

操作系统期末复习

By Olittle

✚ 信号量及 P、V 操作：

信号量：semaphore

- 一个特殊变量
- 用于进程间传递信号的一个整数值
- 定义如下：

```

struc semaphore
{
int count;
queueType queue;
}
    
```

- 信号量说明：semaphore s;

P 操作(semWait)

```

P(s)
{
s.count --;
if (s.count < 0)
{
该进程状态置为等待状态；
将该进程的 PCB 插入相应的等待队列
末尾 s.queue；重新调度
}
}
    
```

V 操作(semSignal)

```

V(s)
{
s.count ++;
if (s.count <= 0)
{
唤醒相应等待队列 s.queue 中等待的一个进程；
改变其状态为就绪态，并将其插入就绪
队列；
}
}
    
```

二元信号量的定义

```

struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}

```

✧ 有关说明:

1. P、V 操作为原语操作

原语(primitive or atomic action)

完成某种特定功能的一段程序，具有不可分割性或不可中断性

即原语的执行必须是连续的，在执行过程中不允许被中断，可以通过软件解法、compare&swap、exchange、屏蔽中断来实现

2. 在信号量上定义了三个操作
初始化(非负数)、P 操作、V 操作

3. 互斥量

一种锁机制

进行加锁(P 操作)和解锁(V 操作)的进程是同一个进程

二元信号量：可能一个进程执行 P 操作，另一个进程执行 V 操作

4. 强信号量与弱信号量

取决于按照什么顺序从等待队列中移走进程

强信号量不会产生“饥饿”

进程的同步: synchronism

指系统中多个进程中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务

● 信号量及 P、V 操作讨论

- 1) 信号量值的物理解释

$s.count > 0$:

$s.count$ 是可以执行 P(s) 而不被阻塞的进程数

$s.count < 0$:

$s.count$ 的大小是阻塞在 $s.queue$ 队列中的进程数

P(s): 表示申请一个资源; V(s): 表示释放一个资源

- 2) P、V 操作应成对出现

- 当为互斥操作时，它们同处于同一进程
- 当为同步操作时，则不在同一进程中出现
- 如果 $P(s1)$ 和 $P(s2)$ 两个操作在一起，那么 P 操作的顺序至关重要，一个同步 P 操作与一个互斥 P 操作在一起时同步 P 操作在互斥 P 操作前
- 两个 V 操作无关紧要

3)P、V 操作的优缺点

优点：简单，而且表达能力强（用 P,V 操作可解决任何同步互斥问题）

缺点：不够安全； $P、V$ 操作使用不当会出现死锁；遇到复杂同步互斥问题时实现复杂



管程

一种同步机制

是关于共享资源的数据及在其上操作的一组过程(或共享数据结构及其规定的所有操作)

管程的四个组成部分：

- 名称
- 数据结构说明
- 对该数据结构进行操作的一组过程/函数
- 初始化语句

管程的三个主要的特性

- (1) 模块化，一个管程是一个基本程序单位，可以单独编译
- (2) 抽象数据类型，管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码
- (3) 信息掩蔽，管程是半透明的，管程中的外部过程（函数）实现了某些功能，至于这些功能是怎样实现的，在其外部则是不可见的

管程有如下几个要素：

- (1) 管程中的共享变量在管程外部是不可见的，外部只能通过调用管程中所说明的外部过程（函数）来间接地访问管程中的共享变量
- (2) 为了保证管程共享变量的数据完整性，规定管程互斥进入
- (3) 管程通常是用来管理资源的，因而在管程中应当设置进程等待队列以及相应的等待/唤醒操作

因为管程是互斥进入的，所以当有一个进程试图进入一个已被占用的管程时，应当在管程的入口处等待，所以在管程的入口处应当有一个进程等待队列，称作**入口等待队列**。如果进程 P 唤醒进程 Q ，则 P 等待 Q 继续，如果进程 Q 在执行又唤醒进程 R ，则 Q 等待 R 继续，……，如此，在管程内部，由于执行唤醒操作，可能会出现多个等待进程，因而还需要有一个进程等待队列，这个等待队列被称为**紧急等待队列**。它的优先级应当高于入口等待队列的优先级

此在管程内部可以说明和使用一种特殊类型的变量，称作**条件变量**：

var c:condition;

对于条件型变量，可以执行 **cwait** 和 **csignal** 操作

cwait (c):

如果紧急等待队列非空，则唤醒第一个等待者；否则释放管程的互斥权，执行此操作的进程进入 c 链尾部

csignal (c):

如果 c 链为空，则相当于空操作，执行此操作的进程继续；否则唤醒第一个等待者，执行此操作的进程进入紧急等待队列的末尾

```

Struct ONE_INSTANCE{
semaphore mutex;//入口互斥队列, 初
值 1
semaphore urgent;//紧急等待队列,初
值 0
inturgent_count; //紧急等待队列计数,
初值 0
};
ClassCMonitorElements{
public:
voidenter(ONE_INSTANCE instance);
voidleave(ONE_INSTANCE instance);
voidcwait(ONE_INSTANCE instance,
semaphore s, intcount);
voidcsignal(ONE_INSTANCEinstance,
semaphore s, intcount);
};
Void enter(ONE_INSTANCE instance)
{
P(instance.mutex);
};
Void leave(ONE_INSTANCE instance)
{
if(instance.urgent_count> 0) {
instance.urgent_count--;
V(instance.urgent);
}
}

else
V(instance.mutex);
};
Void cwait(ONE_INSTANCE instance,
semaphore s, intcount)
{
count++;
if (instance.urgent_count>0){
instance.urgent_count--;
V(instance.urgent);
}
else{
V(instance.mutex);
P(s);
}
};
Void csignal(ONE_INSTANCEinstance,
semaphore s, intcount)
{
if(count>0){
count--;
instance.urgent_count++;
V(s);
P(instance.urgent)
}
};

```

//自己把读写者问题解决掉（能自己写代码）

死锁

死锁的定义

一组进程中，每个进程都无限等待被该组进程中另一进程所占有的资源，因而永远无法得到的资源，这种现象称为**进程死锁**，这一组进程就称为**死锁进程**

可重用资源：可以被多个进程多次使用

- 可抢占资源与不可抢占资源
- 处理器、I/O 通道、内存、文件

可消耗资源：只可使用一次的可以创建和销毁的资源

- 信号、中断、消息

资源分配图

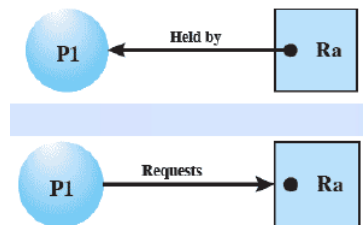
用有向图描述系统资源和进程的状态

资源类：用方框表示

资源实例：用方框中的黑圆点表示

进程：用圆圈中加进程名表示

分配边：资源实例->进程的一条有向边申请边：进程->资源类的一条有向边



产生死锁的条件

➤ **互斥使用(资源独占)**

一个资源每次只能给一个进程使用

➤ **占有且等待(请求和保持，部分分配)**

一个进程在申请新的资源的同时保持对原有资源的占有

➤ **不可抢占(不可剥夺)**

资源申请者不能强行的从资源占有者手中夺取资源，资源只能由占有者自愿释放

➤ **循环等待**

存在一个进程等待队列

{P1, P2, ..., Pn},

其中 P1 等待 P2 占有的资源，P2 等待 P3 占有的资源，…，Pn 等待 P1 占有的资源，形成一个进程等待环路

🚩 **死锁预防**

定义：

在设计系统时，通过确定资源分配算法，排除发生死锁的可能性

具体的做法是防止产生死锁的四个条件中任何一个发生

1. 破坏“**互斥使用/资源独占**”条件（由于是资源独占，才会使得其他进程得不到自愿，导致进程挂起，死锁等）[虚设备技术]

资源转换技术：把独占资源变为共享资源

SPOOLing 技术的引入

解决不允许任何进程直接占有打印机的问题

设计一个“精灵”程序 daemon 负责管理打印机，进程需要打印时，将请求发给该 daemon，由它完成打印任务

2. 破坏“**占有且等待**”条件（这样一个进程如果获得了资源，那么就一定会执行，不会出现占有资源却被挂起，对资源的浪费）[资源的静态分配、释放已占有资源]

实现方案：要求每个进程在运行前必须一次性申请它所要求的所有资源，且仅当该进程所要资源均可满足时才给予一次性分配

问题：资源利用率低；“饥饿”现象

3. 破坏“**不可抢占**”条件（这样当一个进程当前执行的条件得不到满足时，为其他进程的而执行创造条件，从而使其他进程执行完全）[允许抢夺资源]

