

- concurrently 非真正同时处理, 但同时开始处理
- parallel 同时处理

Common Functions of Interrupts 触发 interrupt -> 通过 interrupt vector(存储 routine 起始地址) 查询 -> 运行 interrupt service routine (会保留 context, 通过记录 registers 与 program counter 实现)

interrupt 存在优先级, interrupt 可能会打断 interrupt.

interrupt 有分类. 例如 software-generated interrupt (又称 trap) 触发来源可能是 error 或者 user request (也称 **system call**. OS 分为 user code 与 system code, system call 是从 user code 调用 system code). 又例如 hard interrupt.

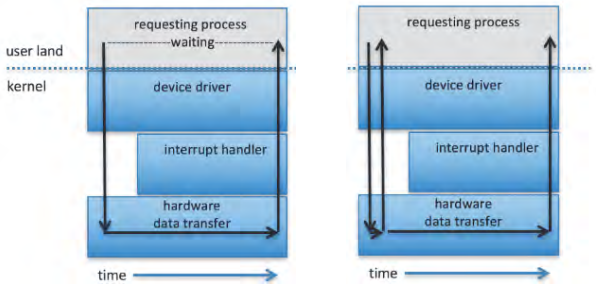
OS 是通过 interrupt 驱动的. 好处有: 封装, 管理等.

在 RISCV 中, interrupt 称为 traps, 分为 interrupt 与 exceptions/ecalls, 分别表示 hard interrupt 与 software interrupt.

I/O Structure Two I/O Methods: 在 I/O 开始后,

1) (Synchronous 同步, block 阻塞) 等待 I/O 完成后再继续运行 user code

2) (Asynchronous 异步, non-block 非阻塞) 不等待 I/O 完成直接继续运行 user code



(a) Synchronous (b) Asynchronous

Figure I.3: I/O Methods

Direct Memory Access Structure CPU 以 block 为单位进行处理 I/O.

Storage Hierarchy 成本问题, 分层处理, 考虑: 速度, 成本, 变更率.

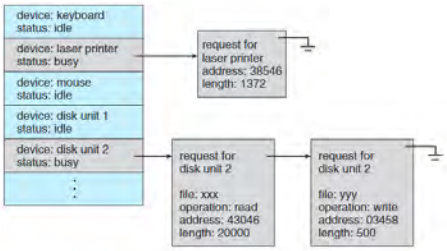


Figure I.4: Device-Status Table (存储设备的状态)

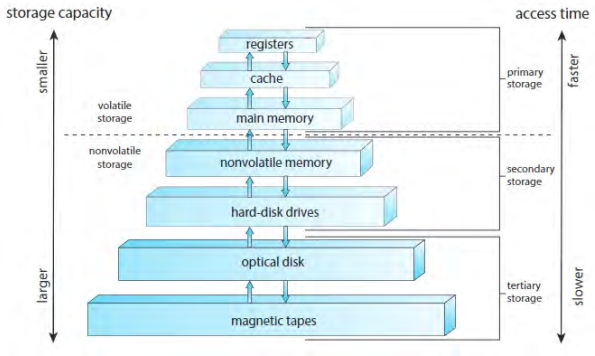


Figure I.5: Storage Hierarchy

Caching 用更高速的设备缓存更低速的设备中的数据的一部分.

可以解决速度读取速率不匹配的问题.

1.3 Computer-System Architecture

Multiprocessor Systems 现在的趋势.

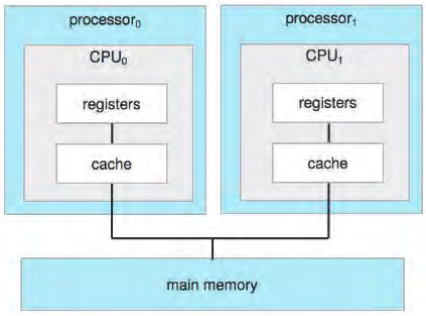


Figure I.6: Multiprocessor Systems

2. Structure

2.1 Operating-System Structure

Multiprogramming (多道程序设计) 可在一个 CPU 上运行多个程序, 提高 CPU 利用率.

Timesharing (分时系统)(multitasking) 有更好的 interactivity. 计算机可同时服务多个用户, 运行多个任务. 有 swapping 与 virtual memory 机制.

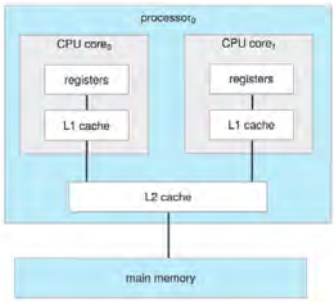


Figure I.7: Multicore Systems

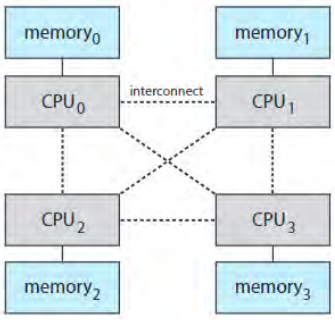


Figure I.8: NUMA multiprocessing architecture

2.2 Operating-System Operations

有两个模型: user mode and kernel(supervisor) mode 或可能存在 machine mode. 使用硬件提供的 mode bit 控制.

有些特权指令只能在 kernel mode 下运行.

Isolation, 模组之间需要有的隔离.

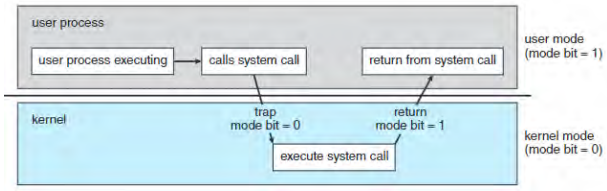


Figure I.9: Transition from User to Kernel Mode

软中断, trap.

3. Main parts

3.1 Process Management

Program is a passive entity, process is an active entity.

process(进程) 是资源的集合, 资源有: CPU, memory, I/O, files, initialization data. 进程中止会回收所有资源.

单线的进程有 program counter (pc), 指明了下一条运行指令的位置.

多线程的进程每个 thread (线程) 有一个 pc.
multiplexing (多路复用) the CPUs.

Activities :

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

3.2 Memory Management

所有数据在 processing 之前与之后需要在 memory 之中.
所有 instructions 需要在 memory 之中.

Activities :

- Keeping track
- Allocating (分配)
- data 在 memory 的进出

3.3 Storage Management

磁盘数据最小单位: files. file system 管理文件.
Mass-Storage Management 海量存储管理.

3.4 I/O Subsystem

OS 的目的之一是 hide peculiarities(歧义) of hardware devices.

I/O subsystem responsible for

- 高效
- I/O 管理统一
- 驱动特定的硬件

4. OS Purposes

- Abstraction
- Multiplex
- Isolation
- Sharing
- Security
- Performance
- Range of uses

II Operating-System Structure

1. Operating System Services

For user

- User interface (UI): Command-Line(CLI), Graphics User Interface (GUI)
- Programe execution
- I/O operations
- File-system manipulation
- Communications
- Error detection

For system

- Resource allocation
- Accounting
- Protection and security

2. System Call

使用高级语言写 system call.

Application Program Interface (API) 与 system call 看作是不同的, API 层级更高.

POSIX API 标准化了 API.

2.1 Implementation

每个 system call 会有一个 number. System-call interface 维护一个 number 索引的 table.

封装好处: 抽象, 可移植 etc.

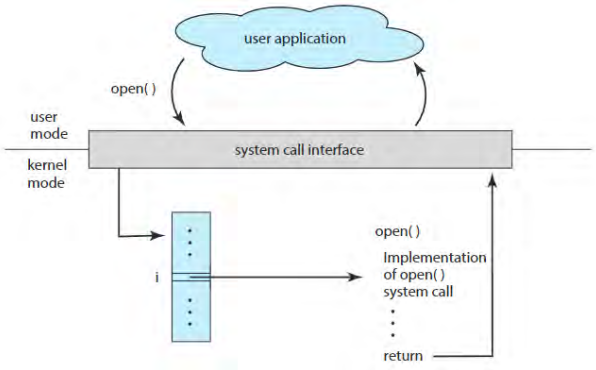


Figure II.1: API –System Call –OS Relationship

2.2 Parameter Passing

Three general methods used to pass parameters to the OS:

- Simplest: pass the parameters in **registers**. 但有时不够用.

- Parameters stored in a **block**, or table, in memory, and address of block passed as a parameter in a register.
- Parameters placed, or pushed, onto the **stack** by the program and popped off the stack by the operating system.

Block and stack methods do not limit the number or length of parameters being passed.

3. Types of System Calls

- Process control
- File management
- Device management
- Information maintenance (e.g. time, date)
- Communications
- Protection

4. System Programs

System programs provide a convenient environment for program development and execution. They can be divided into:

- File manipulation
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Application programs

5. Operating System Design and Implementation

OS 设计与实现没有明确输入输出. Start by defining goals and specifications. Affected by choice of hardware, type of system.

- User goals — operating system should be convenient to use, easy to learn, reliable, safe, and fast
- System goals — operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

5.1 Mechanism and Policy

Important principle to separate: (分离策略与机制)

- Policy: What will be done? 策略（what to do）
- Mechanism: How to do it? 机制（how to do）

6. Operating System Structure

6.1 Layered Approach

仅帮助人类理解.

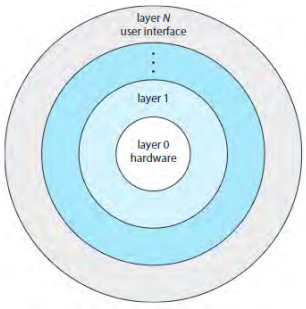


Figure II.2: Layered Operating System

6.2 Monolithic structure

(宏内核) e.g. UNIX

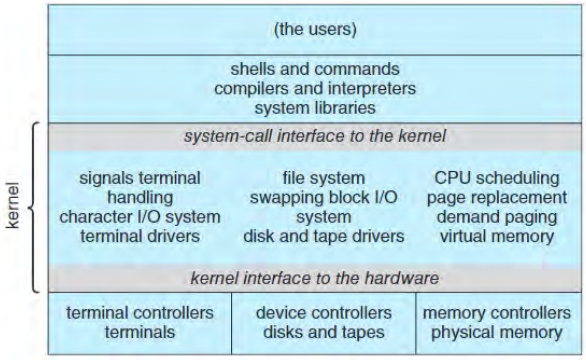


Figure II.3: Traditional UNIX system structure

6.3 Microkernel System Structure

(微内核) 把很多操作分到 user code 之中. Benefits: 代码少, 稳定, 易移植. Detriments: 效率变低 (Performance overhead).

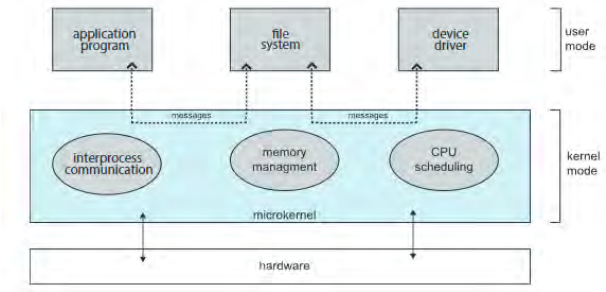


Figure II.4: Architecture of A Typical Microkernel

6.4 Kernel Modules

(内核模块) 现代常用. 提供扩展内核的能力 (loadable kernel modules, LKM).

