# 操作系统

## 第1章 操作系统概述

### 1.1 操作系统的概念、特征、功能

#### 1.1.1 操作系统的概念

操作系统（Operating System, OS）是指控制和管理整个计算机系统的硬件和软件资源，并合理的组织调度计算机的工作和资源的分配，以提供给用户和其他软件方便的接口和环境的程序的集合。

计算机系统包括硬件和软件两个组成部分。操作系统则是配置在计算机硬件上的第一层软件，是对硬件的首次扩充和改造。

#### 1.1.2 操作系统的特征

**1. 并发（Concurrence）**

**并行**是指两个或两个以上事件在同一时刻发生，这要求在硬件上至少要有两个处理器。

**并发**是指两个或两个以上事件在同一时间间隔内发生。

操作系统的并发性是指计算机系统中同时存在着多个运行着的程序。

在单处理器机器上，宏观上有多个程序在同时执行，微观上这些程序还是在分时地交替执行（宏观上并行，微观上串行）。操作系统的并发性是通过分时得以实现的。

**2. 共享（Sharing）**

共享是指系统中的资源（硬件资源和信息资源）可以被多个并发执行的程序共同使用，而不是被其中一个独占。资源共享有两种方式：互斥访问和同时访问。

并发和共享式操作系统的最基本特征，互为依存。并发执行的要求引出了资源的共享；而资源共享的管理又直接影响到程序的并发执行。

**3. 异步（Asynchronism）**

在多道程序环境下，进程的执行不是一贯到底，而是走走停停，已不可预知的速度向前推进，这就是进程的异步性。

**4. 虚拟（Virtual）**

虚拟性是一种管理技术，是把物理上的一个实体变成逻辑上的多个对应物，或把物理上的多个实体变成逻辑上的一个对应物的技术。采用虚拟技术的目的是为用户提供易于使用、方便高效的操作环境。

操作系统利用两种方式来实现虚拟；时分复用技术，如处理器分时、SPOOLing技术；空分复用技术，如虚拟存储器、虚拟机。

#### 1.1.2 操作系统的功能

1. 处理器管理：（1）进程控制；（2）进程同步；（3）进程通信；（4）进程调度。

2. 存储管理：（1）内存分配与回收；（2）地址转换；（3）内存扩充；（4）内存保护。

3. 文件管理：（1）磁盘空间管理；（2）目录管理；（3）文件操作；（4）文件保护。

4. 设备管理：（1）设备分配；（2）设备传输控制；（3）设备独立性。

5. 提供接口：程序接口（即操作系统提供的一些列系统调用，又称为API）和用户接口（包括早期的命令解释程序和现在的图形界面窗口程序）。

### 1.2 操作系统的发展与分类

1. 单道批处理系统

特征：单道，自动性，顺序性

2. 多道批处理系统

特征：脱机，多道，成批处理，无序性，调度性

3. 分时系统

特征：多路性，交互性，独占性，及时性

4. 实时系统

分为两类：实时控制系统和实时信息处理系统

特征：及时响应和高可靠性

5. 网络与分布式系统、多机系统

网络与分布式系统是支持网络和分布处理的系统。多机系统是支持共享内存多处理器的系统。

### 1.3 操作系统的运行环境

核心态（管态）和用户态（目态）

从核心态进入用户态的唯一途径就是通过中断。

## 第2章 进程管理

### 2.1进程与线程

#### 2.1.1 进程概念

**进程**是程序在处理机上的一次执行过程。进程是动态的，程序是静态的。

**进程映像**（Process Image）包括：（1）进程控制块PCB；（2）程序和数据；（3）执行栈区。

为了管理进程，操作系统为每一个进程定义了一个数据结构——**进程控制块PCB**（Process Control Block），用于记录进程的各种信息。

### 2.1.2 进程的状态与转换

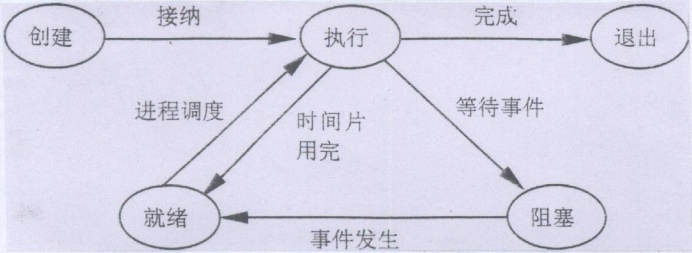


图2.1 三态转换

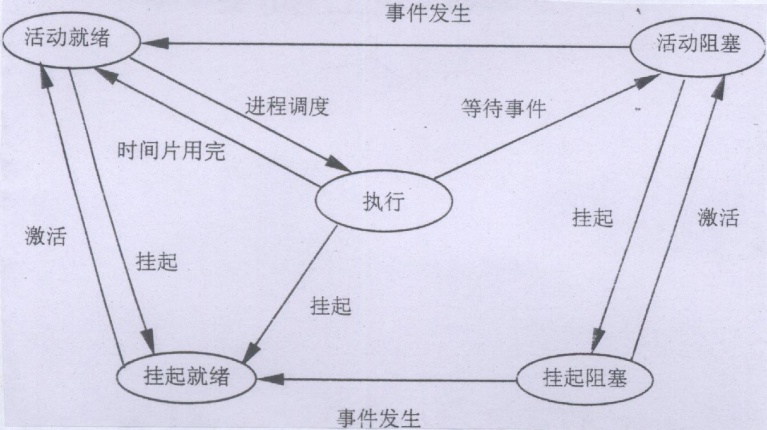


图2.2 五态转换

#### 2.1.3 进程控制

进程的创建与终止，模式切换，进程切换

#### 2.1.4 进程组织

批处理操作系统中：根进程，子进程，进程组。

分时系统中：一个终端一个进程

#### 2.1.5 进程通信

1. 共享存储：共享内存，共享文件

2. 消息传递：发送消息和接收消息原语

3. 管道通信：pipe文件

#### 2.1.6 线程概念与多线程模型

**1. 线程的引入**

回顾一下进程的两个基本职能：

（1）进程是系统资源的分配单位；（2）进程是处理机的分配单位。

进程切换的开销比较大，为了提高操作系统的并发性能，不少操作系统研究者考虑将进程的两个基本属性分离开来，为此引入了线程，让线程去完成第二个基本职能，而进程只完成第一个职能。

**2. 线程的定义**

线程是进程内一个相对独立的、可调度的执行单元。线程自己基本上不拥有资源，只用有一点在运行时必不可少的资源（如程序计数器、一组寄存器和栈），但它可以与同属一个进程的其他线程共享进程拥有的全部资源。

**3. 线程和进程的比较**

（1）调度。线程是调度的单位，进程不是。

（2）拥有资源。进程是资源的分配单位，线程不拥有资源，但线程可以访问其所在进程的所有资源。

（3）并发性。引入线程后，不仅进程之间可以并发执行，而且同一进程内的多个线程之间也可以并发执行。从而使操作系统具有更好的并发性，大大提高了系统的吞吐量。

（4）系统开销。进程切换开销很大，而线程切换只需保存少量寄存器内容，开销很小。

**4. 多线程模型**

多对一：将多个用户级线程映射到一个内核线程。

一对一

多对多

### 2.2 进程调度

#### 2.2.1 调度的基本概念

作业从提交开始直到完成，往往要经历以下三级调度：

**高级调度**，又称长程调度，作业调度，其主要任务是按一定的原则从外存上处于后备状态的作业中选择一个或多个，给它们分配内存、输入输出设备等必要的资源，并创建相应的进程。

**中级调度**，又称中程调度，交换调度，其主要任务是按一定的原则将处于外存对换区的重新又具备运行条件的进程调入内存，或将处于内存的暂时不能运行的进程交换到外存对换区。

**低级调度**，又称短程调度，进程调度，其主要任务是按某种策略从就绪队列中选择一个进程，将处理机分配给它。

#### 2.2.2 调度时机、切换与过程

现代操作系统中，不能进行进程的调度与切换的情况有：

（1）在处理中断的过程中：中断处理过程复杂，在实现上很难做到，而且中断处理是系统工作的一部分，逻辑上不属于某一进程，不应被剥夺处理器资源。

（2）进程在操作系统内核程序临界区中：进入临界区后，需要独占式的访问共享数据，理论上必须加锁，以防止其他并行程序进入，在解锁前不应切换到其他进程，以加快该共享数据的释放。

