操作系统原理笔记

陈鸿峥

2019.07*

目录

*Build 20190709

本课程使用的教材为William Stallings《操作系统—精髓与设计原理（第八版）》。其他参考资料包括Stanford CS140、CMU15-460、*Operating System Concepts (10th ed.)*。

关于计算机系统的内容在此不再赘述，详情参见**计算机组成原理**的笔记。

1 操作系统概述

1.1 概述

操作系统核心即怎么虚拟多几个冯诺依曼计算机出来给程序用。操作系统是控制应用程序执行的程序，是应用程序和计算机硬件间的接口（屏蔽硬件细节）。

这里先解释几个概念

- 并发：两件事情**可以**同时(simultaneously)发生，没有时间限制，$t_1 > t_2$，$t_1 < t_2$，$t_1 = t_2$都可
- 同步：两个事件有确定的时间限制
- 异步：两件事不知道何时发生

并发和共享是操作系统两个最基本的特征

1.2 发展历史

- 串行处理/手工操作(1940s)：没有OS，人工调度，准备时间长
- 简单批处理系统(1950s)：
  - 使用监控程序(monitor)，读入用户程序执行
  - 提供内存保护、计时器、特权指令、中断
  - 两种操作模式：用户态、内核态(mode)
  - 单道程序(uniprogramming)批处理：处理器必须等到IO指令结束后才能继续
- 多道程序批处理(1950s末)：多个作业同时进入主存，切换运行，**充分利用处理器**（大块处理时间，少交换上下文的调度）；用户响应时间长，不提供人机交互能力
- 分时系统(1961)：MIT CTSS(Compatible Time-Sharing System)，满足用户与计算机交互的需要，**减小响应时间**（分时间片小块调度）；多个交互作业，多个用户，把运行时间分成很短的时间片轮流分配

- 实时系统：专用，工业、金融、军事

现代的操作系统通常同时具有分时、实时核多道批处理的功能，因此被称为通用操作系统。而OS也不仅是在PC机上有，网络OS、分布式OS、嵌入式OS层出不穷。

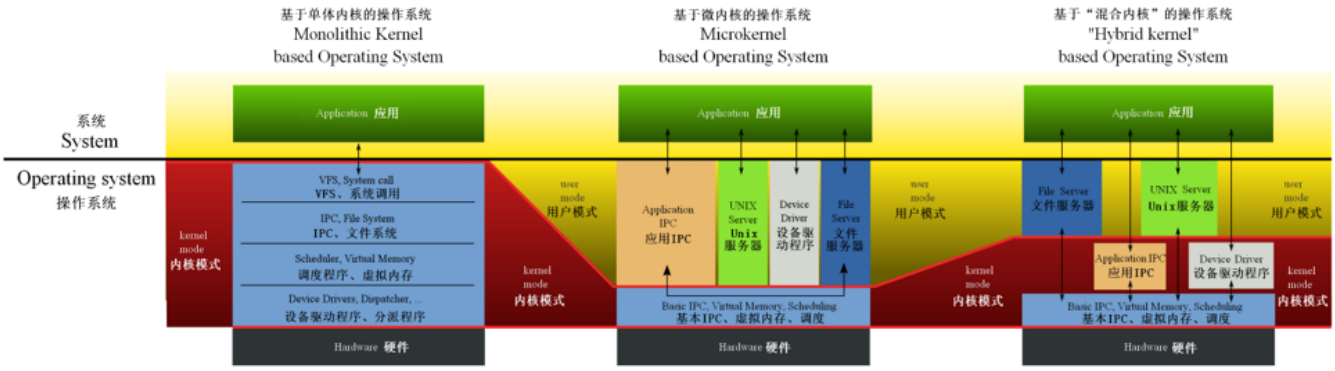
影响现代OS发展的因素：

- 新硬件：多核/处理器结构、高速增长的计算机速度、高速网络连接、容量的不断增加各种存储设备
- 新应用：多媒体应用、互联网/Web访问、客户/服务器计算模式
- 新安全威胁：网络使安全问题更突出（病毒、蠕虫、黑客技术）

内核的分类：

- 单体内核(monolithic kernel)/宏内核(macro kernel)
  - 内核实现操作系统所有基本功能（包括调度、文件系统、连网、设备驱动程序、内存管理等等）
  - 一般用一个进程实现，内核代码共享同一个地址空间，每一模块可以调用任意其它模块和使用内核所有核心数据
  - 效率高，但难于修改和扩充
  - 例子：Unix、Linux、Android（基于Linux）、DOS、Windows 9x、Mac OS 8.6以下
- 微内核(microkernel)
  - 内核只实现最基本功能（包括地址空间、IPC[InterProcess Communication，进程间通信]和基本调度）
  - 更多的功能代码组织为多个进程，各自独立使用自己的地址空间，运行于用户态
  - 一致接口（消息传递[1]）、可扩展性（允许增加新服务）、可移植性（将系统移植到新处理器只需对内核修改），适用于嵌入式与分布式环境，但效率稍低
  - 例子：Mach、QNX
- 混合内核
  - 微内核与宏内核的结合（也可算作单体内核中的一类）
  - 具有微内核结构，按宏内核实现
  - 例子：Windows NT(2000/XP/Vista/7/8)、BSD、XNU（Darwin 的核心，源自Mach和BSD，用于Mac OS X 和iOS）

[1]即使是硬件中断也会被当作消息处理，需要send/receive



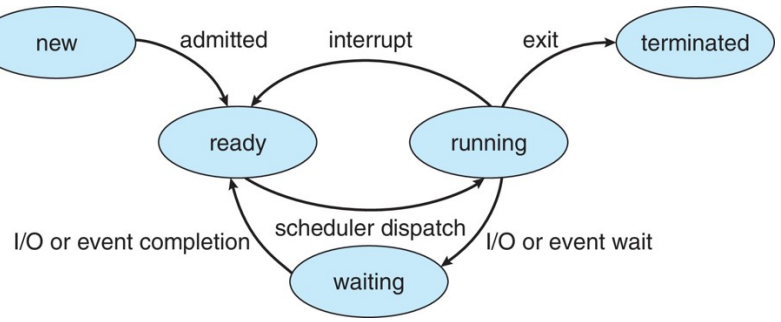
2 进程

进程(process)是**运行时(running/in execution)**程序的实例。

- 程序并发执行的特征（多道程序）：间断性、无封闭性、不可再现性（破坏冯顺序执行特性）
- 进程的特点：动态性、并发性、独立性、异步性
- 进程的作用
  - 提升CPU利用率：将多个进程重叠（一个进程IO时另一个计算）
  - 降低延迟(latency)：并发执行，不断切换，防止卡住