实现方案 1：在允许进程动态申请资源前提下规定，一个进程在申请新的资源不能立即得到满足而变为等待状态之前，必须释放已占有的全部资源，若需要再重新申请

实现方案 2：当一个进程申请的资源被其他进程占用时，可以通过操作系统抢占这一资

源(两个进程优先级不同)

局限性：适用于状态易于保存和恢复的资源，如 CPU、内存

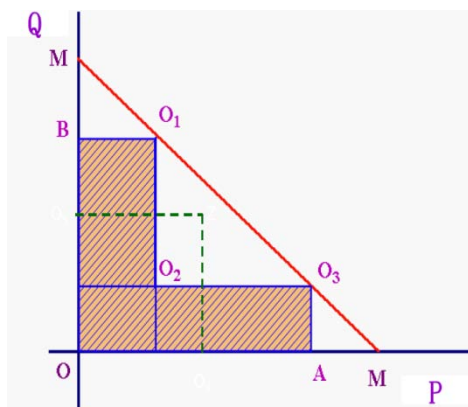
4. 破坏“**循环等待**”条件（这样就不会出现两个进程各自占有另外一个进程所申请的资源，也就是不会有循环等待）[资源的有序分配]

通过定义资源类型的线性顺序实现

实施方案：资源有序分配法

把系统中所有资源编号，进程在申请资源时必须严格按资源编号的递增次序进行，否则操作系统不予分配

(死锁避免)



死锁避免定义: (动态)

在系统运行过程中，对进程发出的每一个系统能够满的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统发生死锁或可能发生死锁，则不予分配，否则予以分配

安全序列（这样保证 P1 能执行，依次下去 P2 也能执行，P3……Pn 这样所有的进程都能够执行下去，所以是安全的，不会发生死锁）

一个进程序列{P1, ..., Pn}是安全的，如果对于每一个进程 Pi(1≤i≤n)，它以后尚需要的资源量不超过系统当前剩余资源量与所有进程 Pj(j < i)当前占有资源量之和，系统处于安全状态

安全状态一定没有死锁发生

银行家算法


当进程请求一组资源时，假设同意该请求，从而改变了系统的状态，然后确定其结果是否还处于安全状态，如果是，同意这个请求，如果不是，阻塞改进程知道同意该请求后仍然是安全的。（前提是：进程数和资源数是确定的）

关于中间问题：当前进程中是否有一个进程可以执行到结束？（对于进程 i, $C_{ij} - A_{ij} < V_j$ ）

Claim 矩阵 (C), Allocation 矩阵 (A), Resource 矩阵 (R), Available 矩阵 (V)

死锁避免相对于死锁预防缺点：

1. 必须声明每个进程请求的最大资源
2. 考虑进程必须是无关系的，也就是说它们的执行顺序没有任何同步要求的限制
3. 分配的资源数目是固定的
4. 占有资源是，进程不能退出。

 **死锁检测：**

允许死锁发生，操作系统不断监视系统进展情况，判断死锁是否发生

一旦死锁发生则采取专门的措施，解除死锁并以最小的代价恢复操作系统运行

检测时机：

❑ 当进程由于资源请求不满足而等待时检测死锁
(其缺点是系统的开销大)

❑ 定时检测

❑ 系统资源利用率下降时检测死锁

检测方法: 1. 利用资源分配表, 查找是否有循环等待; 2. 查找是否存在安全序列(每次检测的时候都要保证进程数是确定的)

死锁的解除方法如下:

1) 撤消所有死锁进程

2) 进程回退(Roll back)再启动

3) 按照某种原则逐一撤消死锁进程, 直到安全状态

4) 按照某种原则逐一抢占资源(资源被抢占的进程必须回退到之前的对应状态), 直到安全状态

死锁定理

• 如果资源分配图中没有环路, 则系统中没有死锁, 如果图中存在环路则系统中可能存在死锁

• 如果每个资源类中只包含一个资源实例, 则环路是死锁存在的充分必要条件(如果每个资源类只有一个资源实例的话, 那当然会是循环等待啦~)

资源分配图化简步骤:

1) 找一个非孤立点进程结点且只有分配边, 去掉分配边, 将其变为孤立结点

2) 再把相应的资源分配给一个等待该资源的进程, 即将某进程的申请边变为分配边

死锁解决方案小结				
原则	资源分配策略	不同的方案	主要优点	主要缺点
预防	保守的; 预提交资源	一次性请求所有资源	<ul style="list-style-type: none"> 对执行一连串活动的进程非常有效 不需要抢占 	<ul style="list-style-type: none"> 低效 延迟进程的初始化 必须知道将来的资源请求
		抢占	<ul style="list-style-type: none"> 用于状态易于保存和恢复的资源时非常方便 	<ul style="list-style-type: none"> 过于经常地没必要地抢占
		资源排序	<ul style="list-style-type: none"> 通过编译时检测是可以实施的 既然问题已经在系统设计时解决了, 不需要在运行时间计算 	<ul style="list-style-type: none"> 禁止增加的资源请求
避免	处于检测和预防中间	操作以发现至少一条安全路径	<ul style="list-style-type: none"> 不需要抢占 	<ul style="list-style-type: none"> 必须知道将来的资源请求 进程不能被长时间阻塞
检测	非常自由; 只要可能, 请求的资源都允许	周期性地调用以测试死锁	<ul style="list-style-type: none"> 不会延迟进程的初始化 易于在线处理 	<ul style="list-style-type: none"> 固有的抢占被丢失

Comparison:

• 死锁: 两个或两个以上的进程因其中的每个进程都在等待其他进程做完某些事情而不能继续执行

• 活锁: 两个或两个以上进程为了响应其他进程中的变化而持续改变自己的状态但不做有用的工作(加锁, 轮询, 没有进展也没有阻塞)

- **饥饿**：是指一个可运行的进程尽管能继续执行，但被调度程序无限期地忽视，而不能被调度执行的情形

哲学家就餐问题的四种解决办法：

1. 直接通过协调处理共享资源~
2. 允许 4 人同时坐在桌子上
3. 使用管程（monitor）
4. 课件~

存储体系

内存管理需求

内存

由存储单元（字节或字）组成的一维连续的地址空间，简称内存空间。用来存放当前正在运行程序的代码及数据

分为：

- **系统区**：用于存放操作系统及相关内容
- **用户区**：用于装入并存放用户程序和数据

为了保证 CPU 执行指令时可正确访问存储单元，需将用户程序中的**逻辑地址**转换为运行时由机器直接寻址的**物理地址**，这一过程称为地址重定位（静态重定位，动态重定位）。

保护

为多个程序共享内存提供保障，使在内存中的各道程序，只能访问它自己的区域，避免各道程序间相互干扰

共享

- 允许多个进程访问内存的同一区域
 - 一个程序派生出多个进程
 - 多个进程需要访问相同数据
- 提供对内存共享区域的保护

逻辑组织

- 内存：线性地址空间
- 程序：由多个模块组成
 - 可独立编写和编译
 - 运行时系统解析从一个模块到其他模块的所有调用
- 内存管理需求
 - 不同模块，不同保护级别
 - 一些模块可被多个进程共享
- 解决方法→ 分段(Segmentation)[对于分段之后，可以按模块对程序进行划分，这样实现不同模块不同的保护级。

覆盖技术与交换技术

- 共同点：进程的程序和数据主要放在外存，当前需要执行的部分放在内存，内外存之间进行信息交换
- 不同点：
 - **覆盖**：一个程序的若干子程序段，或几个程序的某些部分共享某一个存储空间（一般要求程序各模块之间有明确的调用结构，程序员要向系统指明覆盖结构，然后由操作系统完成自动覆盖）

缺点:

对用户不透明, 增加了用户负担

■ 交换技术

- 当内存空间紧张时, 系统将内存中某些进程暂时移到外存, 把外存中某些进程换进内存, 占据前者所占用的区域, 这种技术是进程在内存与外存之间的动态调度 (由于是将内存中的某些进程暂时移到外存, 所以需要有一个盘交换区。[分时系统])

存储管理方案

1. 单一用户(连续区)存储管理方案

单用户系统在一段时间内, 只有一个进程在内存, 故内存分配管理十分简单, 内存利用率低。内存分为两个区域, 一个供操作系统使用, 一个供用户使用

2. 固定分区存储管理方案

系统生成时, 把可分配的内存空间分割成若干个连续区域, 每一区域称为分区

每个分区的大小可以相同也可以不同, 分区大小固定不变, 每个分区装一个且只能装一个进程

存储分配:

如果有合适的空闲区, 则分配给进程

3. 动态(可变)分区存储管理方案(1/6)

• 基本思想

- ② 内存不是预先划分好的(分区长度和数目可变)
- ② 装入程序时, 根据其最大内存需求和当前内存使用状况来决定是否分配
- ② 若有足够的空间, 则按需分割一块区域(一个分区)给该程序; 否则令其等待

• 数据结构

- ② 空闲块表——记录了空闲区起始地址和长度
- ② 已分配区表

• 内存回收算法

当某一块归还后, 前后空间合并, 修改内存空闲块表

考虑: 上邻、下邻、上下相邻、上下不相邻

• “外碎片”问题

经过一段时间的分配回收后, 内存中存在很多很小的空闲块。它们每一个都很小, 不足以满足分配要求; 但其总和满足分配要求。这些空闲块被称为碎片

导致内存利用率下降

• “外碎片”问题的解决

压缩技术: 通过在内存移动程序, 将所有小的空闲区域合并为大的空闲区域

4. 伙伴系统

- 整个可用空间看作一块 $2U$, 如果进程申请的空间大小 s 满足, $2^{U-1} < s \leq 2^U$, 则分配整个块, 否则, 将块划分为两个大小相等的伙伴, 大小为 2^{U-1} , 一直进行下去直到大于或等于 s 的最小块产生

5. 页式存储管理方案

• 用户程序地址空间划分

把用户程序地址空间划分成大小相等的部分, 称为页

内存空间按页的大小划分为大小相等的区域, 称为内存块(物理页面, 页框, 页帧)

• 内存分配(规则)

以页为单位进行分配, 并按进程的页数多少来分配。逻辑上相邻的页, 物理上不一定相邻

• 页表

系统为每个进程建立一个页表，页表给出逻辑页号和具体内存块号相应的关系

页表放在内存（属于进程的现场信息）

•空闲块管理

位示图

内存的分配算法

✓计算一个进程所需要的总块数N

✓查位示图：是否还有N个空闲块

✓如果有足够的空闲块，则页表长度设为N，可填入PCB中；申请页表区，把页表始址填入PCB

✓依次分配N个空闲块，将内存块号和页号填入页表

✓修改位示图

✧ 硬件的支持

•系统设置一对寄存器

② 页表始址寄存器

用于保存正在运行进程的页表的始址

② 页表长度寄存器

用于保存正在运行进程的段表的长度

✧ 相联存储器（associative memory）TLB（Translation Look-aside Buffers）

快表

用途：

保存正在运行进程的页表的子集（部分表项）

特点：按内容并行查找

快表表项：

页号；内存块号；标识位；置换位

6.段式存储管理方案

(1)设计思想

•用户程序地址空间划分

按程序自身的逻辑关系划分为若干个程序段，每个程序段都有一个段名，且有一个段号。段号从0开始，每一段也从0开始编址，段内地址是连续的

•内存划分

内存空间被动态的划分为若干个长度不相同的区域，称为物理段，每个物理段由起始地址和长度确定

•内存分配规则

以段为单位分配内存，每一个段在内存中占据连续空间（内存随机分割，需要多少分配多少），但各段之间可以不连续存放

段表

记录了段号、段的首（地）址和长度之间的关系

每一个程序设置一个段表，放在内存

属于进程的现场信息

段式存储管理方案与可变分区存储管理方案的相同点与不同点？

相同点：内存空间被动态的划分为若干个长度不相同的区域

不同点：段式存储管理只要求每一段在内存中占据连续的空间，而可变分区存储管理方案是每个程序都要在内存中占据连续的空间。

地址映射机制

- 系统设置一对寄存器

▣ 段表始址寄存器：

用于保存正在运行进程的段表的始址

▣ 段表长度寄存器：

用于保存正在运行进程的段表的长度

- 相联存储器——快表

快表项目：段号；段始址；段长度；标识（状态）位；访问位（置换位）

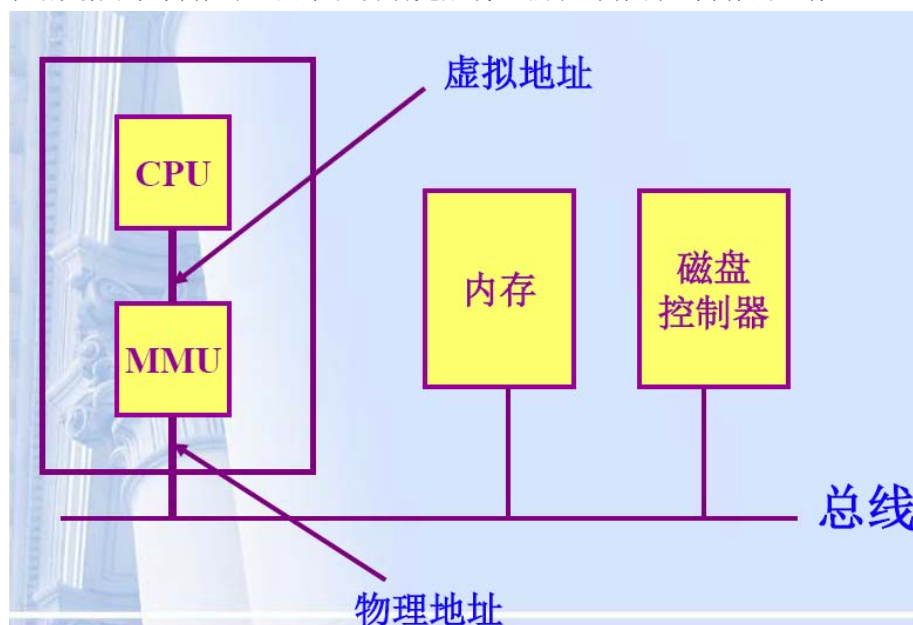
✓ 虚拟内存管理

虚拟存储器

把内存与外存有机地结合起来使用，从而得到一个容量很大的“内存”，即虚存

虚拟存储技术

当进程运行时，先将其一部分装入内存，另一部分暂留在外存，当要执行的指令或访问的数据不在内存时，由系统自动完成将它们从外存调入内存的工作



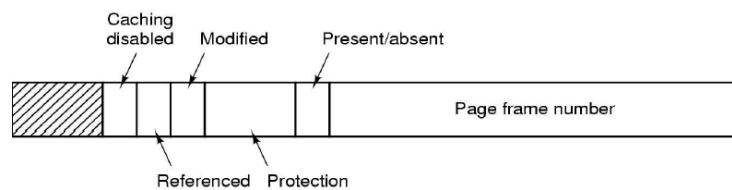
虚拟内存	在存储分配机制中，尽管备用内存是主内存的一部分，它也可以被寻址。程序引用内存使用的地址与内存系统用于识别物理存储站点的地址是不同的，程序生成的地址会自动转换成机器地址。虚拟存储的大小受到计算机系统寻址机制和可用的备用内存量的限制，而不受内存存储位置实际数量的限制
虚拟地址	在虚拟内存中分配给某一位置的地址使该位置可以被访问，仿佛它是主内存的一部分
虚拟地址空间	分配给进程的虚拟存储
地址空间	可用于某进程的内存地址范围
实地址	内存中存储位置的地址