6.5 Other Structures

Exokernel (外核) 高度简化 kernel, 只负责资源分配, 提供了低级的硬件操作, 必须通过定制 library 供应用使用. 高性能, 但定制化 library 难度大, 兼容性差.

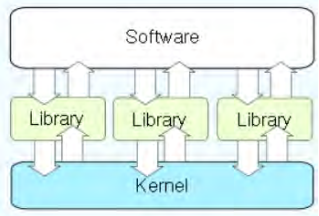


Figure II.5: Exokernel

Unikernel (单核), 来自 include OS. 静态连接所有的 OS 代码. 适用于云服务, app 的 boots, 启动耗时只需几十 ms. 达到**高可用**.

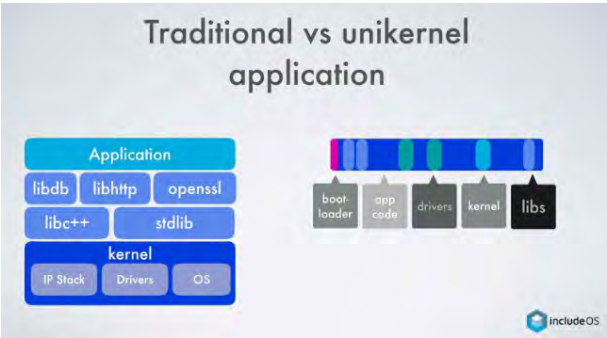


Figure II.6: Unikernel

7. Virtual Machines

(虚拟机) 虚拟机算是分层方法的自然结果. 其把底层的硬件与 OS 都视为自己的硬件. 有全虚拟化 (解释执行) 与物理虚拟化 (底层执行).

有两种:

1) bare-metal Hypervisor (裸金属): 在硬件上架构 Hypervisor, 然后架构虚拟机, 性能高.

2) Hosted Hypervisor: 在 OS 上架构 Hypervisor 后架构虚拟机.

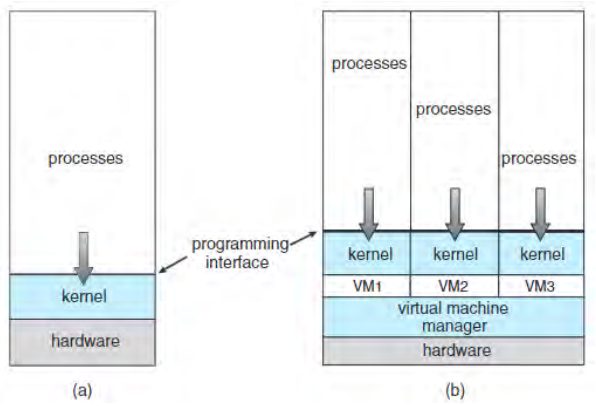


Figure II.7: A computer running (a) a single operating system and (b) three virtual machines.

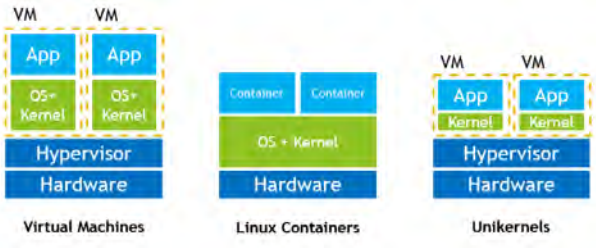


Figure II.8: Different Techniques

III Processes

1. Process Concept

1.1 Concept

又称 jobs/tasks/process. Process 是一个运行的实例.

A process includes:

- text section (code, 代码段)
- data section (global vars, 数据段)
- stack (function parameter, local vars, return addresses, 栈)
- heap (dynamically allocated memory, 堆)

heap 增长需要 OS 操作, stack 是已分配的一块空间, 在其内增长, 超过触发 stack overflow. stack 与 heap 之间的区域称为 hole, 占据超过 90% 的空间.

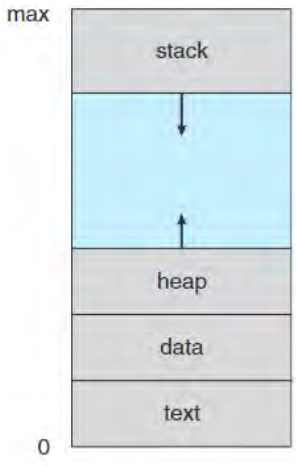


Figure III.1: Layout of a process in memory

kernel stack 与程序的 stack 不是一个. 每个 process 对应一个 kernel stack.

1.2 Process State

As a process executes, it changes state:

- new: The process is being created
- running: Instructions are being executed
- waiting: The process is waiting/blocked for some event to occur
- ready: The process is waiting to be assigned to a processor
- terminated: The process has finished execution

即使 cpu 空闲仍会有 idle 进程一直在运行, 所以无论如何都需要 ready 缓冲.

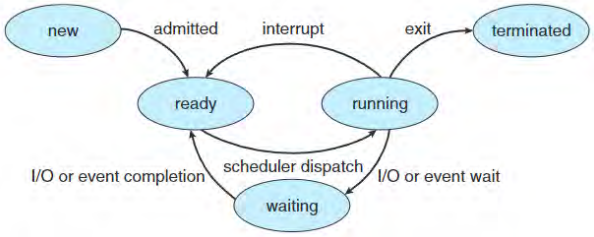


Figure III.2: Diagram of process state transition

1.3 Process Control Block (PCB)

Information associated with each process:

- Process state
- Program counter
- Contents of CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

这些由 PCB 存储.

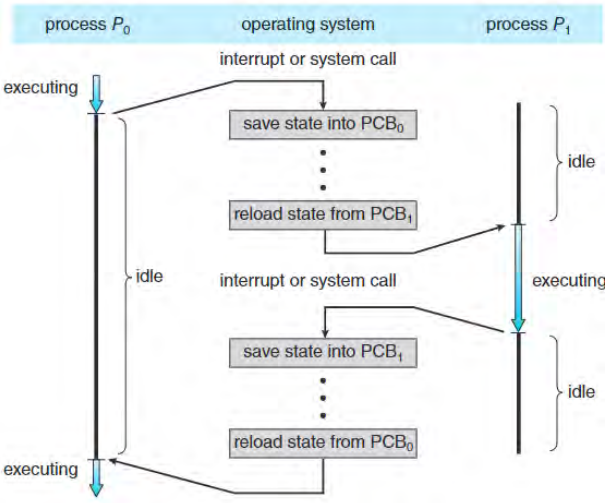


Figure III.3: Diagram showing context switch from process to process

二者 idle 相交称为 overhead, 是为了 multiprocessing 留的冗余.

2. Process Scheduling

- Job queue — set of all processes in the system
- Ready queue — set of all processes residing in main memory, ready and waiting to execute

- Device queues — set of processes waiting for an I/O device

Processes migrate(迁移) among the various queues. 一个 process 仅能在一个 queue 中.

2.1 Schedulers

- Long-term scheduler (or job scheduler) (过时了, 现在由用户决定, 是 user scheduler) — selects which processes should be brought into memory (the ready queue). 控制着 the degree of multiprogramming.
- Short-term scheduler (or CPU scheduler) — selects which process should be executed next and allocates CPU. 频繁触发, 为了保证系统的交互性.

UNIX and Windows do not use long-term scheduling

Medium Term Scheduling 需要时将 PCB 写入磁盘 (swap out), 然后合适时恢复 PCB 到 ready queue (swap in).

Processes can be described as either:

- I/O-bound process (IO 绑定进程) — spends more time doing I/O than computations, many short CPU bursts
- CPU-bound process (CPU 绑定进程) — spends more time doing computations; few very long CPU bursts

2.2 Context Switch

当切换进程时, 需要保存当前进程的上下文, 并加载切换进程的上下文. swtch 是特殊的命令, 不是拼写错误.

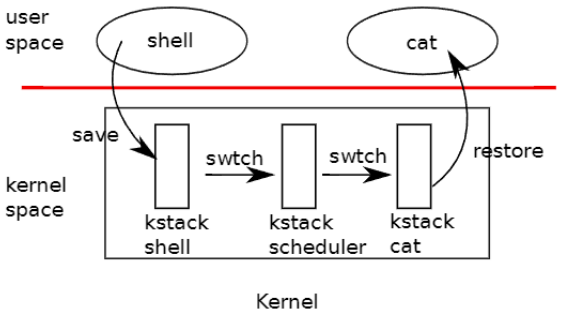


Figure III.4: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

3. Operations on Processes

3.1 Process Creation

Parent process creates children processes, which, in turn create other processes, forming a tree of processes. Parent and children share all resources. Child duplicate of parent.

使用 fork() 产生子进程, 基本相当于复制进程.

wait() 等待子进程的完成, 会返回信息.

exec() 运行另外的程序, 抹杀当前的程序.

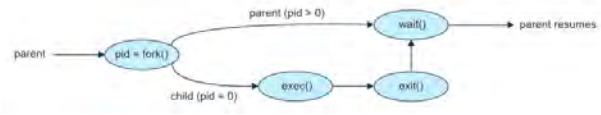


Figure III.5: Process creation using the fork() system call

3.2 Process Termination

exit() 释放进程占用资源, 并通知子进程, the child gets orphaned and its parent becomes the "init" process (PID=1).

abort()

4. Cooperating Processes

有 Independent process 与 Cooperating process.

Advantages of process cooperation:

- Information sharing
- Computation speed-up (Multiple CPUs)
- Modularity
- Convenience

4.1 Producer-Consumer Problem

经典的合作模型, 可描述调用关系. 生产者产生信息, 消费者消费信息. 有两种:

- unbounded-buffer: 信息相比 buffer 微不足道.
- bounded-buffer: 信息相比 buffer 不能忽略.

以 Bounded-Buffer 为例:

- Shared-Memory Solution: Shared data. 共享数据.