进程控制块(Process Control Block, PCB)，在Unix是proc，在Linux是task_struct

- 进程标识符/ID
- 五状态：运行、就绪、等待/阻塞、创建、结束



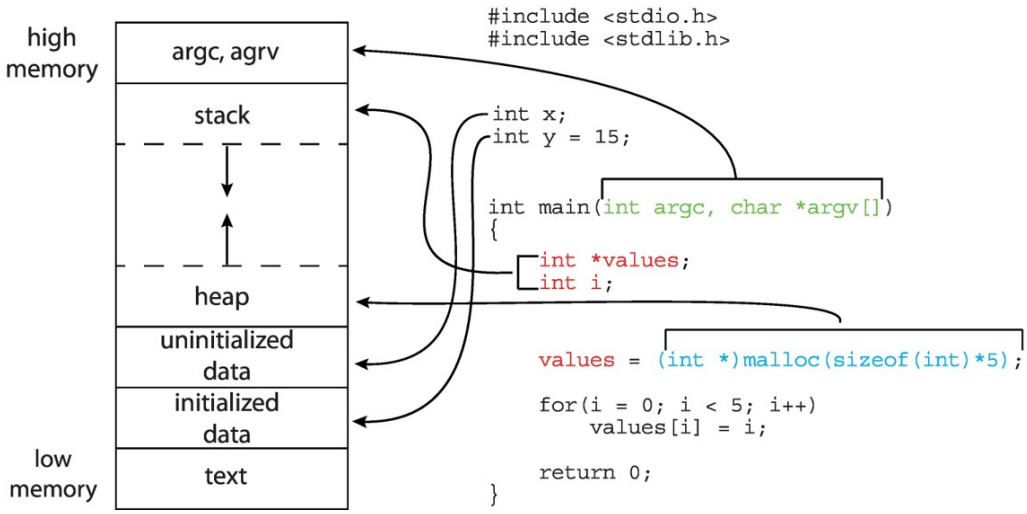
- 优先级
- 程序计数器(PC)
- 内存指针：报错指向程序代码、相关数据和共享内存的指针
- 上下文数据(context)：进程被中断时寄存器中的数据
- IO状态信息
- 记账信息(accounting)：占用处理器时间、时钟数总和、时间限制等
- 链表：各状态的进程形成不同的链表：就绪链表、阻塞链表等

进程间的通信

- 共享存储：进程到共享空间再到进程
- 消息传递：直接进程到进程
- 管道通信：进程到缓冲区到进程，管道即共享文件，半双工通信

进程控制由原语(primitive)完成，由若干条指令完成，也可被视为原子操作

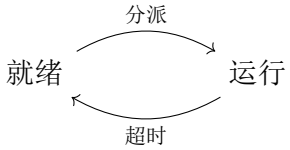
内存组织：代码段、数据段、堆段、栈段（从小地址往大地址）



可以参考原始的UNIX论文[2]。

- 创建进程：fork、waitpid
- 删除进程：exit、kill
- 执行进程：execve

处理器上下文即处理器寄存器的内容



导致OS获得控制权的事件：

- 时钟中断：时间片结束
- IO中断：IO完成
- 硬件中断/陷阱(trap)/异常
- 系统调用：`int`

[2] http://www.scs.stanford.edu/19wi-cs140/sched/readings/unix.pdf

3 线程

在没有线程概念的系统中，进程是**资源分配**、调度/执行的单位；而在有线程概念的系统中，线程就成了**基本调度单位**/程序执行流最小单元，由线程ID、程序计数器、寄存器集合和堆栈组成。

线程的优点：

- 创建速度快
- 终止所用时间少
- 切换时间少
- 通信效率高，同一进程无需调用内核，共享存储空间

用户级线程(ULT)：线程管理都由应用程序完成（线程库），内核不知道线程的存在，优点：

- 线程切换不需要模式切换
- 调度算法可以应用程序专用
- ULT不需要内核支持，线程库可以在任何OS上运行

缺点：

- 一个线程阻塞会导致整个进程阻塞（因用户级线程对操作系统不可见，操作系统对整个进程进行调度）
- 不能利用多核和多处理器技术

内核级线程(KLT)：线程管理由内核完成（提供API），调度基于线程进行，优点：

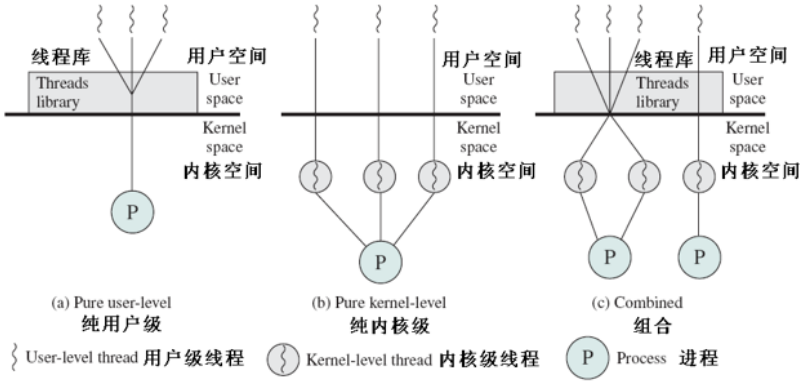
- 线程阻塞不会导致进程阻塞
- 可以利用多核和多处理器技术
- 内核例程本身也可以使用多线程

缺点：

- 线程切换需要进行模式切换

多线程模型

- 多对一：多个用户级线程映射到一个内核级线程，线程管理在用户空间进行，效率高；若内核服务阻塞，则整个进程被阻塞
- 一对一：每个用户级线程映射到一个内核级线程，开销大
- 多对多：折中



User-Level and Kernel-Level Threads
用户级与内核级线程

线程与进程之间的关系

- 1：1，每个进程都有唯一线程，DOS、传统Unix
- M：1，一个进程多个线程，Windows NT、Linux、Mac OS、iOS
- 1：M，一个线程可在多个进程环境中迁移

* Linux并不区分线程和进程，采用Copy On Write (COW)方式

4 并发—互斥与同步

4.1 基本概念

- 原子(atomic)操作：不可分割
- 临界区(critical section)：不允许多个进程同时进入的一段访问**共享资源**的代码
- 死锁(deadlock)：两个及以上进程，因每个进程都在等待其他进程做完某事（如释放资源），而不能继续执行
- 活锁(livelock)：两个及以上进程，为响应其他进程中的变化，而不断改变自己的状态，但是没有做任何有用的工作
- 互斥(mutual exclusion)：当一个进程在**临界区访问共享资源**时，不允许其他进程进入访问
- 竞争条件(race condition, RC)：多个进程/线程读写共享数据，其结果依赖于它们执行的相对速度
- 饥饿(starvation)：可运行的进程长期未被调度执行

核心内容

$$\text{并发} \rightarrow \text{共享} \rightarrow \text{RC问题} \rightarrow \text{互斥}$$

共享数据的最终结果取决于进程执行的相对速度（异步性），需要保证进程的结果与相对执行速度无关

同步：有明确的时间先后限制，两个或多个进程之间的操作存在时间上的约束（也包含互斥）

4.2 互斥

互斥的要求

- 在具有相同资源或共享对象的临界区的所有进程中，一次只允许一个进程进入临界区（强制排它）
- 一个在非临界区停止的进程必须不干涉其他进程（充分并发）
- 没有进程在临界区中时，任何需要访问临界区的进程必须能够立即进入（空闲让进）
- 决不允许出现一个需要访问临界区的进程被无限延迟（有限等待）
- 相关进程的执行速度和处理机数目没有任何要求或限制（满足异步）
- 当进程不能进入临界区，应该立即释放处理机，防止进程忙等待（让权等待）