缺页中断 (Page Fault) 处理

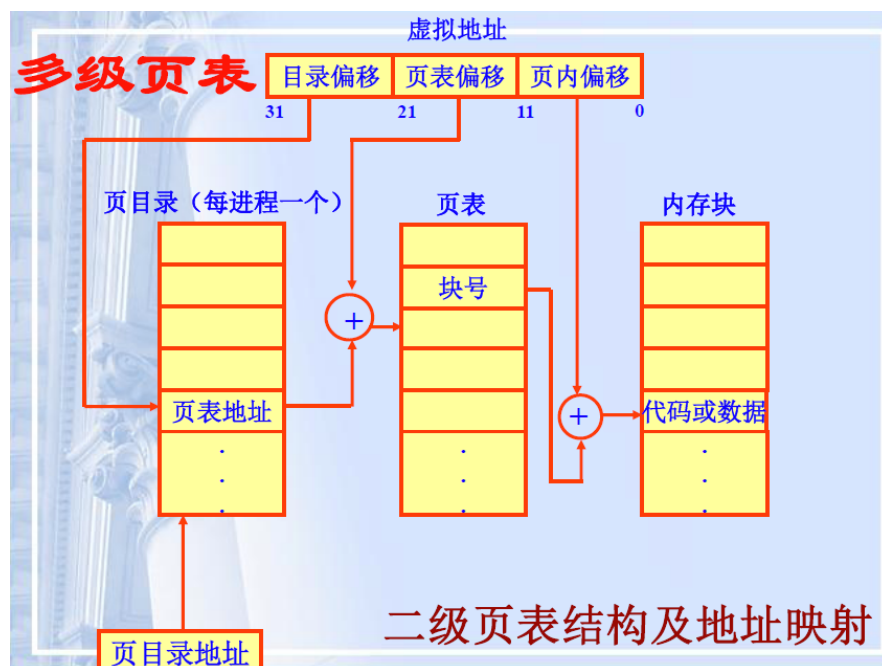
- 在地址映射过程中，在页表中发现所要访问的页不在内存，则产生缺页中断
- 操作系统执行缺页中断处理程序，根据页表获得外存地址，启动磁盘，将该页调入内存，使进程继续运行
- 如果内存中有空闲块，则分配一页，将新调入页装入内存，并修改页表中相应页表项的驻留位及相应的内存块号
- 若此时内存中没有空闲块，则要置换某一页，若该页在内存期间被修改过，则要将其写回外存

页表表项设计

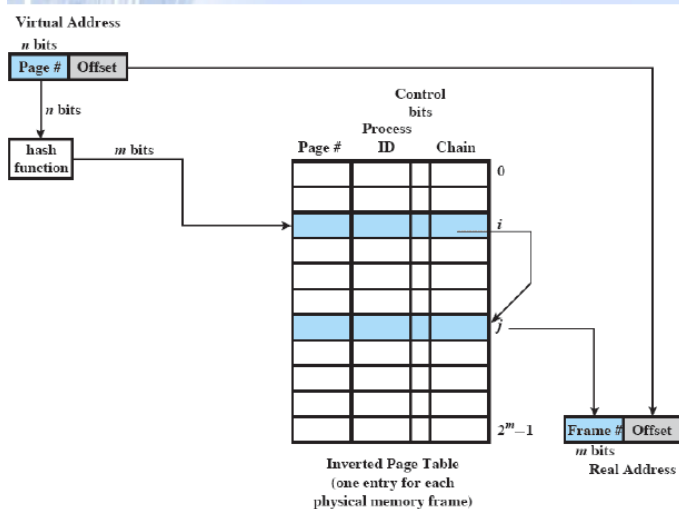
- 驻留位、内存块号、保护位、访问位、修改位等
- 驻留位（中断位）：表示该页是在内存还是在外存
- 访问位：根据访问位来决定淘汰哪页（由不同的算法决定）
- 修改位：查看此页是否在内存中被修改过
- 保护位：读/写/执行
- 禁止缓存位：采用内存映射I/O的机器中需要



一个进程的页表的各页在内存中不连续存放，需要引入地址索引——页目录



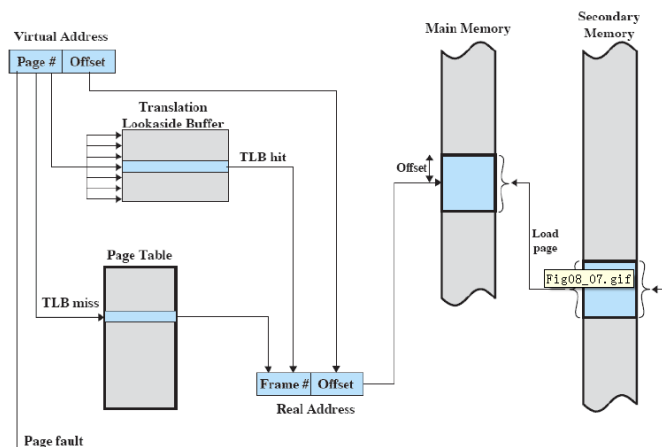
倒排页表



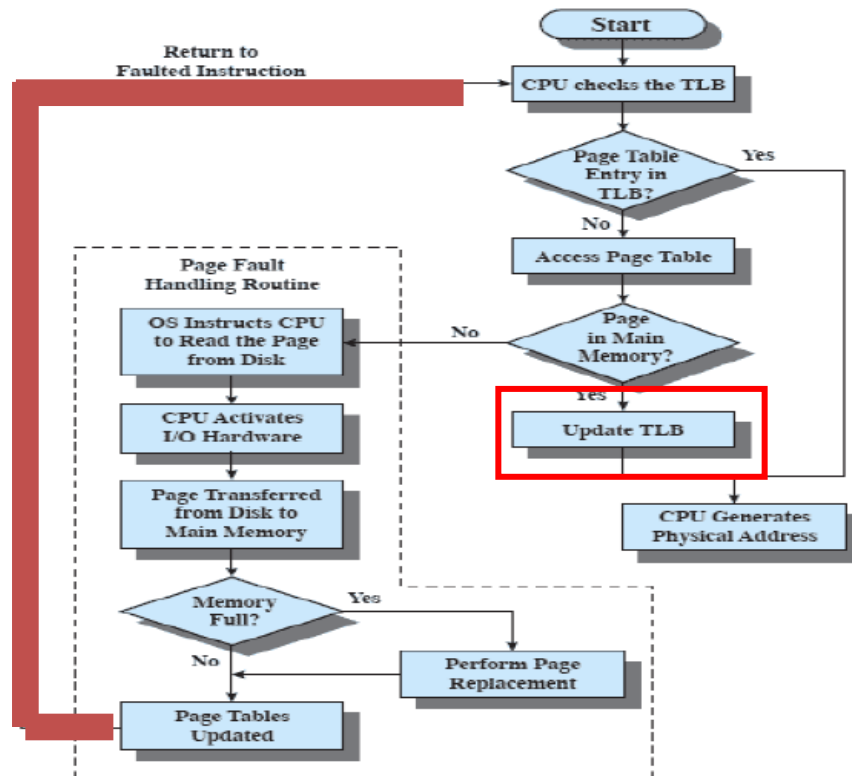
倒排表结构

- 页号
- 进程标识
- 控制位
- 链指针

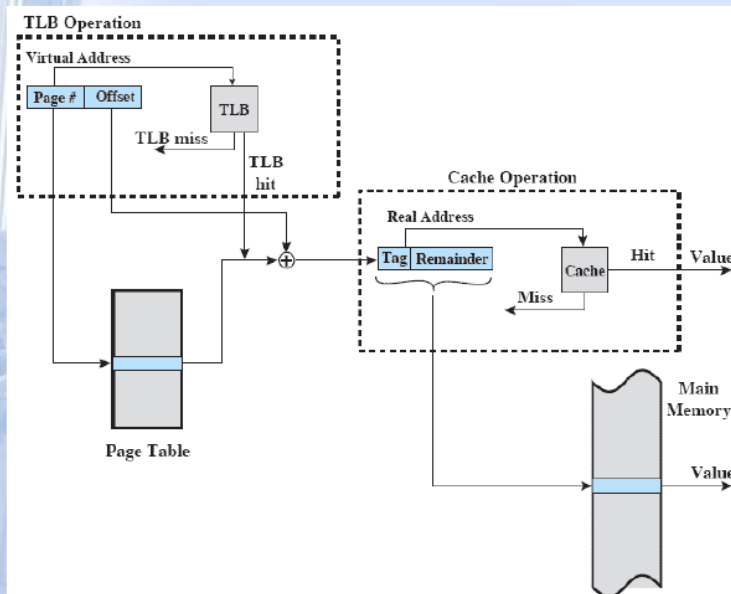
- 将虚拟地址的页号部分映射到一个散列值
 - 散列值指向一个倒排表
 - 倒排表大小与实际内存成固定比例，与进程个数无关
- TLB的使用



分页与TLB的操作



TLB与高速缓存操作



驻留集管理

- 驻留集大小
给每个进程分配多少页框？

✓ 固定分配策略

进程创建时确定

可以根据进程类型（交互、批处理、应用类）或者基于程序员或系统管理员的需要来确定

✓ 可变分配策略

根据缺页率评估进程局部性表现

缺页率高→增加页框

缺页率低→减少页框

系统开销

置换范围

- 局部置换策略

仅在产生本次缺页的进程的驻留集中选择

- 全局置换策略

将内存中所有未锁定的页框都作为置换的候选

可变分配的局部置换：

1. 当一个新进程被装入内存时，给它分配一定数目的页框，使用预先分页或请求分页填满这些页框
2. 当发生一次缺页中断时，从产生缺页中断的进程的驻留集中选择一页用于置换
3. 不断重新评估进程的页框分配情况，增加或减少分配给它的页框，以提高整体性能