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Insert() Method

```
while (true) {
    Produce an item;
    while ((in + 1) % BUFFER_SIZE == out)
        ; /* do nothing -- no free buffers */
```

```
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

- Remove() Method

```
while (true) {
    while (in == out)
        ; //do nothing, nothing to consume
    Remove an item from the buffer;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
```

5. Interprocess Communication (IPC)

进程间进行通讯.

Two models for IPC: message passing and shared memory.

Message-passing facility provides two operations:

- send(message) –message size fixed or variable
- receive(message)

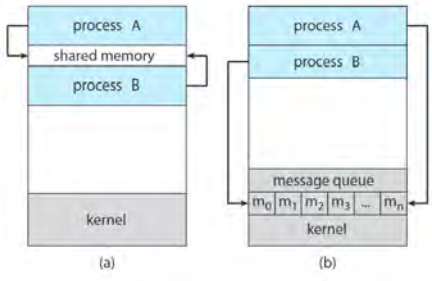


Figure III.6: Communications models. (a) Shared memory. (b) Message passing.

5.1 Direct Communication

Processes must name each other explicitly:

- **send** (P, message) –send a message to process P
- **receive**(Q, message) –receive a message from process Q

5.2 Indirect Communication

Messages are directed and received from **mailboxes** (also referred to as **ports**)

- Each mailbox has a unique id (well-known ID)
- Processes can communicate only if they share a mailbox

Operations:

- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox

Primitives are defined as:

- **send** (A, message) –send a message to mailbox A
- **receive**(A, message) –receive a message from mailbox A

5.3 Synchronization

Message passing may be either blocking or non-blocking.
Blocking is considered synchronous
Non-blocking is considered asynchronous

5.4 Buffering

Queue of messages attached to the link; implemented in one of three ways

1) Zero capacity –0 messages
   Sender must wait for receiver
2) Bounded capacity –finite length of n messages
   Sender must wait if link full
3) Unbounded capacity –infinite length
   Sender never waits

IV Threads

1. Overview

peek: A lot Checks, not efficient. And still not responsive!

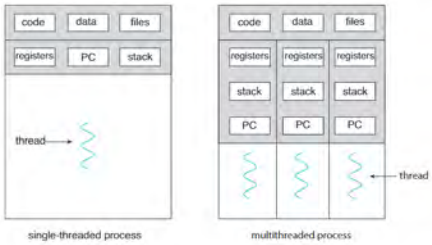


Figure IV.1: Single-threaded and multithreaded processes

Benefits:

- Responsiveness: interactive applications
- Resource Sharing: memory for code and data can be shared.
- Economy: creating processes are more expensive.
- Utilization of MP Architectures: multi-threading increases concurrency

1.1 User Threads

Thread management done by user-level threads library.
Three primary thread libraries:

- POSIX Pthreads (can also be provided as system library)
- Win32 threads
- Java threads

1.2 Kernel Threads

Supported by the Kernel. Almost all contemporary OS implements kernel threads.

2. Multithreading Models

- Many-to-One: thread mgmt is efficient, but will block if making system call, kernel can schedule only one thread at a time (legacy system(kernel))
- One-to-One: more concurrency, but creating thread is expensive
- Many-to-Many: flexible

2.1 Many-to-One Model

Many user-level threads mapped to single kernel thread. 改造老系统, 提供多线程的方式.

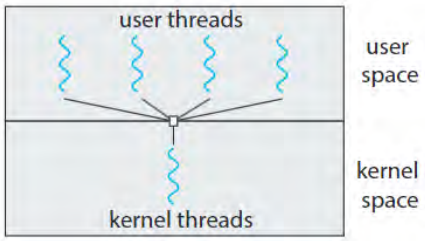


Figure IV.2: Many-to-one model

2.2 One-to-One Model

Each user-level thread maps to kernel thread

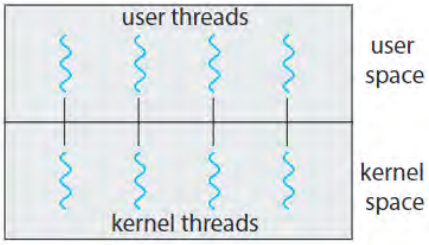


Figure IV.3: One-to-one model

2.3 Many-to-Many Model

Allows many user level threads to be mapped to many kernel threads
Lightweight Process ID

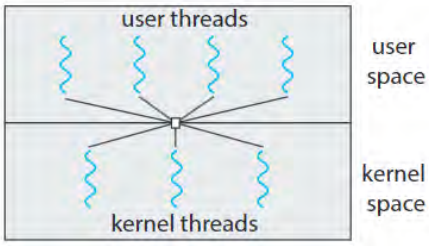


Figure IV.4: Many-to-many model

2.4 Two-level Model

Similar to M:M, except that it allows a user thread to be bound to kernel thread

3. Threading Issues

3.1 Semantics of fork() and exec()

Does fork() duplicate only the calling thread or all threads? 可以复制全部也可以仅复制本线程.
exec() will replace the entire process.

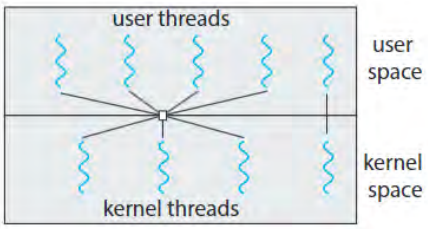


Figure IV.5: Two-level Model

3.2 Signal Handling

signal handler

3.3 Thread Cancellation

Terminating a thread before it has finished. Two general approaches:

- Asynchronous cancellation
- Deferred cancellation

3.4 Thread Pools

Create a number of threads in a pool where they await work

3.5 Thread Specific Data

Thread-Local Storage (TLS). Allows each thread to have its own copy of data

3.6 Scheduler Activations

调度激活. allows an application to maintain the correct number of kernel threads

V CPU Scheduling

1. Basic Concepts

Maximum CPU utilization obtained with multiprogramming.

CPU—I/O Burst Cycle CPU bursts 远短于 I/O bursts.

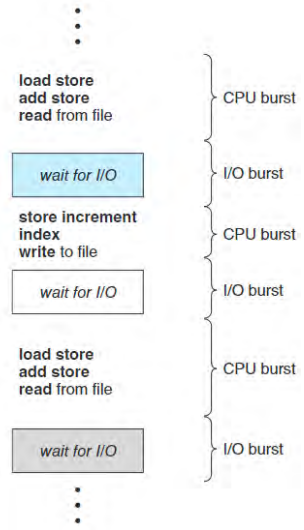


Figure V.1: Alternating sequence of CPU and I/O bursts

CPU bursts time 大致在都在 8ms 以内.

1.1 CPU Scheduler

CPU scheduling decisions may take place when a process:

1) switches from the running state to the waiting state
2) switches from the running state to the ready state
3) switches from the waiting state to the ready state
4) terminates

Scheduling under 1 and 4 is non-preemptive. All other scheduling is preemptive.

1.2 Dispatcher

involves:

- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program

2. Scheduling Criteria

- CPU utilization (CPU 利用率)— keep the CPU as busy as possible
- Throughput (吞吐率)— # of processes that complete their execution per time unit
- Turnaround time (周转时间)— amount of time to execute a particular process

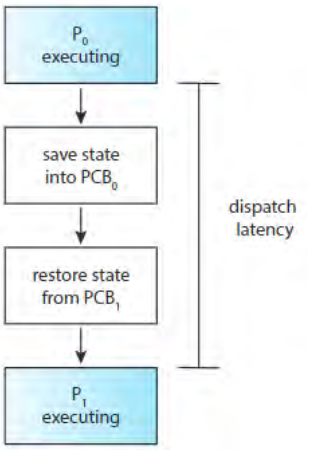


Figure V.2: The role of the dispatcher

- Waiting time (等待时间)— amount of time a process has been waiting in the **ready queue** (不算 I/O 时的时间)
- Response time (响应时间)— amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

- 响应时间 = 第一次响应 - 到达时间
- 周转时间 = 结束时刻 - 到达时间
- 等待时间 = 周转时间 - 运行时间

2.1 Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

3. Scheduling Algorithms

The Gantt Chart

3.1 First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

1) Suppose that the processes arrive in the order: P_1, P_2, P_3

Average waiting time: $(0 + 24 + 27)/3 = 17$

2) Suppose that the processes arrive in the order: P_2, P_3, P_1

Average waiting time: $(6 + 0 + 3)/3 = 3$

Convoy effect short process behind long process.

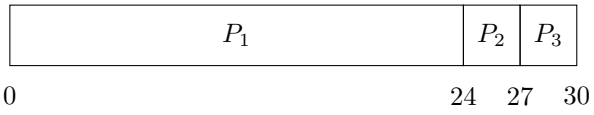


Figure V.3: FCFS Scheduling Example 1

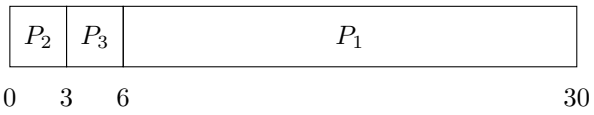


Figure V.4: FCFS Scheduling Example 2

3.2 Shortest-Job-First (SJF) Scheduling

与进程的 burst time 长度相关, 选取长度最小的进程.

SJF 是平均等待时间最优的算法, 但缺点是需要知道所有进程的 burst time, 这几乎不可能.

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Example of Non-Preemptive SJF 进程不能被打断

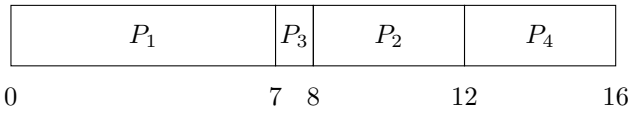


Figure V.5: Non-Preemptive SJF Example

Average waiting time= $(0 + 6 + 3 + 7)/4 = 4$
(arrival → start running)

Example of Preemptive SJF 进程可以被打断

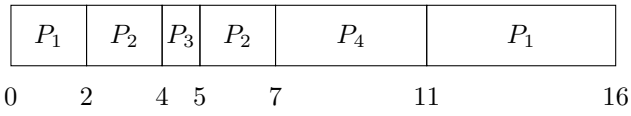


Figure V.6: Preemptive SJF Example

Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$

3.3 Determining Length of Next CPU Burst

Unfortunately, no way to know the length of the next burst. 所以使用先前的 cpu burst 估计长度, using exponential averaging:

1) t_n: actual length of nth CPU burst
2) τ_{n+1}: predicted value for the next CPU burst
3) α: $0 \geq \alpha \geq 1$
4) Define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

之前的权重会越来越小.

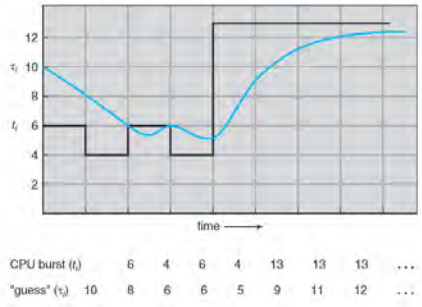


Figure V.7: Prediction of the length of the next CPU burst

3.4 *Priority Scheduling*

smallest integer ≡ highest priority.

Problem: Starvation –low priority processes may never execute

Solution: Aging –as time progresses increase the priority of the process

相当于使用权重当作 burst time 估计的 SJF.

3.5 *Round Robin (RR)*

Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

Performance:

- q large → FIFO
- q small → q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

平均等待时间更长, 但响应更好.

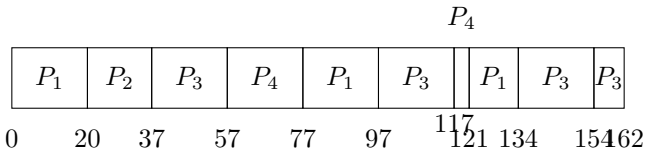


Figure V.8: RR Example

3.6 *Multilevel Queue*

Ready queue is partitioned into separate queues:

- foreground (interactive)
- background (batch)

Each queue has its own scheduling algorithm. Scheduling must be done between the queues.

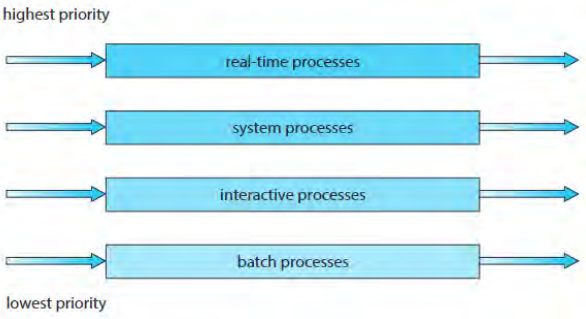


Figure V.9: Multilevel queue scheduling

3.7 *Multilevel Feedback Queue*

A process can move between the various queues; aging can be implemented this way

Multilevel-feedback-queue scheduler defined by the following parameters:

1) number of queues
2) scheduling algorithms for each queue
3) method used to determine when to upgrade a process
4) method used to determine when to demote a process
5) method used to determine which queue a process will enter when that process needs service

Example Three queues:

1) Q_0: RR with time quantum 8 milliseconds
2) Q_1: RR time quantum 16 milliseconds
3) Q_2: FCFS

Scheduling:

- A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.

- At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

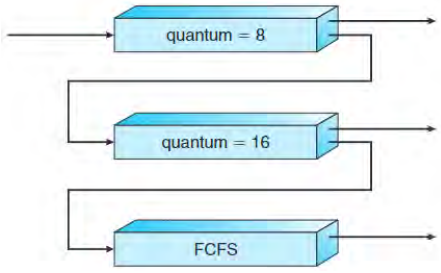


Figure V.10: Multilevel feedback queues

4. Multiple-Processor Scheduling

- Asymmetric multiprocessing: 只有一个处理器访问系统数据结构, 从而减轻了对数据共享的需要; 其他人只执行用户代码.
- Symmetric multiprocessing (SMP): 每个处理器都是自调度的. 多个处理器可能访问和更新一个公共数据结构.

5. Real-Time Scheduling

- Hard real-time systems: 在保证的时间内完成所需关键任务
- Soft real-time computing: 要求关键流程优先于其他的流程

6. Thread Scheduling

Also known as the Contention Scope.

- Local Scheduling (Process-Contention Scope): 线程库如何决定将哪个线程放入可用的 LWP
- Global Scheduling (System-Contention Scope): 内核如何决定下一个运行的内核线程

VI Process Synchronization

1. Background

Concurrent access to shared data may result in data inconsistency. (共享数据上的并发访问导致的不同步)