4.2.1 简单的尝试

第一种尝试（单标志法）：两个进程轮流进入临界区

```
while (turn != 0)
    /* do nothing */;
/* critical section */
turn = 1;
```

```
while (turn != 1)
    /* do nothing */;
/* critical section */
turn = 0;
```

可以保证互斥，硬性规定进入的顺序，但是

- 忙等待，白白消耗CPU时间
- 必须轮流进入临界区，不合理，限制推进速度
- 若一个进程失败，则另一个将永远被阻塞

难以支持并发处理

一共四种尝试

- 单标志法
- 双标志先检查：死锁
- 双标志后检查：不能保证互斥
- 双标志延迟礼让：活锁

例 1. 忙等待效率一定比阻塞等待效率低吗

分析. 一般情况下确实如此，因为忙等待一直在消耗*CPU*资源。但特殊情况下，忙等待能立即响应请求完成（条件判断结束），对于性能要求较高的应用有好处；而阻塞等待还需等*OS*调度才能继续执行下面的工作。

4.2.2 软件方法

Dekker算法：避免无原则礼让，规定各进程进入临界区的顺序；逻辑复杂，正确性难以证明，存在轮流问题，存在忙等待；初始化`flag`都为`false`，`turn`为1

```
void P0()
{
    while(true)
    {
        flag[0] = true; // P0想使用关键区
        while(flag[1]) // 检查P1是不是也想用？
        {
            if(turn == 1) // 如果P1想用，则查看P1是否具有访问权限？
            {
                flag[0] = false; // 如果有，则P0放弃
                while(turn == 1); // 检查turn是否属于P1
                flag[0] = true; // P0想使用
            }
        }
        visit(0); // 访问Critical Partition
        turn = 1; // 访问完成，将权限给P1
        flag[0] = false; // P0结束使用
    }
}

void P1()
{
    while(true)
    {
        flag[1] = true; // P1想使用关键区
        while(flag[0]) // 检查P0是不是也想用？
        {
            if(turn == 0) // 如果P0想用，则查看P0是否具有访问权限？
            {
                flag[1] = false; // 如果有，则P1放弃
                while(turn == 0); // 检查turn是否属于P1
                flag[1] = true; //  P1想使用
            }

        }
          visit(1); // 访问Critical Partition
        turn = 0; // 访问完成，将权限给P0
        flag[1] = false; // P1结束使用
    }
}
```

Peterson算法：flag和turn的含义同Dekker的，但先设turn=别人，且只有flag[别人]和turn=别人同时为真时才循环等待

```
void P0()
{
        while(true)
        {
                flag[0] = true;
                turn = 1;
                while(flag[1] && turn == 1)
                // 退出while循环的条件就是，要么另一个线程
                // 不想要使用关键区，要么此线程拥有访问权限
                {
                        sleep(1);
                        printf("procedure0 is waiting!\n");
                }
                //critical section
                flag[0] = false;
        }
}

void P1()
{
        while(true)
        {
                flag[1] = true;
                turn = 0;
                while(flag[0] && turn == 0)
                {
                        sleep(1);
                        printf("procedure1 is waiting!\n");
                }
                //critical section
                flag[1] = false;
        }
}
```

4.2.3 硬件方法

- 关中断：限制处理器交替执行各进程的能力，不能用于多核
- 专用指令：**比较并交换**，原子指令，一个指令周期内完成，不会被中断
  - TestSet(TS)指令，比较并交换的bool形式

```
int compare_and_swap (int *word, int testval, int newval)
bool testset (int i)
```

– Exchange/swap指令(x86xchg指令)：同上，适用于单核多核，多变量多临界区，但需要忙等待(busy waiting)/自旋等待(spin waiting)，可能饥饿或死锁

```
void exchange (int register, int memory)
```

机器指令方法优点

- 适用于单处理器或共享主存多[核]处理器系统，进程数目任意
- 简单且易于证明
- 可以使用多个变量支持多个临界区

缺点

- 忙等待/自旋等待
- 可能饥饿或死锁

例 2. 利用xchg实现一套互斥机制并给出使用该机制的框架

分析. 由于xchg可以交换两个变量的内容，且为原子操作，故如果临界区未被占用，经过xchg操作后，lock_var被置为*1*；而其他进程再要访问临界区时，lock_var和ax均为*1*，交换后不会发生改变，进而不断进行lock_loop循环。

```
    lock_var db 0 ; not used critical section
lock:
    mov ax, 1
lock_loop:
    xchg [lock_var], ax
    cmp ax, 0
    jnz lock_loop
```

解锁操作则只需将lock_var置*0*即可

```
unlock:
    mov ax, 0
    xchg [lock_var], ax
```

4.3 信号量(semaphore)

解决RC问题一种简单高效的方法

4.3.1 基本操作

记录信号量

- 整数：可用资源数(≥ 0)，需要初始化
- P操作(proberen,semWait)：信号量的值**减1**（申请一个单位的资源），若信号量变为**负数**，则执行P操作进程阻塞，**让权等待**

- V操作(verhogen,semSignal)：信号量的值**加1**（释放一个单位的资源），若信号量**不是正数**（绝对值=现被阻塞的进程数/等待队列的长度），则使一个因P操作被阻塞的进程解除阻塞（唤醒）

需要保证P操作和V操作的**原子性!**

```
struct semaphore {
    int count;
    struct process* L; // 阻塞队列
} s;
void P(semaphore s) { // semWait
    s.count--;
    if (s.count < 0)
        Block(CurruntProcess, s.L);
        // 将当前进程插入该信号量对应的阻塞队列
}
void V(semaphore s) { // semSignal
    s.count++;
    if (s.count <= 0)
        WakeUp(s.L);
}
```

注意P操作是**小于0**，V操作**小于等于0**[3]，且一个进程只会在一个信号量的阻塞队列中。

信号量的优点是简单且表达能力强，用P、V操作可解决多种类型的同步/互斥问题，但不够安全，P、V操作使用不当会产生死锁。

二元信号量省空间，不能代表资源数量，要引入全局变量代表数量。

4.3.2 实现互斥(mutex)

- 对于每一个RC问题，设一个信号量（向系统调用/向内核调用），**初始化为**1
- 所有相关进程在进入临界区**之前**对该信号量进行**P操作**
- 出临界区**之后**进行**V操作**

4.3.3 同步

同步：后续动作必须在前驱动作执行完后才能进行

- 对每一个同步关系都要设一个信号量，初值看具体问题（一般为0）
- 在**前驱动作之后**执行V操作（相当于资源产生了）
- 在**后续动作之前**执行P操作

4.3.4 生产者-消费者问题

[3]理解为先判断是否小于0（阻塞队列非空），然后再++会比较好

```
void producer() {
    while (true) {
        produce();
        P(e);

        P(s);
        append();
        V(s);

        V(n); // first
    }
}

void consumer() {
    while (true) {
        P(n); // after

        P(s);
        take();
        V(s);

        V(e);
        consume();
    }
}

// s = initSem(1); // mutex
// n = initSem(0); // # products
// e = initSem(12); // # empty entries in buffer
```