页框锁定

- 内存锁定：不让操作系统将进程使用的页面换出内存，避免产生由交换过程带来的不确定的延迟

- 例如：操作系统核心代码、关键数据结构、I/O缓冲区

- 给每一页框加一个锁定位

页面置换算法

算法	评价
OPT	不可实现，但可作为基准
NRU	LRU的很粗略的近似
FIFO	可能淘汰重要的页面
Second Chance	比FIFO有很大的改善
Clock	现实的
LRU	很优秀，但很难实现
NFU	LRU的相对粗略的近似
Aging	非常近似LRU的有效算法
Working set	实现起来开销很大
WSClock	好的有效的算法
PFF	一种近似的工作集算法
VSWs	对PFF的改进，解决局部性过渡问题

FIFO页面置换算法会产生异常现象（Belady现象），即：当分配给进程的物理页面数增加时，缺页次数反而增加

时钟算法实现(R,M值，其中R值为使用位，M为修改位)

1. 从指针的当前位置开始，扫描页框缓冲区，选择遇到的第一个页框（r=0；m=0）用于置换（本扫描过程中，对使用位不做任何修改）

- 如果第1步失败，则重新扫描，选择第一个 ($r=0$; $m=1$) 的页框(本次扫描过程中，对每个跳过的页框，将其使用位设置成0)
- 如果第2步失败，指针将回到它的最初位置，并且集合中所有页框的使用位均为0。重复第1步，并且，如果有必要，重复第2步。这样将可以找到供置换的页框。

最近最少使用算法(LRU, Least Recently Used)

选择最后一次访问时间距离当前时间最长的一页并置换，即置换未使用时间最长的一页
实现：时间戳或维护一个访问页的栈→ 开销大

最不经常使用算法(NFU, Not Frequently Used)

选择访问次数最少的页面置换

老化算法(Aging)

改进(模拟LRU)：计数器在加R前先右移一位，R位加到计数器的最左端

(老化算法的最近使用的比重比以前大，而对于最不经常使用算法，最近使用的比重和以前使用的比重完全一样)

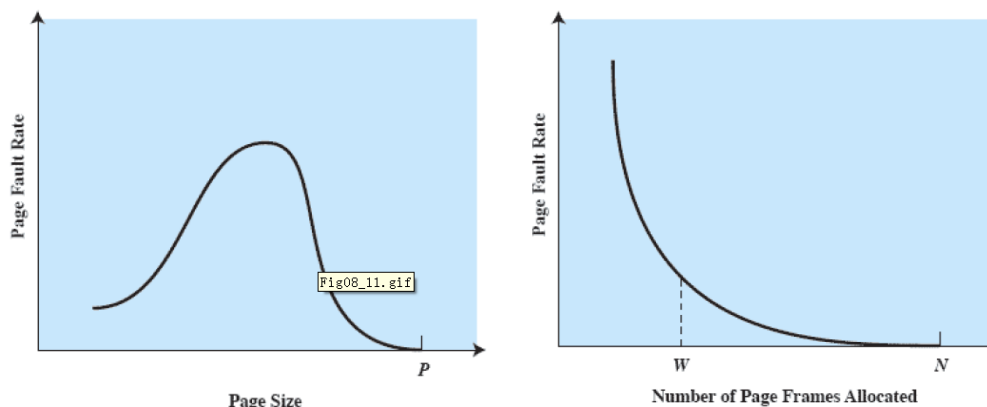
7. 影响缺页次数的因素

- 页面置换算法 (页面置换算法不同，换出的页面不同，缺页次数受影响)
- 页面本身的大小 (页面大小和缺页率的图)
- 程序的编制方法 (如循环变量的问题)
- 分配给进程的物理页面数 (页框数和缺页率的大小的图)

颠簸(抖动) 在虚存中，页面在内存与外存之间频繁调度，若调度页面所需时间比进程实际运行的时间还多，则系统效率急剧下降，这种现象称为颠簸或抖动

页面尺寸问题要考虑的因素：

- 内部碎片
- 页表长度
- 辅存的物理特性 (调度页面是有MMU实现，与物理特性有关)



工作集(Working Set)模型

如果能为进程提供与活跃页面数相等的物理页面数，则可减少缺页中断次数

工作集 $W(t, \Delta)$ = 该进程在过去的 Δ 个虚拟时间单位中访问到的页面的集合
内容取决于三个因素：

- ✓ 访页序列特性
- ✓ 时刻 t
- ✓ 工作集窗口长度 (Δ)

工作集概念与驻留集大小

- 监视每个进程的工作集
- 周期性地从一个进程的驻留集中移去那些不在它的工作集中的页。这可以使用一个LRU策

略

3. 只有当一个进程的工作集在内存中时，才可以执行该进程（也就是说，如果它的驻留集包括了它的工作集）

工作集算法实现：

扫描所有页表项，执行操作（这样选择出来的页面虽然不都是最近未被使用的页面，但是差不多啦~）

1. 如果一个页面的R位是1，则将该页面的最后一次访问时间设为当前时间，将R位清零

2. 如果一个页面的R位是0，则检查该页面的访问时间是否在“当前时间-T”之前

(1) 如果在，则该页面为被置换的页面；

(2) 如果不在，记录当前所有被扫描过页面的最后访问时间里面的最小值。扫描下一个页面并重复1、2

[工作集算法讨论]

1. 根据过去并不总能预测将来。工作集的大小和成员都会随着时间而变化

2. 为每个进程真实地测量工作集是不实际的，它需要为每个进程的每次页访问使用该进程的虚拟时间作时间标记，然后为每个进程维护一个基于时间顺序的页队列

3. Δ 的最优值是未知的，并且它在任何情况下都会变化

缺页中断率(PFF)算法 Page Fault Frequency

- 每一页框设置一个使用位

- 当页框被访问时，相应的使用位置为1

- 发生一次缺页中断时，操作系统记录该进程从上一次缺页中断到现在的虚拟时间，这可以通过维护一个页访问计数器来实现

- 定义一个阈值F，如果从上一次缺页中断到这一次的时间小于F，则这一页被加入到该进程的驻留集中；否则淘汰所有使用位为0的页，缩减驻留集大小；同时，把其余页的使用位重新置为0

- 改进：使用两个阈值：一个是用于引发驻留集大小增加的最高阈值，一个是用于引发驻留集大小缩小的最低阈值

清除策略

目标：分页系统工作的最佳状态：发生缺页中断时，系统中有大量的空闲页框，保存一定数目的页框供给比使用所有内存并在需要时搜索一个页框有更好的性能

实现：使用一个双指针时钟实现清除策略(A. S. Tanenbaum)

- 前指针由分页守护进程控制。当它指向一个“脏”页面时，就把该页面写回磁盘，前指针向前移动。当它指向一个干净页面时，仅仅指针向前移动

- 后指针用于页面置换，与标准时钟算法一样

- 由于分页守护进程的工作，后指针命中干净页面的概率会增加

MMU 将虚拟地址映射为物理地址

页缓冲

- 目的：提高性能

- 思路

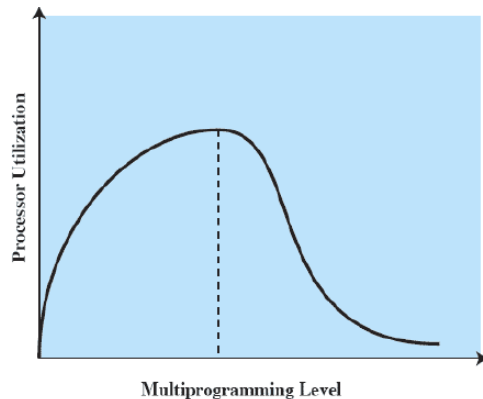
1. 不丢弃置换出的页，将它们放入两个表之一：如果未被修改，则放到空闲页链表中，如果修改了，则放到修改页链表中

2. 被修改的页以**簇方式**写回磁盘（不是一次只写一个，大大减少I/O操作的数量，从而减少了磁盘访问时间）

3. 被**置换的页仍然保留在内存**中，一旦进程又要访问该页，可以迅速将它加入该进程的驻留集合（代价很小）

加载控制

系统并发度：驻留在内存中的进程数目



内存映射文件

- 这种机制的思想是：进程通过一个系统调用将一个文件映射到其虚拟地址空间的一部分，访问这个文件就象访问内存中的一个大数据组，而不是对文件进行读写
- 在多数实现中，在映射共享的页面时不会实际读入页面的内容，而是在访问页面时，页面才会被每次一页的读入，磁盘文件则被当作后备存储
- 当进程退出或显式地解除文件映射时，所有被修改页面会写回文件