（3）其他需要完全屏蔽中断的原子操作过程中：如加锁、解锁、中断现场保护、恢复等原子操作。在原子操作过程中，连中断都要屏蔽，更不应该进行进程调度与切换。

如在上述过程中发生了引起调度的条件，并不能马上响应，应置系统的请求调度标志，直到上述过程结束后才进行相应的调度与切换。

应该进行进程的调度与切换的情况有：

（1）正在运行的进程的时间片用完了。

（2）一个更高优先级进程到达系统或从阻塞状态变为就绪状态。

（3）一个就绪进程的优先级被提升至当前最高优先级。

（4）正在运行进程因为等待资源或事件变成阻塞状态。

（5）进程执行“放弃CPU”的系统调用。

（6）进程运行结束。

剥夺调度是由于运行进程的外部原因（如前三种事件），内核强行地剥夺正在运行进程的CPU，将CPU分配给另一个就绪进程。非剥夺调度是由于运行进程的自身原因（即后三种事件），如进程运行结束或进程发生阻塞，在进程自愿放弃CPU后，内核将CPU分配给另一个就绪进程。

#### 2.2.3 调度的基本准则

1. 用户角度

（1）周转时间短。周转时间——作业从被提交给系统开始到作业完成为止的全部时间。带权周转时间——周转时间/运行时间。

（2）响应时间短。响应时间——作业从提交到系统第一次响应的时间间隔。

（3）截止时间保证。截止时间——一个任务开始得到执行或执行完毕的最迟时间。

（4）优先权准则。

2. 系统角度

（1）系统吞吐量高。吞吐量——系统在单位时间内完成的作业数量。

（2）处理器利用率高。

（3）各类资源平衡利用。

#### 2.2.4 调度方式

剥夺方式（抢占式）和非剥夺方式（非抢占式）

#### 2.2.5 进程调度算法

1. 先来先服务算法（FCFS）

按照进入就绪队列的先后顺序进行调度，非剥夺方式

2. 短进程优先调度算法（SJF）

从就绪队列中选择那些所需运行时间（估计时间）最短的进程运行，非剥夺方式。

3. 优先级调度算法

按进程的优先级进行调度，优先级有静态优先级和动态优先级，调度方式可以是剥夺方式和非剥夺方式。

4. 最高响应比优先调度算法（HRN）

响应比=（等待时间+请求服务时间）/请求服务时间，非剥夺方式

5. 时间片轮转调度算法（RR）

按照先来先服务原则，选择一个进程运行一段时间，当该进程用完这段时间后（即使未完成），系统将其送入就绪队列队尾。它相当于剥夺方式的先来先服务算法。

6. 多级反馈队列调度算法

本算法基于以下考虑：

（1）为提高系统吞吐量和降低进程平均等待时间而照顾短进程。

（2）为得到较好的输入/输出设备利用率和对交互用户的及时响应而照顾输入/输出型进程。

（3）在进程运行过程中，按进程运行情况来动态地考虑进程的性质（是输入/输出型还是计算型进程），同时进行相应的调度。

具体来说，① 系统中有多个进程就绪队列，每个就绪队列对应一个优先级。第1级队列的优先级最高，以下各级队列的优先级逐次降低。优先级最高的第1级队列中的进程的时间片最小，随着队列的级别增加，其进程的优先级降低了，但时间片却增加了。通常下放一级，其时间片增加1倍。各级队列均按先来先服务原则调度。② 当一个新进程进入内存后，首先将它放入第一级队列队尾，等待调度，当轮到该进程执行时，若它能在该时间片内完成，便可撤离系统；若它不能，则进入下一级队列，依次类推。

### 2.3 进程同步

#### 2.3.1 进程同步的基本概念

**同步关系**：若某一进程收不到另一进程提供的信息就不能继续运行下去，这种制约关系的基本形式是“进程——进程”，是一种直接制约关系，称为同步关系。

**互斥关系**：若某一进程要求使用某一资源，而该资源正在被另一进程使用，并且该资源不允许两个进程同时使用，那么该进程只好等待。这种制约关系的基本形式是“进程——资源——进程”，是一种间接制约关系，称为互斥关系。

**临界资源**（Critical Resource, CR）：一次仅允许一个进程使用的资源。

**临界区**（Critical Section, CS）：各进程必须互斥执行的那种程序段，该程序段实施对临界资源的操作。

临界区应遵循的原则：空闲让进，忙则等待，有限等待，让权等待。

#### 2.3.2 实现临界区互斥的基本方法

1. 软件实现方法

设置以一些标志位来表明是否有进程在临界区中。

采用软件方法解决互斥既困难又复杂，目前很少使用。

2. 硬件实现方法

硬件方法的主要思想是用一条指令完成标志的检查和修改两个操作，因而保证了检查操作和修改操作不被打断，或通过中断屏蔽的方式来保证检查和修改作为一个整体执行。

（1）中断屏蔽方法

CPU只有在发生中断时才会引起进程切换，因此简单而直接的方法是禁止一切中断发生，这样就能保证当前运行的进程能将临界段顺利执行完，从而保证了互斥的正确实现，然后在开中断。

下面是用中断屏蔽方法实现临界区的典型模式：

...

关中断

临界区

开中断

...

（2）硬件指令方法

① “Test-and-Set”指令

bool Test\_and\_Set(bool \*lock)

{

bool old = \*lock;

\*lock = true;

return old;

}

临界段代码：

...

while(Test\_and\_Set(&lock));

critical section;

lock = false;

...

② “Swap”指令

void Swap(bool \*a, bool \*b)

{

bool tmp = \*a;

\*a = \*b;

\*b = tmp;

}

临界段代码：

...

key = true;

while(key){

Swap(&key, &lock);

}

critical section;

lock = false;

...

硬件方法的缺点是，进程在等待进入临阶段时需要耗费处理机时间，不能实现让权等待。

#### 2.3.3信号量

1965年，Dijkstra提出了一种称为信号量(semaphore)的同步互斥工具，称为信号量机构。信号量机构由信号量和P操作（又称为wait操作）、V操作（又称为signal操作）两部分组成。信号量（S）为一整型变量，表示资源的数目，只能被两个标准的原语操作访问。

P操作和V操作可以用如下函数描述：

struct semaphore{

int count;

queue\_t q;

};

P(semaphore \*s)

{

s->count--;

if(s->count < 0){

阻塞该进程;

将该进程插入阻塞队列s->q;

}

}

V(semaphore \*s)

{

s->count++;

if(s->count <= 0){

从阻塞队列s->q中取出第一个进程;

将该进程插入就绪队列;

}

}

#### 2.3.4 管程

用信号量可以实现进程间的同步和互斥，但由于信号量的控制代码分散地分布在整个程序中，其正确性分析很困难，使用不当还可能导致进程死锁。针对信号量机制中存在的这些问题，Dijkstra于1971年提出，为每一个共享资源设计一个“秘书”来管理对它的访问，一切来访者都要通过秘书，而秘书每次仅允许一个来访者（进程）访问共享资源。这样既便于系统管理共享资源，又能保证互斥访问和进程间同步。1973年，Hanson和Hoare又把“秘书”概念发展为管程概念。

1. 管程的定义

管程定义了一个数据结构和能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据。

2. 管程的组成

由定义可知，管程有4个组成部分：管程有有一个全局的名字，管程类似于一个封装类；局部于该管程的数据结构——共享变量；对该变量进行的一组操作；对该变量的初始设置语句。