4.3.5 读者写者问题

可以有多个读者，但只有一个写者

读者优先

```
int readcount;
semaphore x=1, wsem=1;
void reader() {
    while(true) {
        P(x);
        readcount++;
        if (readcount==1) P(wsem);
        V(x);
```

```
        READUNIT();
        P(x);
        readcount--;
        if (readcount==0) V(wsem);
        V(x);
    }
}

void writer() {
   while(true) {
       P(wsem);
       WRITEUNIT();
       V(wsem);
   }
}
```

写者优先

```
int readcount, writecount;
semaphore x=1, y=1, z=1, rsem=1, wsem=1;
void reader() {
    while(true) {
        P(z); P(rsem);
        P(x);
        readcount++;
        if (readcount==1) P(wsem);
        V(x);
        V(rsem); V(z);
        READUNIT();
        P(x);
        readcount--;
        if (readcount==0) V(wsem);
        V(x);
    }
}

void writer() {
    while(true) {
        P(y);
        writecount++;
        if (writecount==1) P(rsem);
        V(y);
        P(wsem);
        WRITEUNIT();
        V(wsem);
        P(y);
        writecount--;
        if (writecount==0) V(rsem);
        V(y);
    }
}
```

4.4 管程(monitor)

管程(monitor)：通过集中管理（封装同步机制与同步策略）以保证安全（类似于OOP中的抽象类）

主要特点：

- 本地变量只能由管程过程访问（封装）
- 进程通过调用管程过程进入管程（调用）
- 每次只能一个进程在执行相关管程的过程（互斥）

主要缺陷

- 可能增加了两次多余的进程切换
- 对进程调度有特殊要求（不允许插队）

4.5 消息传递

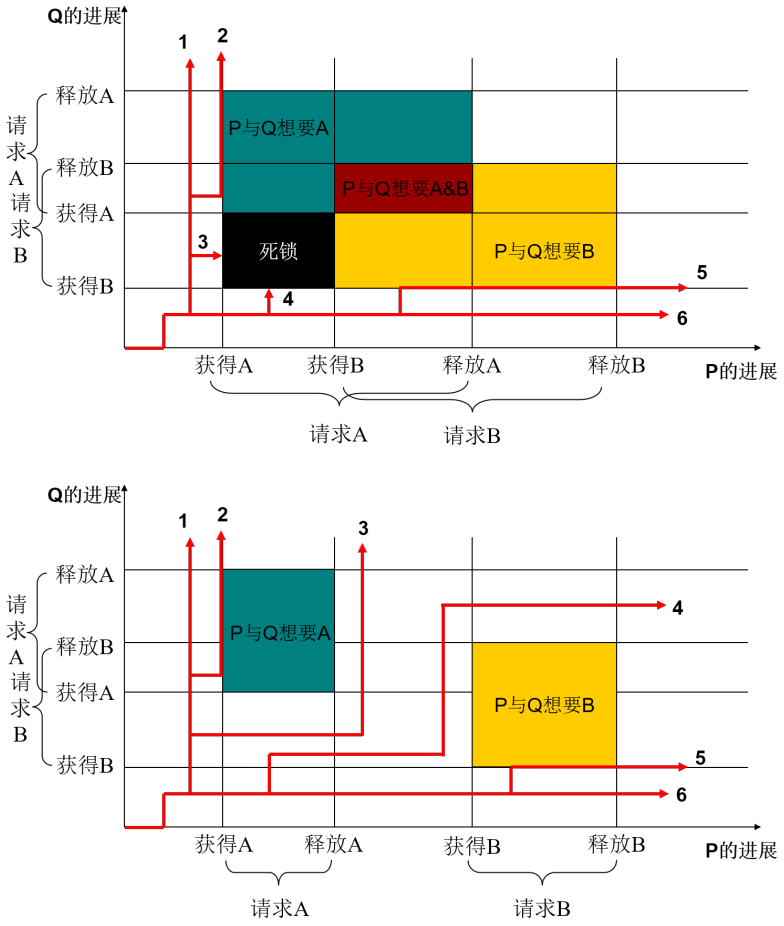
send和receive。

同样可以实现互斥，相当于在进程间传递一个可使用临界区的令牌

5 并发—死锁与饥饿

5.0.1 死锁

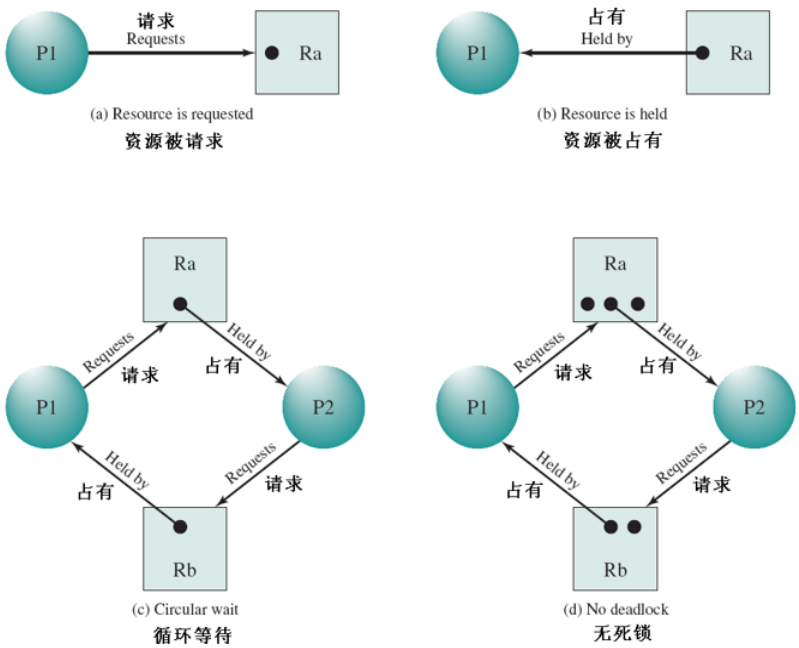
死锁(deadlock)：一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件（释放占有资源/进行某项操作）



联合进程图，上图死锁，下图无死锁

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，预提交资源	一次性请求所有资源	• 对执行单一突发行为的进程有效 • 不需抢占	• 低效 • 延时进程的初始化 • 进程必须知道未来的资源请求
		抢占	• 便于用于状态易保存和恢复的资源	• 过多的不必要抢占
		资源排序	• 通过编译时检测可实施 • 由于问题已在系统设计时解决，不需运行时再计算	• 不允许增加资源请求
避免	位于检测和预防中间	操纵以发现至少一条安全路径	• 不需抢占	• OS必须知道未来的资源请求 • 进程可能被长期阻塞
检测	自由	周期性地调用以测试死锁	• 不会延时进程的初始化 • 易于在线处理	• 丢失固有抢占