```
while (true) {
    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Code 1: Producer

```
while (true) {
    while (count == 0)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /* consume the item in nextConsumed */
}
```

Code 2: Consumer

1.1 Race Condition(竞态条件)

多个进程对寄存器中的内容进行并发的访问会发生竞态条件.

e.g. 当 count 同时 $++$ $--$, 其结果会不一致.

Definition VI.1 (Race Condition) *A race condition is a situation in which a memory location is accessed concurrently, and at least one access is a write.*

2. The Critical-Section Problem

To design a protocol that the processes can use to cooperate

```
do{
    Entry section;
    Critical section; // 临界区段
    Exit section;
    Remainder section;
}while(TRUE);
```

Code 3: General structure of a typical process P_j

仅在 Critical section 中修改 register. 细粒度的 critical section 并发性更好.

2.1 Solution to Critical-Section Problem

1) Mutual Exclusion (互斥): 对一个 critical section, 仅有一个相关进程可以运行.

2) Progress (空闲让进): If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely (当无进程处于临界区，可允许一个请求进入临界区的进程立即进入自己的临界区)

3) Bounded Waiting (有限等待): 进程运行 entry 请求进入临界区, 使用 exit 退出. 在请求后, 其他进程进入临界区的次数是有限的, 即等待不会饿死这个进程.

3. Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic
- The two processes share two variables
  - int turn: indicates whose turn it is to enter the critical section
  - Boolean flag[2]: is used to indicate if a process is ready to enter the critical section.

```
while (true) {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
        CRITICAL SECTION;
    flag[i] = FALSE;
    REMAINDER SECTION;
}
```

Code 4: The Algorithm for Process P_i

但遇到指令重排会失效, 可在 flag[i]=ture 后增加个内存屏障, 保证 flag 都处理完才能进行下面的指令.

4. Synchronization Hardware

Many systems provide hardware support for critical section code.

Uniprocessors – could disable interrupts. 在临界区关中断, 出了后再打开.

Modern machines provide special atomic hardware instructions.

4.1 Solution using TestAndSet

```
bool TestAndSet (bool *target){
    bool rv = *target;
    *target = TRUE;
    return rv:
}
```

Code 5: TestAndSet Instruction

Shared boolean variable lock., initialized to false.

```
while (true) {
    while(TestAndSet(&lock))
        ; /* do nothing */
    critical section;
    lock = FALSE;
    remainder section;
}
```

Code 6: Solution using TestAndSet

4.2 Solution using Swap

```
void Swap (boolean *a, boolean *b){
    bool temp = *a;
    *a = *b;
    *b = temp:
}
```

Code 7: Swap Instruction

```
while (true) {
    key = TRUE;
    while (key == TRUE)
        Swap (&lock, &key );
    critical section;
    lock = FALSE;
    remainder section;
}
```

Code 8: Solution using Swap

4.3 Solution with Compare and Swap

```
int compare_and_swap(int *value, int expected, int
↪ new_value){
    int tmp = *value;
    if(*value == expected)*value = new_value;
    return tmp;
}
```

Code 9: Compare and Swap Instruction

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
    critical section;
    lock = 0;
    remainder section;
}
```

Code 10: Solution with CAS

不满足 Bounded Waiting. 可以解决但需要更改逻辑, 加个检查

5. Solution with Mutex Locks

Has atomic operations acquire and release.

```
void acquire(){
    while(!available)
        ; /* busy wait */
    available=false;
}

void release(){
    available=true;
}
```

Code 11: acquire and release

```
while(true){
    acquire();
    critical section;
    release();
    remainder section;
}
```

Code 12: Solution with mutex lock

自旋锁 (spin lock), 忙等待 (busy waiting).

6. Semaphores(信号量)

- Semaphore S – integer variable
- Two atomic standard operations modify S: wait() and signal().

  Originally called P() and V().
- Can only be accessed via two indivisible (atomic) operations:

```
struct Semaphore{
    int S;
public:
    S(int _S)S(_S){}
    void wait(){
        while(S<=0)
            ; // no-op
        S--;
    }
    void signal(){
        S++;
    }
};
```

Code 13: Semaphores definition

6.1 Usage as General Synchronization Tool

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement (mutex locks)
- Counting (计数) semaphore – integer value can range over an unrestricted domain

Provides mutual exclusion