由于管程是一个语言成分，所以管程的互斥访问完全由编译程序在编译时自动添加，无需程序员关心，而且保证正确。

#### 2.3.5 经典同步互斥问题

**1.生产-者消费者问题**

（1）将缓冲区当成一个整体

semaphore full = 0;

semaphore empty = N;

semaphore mutex = 1;

main()

{

producer();

consumer();

}

producer()

{

while(true){

生产一个产品;

P(empty);

p(mutex);

将产品放入缓冲区

V(mutex);

V(full);

}

}

consumer()

{

while(true){

P(full);

P(mutex);

从缓冲区取出一个产品;

V(mutex);

V(empty);

消费这个产品;

}

}

（2）将n个缓冲区当成一个循环队列

semaphore full = 0;

semaphore empty = N;

semaphore pmutex = 1; //生产者之间的互斥信号量

semaphore cmutex = 1; //消费者之间的互斥信号量

int in = 0, out = 0;

main()

{

producer();

consumer();

}

producer()

{

while(true){

生产一个产品;

P(empty);

p(pmutex);

将产品放入buffer[in];

in = (in + 1) % N;

V(pmutex);

V(full);

}

}

consumer()

{

while(true){

P(full);

P(cmutex);

从buffer[out]取出一个产品;

out = (out + 1) % N;

V(cmutex);

V(empty);

消费这个产品;

}

}

程序中有一点要请读者注意，无论在生产者进程还是在消费者进程中，P操作的次序都不能颠倒，否则可能造成死锁。例如，在某一时刻缓冲区中已装满了产品且缓冲区没有进程工作（这时full=N, empty=0, mutex=1），若系统此时调度生产者进程运行，生产者进程生产了一个产品，执行P(mutex)并顺利进入临界区（这时mutex变为0），随后它执行P(empy)时因没有空闲缓冲单元而受阻等待，等待消费者进程进入缓冲区取走产品，以释放缓冲单元；而消费者进程执行P(full)后再执行P(mutex)时，因缓冲区被生产者进程占据而无法进入。这样就形成了生产者进程在占有临界资源的情况下等待消费者进程取走产品，而消费者进程又无法进入临界区取走产品的僵局，此时两进程陷入死锁。

**2.读者-写者问题**

读者写者问题分为两类，读者优先和写者优先。

读者优先的读者写者问题的具体描述如下：

（1）如果当前无进程访问数据，无论读者还是写者欲访问数据都可以直接进行访问。

（2）如果已有一个读者正在访问数据，那么其他与访问数据的读者可直接进行访问；而当前欲访问数据的写者则必须无条件等待。

（3）若某个写者正在访问数据，则当前欲访问数据的读者和写者均须等待。

（4）当最后一个结束访问数据的读者发现有写者正在等待时，则将其中的一个唤醒。

（5）当某个写者结束访问数据时发现存在等待者，那么若此时只有写者处于等待，则唤醒某个写者。若此时有读者和写者同时处于等待，则按照FIFO或其他原则唤醒一个写者或唤醒所有读者。

该问题中“读者优先”主要表现在：除了某个写者正在访问数据之外，任何情况下读者欲访问数据均可以直接进行访问，即只要存在读者正在访问数据，后续到达的那些欲访问数据的读者就无需顾忌此时是否已存在等待访问数据的写者，均直接进行访问。

写者优先的问题中，“写者优先”表现在：写者欲访问数据时，尽可能早地让它访问。只要存在一个写者正在等待访问数据，那么任何后续欲访问数据的读者均不能进行访问。

（1）读者优先

semaphore rmutex = 1;

semaphore wmutex = 1;

int reader\_count = 0;

main()

{

reader();

writer();

}

reader()

{

while(true){

P(rmutex);

if(reader\_count == 0) P(wmutex);

reader\_count ++;

V(rmutex);

读数据;

P(rmutex);

reader\_count--;

if(reader\_count == 0) V(wmutex);

V(rmutex);

}

}

writer()

{

while(true){

P(wmutex);

写数据;

V(wmutex);

}

}

（2）写者优先

semaphore rmutex = 1;

semaphore wmutex = 1;

semaphore s = 1; // 用于控制写者优先

int reader\_count = 0;

main()

{

reader();

writer();

}

reader()

{

while(true){

P(s);

P(rmutex);

if(reader\_count == 0) P(wmutex);

reader\_count ++;

V(rmutex);

V(s);

读数据;

P(rmutex);

reader\_count--;

if(reader\_count == 0) V(wmutex);

V(rmutex);

}

}

writer()

{

while(true){

P(s);

P(wmutex);

写数据;

V(wmutex);

V(s);

}

}

**3.理发师睡眠问题**

一个理发店由一个有N张椅子的等候室和一间理发室组成。若没有要理发的顾客，则理发师就去睡觉；若一个顾客走进理发店且所有的椅子都被占用了，则该顾客就离开理发店；若理发师正在为人理发，则该顾客就找一张空椅子坐下等待；若理发师在睡觉，则顾客就唤醒他。使用信号量设计一个协调理发师和顾客的程序。

【解】customers 用来记录等候的顾客数（不包括正在理发的顾客），barbers记录正在等候顾客的理发师数，其值为0或1，mutex用于互斥地访问count，count记录等候的顾客数，它实际上是customers的一份拷贝，之所以使用count是因为无法读取信号量的当前值。算法伪代码如下：

semaphore customers = 0; // 等候的顾客数

semaphore barbers = 0; // 等候顾客的理发师数

semaphore mutex = 1; // 用于理发师和顾客互斥访问 count

int count = 0; // 等候的顾客数，是customers的一份拷贝

main()

{

barbers();

customers();

}

barbers()

{

while(true){

P(customers); // 是否有等候的顾客

P(mutex);

count --; // 顾客数减 1

V(barbers); // 理发师开始理发

V(mutex);

理发;

}

}

customers()

{

P(mutex);

if(count < N){

count ++; // 若有空椅子则等候的顾客数加 1

V(customers);

V(mutex);

P(barbers);

理发;

}

else{

V(mutex); // 无空椅子则离开

}

}

**4.哲学家就餐问题**

semaphhore chopsticks[5] = {1, 1, 1, 1, 1};

main()

{

philosopher(0);

philosopher(1);

philosopher(2);

philosopher(3);

philosopher(4);

}

philosopher(int i)

{

while(true){

思考;

P(chopstick[i]);

P(chopstick[(i+1)%5]);

进餐;

V(chopstick[i]);

V(chopstick[(i+1)%5]);

}

}

上述算法可以保证不会有相邻的哲学家同时进餐，但有可能死锁。如果每个哲学家都同时拿起了自己左边的筷子，当他们试图去那右边的筷子时，都将因无筷子而无限期地等待下去。

在应用信号量时必须注意两点：

（1）对信号量只能执行P、V操作，且P、V操作必须称对出现：当实现互斥时，它们出现在同一进程中；当实现同步时，则出现在不同的进程中。

（2）同步P操作和互斥P操作在一起时，同步P操作应该放在互斥P操作之前。

### 2.4 死锁

#### 2.4.1 死锁的概念

1. 死锁的定义

所谓**死锁**是指多个进程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。

2. 死锁的必要条件

互斥，占有等待，非剥夺，循环等待

#### 2.4.2 死锁处理策略

死锁预防，死锁避免，死锁检测及解除

#### 2.4.3 死锁预防

要想防止死锁发生，只要破坏死锁的四个必要条件之一即可。

1.互斥

有些资源根本不能同时访问，例如打印机，因此破坏这个条件不太可能。

2.占有等待

（1）资源静态分配法，进程只有在申请到了所有资源后才能开始运行。

（2）进程提出新的资源请求但不能立刻得到满足时，必须主动释放所有已获得的资源。

3.非剥夺

允许操作系统剥夺进程所占有的资源（被动释放）

4.循环等待

资源按序分配法，其基本思想是将所有资源都按类型赋予一个编号，进程称按照编号递增的顺序申请资源，同类资源一次申请完。也就是说，进程提出请求资源Ri，则在以后的请求中，只能请求排在Ri后面的资源。