死锁定理：资源分配图中存在环路是存在死锁的充分必要条件



Examples of Resource Allocation Graphs

资源分配图的例子

死锁的充分必要条件

- 互斥
- 不可抢占（不剥夺）
- 占有且等待（请求和保持）
- 循环等待

5.1 死锁预防

5.1.1 破坏互斥条件

允许多个进程同时使用资源，但不适用于绝大多数资源，适用条件：

- 资源的固有特性允许多个进程同时使用（如文件允许多个进程同时读）
- 借助特殊技术允许多个进程同时使用（如打印机借助Spooling技术）

5.1.2 破坏占有且等待条件

禁止已拥有资源的进程再申请其他资源，如要求所有进程在开始时一次性地申请在整个运行过程所需的全部资源；或申请资源时要先释放其占有资源后，再一次性申请所需全部资源

- 优点：简单、易于实现、安全
- 缺点：进程延迟运行，资源严重浪费

5.1.3 破坏不可剥夺条件

一个已经占有了某些资源的进程，当它再提出新的资源请求而不能立即得到满足时，必须释放它已经占有的所有资源，待以后需要时再重新申请；OS可以剥夺一个进程占有的资源，分配给其他进程

适用条件：资源的状态可以很容易地保存和恢复（如CPU）缺点：实现复杂、代价大，反复申请/释放资源、系统开销大、降低系统吞吐量

5.2 破坏环路等待条件方法

- 要求每个进程任何时刻只能占有一个资源，如果要申请第二个则必须先释放第一个（不现实）
- 对所有资源按类型进行线性排队，进程申请资源必须严格按资源序号递增的顺序（可避免循环等待）

缺点

- 很难找到令每个人都满意的编号次序，类型序号的安排只能考虑一般作业的情况, 限制了用户简单、自主地编程
- 易造成资源的浪费（会不必要地拒绝对资源的访问）
- 可能低效（会使进程的执行速度变慢）

5.3 死锁避免

进程启动拒绝：考虑一个有n个进程和m种不同类型资源的系统。定义以下向量和矩阵：

Resource=$\mathbf{r}=\begin{bmatrix}R_1 & R_2 & \cdots & R_m\end{bmatrix}$	系统中每种资源的总量
Available=$\mathbf{v}=\begin{bmatrix}V_1 & V_2 & \cdots & V_m\end{bmatrix}$	未分配给进程的每种资源的总量
Claim=$C=\begin{bmatrix}C_{11} & \cdots & C_{1m}\\ \vdots & \ddots & \vdots\\ C_{n1} & \cdots & C_{nm}\end{bmatrix}$	C_{ij}为进程i对资源j的请求
Allocation=$A=\begin{bmatrix}A_{11} & \cdots & A_{1m}\\ \vdots & \ddots & \vdots\\ A_{n1} & \cdots & A_{nm}\end{bmatrix}$	A_{ij}为分配给进程i资源j的数目

有下列关系式成立：

1. $\forall j:\ r_j = v_j + \sum_{i=1}^{n} A_{ij}$。所有资源要么可用，要么已经被分配
2. $\forall i,j:\ A_{ij} \le C_{ij} \le R_i$：分配小于等于最大请求，最大请求小于等于资源总量

进而可以定义死锁避免策略：若一个新进程的资源需求会导致死锁，则拒绝启动这个进程，当且仅当

$$\forall j:\ R_j \ge C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$$

进程分配拒绝：银行家算法(Dijkstra)

$$\forall j:\ C_{ij} - A_{ij} \le v_j$$

- 只要系统处于安全状态（至少有一个资源分配序列不会导致死锁，即所有进程都能运行到结束），必定不会进入死锁状态
- 不安全状态不一定是死锁状态，但不能保证不会进入死锁状态
- 如果一个新进程的资源请求会导致不安全状态，则拒绝启动这个进程

优点

- 比死锁预防限制少
- 无须死锁检测中的资源剥夺和进程重启

缺点

- 必须事先声明每个进程请求最大资源
- 进程必须无关，没有同步要求
- 分配的资源数目是固定的
- 占有资源时进程不能退出

例 3. *在如下条件下考虑银行家算法。*

6个进程：P0-P5

4种资源：A（15单位）、B（6单位）、C（9单位）、D（10单位）

时间$T0$时的情况，左侧为分配矩阵A，右侧为请求矩阵C

	A	B	C	D	A	B	C	D
$P0$	2	0	2	1	9	5	5	5
$P1$	0	1	1	1	2	2	3	3
$P2$	4	1	0	2	7	5	4	4
$P3$	1	0	0	1	3	3	3	2
$P4$	1	1	0	0	5	2	2	1
$P5$	1	0	1	1	4	4	4	4
v	6	3	5	4				
All	15	5	9	10				

分析. 需求矩阵$C - A$

$P0$	7	5	3	4
$P1$	2	1	2	2
$P2$	3	4	4	2
$P3$	2	3	3	1
$P4$	4	1	2	1
$P5$	3	4	3	3

看有没有进程所请求的资源都小于等于可用资源，用完则将当前分配资源还回可用资源中

原来	6	3	5	4
$P1$	6	4	6	5
$P2$	10	5	6	7
$P3$	11	5	6	8
$P4$	12	6	6	8
$P5$	13	6	7	9
$P0$	15	6	9	10

如果某一个进程的请求后，需求矩阵每一行均小于等于可用资源的话，基于死锁避免原则，则该请求应该被拒绝

5.4 死锁检测

检测方法：

- 单个资源实例：检测资源分配图中是否存在环路（依据——死锁定理）
- 多个资源实例：类似银行家算法的安全检查

恢复：

- 剥夺法：连续剥夺资源指导不存在死锁

- 回退法：将每个死锁进程回滚到前面定义的某些检查点(checkpoint)，并重启所有进程（死锁可能重现）
- 杀死进程法

5.4.1 饥饿

饥饿：一组进程中，某个或某些进程**无限等待**该组进程中其他进程所占用的资源
进入饥饿状态的进程可以只有一个，而由于循环等待条件而进入死锁状态的进程必须大于等于两个