```
Semaphore S(1); // initialized to 1
S.wait();
Critical Section;
S.signal();
```

Code 14: Semaphore Usage

6.2 Semaphore Implementation

with Busy waiting wait 与 signal 也成为了 critical section. 需要保护. 这属于 busy Waiting

with no Busy waiting With each semaphore there is an associated waiting queue. Each semaphore has two data items:

- value
- pointer to a linked-list of PCBs.

Two operations (provided as basic system calls):

1) block - place the process invoking the operation on the appropriate waiting queue.

2) wakeup – remove one of processes in the waiting queue and place it in the ready queue.

可能导致 lost wake-up problems

6.3 Deadlock and Starvation

- Deadlock
- Starvation

7. Classic Problems of Synchronization

7.1 Bounded-Buffer Problem

7.2 Readers-Writers Problem

7.3 Dining-Philosophers Problem

8. Monitors(管程)

A high-level abstraction that provides a convenient and effective mechanism for process synchronization.

Only one process may be active within the monitor at a time.

8.1 Condition Variables

Two operations on a condition variable:

- x.wait(): a process that invokes the operation is suspended.
- x.signal(): resumes one of processes (if any) that invoked x.wait ()

8.2 Solution to Dining Philosophers

9. Pthreads Synchronization

VII Deadlocks

1. The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

e.g. semaphores A and B, initialized to 1

P_0	P_1
wait(A)	wait(B)
wait(B)	wait(A)

死锁由资源的共享造成.

2. System Model

Resource types $R_1, \ldots, R_m$. Each resource type R_i has W_i instances. Each process utilizes a resource as follows:

- request
- use
- release

3. Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

1) Mutual exclusion(互斥性): 一个 resource 只能同时被一个 process 使用.
2) Hold and wait: 进程手里至少有一个 resource, 且等待其他进程的 resource.
3) No preemption: 不能抢占
4) Circular wait: 循环等待.

3.1 Resource-Allocation Graph

- Process
- Resource Type with 4 instances
- P_i requests instance of R_j
- P_i is holding an instance of R_j

3.2 Basic Facts

If graph contains no cycles → no deadlock.
If graph contains a cycle →

- if only one instance per resource type, then deadlock.
- if several instances per resource type, possibility of deadlock.

4. Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state(死锁避免).

- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX. (KISS)

5. Deadlock Prevention (预防)

Restrain the ways request can be made

1) Mutual Exclusion: 难以打破.

2) Hold and Wait: must guarantee that whenever a process requests a resource, it does not hold any other resources.

3) No Preemption: 也不太行

4) Circular Wait: impose a **total ordering** of all resource types, and require that each process requests resources in an increasing order of enumeration. (但难以做到)

6. Deadlock Avoidance (避免)

Requires that the system has some additional a priori information available.

1) Simplest and most useful model requires that each process declares the maximum number of resources of each type that it may need

2) The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

3) Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

6.1 Safe State

6.2 Avoidance algorithms

For single instance of a resource type, use a resource-allocation graph.

For multiple instances of a resource type, use the banker's algorithm

Resource-Allocation Graph

Banker's Algorithm Assumptions:

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures: Let n = number of processes, and m = number of resources types

- Available
- Max
- Allocation
- Need

First part: Safety Algorithm

1)

Second part: Resource-Request Algorithm for Process P_i:

1) 违反约定
2) 等待资源
3) 假装修改后, 调用 part 1
    - safe, 修!
    - unsafe, 拒!

e.g.

7. Deadlock Detection

Allow system to enter deadlock state.

7.1 Single Instance of Each Resource Type

Maintain wait-for graph

-

rag -> wait-for

7.2 Several Instances of a Resource Type

Detection Algorithm

1)

e.g.

VIII Main Memory

Program must be brought (from disk) into memory and placed within a process for it to be run.

Main memory, Register, Cache

1. Background

Memory Hierarchy

Memory Wall

Emerging NVM Technologies

Processing-in-Memory

1.1 Base and Limit Registers

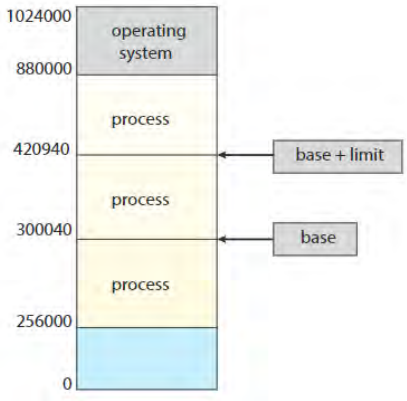


Figure VIII.1: A base and a limit register define a logical address space

1.2 Addresses Given in Different Ways

- Symbolic Address (such as the variable count)
- Relocatable Addresses (such as "14 bytes from the beginning of this module").
- Absolute Addresses (such as 74014)

1.3 Address Binding

Address binding of instructions and data to memory addresses can happen at three different stages

- Compile time: If memory location known a priori, absolute code can be generated
- Load time: Must generate relocatable code if memory location is not known at compile time
- Execution time: Binding delayed until run time

1.4 Logical vs. Physical Address Space

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management

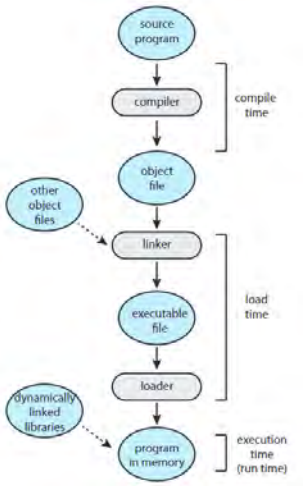


Figure VIII.2: Multistep processing of a user program

- Logical address(CPU, also referred to as virtual address)
- Physical address

be the same in compile-time and load-time.
differ in execution-time

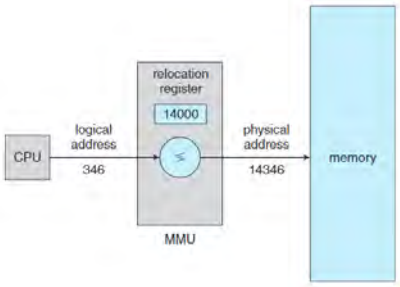


Figure VIII.3: Dynamic relocation using a relocation register.

Memory-Management Unit (MMU): Hardware device that maps virtual to physical address

1.5 Dynamic Loading

Routine is not loaded until it is called

1.6 Dynamic Linking

Linking postponed until execution time.

1) Small piece of code, stub, used to locate the appropriate memory-resident library routine

2) Stub replaces itself with the address of the routine, and executes the routine

Dynamic linking is particularly useful for libraries.

also known as shared libraries

2. Contiguous Memory Allocation

每个进程自己的 view 实现物理内存的隔离.

Main memory usually into two partitions:

1) Resident operating system, usually held in low memory with interrupt vector
2) User processes then held in high memory

2.1 Memory Protection

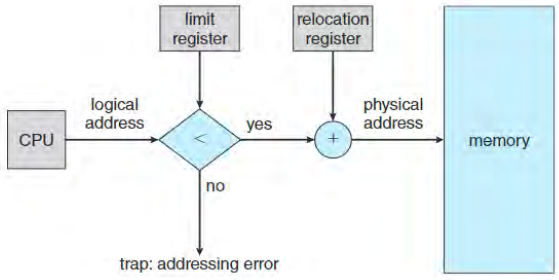


Figure VIII.4: Hardware support for relocation and limit registers

- Relocation register contains value of smallest physical address
- Limit register contains range of logical addresses

MMU maps logical address dynamically

2.2 Multiple-partition allocation

When a process arrives, it is allocated memory from a hole large enough to accommodate it.

Operating system maintains information about:

- allocated partitions
- free partitions (hole)

Hole — block of available memory; holes of various size are scattered throughout memory

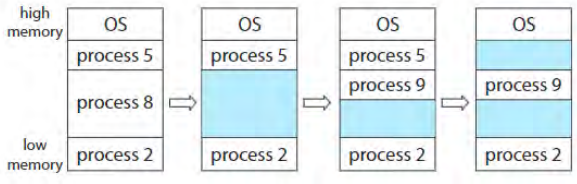


Figure VIII.5: Variable partition

How to satisfy a request of size n from a list of free holes:

- First-fit: Allocate the first hole that is big enough
- Best-fit: Allocate the smallest hole that is big enough
- Worst-fit: Allocate the largest hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

2.3 Fragmentation

- External Fragmentation: total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation: allocated memory may be slightly larger than requested memory

Reduce external fragmentation by compaction. 但 move 的代价很大.

3. Paging

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