存储管理系统被分为三个部分：

- 底层MMU处理程序(与机器相关)
- 作为内核一部分的缺页中断处理程序(与机器无关)
- 运行在用户空间中的外部页面调度程序(策略)

Windows 内存管理：

内存管理器的组成部分

主要任务：

完成地址转换，页面在内外存之间的调度

提供一组核心服务：内存区对象、写时复制等

页目录

- 内存管理器创建的特殊页，用于映射进程所有页表的位置

内存分配方式

- 以页为单位的虚拟内存分配方式
 - 函数 (Virtualxxx)
- 内存映射文件
 - 函数 (CreateFileMapping, MapViewOfFile)
- 内存堆方法
 - (Heapxxx 和早期的接口Localxxx 和Globalxxx)

区域对象(section object)区域对象和其他对象一样由对象管理器创建和删除，对象管理器负责用对象头来管理区域对象，而虚拟内存管理器定义区域对象的实体并实现对象服务使用区域对象将一个文件映射到进程的地址空间，之后访问这个文件就象访问内存中的一个大数据组，而不是对文件进行读写

工作集——驻留在物理内存中的虚拟页面的子集

- ▣ 进程工作集：为每个进程分配的一定数量的页框
- ▣ 系统工作集：为可分页的系统代码和数据分配的页框

写时复制机制 (Linux)

- 写时复制 (Copy on Write) 是Linux目前常用的一种内存管理机制
 - 父进程和子进程共享页框而不是复制页框，但是共享的页框是只读的（设置写保护）
 - 父进程或子进程试图写一个共享页框时，就产生一个异常此时内核会把这个页复制到另一个页框并且标记为可写（原来的页框仍然是写保护的）
- 这种机制可以避免不必要的复制开销，而且通过页框共享节省物理内存

I/O管理和磁盘调度

设备分类：

按数据组织分

- 块设备
- 字符设备

按资源分配角度分

- 独占设备
- 共享设备
- 虚设备

在一类设备上模拟另一类设备，常用共享设备模拟独占设备，用高速设备模拟低速设备，被模拟的设备称为虚设备

目的：将慢速的独占设备改造成多个用户可共享的设备，提高设备的利用率

设备管理的目标和任务

(1)

- 设备分配与回收

记录设备的状态

根据用户的请求和设备的类型，采用一定的分配算法，选择一条数据通路

- 执行设备驱动程序，实现真正的I/O操作
- 设备中断处理：处理外部设备的中断
- 缓冲区管理：管理I/O缓冲区

(2) 建立方便、统一的独立于设备的接口[逻辑设备与物理设备、屏蔽硬件细节（设备的物理细节，错误处理，不同I/O的差异性）]

(3) 充分利用各种技术（通道，中断，缓冲，异步I/O等）提高CPU与设备、设备与设备之间的并行工作能力，充分利用资源，提高资源利用率

- 并行性
- 均衡性（使设备充分忙碌）

(4) 保护

设备传送或管理的数据应该是安全的、不被破坏的、保密的

设备组成

I/O设备一般由机械和电子两部分组成

通常，两部分分开处理，以提供更加模块化、更加通用的设计

(1) 物理设备机械部分是设备本身（物理装置） (2) 设备控制器(适配器)

电子部分完成的工作

- ② （端口）地址译码
- ② 按照主机与设备之间约定的格式和过程
- ② （接受计算机发来的数据和控制信号||向主机发送数据和状态信号）
- ② 将计算机的数字信号转换成机械部分能识别的模拟信号，或反之
- ② 实现设备内部硬件缓冲、数据加工等提高性能或增强功能

设备接口

操作系统将命令写入**控制器的接口寄存器**（或接口缓冲区）中，以实现输入 / 输出，并从接口寄存器读取状态信息或结果信息

控制器的任务是把串行的位流转换为字节块，并进行必要的错误修正：首先，控制器按位进行组装，然后存入控制器内部的缓冲区中形成以字节为单位的块。在对块验证检查和并证明无错误时，再将它复制到内存中

I/O端口地址

接口电路中：多个寄存器。每个寄存器有唯一的一个地址，该地址称为I/O端口地址，是一个整数。所有I/O端口地址形成I/O端口空间(受到保护)

I/O指令形式与I/O地址是相互关联的，主要有两种形式：

	编址方式	优点	缺点
内存映像编址 (内存映像I/O模式)	分配给系统中所有端口的地址空间是完全独立的，与内存的地址空间无关。使用专门的I/O指令对端口进行操作	外部设备不占用内存的地址空间。程序设计时，易于区分是对内存操作还是对I/O端口操作	对I/O端口操作的指令类型少，操作不灵活
I/O独立编址 (I/O专用指令)	分配给系统中所有端口的地址空间与内存的地址空间统一编址。把I/O端口看作一个存储单元，对I/O的读写操作等同于对存储器的操作	凡是可对存储器操作的指令都可对I/O端口操作，不需要专门的I/O指令，I/O端口可占有较大的地址空间	占用内存空间

内存映像编址的讨论：

- 对于内存映射I/O，不需要特殊的保护机制来阻止用户进程执行I/O操作。操作系统必须要做的事情：避免把包含控制寄存器的那部分地址空间放入任何用户的虚拟地址空间之中。对于内存映射I/O，可以引用内存的每一条指令也可以引用控制寄存器。例如，如果存在一条指令TEST可以测试一个内存字是否为0，那么它也可以用来测试一个控制寄存器是否为0LOOP
- 考虑以下汇编代码循环，第一次引用PORT_4将导致它被高速缓存，随后的引用将只从高速缓存中取值并且不会再查询设备，之后当设备最终变为就绪时，软件将没有办法发现这一点，结果循环将永远进行下去。为避免这一情形，硬件必须针对每个页面具备选择性禁用高速缓存的能力，操作系统必须管理选择性高速缓存，所以这一特性为硬件和操作系统两者增添了额外的复杂性。
- 如果只存在一个地址空间，那么所有的内存模块和所有的I/O设备都必须检查所有的内存引用，以便了解由谁做出响应。如果计算机具有单一总线，那么让每个内存模块和I/O设备查看每个地址是简单易行的。

I/O控制方式

(1) 可编程I/O（轮询）

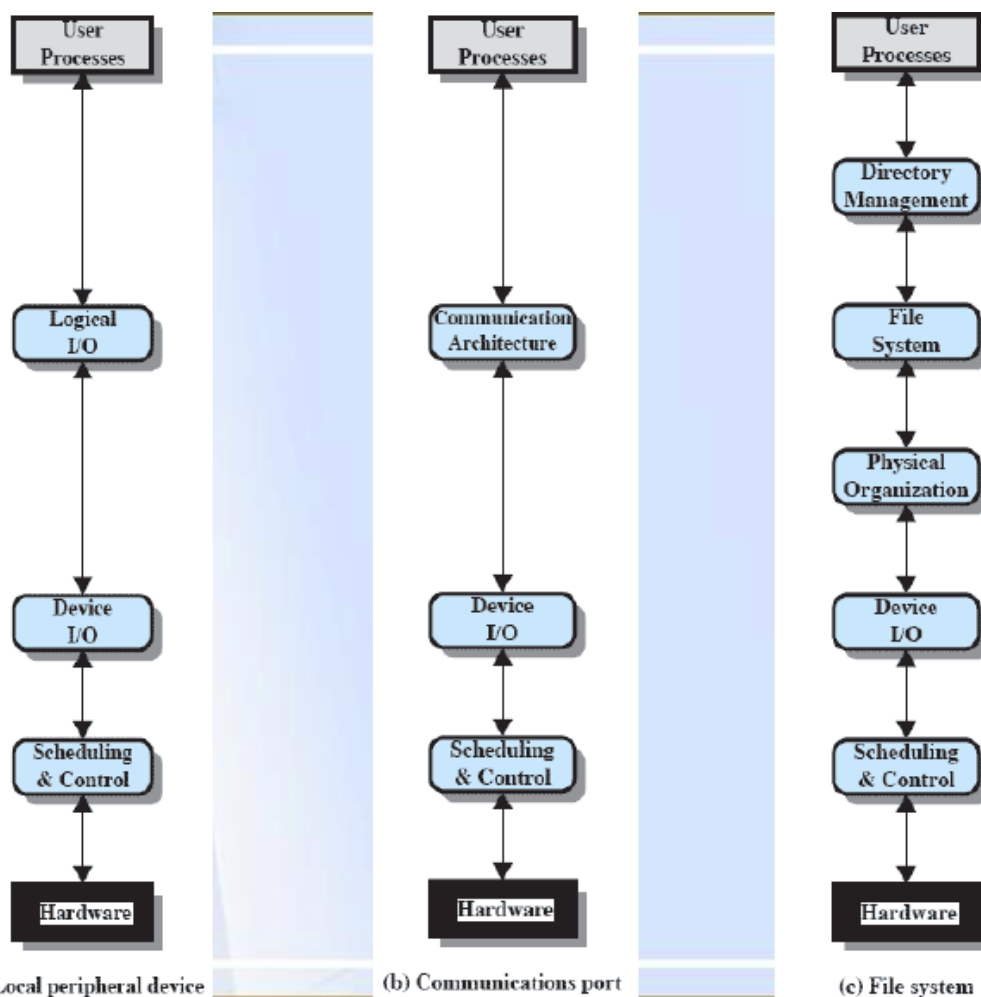
(2) 中断驱动I/O [中断处理方式的缺点是每传送一个字符都要进行中断，启动中断控制器，还要保留和恢复现场以便能继续原程序的执行，花费的工作量很大，这样如果需要大量数据交换，系统的性能会很低。]