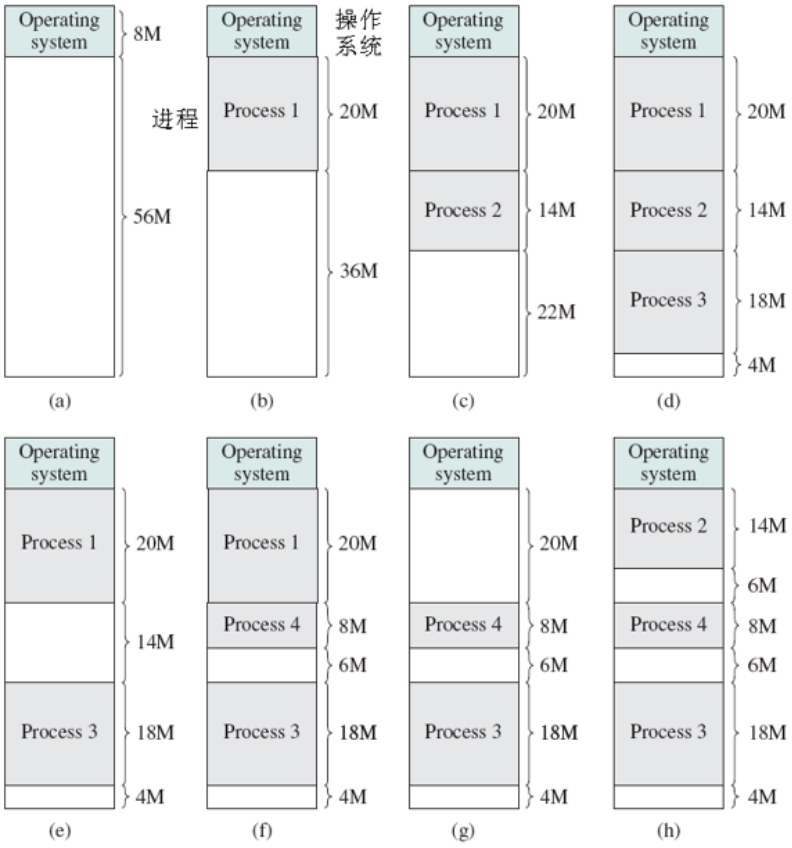
6 内存管理

6.1 分区存储管理

固定分区

- 等长分区：大进程则只能部分载入，小进程将产生内碎片
- 不等长分区：一定程度上缓解等长分区的问题

动态分区：会存在外碎片；若采用压缩方式移动进程使其紧靠，则非常耗时，需要进行动态重定位



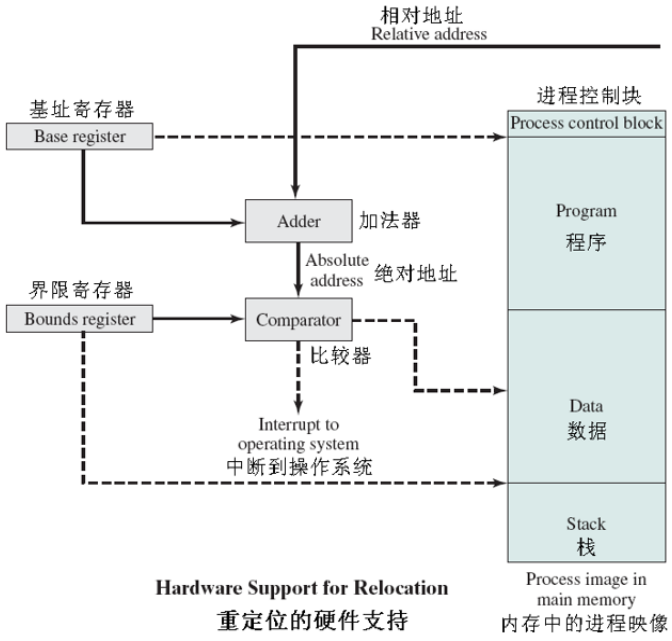
The Effect of Dynamic Partitioning
动态分区的效果

动态分区放置算法：

- 首次适配(first fit)：从前端开始扫描内存，直到找到一个足够大的空闲区，通常性能比较好
- 下次适配/邻近适配(next fit)：从上次分配结束的地方开始扫描内存，直到找到一个足够大的空闲区
- 最佳适配(best fit)算法：扫描整个内存，找出一个足够大的最小的空闲区，会产生很多外部碎片

伙伴系统(buddy system)：固定分区和动态分区的折中方案

- 可用内存块大小为$2^K, L \leq K \leq U$
- 初始空间大小为2^U
- 若请求空间大小$s < 2^{U-1}$，则对分现有块



重定位的硬件机制

6.2 页式存储管理

分页(paging)

- 将主存划分为许多等长的帧/页框(frame)
- 将进程划分为若干页(page)
- 进程加载时，所有页面被载入可用帧，同时建立页表

设页大小为L，逻辑地址A，物理地址E，则

$$\text{页号}P = A/L \qquad \text{页内偏移量}W = A\%L$$

例 4. *16位编址，若页面大小为1K(1024)，则需（低）10位表示页内偏移，剩下（高）6位表示页号，则*

- 相对地址为1502的逻辑地址$= 1024 + 478 = (1, 478)$
- 逻辑地址为$(1, 478)$的相对地址$= 1 * 1024 + 478 = 1502$

类似固定分区，不同在于：

- 分页中的“分区”（页帧）非常小（从而内碎片也小）
- 分页中一个进程可占用多个“分区” （页帧）（从而不需要覆盖）
- 分页中不要求一个进程占用的多个“分区”（页帧）连续（充分利用空闲“分区”）

存在问题：

- 不易实现共享和保护（不反映程序的逻辑组织）
- 不便于动态链接（线性地址空间）
- 不易处理数据结构的动态增长（线性地址空间）

6.3 段式存储管理

将程序及数据划分成若干段(segment)（不要求等长，但不能超过最大长度）

- 分页是出于系统管理的需要，分段是出于用户应用的需要：一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处
- 页大小是系统固定的，而段大小则通常不固定
- 逻辑地址表示
  - 分页是一维的，各个模块在链接时必须组织成同一个地址空间
  - 分段是二维的，各个模块在链接时可以每个段组织成一个地址空间
- 通常段比页大，因而段表比页表短，可以缩短查找时间，提高访问速度
- 分段对程序员可见，从而可用来对程序和数据进行模块化组织
- 分段方便实现模块化共享和保护，如程序可执行、数据可读写（段表表项要有保护位）
- 都存在外碎片，但分段中可通过减少段长来减轻外碎片浪费程度
- 分段中一个进程可占用多个“分区”，不要求一个进程占用的多个“分区”连续（但一般要求一个段所占用的多个“分区”连续）
- 分段克服了分页存在的问题（数据结构的动态增长、动态链接、保护和共享）
- 分段存在外碎片，分页只有小的内碎片，分页内存利用率比分段高