- Divide physical memory into fixed-sized blocks called frames
- Divide logical memory into blocks of same size called pages

Internal fragmentation

3.1 Address Translation Scheme

Address generated by CPU is divided into:

- Page number (p)
- Page offset (d)

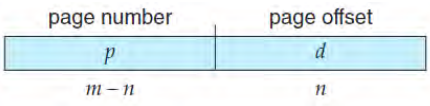


Figure VIII.6: address

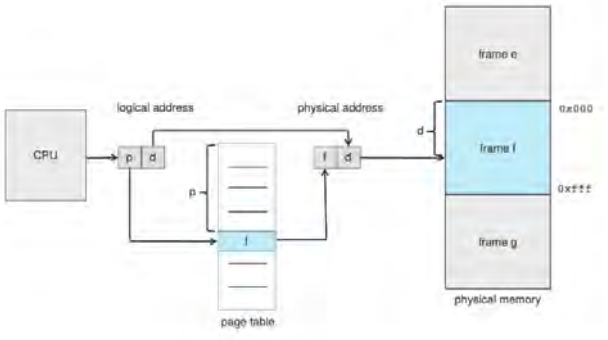


Figure VIII.7: Paging hardware

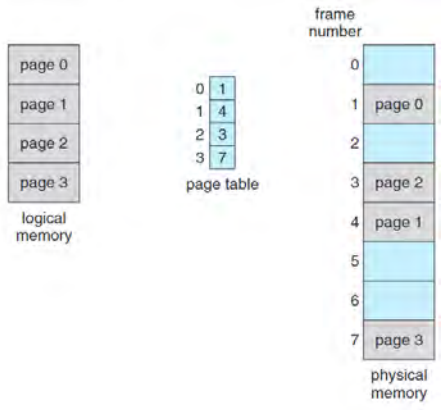


Figure VIII.8: Paging model of logical and physical memory.

frame number (or called PPN)

free-frame list 记录空闲的 frame.

3.2 *Hardware Implementation of Page Table*

Page table is kept in main memory.

- Page-table base register (PTBR) points to the page table
- Page-table length register (PTLR) indicates size of the page table

这样每次访问 data/instruction 需要两次 memory, 一次 page table, 一次 data/instruction.

Solution: a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs 转换旁视缓冲, 一称快表)

Some TLBs store address-space identifiers (ASIDs) in each TLB entry. (唯一标识每个进程，为该进程提供地址空间保护)

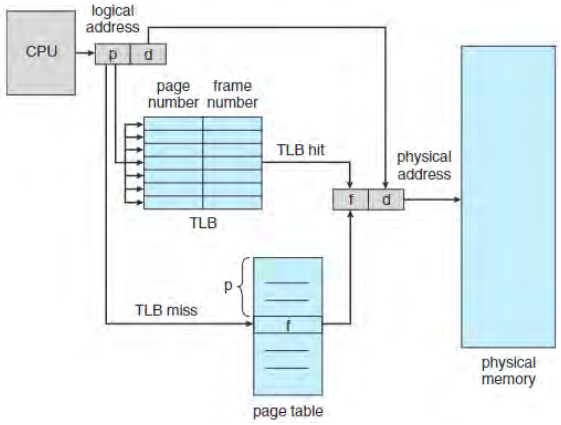


Figure VIII.9: Paging hardware with TLB

3.3 *Effective Access Time*

Associative Lookup = ϵ time unit. Assume memory cycle time is 1 microsecond. Hit ratio = α

Effective Access Time (EAT):

$$EAT = (1+\epsilon)\alpha + (2+\epsilon)(1-\alpha)$$
$$= 2 + \epsilon - \alpha$$

3.4 *Memory Protection in Paged Scheme*

Valid-invalid bit on PTE(page table entry). 访问 invalid 会产生 page fault.

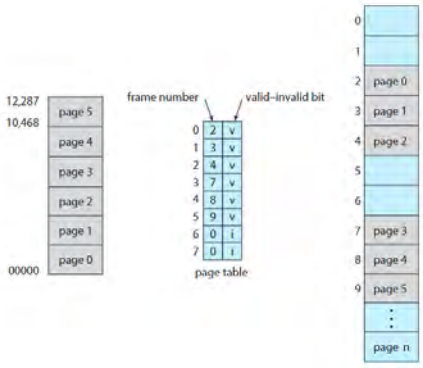


Figure VIII.10: Valid (v) or invalid (i) bit in a page table.

3.5 *Shared Pages*

- Shared code
- Private code and data

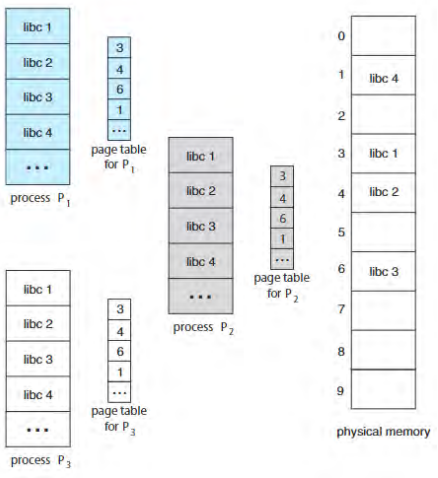


Figure VIII.11: Sharing of standard C library in a paging environment

4. Structure of the Page Table

4.1 Hierarchical Paging

Break up the logical address space into multiple page tables — to page the page table

A simple technique is a two-level page table

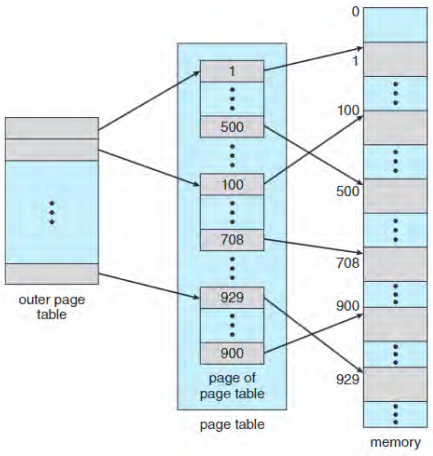


Figure VIII.12: A two-level page-table scheme

Two-Level Paging Example A logical address (on 32-bit machine with 1K page size) is divided into:

- a page number consisting of 52 bits
- a page offset consisting of 12 bits

Since the page table is paged, the page number is further divided into:

- a 42-bit page number
- a 10-bit page offset

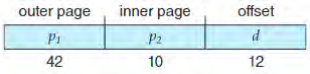


Figure VIII.13: Two-Level Paging

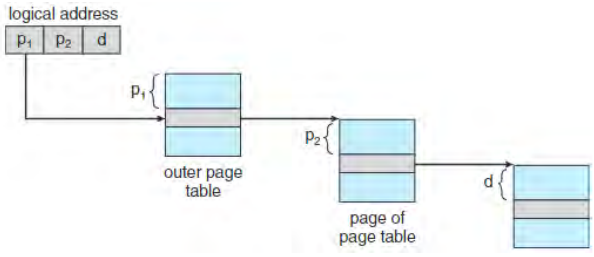


Figure VIII.14: Address translation for a two-level 32-bit paging architecture

最多可能达到 7 级.

在 linux 中: PGD → P4D → PUD → PMD → PTE

多级页表可以节省内存, 因为其空间是动态的.

4.2 Hashed Page Tables

Common in address spaces > 32 bits. the clustered page table 处理 hash 冲突.

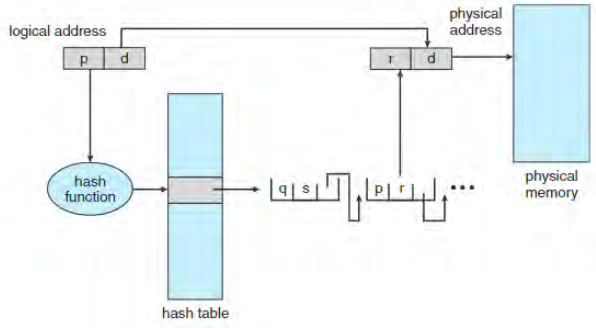


Figure VIII.15: Hashed page table

4.3 Inverted Page Tables

One entry for each real page of memory. 只有一个表, 相当于 memory 的微缩. 用物理的 i 查 p.

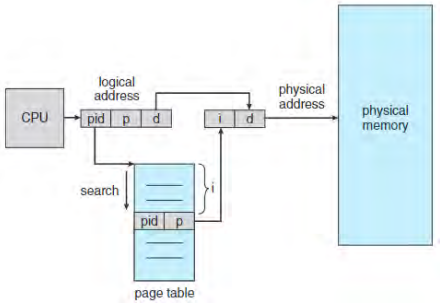


Figure VIII.16: Inverted page table

Search is slow, so put page table entries into a hash table. TLB can be used to speed up hash-table reference. 使用 content-addressible memory (CAM) 特殊硬件加速.

5. Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- Backing store
- Roll out, roll in

System maintains a ready queue of ready-to-run processes which have memory images on disk.

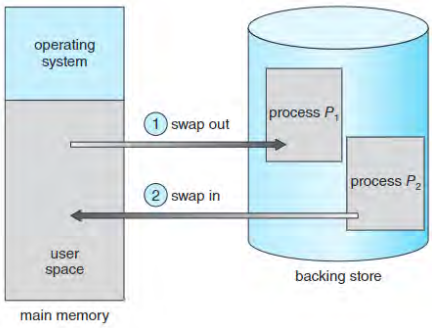


Figure VIII.17: Standard swapping of two processes using a disk as a backing store

只换一部分的 page, called page-in / page-out.
demand-paging 请求式调页.

6. Segmentation

Memory-management scheme that supports user view of memory.

A program is a collection of segments.

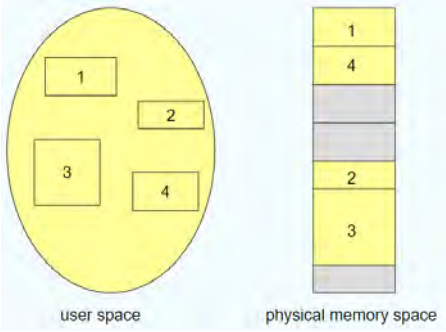


Figure VIII.18: Logical View of Segmentation

Logical address consists of a two tuple:
<segment-number, offset>,

Segment table — maps two-dimensional physical addresses; each table entry has:

- base: contains the starting physical address where the segments reside in memory
- limit: specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program

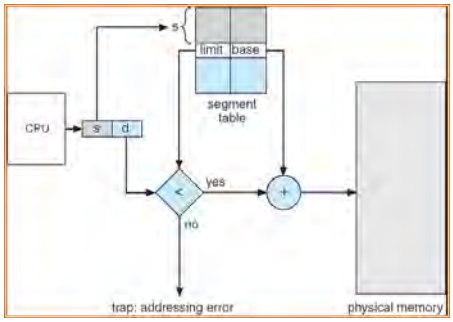


Figure VIII.19: Segmentation Hardware

IX Virtual Memory

1. Background

Virtual memory – separation of user logical memory from physical memory.

仅仅一部分在 memory 的程序需要运行. 所以逻辑内存可以大于物理内存. 甚至可以把虚地址扩展到和外存一样大.

Definition IX.1 *Virtual memory isn't a physical object, but refers to the collection of abstractions and mechanisms the kernel provides to manage physical memory and virtual addresses.(Xv6 book)*

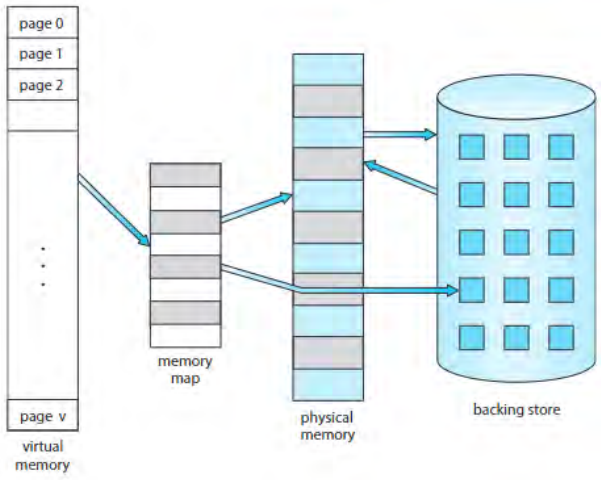


Figure IX.1: Diagram showing virtual memory that is larger than physical memory

Other benefits:

- 动态链接
- 共享内存
- 共享 page

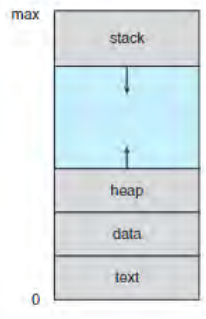


Figure IX.2: Virtual address space of a process in memory

2. Demand Paging(请求式调页)

Bring a page into memory only when it is needed.

Lazy swapper – never swaps a page into memory unless page will be needed. Swapper that deals with pages is a pager.

与之相对的是 eager swapper, 会预取.

Transfer of a Paged Memory to Contiguous Disk Space. swap 连续的 memory.

2.1 Valid-Invalid Bit

With each page table entry a valid–invalid bit is associated

- v: in memory
- i: not in memory

save in table, called another table

2.2 Page Fault

If there is a reference to a page, first reference to that page will trap to operating system: page fault

1) Operating system looks at another table (kept with PCB) to decide:
   Invalid reference → abort
   Just not in memory
2) Get empty frame
3) Swap page into frame
4) Reset tables
5) Set validation bit = v
6) Restart the instruction that caused the page fault

但会有问题: block move

2.3 A Page Fault Causes The Following

1)

2.4 Performance of Demand Paging

Page Fault Rate: $0 \le p \le 1$

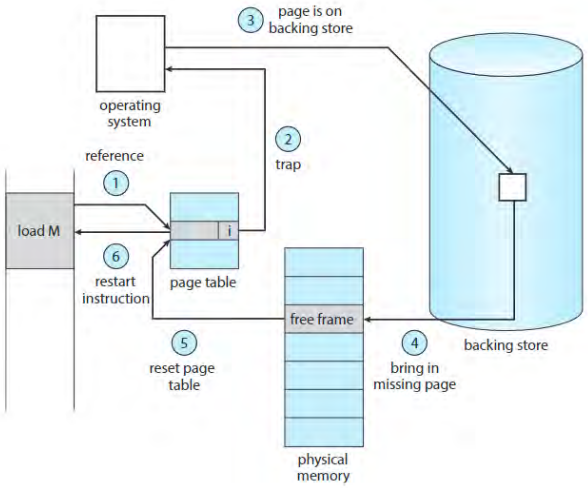


Figure IX.3: Steps in handling a page fault

Effective Access Time (EAT)

$$EAT = (1-p) \times \text{memory access} + p \begin{pmatrix} & \text{page fault overhead} \\ + & \text{swap page out} \\ + & \text{swap page in} \\ + & \text{restart overhead} \end{pmatrix}$$

Process Creation

Virtual memory allows other benefits during process creation

- Copy-on-Write
- Memory-Mapped Files

3. Copy-on-Write

Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory.Free pages are allocated from a pool of zeroed-out pages.

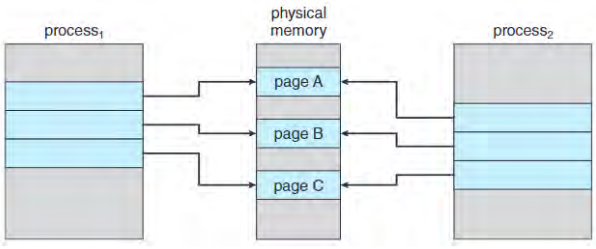


Figure IX.4: Before process 1 modifies page C

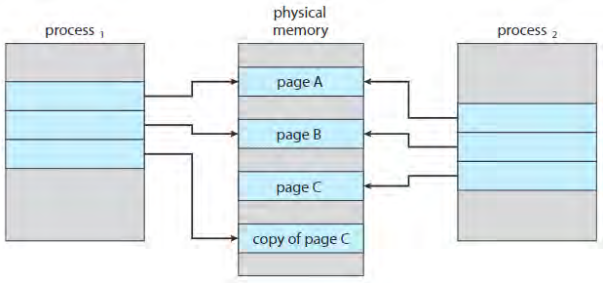


Figure IX.5: After process 1 modifies page C

4. Page Replacement

If there is no free frame: Page replacement – find some page in memory, but not really in use, swap it out.

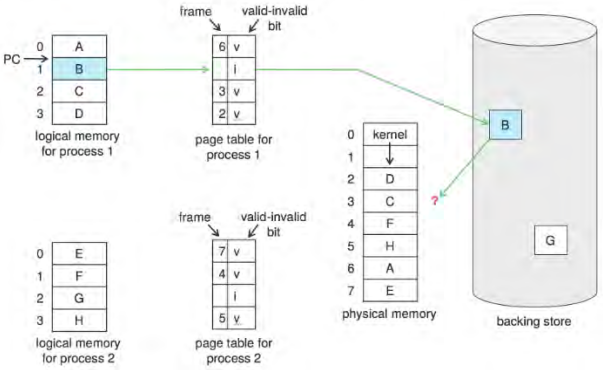


Figure IX.6: Need for page replacement

4.1 Basic Page Replacement

1) Find the location of the desired page on secondary storage.

2) Find a free frame:

- If there is a free frame, use it.
- If there is no free frame, use a page-replacement algorithm to select a victim frame.
- Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly. (Use dirty bit)

3) Read the desired page into the newly freed frame; change the page and frame tables.

4) Continue the process from where the page fault occurred.

Page Replacement Algorithms: Want lowest page-fault rate.

4.2 First-In-First-Out (FIFO) Algorithm

First-In-First-Out