(3) DMA

(4) 通道

I/O软件的组成

I / O软件设计的基本思想——分层



本地设备

- ▣ **逻辑I/O**: 把设备当作一个逻辑资源来处理，不关心实际控制设备的细节
逻辑I/O模块代表用户进程管理的一般的I/O功能，允许用户进程根据设备标识符以及诸如打开、关闭、读、写之类的简单命令与设备打交道
- ▣ **设备I/O**: 请求的操作和数据（缓冲的数据、记录等）被转换成适当的I/O指令序列、通道命令和控制器指令
可以使用缓冲技术，来提高利用率
- ▣ **调度和控制**: I/O操作的排队、调度
本层处理中断，收集并报告I/O状态，是与I/O模块和设备硬件真正发生交互的软件层

文件系统

- ▣ **目录管理**: 符号文件名被转换成标识符，用标识符可以通过文件描述符表或索引表直接或间接地访问文件
本层还处理文件目录的用户操作，如添加、删除、重新组织等
- ▣ **文件系统**: 处理文件的逻辑结构以及用户指定的操作，如打开、关闭、读、写等，还管理访问权限
- ▣ **物理组织**: 考虑到辅存设备的物理磁道和扇区结构，对于文件和记录的逻辑访问也必须转换成物理外存地址
辅助存储空间和内存缓冲区的分配通常也在这一层处理
- 🚩 **设备独立性（设备无关性）**
用户编写的程序可以访问任意I/O设备，无需事先指定设备

从用户角度：用户在编制程序时，使用逻辑设备名，由系统实现从逻辑设备到物理设备（实际设备）的转换，并实施I/O操作

从系统角度：设计并实现I/O软件时，除了直接与设备打交道的低层软件之外，其他部分的软件不依赖于硬件

好处：设备分配时的灵活性易于实现I/O重定向

1. Spooling（虚拟设备）技术：一个虚拟设备，一个资源转换技术，（用空间，如输入，输出等换取CPU时间）

解决问题：在进程所需物理设备不存在或被占用时使用该设备

注意：SPooling只提高设备利用率，缩短用户程序执行时间，并不提高CPU利用率

2. 通道技术

（1）定义：通道是独立于CPU的专门负责数据输入/输出传输工作的处理机，对外部设备实现统一管理，代替CPU对输入/输出操作进行控制，从而使输入，输出操作可与CPU并行操作。可以执行通道程序

（2）引入通道的目的

为了使CPU从I/O事务中解脱出来，同时为了提高CPU与设备，设备与设备之间的并行工作能力

a. 字节多路通道

字节多路通道以字节为单位传输信息，它可以分时地执行多个通道程序。当一个通道程序控制某台设备传送一个字节后，通道硬件就控制转去执行另一个通道程序，控制另一台设备传送信息

主要连接以字节为单位的低速I/O设备，如打印机，终端以字节为单位交叉传输，当一台传送一个字节后，立即转去为另一台传送字节

b. 选择通道

选择通道是以成组方式工作的，即每次传送一批数据，故传送速度很高。选择通道在一段时间内只能执行一个通道程序，只允许一台设备进行数据传输

当这台设备数据传输完成后，再选择与通道连接的另一台设备，执行相应的通道程序

主要连接磁盘，磁带等高速I/O设备

c. 成组多路通道

它结合了选择通道传送速度高和字节多路通道能进行分时并行操作的优点。它先为一台设备执行一条通道指令，然后自动转接，为另一台设备执行一条通道指令

主要连接高速设备

这样，对于连接多台磁盘机的数组多路通道，它可以启动它们同时执行移臂定位操作，然后，按序交叉地传输一批批数据。数据多路通道实际上是对通道程序采用多道程序设计的硬件实现

（3）通道连接：执行通道程序，向控制器发出命令，并具有向CPU发中断信号的功能。一旦CPU发出指令，启动通道，则通道独立于CPU工作。一个通道可连接多个控制器，一个控制器可连接多个设备，形成树形交叉连接

主要目的是启动外设时：提高了控制器效率；提高可靠性；提高并行度

（4）工作原理【与CPU公用内存，通过周期窃取的方式获得】

CAW CCW CDW CSW

I/O指令：启动通道程序

通道指令：对I/O进行操作

CPU：执行用户程序，当遇到I/O请求时，可根据该请求生成通道程序放入内存（也可事先编好放入内存），并将该通道程序的首地址放入CAW中；之后执行“启动I/O”指令，启动通道

工作

通道：接收到“启动I/O”指令后，从CAW中取出通道程序的首地址，并根据首地址取出第一条指令放入CCW中，同时向CPU发回答信号，使CPU可继续执行其他程序，而通道则开始执行通道程序，完成传输工作

（通道程序完成实际I/O，启动 I/O设备，执行完毕后，如果还有下一条指令，则继续执行， 否则表示传输完成）

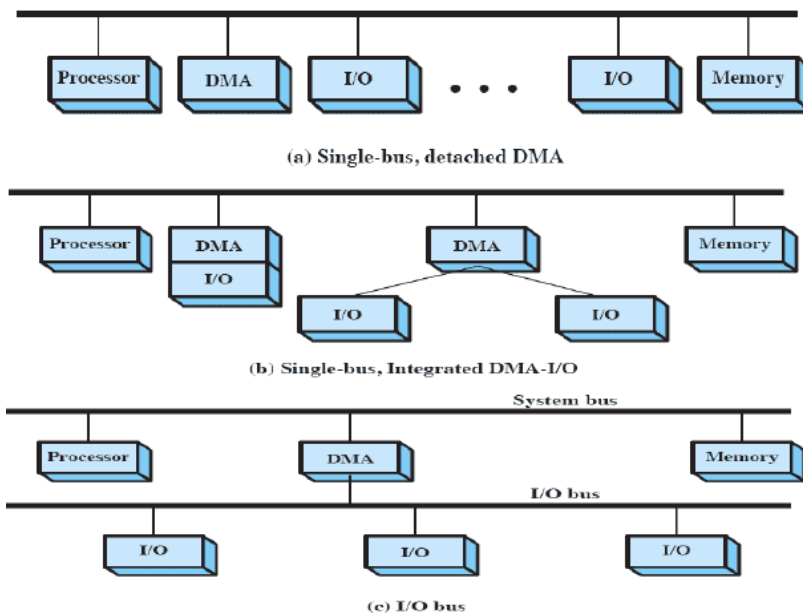
当通道传输完成最后一条指令时，向CPU发I/O中断，并且通道停止工作。CPU接收中断信号，从CSW中取得有关信息，决定下一步做什么

3. DMA技术

（1）工作流程：

1. 当CPU想读或写一块数据时，它向DMA模块发出命令
(读/写，设备地址，内存起始地址，读写字数)
2. DMA模块完成对内存/设备之间的数据传送，一次一个字，直至所有字数传送完成
3. DMA模块给CPU发中断信号