段表只能有一个，而页表可以有多个。
段页式系统中，逻辑地址被分为段号S、页号P和页内偏移量W。
逻辑地址偏移量只需小于段的长度即可。

6.4 虚拟存储

传统的存储方式都是一次性加载，并且驻留在内存中。而虚拟存储器则是基于程序的局部性原理，在程序装入时，将程序一部分装入内存，其余部分留在外存。

- 采用部分加载，内存中可同时容纳更多的进程。每个进程都只加载一部分，更多进程中应该也会有更多的就绪进程，从而提高CPU利用率

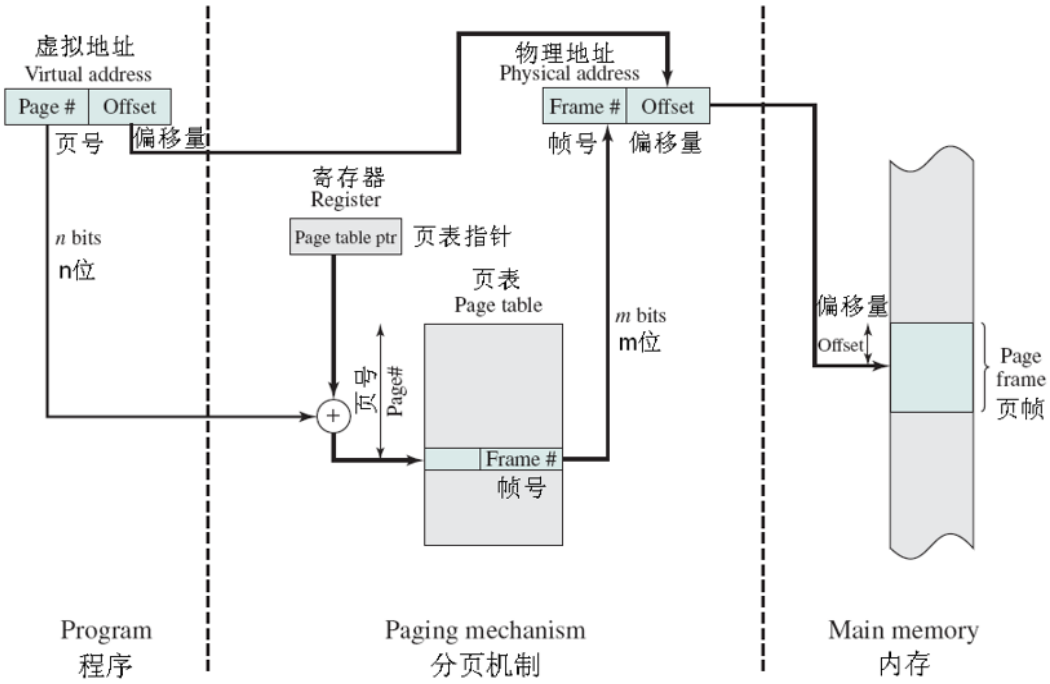
- 采用部分加载，进程可以比内存大，实现了虚拟存储
  - 用户程序可以使用的独立于物理内存的逻辑地址单元组成存储空间(虚拟存储)
  - 逻辑地址空间可以比物理地址空间大，例如，设物理内存64KB，1KB/页，则物理地址需要16位，而逻辑地址可以是28位!
  - 虚拟存储由内存和外存结合实现

虚拟存储技术的特征：不连续性、部分交换、大空间
抖动(thrashing)问题：交换操作太过频繁
页表

- 页表项（Page Table Entry，简称为PTE）的一般内容：
  - Present：在/不在内存
  - Modified：有没有被修改
  - Protection：保护码，1位或多位(rwe：读/写/执行)
  - Referenced：有没有被访问
  - Cache：是否禁止缓存
- 页表长度不定，取决于进程大小
- 不适合用寄存器存储页表，而是存放在内存
- 页表起始地址保存在一个CPU专用寄存器里(`cr3`)

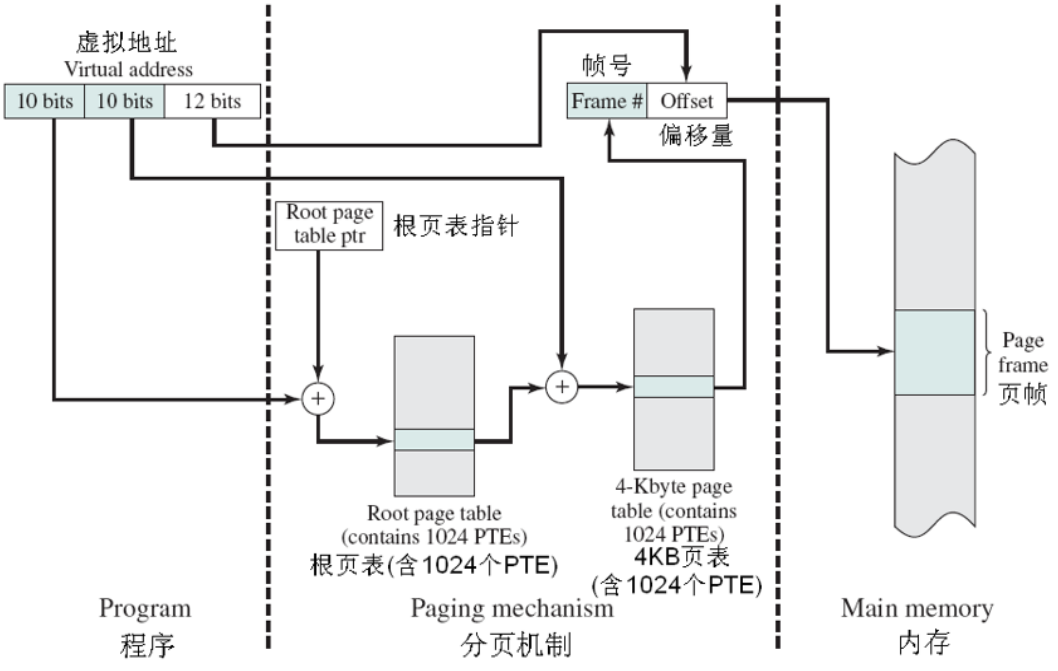
虚拟地址转换为物理地址流程：

- 将虚拟地址分割为虚页号和偏移量两个部分
- 通过虚页号在页表中寻找对应的表项
- 从中获取页框号后，乘页尺寸，加上偏移量，得到物理地址



Address Translation in a Paging System

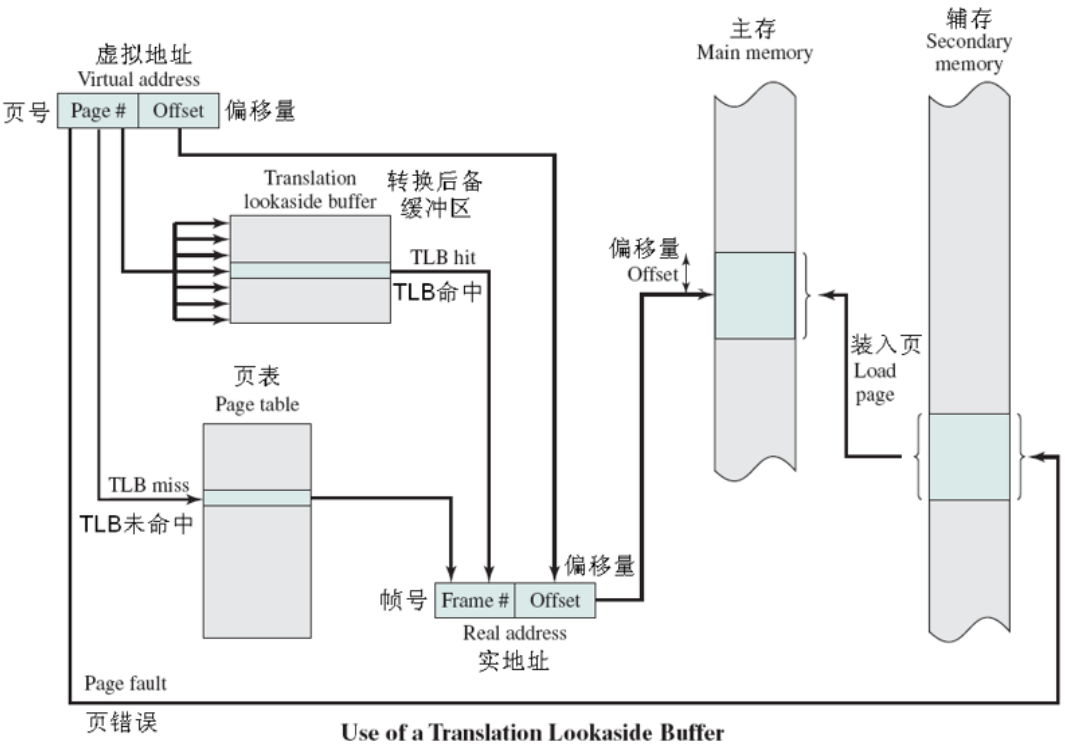
分页系统中的地址转换



Address Translation in a Two-Level Paging System

二级分页系统中的地址转换

快表/联想存储器(TLB)