Belady's Anomaly: more frames ⇒ more page faults.

sequential flood.

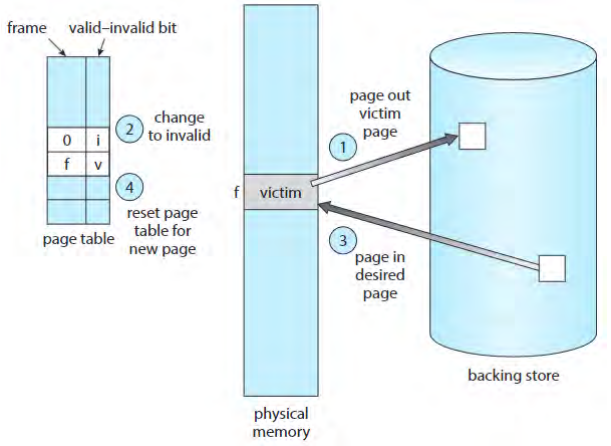


Figure IX.7: Page replacement

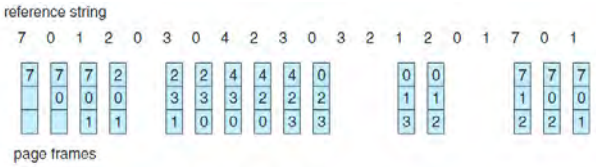


Figure IX.8: FIFO page-replacement algorithm

4.3 Optimal Algorithm

Replace page that will not be used for longest period of time. Just used for measuring how well your algorithm performs.

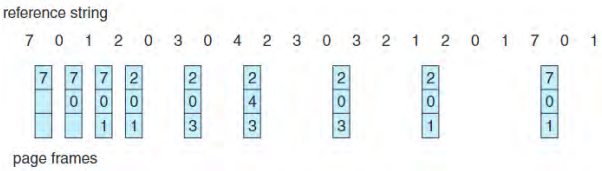


Figure IX.9: Optimal page-replacement algorithm

4.4 Least Recently Used (LRU) Algorithm

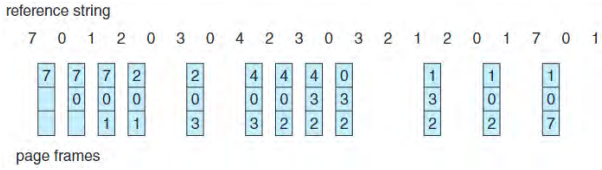


Figure IX.10: LRU page-replacement algorithm

- Counter implementation: write clock into counter, look at counters to determine which is to change.

- Stack implementation: keep a stack of page numbers in a double link form. When Page referenced, move it to the top.

4.5 *LRU Approximation Algorithms*

Reference bit:

- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace the one which is 0 (if one exists)

Second chance:

- Need reference bit
- Clock replacement
- If page to be replaced (in clock order) has reference bit = 1 then:
  - set reference bit 0
  - leave page in memory
  - replace next page (in clock order), subject to same rules
- 扫描, 若 1 则置 0, 若 0 替换.

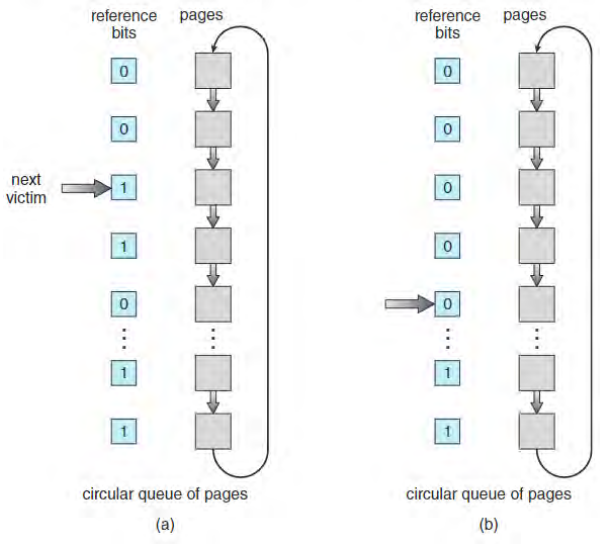


Figure IX.11: Second-chance (clock) page-replacement algorithm

4.6 *Counting-based Algorithms*

Keep a counter of the number of references that have been made to each page

Other:

- LFU Algorithm: replaces page with smallest count
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

5. Allocation of Frames

Each process needs minimum number of pages — usually determined by computer architecture

Two major allocation schemes:

- fixed allocation
  - Equal allocation
  - Proportional allocation - Allocate according to the size of process
- priority allocation

  Use a proportional allocation scheme using priorities rather than size

5.1 *Global vs. Local Allocation*

- Global replacement
- Local replacement

6. Thrashing(颠簸)

Thrashing: a process is busy swapping pages in and out. Total size of locality > total memory size

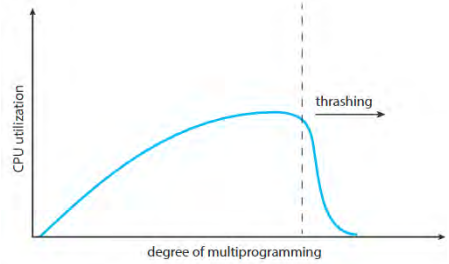


Figure IX.12: Thrashing

To limit the effect of thrashing: local replacement algo cannot steal frames from other processes. But queue in page device increases effective access time.

To prevent thrashing: allocate memory to accommodate its locality

6.1 *Working-Set Model*

Let Δ be working-set window, which is a fixed number of page references.

WSS_i(working-set size of Process P_i) = total number of pages referenced in the most recent Δ (varies in time).

Let $D = \sum WSS_i$ be total demand frames for all processes in the system. If $D > m$, thrashing, then suspend one of the processes.

6.2 *Keeping Track of the Working Set*

Approximate with interval timer + a reference bit.
e.g.

7. Memory-Mapped Files

8. Allocating Kernel Memory

Often allocated from a free-memory pool

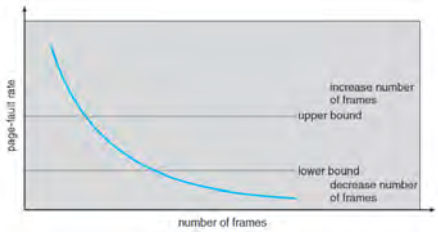


Figure IX.13: Page-fault frequency

8.1 Buddy System

Allocates memory from fixed-size segment consisting of physically- contiguous pages

Memory allocated using power-of-2 allocator.

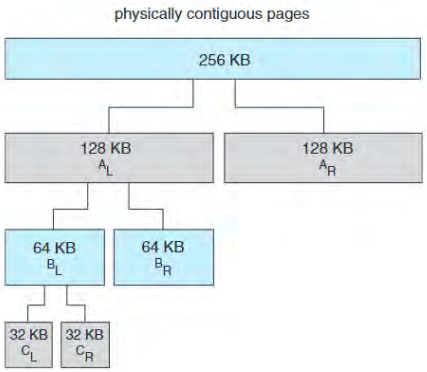


Figure IX.14: Buddy system allocation

8.2 Slab Allocator

Slab is one or more physically contiguous pages. Cache consists of one or more slabs. Each cache filled with objects – instantiations of the data structure

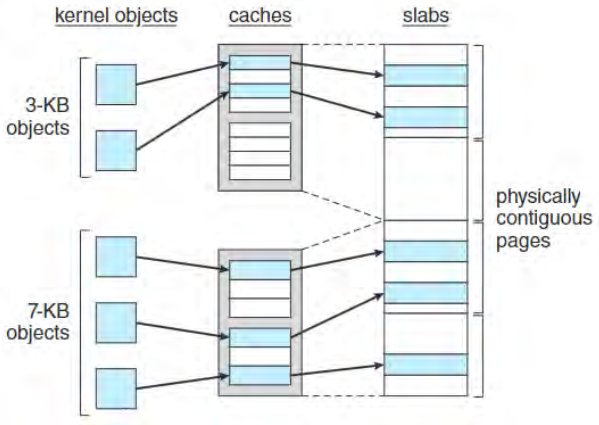


Figure IX.15: Slab allocation

9. Other Issues

9.1 Prepaging/Prefetching

To reduce the large number of page faults that occurs at process startup

9.2 Page Size

Page size selection must take into consideration

9.3 TLB Reach

9.4 Program Structure

X File-System Interface

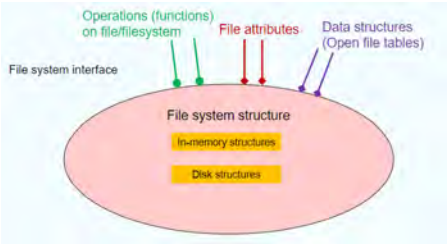


Figure X.1: Mind Map for Ch10 and 11

1. File Concept

Definition X.1 (File System) *The way that controls how data is stored and retrieved in a storage medium*

- *File naming*
- *Where files are placed*
- *Metadata*
- *Access rules*

Definition X.2 (File) *Contiguous logical address space.*

A sequence of bits, bytes, lines, or records. The meaning is defined by the creator and user.

File Structure:

- None: sequence of words, bytes
- Simple record structure: Lines, Fixed length, Variable length
- Complex Structures: Formatted document, Relocatable load file

1.1 File Attributes

- Name: only information kept in human-readable form
- Identifier: unique tag (number) identifies file within file system
- Type: needed for systems that support different types
- Location: pointer to file location on device
- Size: current file size
- Protection: controls who can do reading, writing, executing
- Time, date, and user identification: data for protection, security, and usage monitoring

Information about files are kept in the directory structure, which is maintained on the disk

1.2 File Operations

File is an abstract data type

- Create
- Write: define a pointer
- Read: use the same pointer Per-process current file-position pointer
- Reposition within file (file seek)
- Delete
- Truncate(截取)
- Open(F_i): search the directory structure on disk for entry F_i, and move the content of entry to memory
- Close (F_i): move the content of entry F_i in memory to directory structure on disk

1.3 Open-file table(打开文件表)

Open() system call returns a pointer to an entry in the open-file table. 即打开文件就在 table 中写一个 entry.

- Per-process table, for P_1, P_2
- System-wide table, for OS kernel

若多个 process 打开一个文件, 需要使用 system-wide table 处理.

1.4 Open File Locking

Provided by some operating systems and file systems. Mediates access to a file. Mandatory or advisory.

1.5 File Types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure X.2: Common file types

Linux 使用 Magic number 判断文件类型.

2. Access Methods

- Sequential Access
- Direct (Random) Access

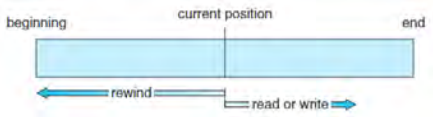


Figure X.3: Sequential-access file

3. Directory Structure

A collection of nodes containing (management) information about all files.

有 volume, 其中用 partition 存储 file.

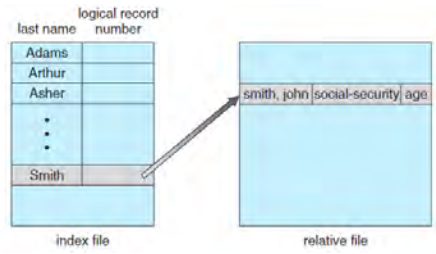


Figure X.4: Example of index and relative files

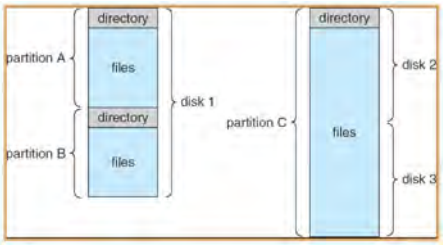


Figure X.5: A Typical File-system Organization

3.1 Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system -access every dir and file for backing up.

3.2 Organize the Directory (Logically) to Obtain

- Efficiency: locating a file quickly
- Naming: convenient to users
- Grouping: logical grouping of files by properties

3.3 Directory

- Single-Level Directory
- Two-Level Directory
- Tree-Structured Directories
- Acyclic-Graph Directories
- General Graph Directory

3.4 Soft Link v.s. Hard Link

- Soft Link: a string
- Hard Link: a link

4. File-System Mounting

A file system must be mounted before it can be accessed.
An un-mounted file system is mounted at a mount point.
chroot: linux 中切换根目录.

5. Protection

Types of access:

- Read
- Write
- Execute
- Append
- Delete
- List

5.1 Access Lists and Groups

Mode of access: read, write, execute

a) **owner access**	7	⇒	RWX 1 1 1
b) **group access**	6	⇒	RWX 1 1 0
c) **public access**	1	⇒	RWX 0 0 1

Figure X.6: Three classes of users