#### 2.4.4 死锁避免

银行家算法

例2.1 在某时刻系统的资源分配情况如表2.1所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 资源  进  程 | Max | Allocation | Need | Available |
| R1R2 R3R4 | R1R2R3R4 | R1R2R3 R4 | R1R2R3R4 |
| P0  P1  P2  P3  P4 | 0 0 4 4  2 6 5 0  3 6 10 10  0 6 8 4  0 6 6 10 | 0 0 3 2  1 0 0 0  1 3 5 4  0 0 3 2  0 0 1 4 | 0 0 1 2  1 6 5 0  2 3 5 6  0 6 5 2  0 6 5 6 | 1 6 2 2 |

请问

（1）系统状态是否安全？

（2）若进程P2提出资源请求Rquest(1, 2, 2, 2)后，系统能否将资源分配给它？

（3）如果系统立即满足P2的上述要求，请问，系统是否立即进入死锁状态？

【解】

（1）使用安全性算法对上表的状态进行分析，如下表，找到一个安全序列{P0, P3, P1, P2}，故系统是安全的。

开始时刻的安全性检查

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 资源  进  程 | Work | Need | Allocation | Work+Allocation | Finish |
| A B C D | A B C D | A B C D | A B C D |
| P0 | 1 6 2 2 | 0 0 1 2 | 0 0 3 2 | 1 6 5 4 | True |
| P3 | 1 6 5 4 | 0 6 5 2 | 0 0 3 2 | 1 6 8 6 | True |
| P4 | 1 6 8 6 | 0 6 5 6 | 0 0 1 4 | 1 6 9 10 | True |
| P1 | 1 6 9 10 | 1 6 5 0 | 1 0 0 0 | 2 6 9 10 | True |
| P2 | 2 6 9 10 | 2 3 5 6 | 1 3 5 4 | 3 9 14 14 | True |

（2）进程P2发出资源请求Rquest2(1, 2, 2, 2)后，按银行家算法进行检查：

① Rquest2(1, 2, 2, 2)≤Need2(2,3 ,5,6)；

② Rquest2(1, 2, 2, 2)≤Available2(1 ,6, 2, 2)；

③ 系统尝试满足P2的请求，并修改Allocation2，Need2和Available向量：

Allocation2 = Allocation2 + Rquest2 = (2, 5, 7, 6)；

Need2 = Need2 - Rquest2 = (1, 1, 3, 4)；

Available = Available - Rquest2 = (0, 4, 0, 0)；

④系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。

P2申请资源后的资源分配表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 资源  进  程 | Max | Allocation | Need | Available |
| R1R2 R3R4 | R1R2 R3R4 | R1R2 R3R4 | R1R2 R3R4 |
| P0  P1  P2  P3  P4 | 0 0 4 4  2 6 5 0  3 6 10 10  0 6 8 4  0 6 6 10 | 0 0 3 2  1 0 0 0  1 3 5 4  0 0 3 2  0 0 1 4 | 0 0 1 2  1 6 5 0  1 1 3 4  0 6 5 2  0 6 5 6 | 0 4 0 0 |

此时对所有的进程，Available<Needi，即Available不能满足任何进程的请求，故系统进入不安全状态。因此系统不能将资源分配给P2。

（3）系统立即满足P2的请求(1, 2, 2, 2)后，并没有立即进入死锁状态，因为此时没有进程提出新的资源请求，并因得不到资源而进入阻塞状态。只有当上述进程提出新的请求，并导致所有没执行完的多个进程因得不到资源而阻塞，系统才进入死锁状态。

#### 2.4.5死锁检测及解除

1.死锁检测算法

死锁检测算法与死锁避免算法很相似，差别仅在于死锁避免算法将进程今后所有可能的资源请求都考虑在内，而死锁检测算法仅考虑当前资源分配状态和当前资源请求。

2.死锁解除

（1）剥夺资源法；（2）撤销进程法。

## 第3章 内存管理

### 3.1内存管理基础

#### 3.1.1内存管理概念

**1. 程序装入和链接**

每个源程序由编译器进行编译并产生目标代码，而应用程序由许多编译过的目标代码模块组成，这些目标代码模块由链接器进行链接来产生一个可加载的绝对模块。在这个模块中，进程使用逻辑空间地址来引用程序的指令、数据和栈。绝对模块在辅存中以文件形式存储，当系统通过内存管理器获得一块内存后，便调用加载器将其从外存载入内存中。

程序加载（也称装入）一般有绝对加载、静态重定位加载和动态重定位加载三种方式：

**绝对加载**即加载的绝对模块中的地址就是主存中的物理地址。

**静态重定位**就是在加载时，把所有地址都修改为“逻辑地址+模块所在主存起始地址”，然后模块可全部载入主存并运行。

**动态重定位**推迟地址的计算，在运行时执行到一条访存指令再把逻辑地址转换为主存中的物理地址，实际中是通过硬件地址转换机制实现，比如利用一个重定位寄存器，当程序的当前模块加载到主存时，将模块内存起始地址送入重定位寄存器，以后在执行访存指令时自动按“原地址+重定位寄存器”进行访问。

动态重定位的特点是允许程序分配到不连续的内存块，程序只需要装入部分代码即可运行。

**2. 逻辑地址和物理地址**

**3. 内存保护**

内存保护必须通过处理器硬件来实现，因为操作系统无法预测程序可能的访存操作，而且软件实现在时间上非常费时，不合实际。

#### 3.1.2 交换与覆盖

所谓**覆盖**技术，就是把一个大的程序划分为一系列的覆盖，每个覆盖就是一个相对独立的程序单位，把程序执行时并不要求同时装入内存的覆盖组成一组，称为覆盖段，这个覆盖段分配到同一个存储区域，这个存储区域称为覆盖区，它与覆盖段一一对应。

**交换**技术就是把暂时不用的某个程序及数据部分（或全部）从内存移到外存中去，以便腾出必要的内存空间，或把指定的程序或数据从外存读到内存中，并将控制权转给它，让其在系统上运行的一种内存扩充技术。

二者的区别：覆盖由程序员负责，需要程序员提供程序的覆盖结构，交换由操作系统负责；覆盖主要在同一个作业或进程内进行，而交换主要是在作业或进程之间进行。

交换技术和虚拟存储器的相同之处是：都将本应处于实际内存的内容调至辅存，提高系统效率，不同之处是：交换技术并未提供大于实际内存空间的逻辑空间，该技术并不是直接面向用户的；而虚拟存储器则是提供更大的逻辑空间供用户使用，是直接面向用户的。简言之，交换节约了内存，而虚存扩充了内存。

#### 3.1.3 连续分配管理方式

连续分配管理方式首先将内存分为系统区和用户区，系统区分配给操作系统，用户区分配给用户程序使用。用户区可进一步划分。

**1. 单一连续分配**

用户区作为一个整体不再划分，即为单一连续分配。它只能用于单任务的操作系统中。

**2. 固定分区分配**

固定分区将用户区划分为若干个固定大小的分区，每个分区中可以装入一道程序。分区的大小可以不等，但事先必须确定，在运行时不能改变。

为了实现固定分区分配，系统需要建立一张分区说明表，以记录内存的分配情况，表项一般包含分区号、分区的大小、分区的起始地址及状态。

**3. 动态分区分配**

动态分区并不事先将用户区划分成一块块的分区，而是在作业进入主存时，根据作业的大小动态地建立分区，并使分区的大小正好适应作业的需要。因此系统中分区的大小和数目是可变的。