Use of a Translation Lookaside Buffer

转换后备缓冲区的用法

常见页面大小介于1KB-8KB

- 调页策略：按需调页、预先调页
- 替换策略：
  - Opt(Belady)：置换下次访问距当前时间最长的页
  - LRU：最近最少使用
  - FIFO：先进先出
  - Clock：时钟，需要附加位，首次/被置换放入置为1，顺序循环扫描页表，将1置0，并选择第一个原来就是0的页表项放置

Linux的内存管理

- 虚拟存储采用三级页表
- 页面分配采用伙伴系统
- 页面替换采用时钟算法

7 调度

7.1 单处理器调度

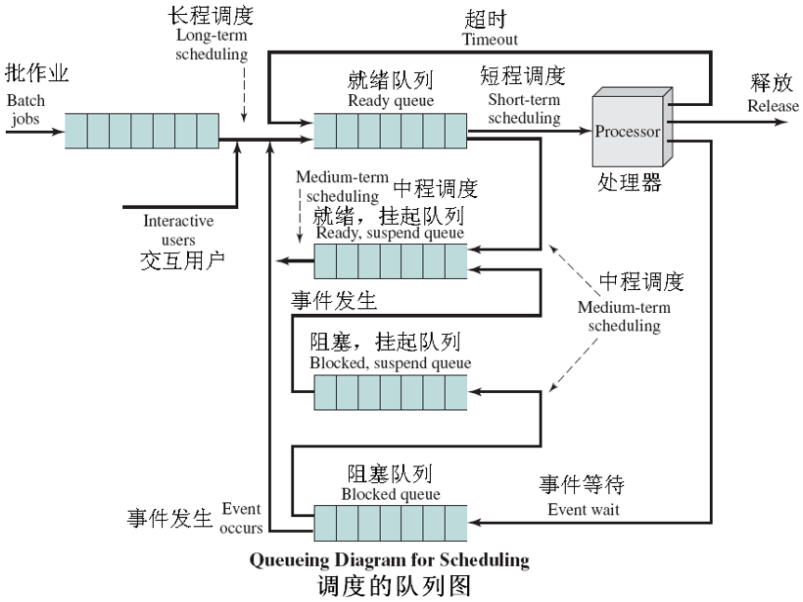
三级调度层次

- 长程调度(Long-term scheduling)/任务调度

  - 决定哪些新建进程可进入系统**准备执行**(ready)
  - 控制多道程序系统的并发程度
  - 进程越多则各进程对CPU的使用百分比越小

- 中程调度(Medium-term scheduling)
  - 决定交换哪些主存-辅存（内存-外存）进程
  - 基于多道程序设计的管理需要

- 短程调度(Short-term scheduling)/CPU调度
  - 决定下一个使用CPU的进程（dispatcher，分派程序）



Queueing Diagram for Scheduling
调度的队列图

进程调度方式

- 剥夺式：立即分配
- 非剥夺式：当前进程执行完再分配给新进程

进程调度算法

- 先来先服务(First Come First Served，FCFS)：公平
- 最短进程优先(Shortest Process Next，SPN)：效率，EWMA

$$S_{n+1} = aT_n + (1-a)S_n$$

- 最短剩余优先(Shortest Remaining Time，SRT)
- 最高响应比优先(Highest Response Ratio Next，HRRN)

$$R_P = \frac{T_{wait} + T_{serve}}{T_{serve}}$$

- 时间片(time slicing)轮转(Round Robin，RR)
- 最高优先级优先(Highest Priority First，HPF)
- 多级队列反馈(Multilevel Feedback，MF/FB)

周转时间：作业提交到作业完成所经历的时间

7.2 多处理器调度

多处理器线程调度方案

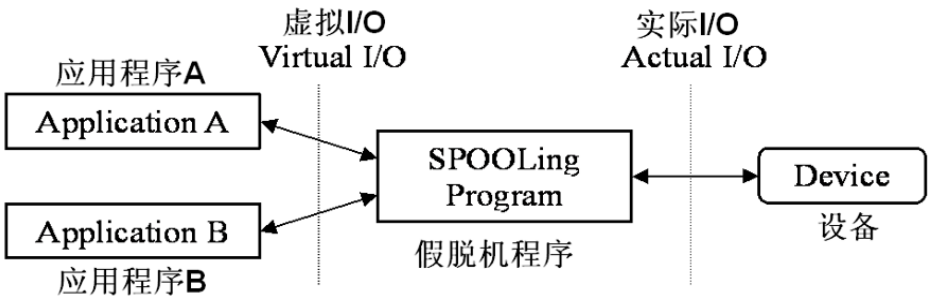
- 负载共享(load sharing)
- 组调度(gang scheduling)
- 专用处理器分配
- 动态调度

8 IO管理与磁盘调度

控制设备和内存或CPU之间的数据传送方式：

- 程序控制(programmed IO)
- 中断驱动方式(interrupt-driven IO)
- 直接内存访问(Direct Memory Access, DMA)

假脱机技术(Simultaneous Peripheral Operations On Line, SPOOL)/虚拟设备技术：专门利用一道程序来完成对设备的IO操作，而无需使用外围IO处理机



优点是高速虚拟IO操作，实现对独享设备的共享

IO缓冲：

- 单方向缓冲：单缓冲、双缓冲、环形缓冲
- 双方向缓冲：缓冲池(buffer pool)
- 循环缓冲

磁盘调度算法

- 先进先出FIFO
- 优先级PRI

- 后进先出LIFO
- 最短服务时间优先(shortest-service-time-first, SSTF)
- 电梯算法/扫描算法SCAN

9 文件管理

文件目录：将所有文件控制块(File Control Block, FCB)组织在一起，一个FCB称之为一个目录项

一级目录：整个目录组织是一个线性结构，系统中所有文件都建立在一张目录表中

优缺点：

- 结构简单、易实现
- 文件多时目录检索时间长，从而平均检索时间长
- 有命名冲突：如多个文件有相同的文件名或一个文件有多个不同的文件名

二级目录：在根目录/第一级目录/主文件目录MFD下，每个用户对应一个第二级目录/用户目录UFD，在用户目录下是该用户的文件，而不再有下级目录

多级目录：上下级关系

- 当前目录/工作目录.
- 父目录..
- 子目录(subdirectory)
- 根目录(root directory)/

Unix文件系统

- Unix磁盘文件系统结构
- 引导块（块0）
- 超级块（块1）
- i-索引结点表
- DOS文件系统：FAT12、FAT16、FAT32
- Windows文件系统：NTFS
- Linux文件系统：ext2/3