```
-rw-rw-r--   1 pbg   staff     31200  Sep 3 08:30   intro.ps
drwx------   5 pbg   staff       512  Jul 8 09.33   private/
drwxrwxr-x   2 pbg   staff       512  Jul 8 09:35   doc/
drwxrwx---   2 jwg   student     512  Aug 3 14:13   student-proj/
-rw-r--r--   1 pbg   staff      9423  Feb 24 2017   program.c
-rwxr-xr-x   1 pbg   staff     20471  Feb 24 2017   program
drwx--x--x   4 tag   faculty     512  Jul 31 10:31  lib/
drwx------   3 pbg   staff      1024  Aug 29 06:52  mail/
drwxrwxrwx   3 pbg   staff       512  Jul 8 09:35   test/
```

Figure X.7: A Sample UNIX Directory Listing

XI File System Implementation

1. File-System Structure

File structure: 需要解决接口与内部数据结构.

- Logical storage unit
- Collection of related information

File system resides on secondary storage (disks). File system organized into layers.

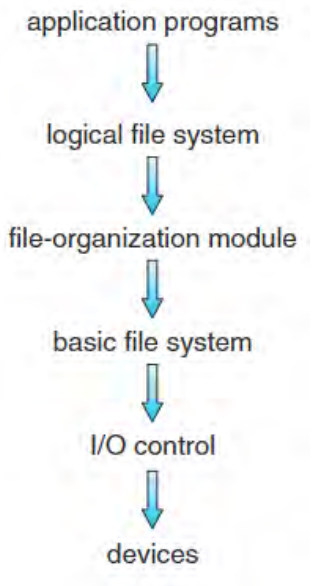


Figure XI.1: Layered file system

Table XI.1: Layered file system

logical file system	Manages metadata of files. Protection and security
file-organization module	Translates logical block addr to physical addr. Free space mgmt
basic file system	Commands to r/w physical blocks
I/O control	Translates 'r/w block' to low-level hw instructions

FTL (flash) is I/O control

2. File-System Implementation

2.1 Data Structures Used to Implement FS

Disk structures:

- Boot control block
- Volume control block (superblock in Unix)
- Directory structure per file system
- Per-file FCB (inode in Unix)

In-memory structures:

- In-memory mount table about each mounted volume
- Directory cache
- System-wide open-file table
- Per-process open-file table

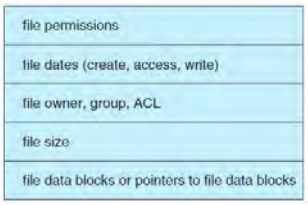


Figure XI.2: A typical file-control block: storage structure consisting of information about a file

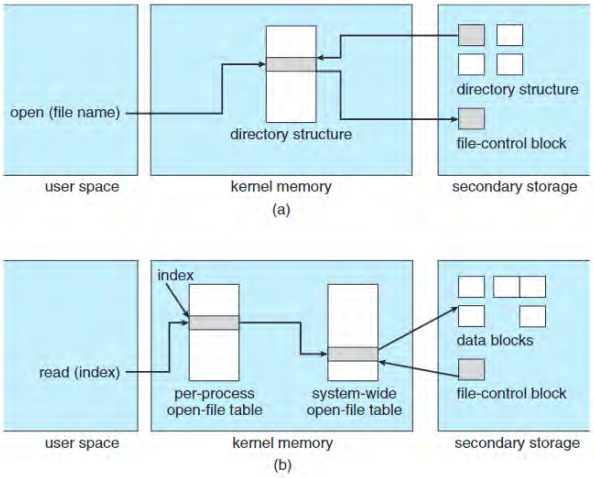


Figure XI.3: In-memory file-system structures.(a) File open. (b) File read.

2.2 Virtual File Systems

Virtual File Systems (VFS) provide an object-oriented way of implementing file systems. VFS isn't a disk file system.

VFS allows the same system call interface (the API) to be used for different types of file systems.

Defines a network-wide unique structure called vnode.

VFS is logical file system

3. Directory Implementation

Linear list of file names with pointer to the data blocks

- simple to program
- time-consuming to execute

Hash Table: linear list with hash data structure

- decreases directory search time

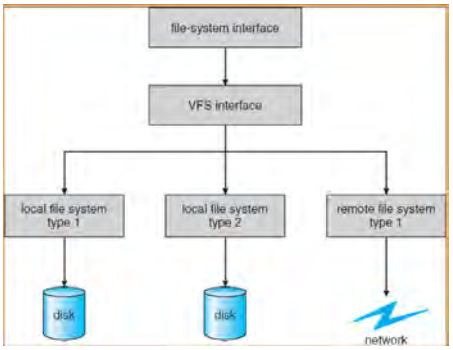


Figure XI.4: Schematic View of Virtual File System

- collisions: situations where two file names hash to the same location
- fixed size: can use chained-overflow hash table

4. Allocation Methods

An allocation method refers to how disk blocks are allocated for files.

4.1 Contiguous allocation

Each file occupies a set of contiguous blocks on the disk. But it's Wasteful of space (dynamic storage-allocation problem). Random access supported

Mapping from logical to physical:

$$\text{Logic Address} = 512Q + R$$

- Block to be accessed = Q+ start address
- Displacement into block = R

Extent-Based Systems a modified contiguous allocation scheme.

Extent-based file systems allocate disk blocks in extents. An extent is a contiguous block of disks.

4.2 Linked allocation

Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk. No random access, poor reliability

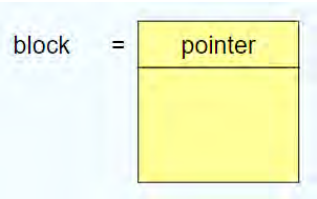


Figure XI.5: Linked Allocation

Mapping:

$$\text{Logic Address} = 511Q + (R - 1)$$

511 because of pointer

- Block to be accessed is the Qth block in the linked chain of blocks representing the file
- Displacement into block = $R + 1$

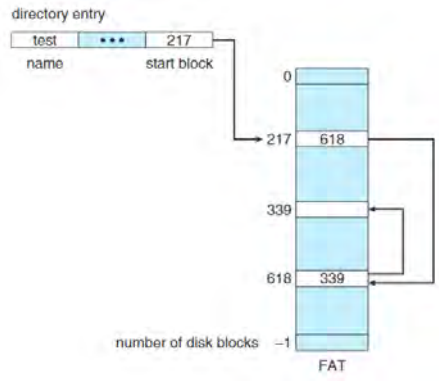


Figure XI.6: File-allocation table(改进的 Linked Allocation, 将 linked 放入内存)

4.3 Indexed allocation

Brings all pointers together into the index block. Random access

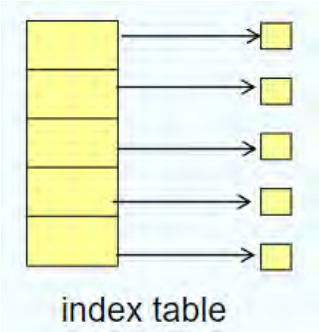


Figure XI.7: Indexed allocation

Mapping:

1) When mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table

$$\text{Logic Address} = 512Q + R$$

- Q = displacement into index table
- R = displacement into block

2) When mapping from logical to physical in a file of unbounded length (block size of 512 words). – more pointers are needed

$$\text{Logic Address} = (512 \times 511)Q_1 + R_1$$
$$R_1 = 512Q_2 + R_2$$

- Linked scheme - Link blocks of index table
  Q_1 = block of index table
  Q_2 = displacement into block of index table
  R_2 = displacement into block of file
- Two-level index
  Q_1 = displacement into outer-index
  Q_2 = displacement into block of index table
  R_2 = displacement into block of file

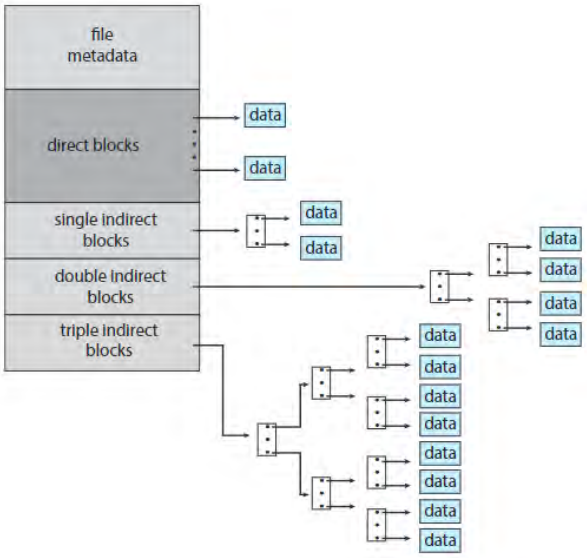


Figure XI.8: The UNIX inode(Combined Scheme)

5. Free-Space Management

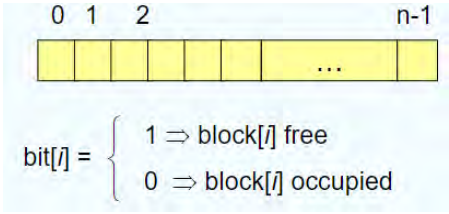


Figure XI.9: Bit vector

e.g.

- Linked list (free list)
- Grouping: a modification of the Linked List
- Counting: Address of the first free block and number n contiguous blocks

6. Efficiency and Performance

Efficiency dependent on:

- disk allocation and directory algorithms
- types of data kept in file's directory entry

Generally, every data item has to be considered for its effect.

Performance:

- disk cache: separate section of main memory for frequently used blocks
- free-behind and read-ahead: techniques to optimize sequential access
- improve PC performance by dedicating section of memory as virtual disk, or RAM disk

6.1 Page Cache

A page cache caches pages rather than disk blocks using virtual memory techniques.

将 page cache 与 buffer cache 合并.

XII Mass-Storage Systems

1. Overview of Mass Storage Structure

Magnetic disks provide bulk of secondary storage of modern computers

- Drives rotate at 60 to 200 times per second
- Transfer rate is rate at which data flow between drive and computer
- Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time, 寻道 + 旋转) and time for desired sector to rotate under the disk head (rotational latency)
- Head crash results from disk head making contact with the disk surface

XIII I/O Systems