（1）动态分区中的数据结构

为了实现动态分区分配，系统中必须设置相应的数据结构来记录内存的使用情况，常用的数据结构有：

① **空闲分区表**。设置一个表格来登记系统中的空闲分区，每个空闲分区对应一个表项，每个表项包含分区号、分区起始地址、分区大小及状态，如下表所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 分区号 | 大小 | 起始地址 | 状态 |
| 1  2  3  4  5 | 32KB  ...  520KB  ...  ... | 352KB  ...  504KB  ...  ... | 空闲  空表目  空闲  空表目  ... |

② 空闲分区链。用单链表将空闲分区链接起来，构成空闲分区链。

（2）**分区分配算法**

首次适应算法（First Fit），循环首次适应算法（Next Fit），最佳适应算法（Best Fit），最大适应算法（Largest Fit）

工程实践表明：首次适应算法可能比最佳适应算法效果好，而它们两者一定比最大适应算法效果好。

（3）分区的回收

系统在空闲分区表（或空闲分区链）中检查是否有相邻的空闲分区，如有相邻空闲分区，则合并为一个大的空闲区。

（4）紧凑技术

**内部碎片**是指分配给作业的存储空间中未被利用的部分，又称内零头；**外部碎片**是指系统中无法利用的小存储块，又称外零头。如固定分区分配中存在内部碎片，而动态分区分配中存在外部碎片。

解决碎片问题的办法之一是将存储器中所有已分配的分区移动到主存的一端，使本来分散的多个小空闲区连成一个大的空闲区，这种方法称**紧凑（Compaction）**。紧凑技术需要动态重定位的支持。

#### 3.1.4 非连续分配管理方式

**1. 分页管理方式**

（1）分页原理

回顾一下前面提到的早期操作系统所使用的内存管理方式，固定分区会产生内部碎片，动态分区会产生外部碎片，两种技术对内存的利用率都比较低。我们希望内存的使用尽量避免碎片的产生，这就引出了分页的思想：把主存空间划分为大小相等且固定的块，块相对较小，作为主存的基本单位。每个进程也以块为单位进行划分，进程在执行时，以块为单位逐个申请主存中的块空间。分页方式从形式上看，像分区相等的固定分区技术，但它又有本质的不同点：块的大小相对分区要小很多，而且进程也按照块进行划分，进程运行时按块申请主存空间。

这里，我们把进程中的块称为**页**（Page），内存中的块称为页框（Page Frame，或**页帧**）。外存也以同样单位进行划分，直接称为**块**（Block）。

在调度作业运行时，必须将它的所有页面一次调入主存；若主存没有足够的主存空间，则作业等待。这样的分页方式称为**简单分页**或纯分页。

简单分页中的**逻辑地址**结构：

为了便于在内存中找到进程的每个页面所对应的页帧，系统为每个进程建立一张页面映射表，简称**页表**，记录页面对应的页帧号。页表一般放在内存中，图3.2说明了页表的作用。



图3.2 页表的作用

（2）地址变换过程

逻辑地址到物理地址的变换要借助页表来实现，页表通常存放在内存中。为了实现上的方便，系统中设置了一个**页表寄存器**，其中存放页表在内存的起始地址和页表的长度。进程未执行时，页表的起始地址和长度存放在进程控制块中。当进程执行时，才将页表起始地址和长度存入页表寄存器中。

当进程要访问某个逻辑地址时，分页系统的地址变换机构自动地将逻辑地址分为页号和页内偏移两部分，再以页号为索引去检索页表。在执行检索之前，先将页号与页表长度进行比较，如果页号超过了页表长度，则表示本次所访问的地址已超越进程的地址空间，系统产生地址越界中断。若未出现越界，则由页表始址和页号计算出相应页表项的位置，从中得到该页的帧号。最后，将帧号与逻辑地址中的页内偏移拼接在一起，就形成了该逻辑地址的物理地址。图3.3给出了分页存储管理系统中的地址变换机构。

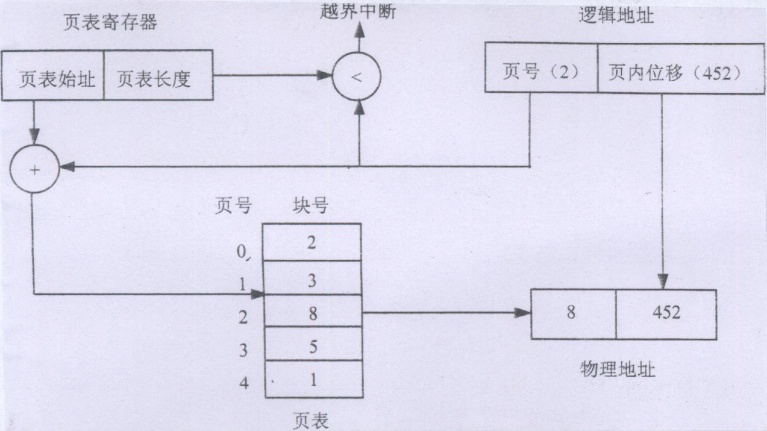


图3.3 分页系统的地址变换过程

若页表放在内存中，则存取一个数据或指令至少需要访问两次主存，第一次是访问页表，得到所访问逻辑地址的物理地址，第二次才根据得到的物理地址存取数据或指令。显然，这种情况下访存速度慢了一倍。为了提高地址变换的速度，可以在地址变换机构中增设一个具有并行查找能力的高速缓冲存储区（一般用联想存储器实现），称为**快表**，将最近使用过的那些页表项存放在快表中，页表的其余部分仍存放在内存中。

**2. 分段管理方式**

（1）分段原理

在分段存储管理系统中，作业的地址空间由若干个逻辑分段组成，每个分段是一组逻辑意义相对完整的信息集合，每个分段都有自己的名字，每个分段都从0开始编址，并采用一段连续的地址空间。因此，整个作业的地址空间是二维的。

分段系统中的逻辑地址结构：

系统为每个进程建立一张**段表**，其中每个表项描述一个分段的信息，表项中包含段号、段长和该段的内存起始地址。

段表项：

（2）地址变换过程

系统中设置了一个段表寄存器，用于存放段表起始地址和段表长度。在进行地址变换时，系统将逻辑地址中的段号与段表长度进行比较，若段号超过了段表长度，则表示段号越界，于是产生越界中断信号；若未越界，则根据段表起始地址和段号计算出该段对应段表项的位置，从中读出该段在内存的起始地址，然后，在检查段内偏移是否超过该段的段长度，若超过则同样发出越界中断信号；若未越界，则将该段的起始地址与段内偏移相加，从而得到了要访问的物理地址。为了提高内存的访问速度，也可以使用快表。

（3）分段与分页的区别

① 页是信息的物理单位，分页是为了实现离散分配方式，仅仅是出于系统管理的需要，而不是用户的需要。段是信息的逻辑单位，分段是为了更好地满足用户的需要。

② 页的大小固定且由系统决定，把逻辑地址划分为页号和页内偏移两部分，是由机器硬件实现的。段的长度不固定，且由用户所编写的程序决定，通常由编译系统在对源程序进行编译时根据信息的性质来划分。

③ 分页系统中作业的地址空间是一维的，程序员只需利用一个值来表示地址；分段系统中作业的地址空间是二维的，程序员在标识一个地址时，既要给出段名，又要给出段内偏移。

**3. 段页式管理方式**

逻辑地址：

每个进程一张段表，每个段一张页表。

在进行地址转换时，首先通过段表查到页表起始地址，然后通过页表找到帧号，最后形成物理地址。

若段表和页表全放在主存中，则访问主存中的一条指令或数据，至少需要访问主存三次，应考虑使用联想存储器。

### 3.2 虚拟内存管理

#### 3.2.1 虚拟内存基本概念

基于程序访问的局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其余部分放在外存，就可以启动程序开始执行。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存，然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换出到外存上，从而腾出空间。对用户而言，用户可用的逻辑地址空间已经不受主存大小的限制，用户感觉到的是一个存储容量比实际物理内存大得多的存储器，这个存储器称为**虚拟内存**。