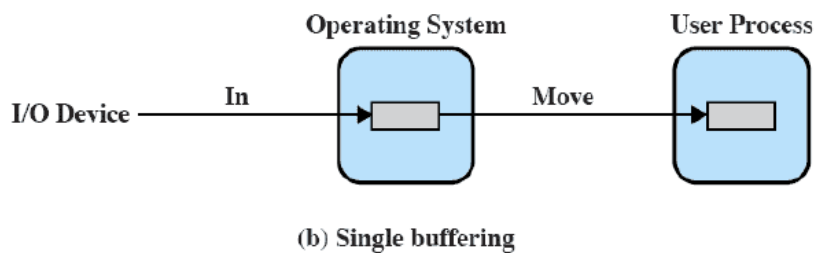
（2）DMA配置



缓冲技术

•单缓冲

当用户进程发出I/O请求时，操作系统在内存的系统空间为该操作分配一个缓冲区，可以实现提前读和延迟写

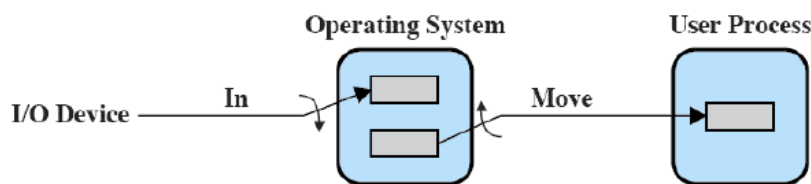


•双缓冲

一个进程往一个缓冲区中传送数据（从这个缓冲区中取数据）的同时，操作系统正在清

空（或者填充）另一个缓冲区

可以实现用户数据区—缓冲区之间交换数据和缓冲区—外设之间交换数据的并行



(c) Double buffering

• 循环缓冲:

又称缓冲池

多个缓冲区连接起来统一管理

引入系统缓冲池，采用有界缓冲区的生产者/消费者模型对缓冲池中的缓冲区进行循环使用



(d) Circular buffering

目的:

- 缓解CPU与I/O设备之间速度不匹配的矛盾
- 提高CPU与I/O设备之间的并行性
- 减少了I/O设备对CPU的中断请求次数，放宽CPU对中断响应时间的要求 (?)
- 平滑I/O需求峰值

🔧 磁盘调度【公平、高效】

一次访盘时间 = 寻道时间 + 旋转时间 + 存取时间

Table 11.3 Disk Scheduling Algorithms

Name	Description	Remarks
Selection according to requestor		
RSS	Random scheduling	For analysis and simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management
LIFO	Last in first out	Maximize locality and resource utilization
Selection according to requested item		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One way with fast return	Lower service variability
N-step-SCAN	SCAN of N records at a time	Service guarantee
FSCAN	N-step-SCAN with N = queue size at beginning of SCAN cycle	Load sensitive

磁盘调度算法，扇区和磁道的优先级的的问题，对于扇区不同，那么优先对都挨打磁头位置下的扇区先进行传送操作，对于扇区号相同，磁道不同，那么任意选择一个磁道。

🔧 RAID（独立磁盘冗余阵列）

具体实现:

1. 把多个磁盘组织在一起，作为一个逻辑卷，提供磁盘跨越功能。
2. 通过把数据分成多个数据块，**并行**写入/读出多个磁盘，以提高数据传输率（数据分条stripe）
3. 通过镜像或校验操作，提供容错能力（冗余）

RAID 级	说明	磁盘请求	数据可用性	大 I/O 数据量传输能力	小 I/O 请求率
RAID 0	条带化	N	低于单个磁盘	很高	读和写都很高
RAID 1	镜像	2N	高于 RAID2, 3, 4 或 5; 低于 RAID6	读时高于单个磁盘; 写时与单个磁盘相近	读时最快可以为单个磁盘的两倍; 写时与单个磁盘相近
RAID 2	并行访问—海明码（并行访问）	N+m	明显高于单个磁盘; 与 RAID3, 4 或 5 可比	所有列出方案中最高的	大概是单个磁盘的两倍
RAID 3	交错位奇偶校验（并行访问）	N+1	明显高于单个磁盘; 与 RAID2, 4 或 5 可比较	所有列出方案中最高的	大概是单个磁盘的两倍
RAID 4	交错块奇偶校验（独立访问）	N+1	明显高于单个磁盘; 与 RAID2, 3 或 5 可比较	读时与 RAID0 相近; 写时明显慢于单个磁盘	读时与 RAID0 相似; 写时显著慢于单个磁盘
RAID 5	交错块分布奇偶校验（独立访问）	N+1	明显高于单个磁盘; 与 RAID2, 3 或 4 可比较	读时与 RAID0 相近; 写时慢于单个磁盘	读时与 RAID0 相似; 写时通常慢于单个磁盘
RAID 6	交错块双重分布奇偶校验（独立访问）	N+2	所有列出方案中最高的	读时与 RAID0 相近; 写时慢于 RAID5	读时与 RAID0 相近; 写时显著慢于 RAID5

🌈 磁盘高速缓存

内存中为磁盘扇区设置的一个缓冲区，它包含有磁盘中某些扇区的副本

☐ 当出现一个请求某一特定扇区的I/O请求时，首先进行检测，以确定该扇区是否在磁盘高速缓存中。

🌈 设备管理

1. 设备的分配与回收(1/3)

当某进程向系统提出I/O请求时，设备分配程序按一定策略分配设备、控制器和通道，形成一条数据传输通路，以供主机和设备间信息交换

（1）设备管理有关的数据结构

- 描述设备、控制器等部件的表格：设备表或部件控制块。这类表格具体描述设备的类型、标识符、状态，以及当前使用者的进程标识符等
- 建立同类资源的队列：相同物理属性的设备连成队列（称设备队列）
- I/O请求包
- 建立I/O队列

（2）根据用户请求的I/O设备的逻辑名，查找逻辑设备和物理设备的映射表；以物理设备为索引，查找SDT→SDT表项，找到该设备所连接的DCT→DCB；继续查找与该设备连接的COCT→COCB 和CHCT→CHCB，就找到了一条通路

设备的分配策略

独占设备：静态分配策略（效率低），动态分配策略（效率高，容易死锁）

分时共享设备分配策略

SPooling技术

驱动程序:

- (1) 接受来自上层、与设备无关软件的抽象I/O请求，将该请求排在请求队列末尾，检查I/O请求的合法性
- (2) 执行特定的缓冲区策略：根据请求传输的数据量，组织I/O缓冲队列，利用I/O缓冲对数据进行加工，包括数据格式处理和编码转换
- (3) 对各种可能的有关设备排队、挂起、唤醒等操作进行处理
- (4) 向有关的输入输出设备的各种控制器发出控制命令，并且监督它们的正确执行，进行必要的错误处理
- (5) 处理来自设备的中断

I/O进程：专门处理系统中的I/O请求和I/O中断工作

(1) I/O请求的进入

用户程序：调用send将I/O请求发送给I/O进程；调用block将自己阻塞，直到I/O任务完成后被唤醒

系统：利用wakeup唤醒I/O进程，完成用户所要求的I/O处理

(2) I/O中断的进入

当I/O中断发生时，内核中的中断处理程序发一条消息给I/O进程，由I/O进程负责判断并处理中断

(3) I/O进程

是系统进程，一般赋予最高优先级。一旦被唤醒，它可以很快抢占处理机投入运行

I/O进程开始运行后，首先关闭中断，然后用receive去接收消息。两种情形：

- 没有消息，则开中断，将自己阻塞；
- 有消息，则判断消息（I/O请求或I/O中断）；

a. I/O请求

准备通道程序，发出启动I/O指令，继续判断有无消息

b. I/O中断，进一步判断正常或异常结束

正常：唤醒要求进行I/O操作的进程

异常：转入相应的错误处理程序

Linux的磁盘调度算法：

电梯算法

如果新请求和现有的请求在同一个扇区或者相邻的话，合并为一个扇区，如果现有的请求已经等待很长时间啦，那么把新请求放在队尾

最后期限调度程序：

维护三个队列，排序的电梯队列，FIFO读队列，FIFO写队列

每个请求都有一个到期时间

预期I/O队列，两次读请求之间的延迟，看是否发生相邻磁盘上的读请求，提高系统的性能（局部性原理）

Windows的异步I/O技术

数据处理和I/O操作不存在严格的时序要求，只有逻辑关系。数据就绪就可以对其进行操作。

数据没有就绪可以对其他数据进行操作

异步模式的实现

异步传输模式的实现

• 系统实现

- 通过切换到其他线程保证CPU利用率
- 对少量数据的I/O操作会引入切换的开销
- 用户实现
 - 将访问控制分成两段进行
 - 发出读取指令后继续做其他操作
 - 当需要用读入的数据的时候，再使用wait命令等待其完成
 - 不引入线程切换，减少开销