常用的虚拟内存技术有请求分页管理方式，请求分段管理方式和请求段页式管理方式。

#### 3.2.2 请求分页管理方式

在简单分页的基础上，增加了请求调页、页面置换功能，形成了一种虚拟存储器。

扩充后的页表表项如下图所示。



图3.4 扩充后的页表项

地址变换过程如图3.5所示。

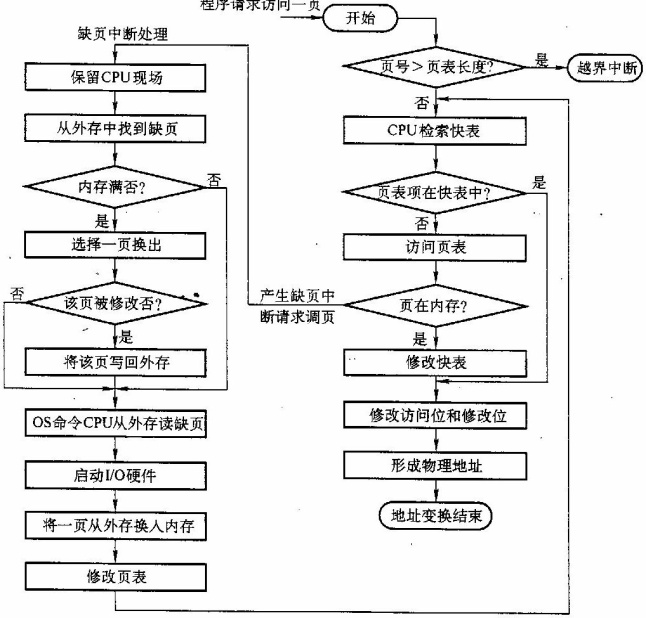


图3.5 请求分页的地址变换过程

#### 3.2.3页面置换算法

1. 最佳置换算法（OPT, OPTimal）

最佳算法是从主存中年选择永远不再需要的页面或在最长的时间以后才需要访问的页面予以淘汰。

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  |  | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 |

× × × × × × × ×

图3.6 OPT算法下驻留集的变化（从上而下，指针动）

2. 先进先出置换算法（FIFO, First In, First Out）

在主存中选择驻留时间最长的页面予以淘汰。

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
|  |  | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 |

× × × × × × × × × × × ×

图3.7 FIFO算法下驻留集的变化（循环队列，指针动）

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 0 | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 0 | 0 | 1 | 2 | 2 | 2 |
|  | 7 | 0 | 1 | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 3 | 3 | 0 | 1 | 1 | 1 |
|  |  | 7 | 0 | 0 | 1 | 2 | 3 | 0 | 4 | 2 | 2 | 2 | 3 | 0 | 0 | 0 |

× × × × × × × × × × × ×

图3.8 FIFO算法下驻留集的变化（从上而下，挤牙膏）

FIFO算法一种异常现象，即在某些情况下会出现分配给进程的页面增多，缺页次数反而增多的奇怪现象，这种现象称**Belady异常**。

堆栈类算法不可能出现Belady异常。

3. 最近最少使用置换算法（LRU, Least Recently Used）

选择最近一段时间内最长时间没有被访问过的页面予以淘汰。

7 0 1 2 0 3 0 4 2 3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0/7 | 1/7 | 2/7 | 0/2 | 1/2 | 2/2 | 3/2 | 0/4 | 1/4 | 2/4 |
|  | 0/0 | 1/0 | 2/0 | 0/0 | 1/0 | 0/0 | 1/0 | 2/0 | 0/3 |
|  |  | 0/1 | 1/1 | 2/1 | 0/3 | 1/3 | 2/3 | 0/2 | 1/2 |

× × × × × × × ×

0 3 2 1 2 0 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0/0 | 1/0 | 2/0 | 0/1 | 1/1 | 2/1 | 0/1 |
| 1/3 | 0/3 | 1/3 | 2/3 | 2/3 | 0/0 | 1/0 |
| 2/2 | 3/2 | 0/2 | 1/2 | 0/2 | 1/2 | 2/2 |

× × ×

图3.9 LRU算法下驻留集的变化（从上而下，计数器）

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 |
|  | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 |
|  |  | 7 | 0 | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 2 | 0 | 3 | 3 | 1 | 2 |

× × × × × × × × × × ×

图3.10 LRU算法下驻留集的变化（从上而下，挤牙膏，被访问则置顶）

4. 最近未使用置换算法（Clock算法）

由硬件给每个页帧增设一个标志位，称为使用位。每访问一页就对应的使用位为1，页面管理软件则定时的（如在处理页故障时）把所有的使用位重新置0。需淘汰时，然选一个使用位为0（可能不唯一旦表明该页最近没有被访问）的页。若所有的页的使用为都为1，则按FIFO规则淘汰。

2 3 2 1 5 2 4 5 3 2 5 2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2\* | 2\* | 2\* | 2\* | 5\* | 5\* | 5\* | 5\* | 3\* | 3\* | 3\* | 3\* |
|  | 3\* | 3\* | 3\* | 3 | 2\* | 2\* | 2\* | 2 | 2\* | 2\* | 2\* |
|  |  |  | 1\* | 1 | 1 | 4\* | 4\* | 4 | 4 | 5\* | 5\* |

× × × × × × × ×

图3.11 NRU算法下驻留集的变化（从上而下，标志位）

例3.1 在页式虚存管理系统中，假定驻留集为m个页，初始所有页均为空，在长为p的引用串中具有n个不同页号（n>m），对于FIFO、LRU两种页面置换算法，试给出页故障数的上限和下限，说明理由，并举例说明。

【解】对于FIFO、LRU置换算法，页故障（即缺页中断）上限均为p，下限均为n。

由于驻留集初始为空，所以n个不同的页面调入内存都要发生缺页中断，因此下限是n。

最坏的情况是，每当访问一个页面时，该页都不在内存，所以上限是p。举例如下。

m=3, p=12, n=4，有引用串1、1、1、2、2、3、3、3、4、4、4、4，则两种置换算法产生的缺页中断数均为4。

#### 3.2.4 页面分配策略

1. 驻留集大小

固定分配策略：为每一个进程分配固定数目的帧，在进程的整个运行期间进程拥有的帧不再变化。

可变分配策略：允许进程拥有的帧在运行期间发生变化。

2. 替换范围

局部替换策略：当某个进程发生缺页时，从本进程的几个页中选择一页换出，然后再调入缺页。

全局替换策略：由操作系统从空闲帧队列中取出一个分配给该进程，当系统的帧已用完时，操作系统才选择某个进程的一页换出。

固定驻留即意味着使用局部替换策略，可变驻留集可使用两种替换策略。

#### 3.2.5 抖动

驻留集（工作集）：指进程在主存的页帧的集合。

**抖动**现象（Thrashing）：指频繁地引起主存页面淘汰后又立即调入，调入后又很快淘汰的现象。

#### 3.2.6 请求分段管理方式

段表项：



#### 3.2.7 请求段页式管理方式

在段页式管理方式上增加了请求调页，页面置换功能。

## 第4章 文件管理

### 4.1文件系统基础

#### 4.1.1文件概念

**文件**是具有文件名的一组相关信息的集合，设备也可以看作一种特殊的文件。**文件系统**是操作系统中与文件管理有关的软件和数据的集合。

#### 4.1.2 文件的逻辑结构与物理结构

文件的**逻辑结构**可分为无结构的流式文件和有结构的记录式文件。

目前的操作系统都不再关心文件的内容，一律当成流式文件。

文件的**物理结构**是指文件在外存上的存储组织形式。常见的文件物理结构有：顺序结构、链接结构和索引结构。

表4-1 文件物理结构的比较

|  |  |  |  |
| --- | --- | --- | --- |
|  | 访问第n个记录 | 优点 | 缺点 |
| 顺序结构 | 需访问磁盘1次 | 顺序存取时速度快，当文件是定长时可以根据文件起始地址及记录长度进行随机访问 | 文件存储要求连续的存储卡空间，会产生碎片，也不利于文件的动态扩充 |
| 链接结构 | 需访问磁盘n次 | 可以解决外存的碎片问题，提高了外存空间的利用率，动态增长较方便 | 只能按照文件的指针链顺序访问，查找效率低，指针信息消耗外存空间 |
| 索引结构 | m级需访问磁盘m+1次 | 可以随机访问，易于文件的增删 | 索引表增加存储空间的开销，索引表的查找策略对文件系统效率影响较大 |

#### 4.1.3目录结构

1. 文件控制块和文件目录

文件由文件控制块和文件体两部分组成。**文件体**即文件本身，**文件控制块**（FCB）用于存放文件的各种属性信息。

文件控制块的集合称为**文件目录**，文件目录是一张表格，每个表项可以是文件控制块，也可以是指向文件控制块的指针。

2. 单级目录结构

文件系统只有一张目录表

3. 二级目录结构

文件系统中有一个主文件目录MFD(Master File Directory)，每个用户占用一个表项，每一项指向一个用户文件目录UFD(User File Directory)，用户文件目录下才是该用户的文件。

4. 树形目录结构

将二级目录加以推广，就形成了多级目录结构，即树形目录结构。

当前目录：考虑到进程在一段时间内所访问的文件通常具有局部性，可在这一段时间内指定某个目录作为当前目录（或称为工作目录）。进程对个文件的访问都是相对于当前目录进行的。这样可以节省很多重复的查找。

5. 图形目录结构

在树形目录的基础上增加了一些指向同一结点的有向边，使整个目录成为一个有向无环图。无环图目录的引入是为了实现文件共享，这种结构允许若干目录共同指向一个被共享的子目录或文件。

#### 4.1.4 文件共享

文件共享有三种典型的方法：

**1. 硬链接（hard linking）**

允许多个目录项指向同一个文件，形成了无环图结构。

**2. 间接链接（indirect linking）**

此时目录项不再是一个FCB，而是一个FCB指针。将原来的FCB中的文件名和文件属性信息分开，文件属性单独构成一个数据结构，称为**索引节点**（又称i节点），而文件目录中的每个目录项仅由文件名和该文件的索引节点指针构成。UNIX采用了这种方式。

目录项是一个FCB，则共享方式是硬链接；目录项是一个FCB指针，则共享方式是间接链接。

使用硬链接时，共享的文件同时有若干相同的FCB，当文件内容发生变化时会带来一致性维护开销。

**3. 符号链接**

例如，目录B共享A目录中的文件C时，在目录B中创建一个文件D，D的内容是C的路径（相对路径或绝对路径）。Windows采用了这种方式，即快捷方式。

#### 4.1.5 文件保护

1.访问控制

访问控制就是对不同的用户访问同一个文件采取不同的访问类型。

访问类型有：读、写、执行、添加、删除、列表清单等。

访问控制矩阵，访问控制表（相当于矩阵的一行），用户权限表（相当于矩阵的一列）

2.口令

3. 密码

4. 文件转储和恢复

文件转储：全量转储，增量转储

文件恢复：首先从最近一次全量转储文件恢复全部文件，然后由旧到新从增量转储文件恢复文件（可能一个文件被转储过若干次，但只恢复最近一次转储的副本，其他则被覆盖掉）。

### 4.2 文件系统实现

#### 4.2.1 文件系统层次结构



图4.1 文件系统层次结构

#### 4.2.2 目录实现与文件实现

略。

### 4.3 磁盘组织与管理

#### 4.3.1 磁盘的结构

磁盘地址用“柱面号-盘面号-扇区”表示。

#### 4.3.2 磁盘调度算法

（1）先来先服务算法（FCFS）

（2）最短寻道时间优先（SSTF）算法

（3）扫描算法（SCAN），又称为电梯算法

（4）循环扫描算法（CSCAN）

#### 4.3.3 存储空间的管理

1. 空闲文件目录

存储设备上的一个空闲块也可以看作是一个文件，称为空闲文件或空白文件。空闲文件目录法为这些空闲文件单独建立一个目录，每个空闲文件在这个目录中占一个表目。

2. 空闲表法和空闲链表法

与内存动态分区管理方式中的空闲分区表和空闲链表类似。

3. 位示图

这种方法是为文件存储设备建立一张位示图（也称位图），以反映整个存储空间的分配情况。在位示图中，每一个二进制位都对应一个页帧，当某位为“1”时，表示对应的页帧分配，若某位为“0”，则表示对应的页帧空闲。

4. 成组链接法

空闲表法和空闲链表法度不适合用于大型文件系统，因为这会使空闲表或空闲链表太大。在UNIX系统中采用的是成组链接法，这种方法结合了空闲表和空闲链表两种方法，克服了表太大的缺点。假设磁盘最初全为空闲扇区，其成组链接如图4-2所示。



图4-2 成组链接法示意图

## 第5章 输入输出（I/O）管理

### 5.1 I/O管理概述

#### 5.1.1 I/O设备

1. 人机交互设备

又称为慢速I/O设备。这类设备主要与用户打交道，数据交换速度相对较慢，通常是以字节为单位进行数据交换。如键盘等。

2. 存储设备

这类设备主要用于存储数据，数据交换速度较快，通常以多字节组成的块为单位进行数据交换。主要有磁带，磁盘，光盘等。

3. 网络通信设备

这类设备数据交换速度通常高于人机交互设备，低于存储设备，主要有各种网络接口、调制解调器等。

#### 5.1.2 I/O目标

方便使用，提高效率，方便控制

#### 5.1.3 I/O管理功能

设备分配，设备控制，缓冲管理，设备独立性

#### 5.1.4 I/O应用接口

即提供一组系统调用来访问设备。

#### 5.1.5 I/O控制方式

1. 程序查询方式

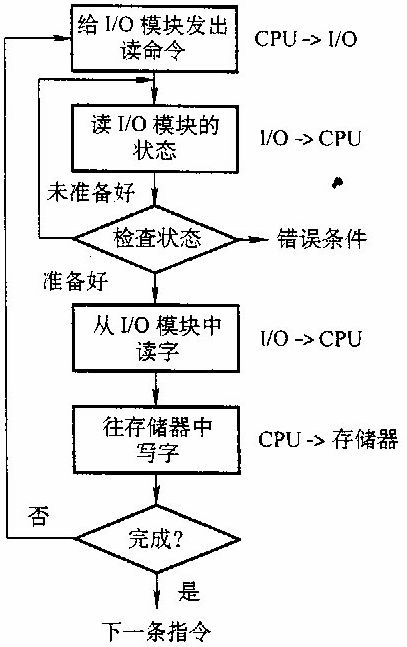
如图5.1a所示，在该方式中，由于CPU的高速性和I/O设备的低速性，致使CPU的绝大部分时间都处于等待I/O设备完成数据I/O的循环测试中，造成CPU的极大浪费。在该方式中，CPU之所以不断地测试I/O设备的状态，就是因为在CPU中无中断机构，使I/O设备无法向CPU报告它已经完成了一个字符的输入操作。

2. 中断方式

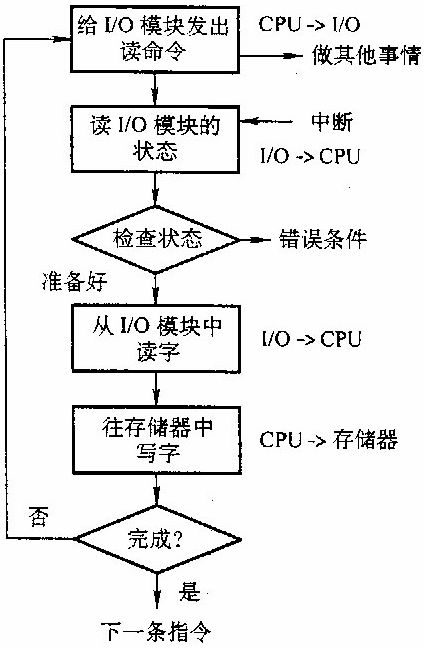
中断方式的思想是，允许I/O模块主动打断CPU的运行并请求服务，从而“解放”CPU，使得其向I/O模块发送命令后可以继续做其他事情。如图5.1b所示。

3. DMA方式

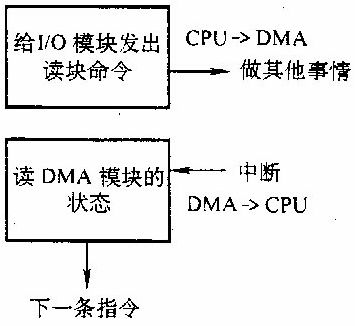
DMA模块直接与存储器交互，传送整个数据块，每次传送一个字，这个过程不需要CPU参与。当传送结束后，DMA模块发送一个中断信号给处理器。因此只有在传送开始和结束时才需要CPU的参与。如图5.1c所示。



(a)



(b)



(c)

图5.1 I/O控制方式

a) 程序查询方式 b) 中断方式 c) DMA方式

### 5.2 I/O核心子系统

#### 5.2.1 I/O层次结构

|  |
| --- |
| 用户层I/O |
| 系统调用接口，设备无关的操作系统软件 |
| 设备驱动程序及中断处理程序 |
| 硬件 |

图5.2 I/O层次结构

#### 5.2.2 高速缓存与缓冲区

1. 磁盘高速缓存（Disk Cathe）

指利用内存中的存储空间来暂存从磁盘读取的数据。

2. 缓冲区

引入I/O缓冲区的主要原因如下：

（1）缓和CPU与I/O设备速度不匹配的矛盾。一般情况下，程序的运行过程是时而进行计算，时而进行输入或输出的。以输出为例，如果没有输出缓冲区，则程序在输出时，必然由于打印的速度跟不上而是CPU停下来等待；然而在计算阶段，打印机又无事可做。如果设置一个缓冲区，程序可以将带输出的数据先输出到缓冲区中，然后继续执行；而打印机则可以从缓冲区取出数据慢慢打印。

（2）减少中断CPU的次数。例如，假如某设备每接收到一个字节就中断CPU一次，若设置一个具有4字节的缓冲寄存器，则可使CPU被中断的次数降低为前者的1/4。

（3）提高CPU和I/O设备之间的并行性。由于在CPU和设备之间引入了缓冲区，CPU可以从缓冲区读取或向缓冲区写入信息，相应地设备也可以向缓冲区写入或从缓冲区读取信息。在CPU工作的同时，设备也能进行输入输出操作，这样CPU和I/O设备及可以并行工作。

根据系统设置的缓冲区个数，可以将缓冲技术分为单缓冲、双缓冲、循环缓冲和缓冲池。

（1）单缓冲

操作系统只设置了一个缓冲区，因此设备与处理机对缓冲区的操作时串行的。

（2）双缓冲

一个输入缓冲，一个输出缓冲。

（3）循环缓冲

循环缓冲包含多个大小相等的缓冲区，每个缓冲区有一个链接指针指向下一个缓冲区，最后一个缓冲区指向第一个缓冲区，这样构成一个环形。循环缓冲用于输入输出时，还需要设置两个指针in和out。

（4）缓冲池

缓冲池有多个缓冲区组成，缓冲池中的缓冲区按其使用状况可分成三个队列：空缓冲队列、装满输入数据的缓冲队列（输入队列）和装满输出数据的缓冲队列（输出队列）。除上述三个队列之外，还应具有四种工作缓冲区：用于收容输入数据的工作缓冲区、用于提取输入数据的工作缓冲区、用于收容输出数据的工作缓冲区和用于提取输出数据的工作缓冲区。

当输入进程需要输入数据时，便从空缓冲区队列的队首摘下一个空缓冲区，把它作为收容输入工作缓冲区，然后把数据输入其中，装满后再将它挂到输入队列队尾。当计算进程需要输入数据时，便从输入队列取得一个缓冲区作为提取输入工作缓冲区，进程从中提取数据，数据用完后再将它挂到空缓冲队列对为。当计算进程需要输出数据时，便从空缓冲队列的队首摘下一个空缓冲区，把它作为收容输出工作缓冲区，当其中装满输出数据后，再将它挂到输出队列队尾。当要输出时，由输出进程从输出队列中取得一个装满输出数据的缓冲区，作为提取输出工作缓冲区，当数据提取完后，再将它挂到空缓冲区队列的队尾。

3. 高速缓存与缓冲区的对比

表5.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 高速缓存 | | 缓冲区 |
| 相同点 | | 都是介于高速设备和低速设备之间 | | |
| 区别 | 存放数据 | 存放的是低速设备上的某些数据的拷贝，也就是高速缓存上有的低速设备上必然有 | 存放的是低速设备传递给高速设备的数据（或者反过来），而这些数据在低速设备（或者高速设备）上却不一定有备份，这些数据正在从缓冲区传送到高速设备（或者低速设备） | |
| 目的 | 高速缓存放的是高速设备经常要访问的数据，如果高速设备要访问的数据不在高速缓存中，高速设备就需要访问低速设备 | 高速设备和低速设备的通信都要经过缓冲区，高速设备永远不会直接去访问低速设备。 | |

#### 5.2.3 设备分配与回收

1. 设备的使用方式：独占式，分时共享式，SPOOLing方式

2. 设备分配方式

（1）静态分配：在用户作业开始执行前，由系统一次性分配该作业所要求的全部设备。一旦分配后，这些设备就一直未改作业所占用，直到该作业结束。静态分配方式不会出现死锁，但设备的利用率很低。

（2）动态分配：在进程执行过程中按需分配。

对于独占设备，既可采用动态分配方式也可以采用静态分配方式，往往采用静态分配方式。共享设备可以被多个进程所共享，一般采用动态分配方式，常用的分配算法有先来先服务和优先级高者优先。

#### 5.2.4 假脱机技术（SPOOLing）

假脱机技术（SPOOLing）是操作系统的采用的一项将独占设备改造成共享设备的技术。

SPOOLing系统的组成如图5.3所示，主要包括三个部分：输入井和输出井，输入缓冲区和输出缓冲区，输入进程和输出进程。



图5.3 Spooling系统的组成

在该系统中，输入进程模拟脱机输入时外围设备控制器的功能，把低速输入设备上的数据通过过输入缓冲区传送到输入井。当CPU需要输入数据时，直接将数据从输入井读入内存。

输出进程模拟脱机输出时外围设备控制器的功能，把用户要求的数据先从内存传送到输出井，待输出设备空闲时，再将输出井中的数据通过输出缓冲区传送到输出设备。

#### 5.2.5 出错处理

操作系统可以采用内存保护，这样一来就可以预防许多硬件和应用程序的错误。一些设备硬件上的故障也不会导致系统的完全崩溃。

## 参考资料

[1] 操作系统考研指导，曾平，清华大学出版社，2003年1月第1版

[2] 2011计算机专业基础综合考试指导全书，王道论坛，2010年9月第1版

由于时间仓促，错误和疏漏之处在所难免，欢迎414的同学指出错误。若

(1) 指出10个以上错误；或

(2) 添加5个以上知识点；

(3) 添加3个以上经典例题。

将会被载入~~史册~~感谢名单。

版本：2.1，最后更新日期：2010/12/02

官方地址：<http://www.gotothu.com> （未来清华人）

作者：戴方勤

个人博客：<http://www.yanjiuyanjiu.com/> （研究研究）

本手册遵循创作共享协议2.0，禁止一切商业用途。