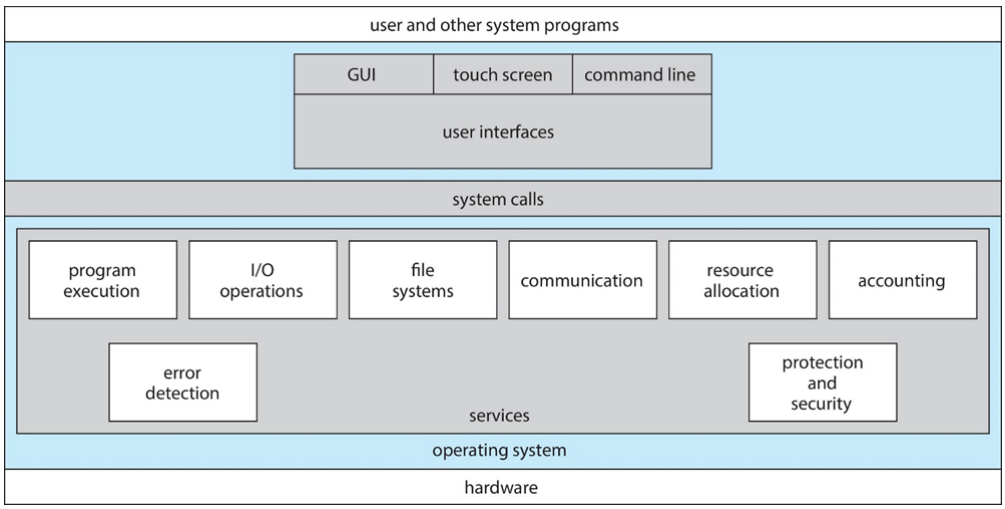
操作系统

绪论

- 操作系统属于系统软件
- 操作系统是中断驱动的。中断会抛出一个中断向量，包含着地址信息
- 陷入(trap)是软件中断
- cache: 逻辑概念，只是速度更快、容量更小
- 存储结构：寄存器-高速缓存-主寄存器-闪存-磁盘-光盘-磁带
- I/O
    - 程序I/O: scanf/printf
    - 中断I/O: 设备-CPU
    - DMA: 设备-内存
    - 通道方式
- processor(处理器)：一个或多个CPU+cache组成
    - 单处理器（利用时钟中断来处理）
    - 多处理器（对称、非对称)
- core(核)：CPU中的计算单元
    - 单核
    - 多核：一个处理器中有多个处理单元。
- 集群系统(cluster)：一组互联主机构成的统一的计算机资源
- 用户态：执行用户程序，普通指令。
- 内核态：访问系统资源，管理硬件设备，系统指令。
- 定时器(Timer)：防止无限循环以及被挂起很久的程序。
- 进程：执行中的程序。 线程：依附于进程存在。
- 操作系统工作：创建/删除 用户/系统进程。挂起/恢复进程。进程的同步/沟通/死锁。
- 虚拟化：可以运行多个不同的OS
- 文件系统：树结构
- 大型机：批处理系统。分时系统(将工作时间分开给不同的进程，一个重要指标为响应时间)。

操作系统结构

操作系统服务

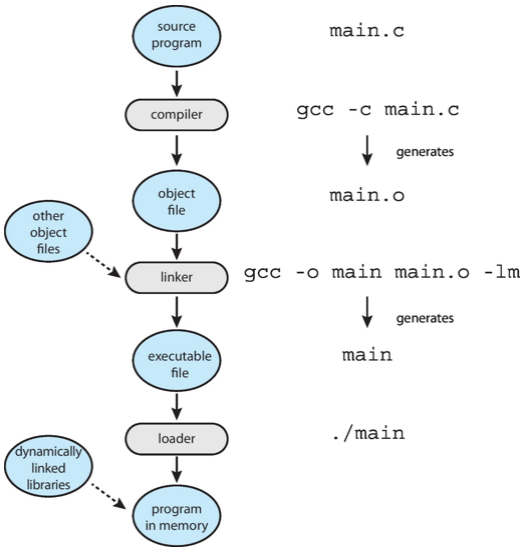


System Calls（系统调用）

使用C或C++的编程接口。介于processes和OS kernel之间。

应用编程接口(API)其实是一组函数定义。用来获得系统内核函数的服务。

link and load

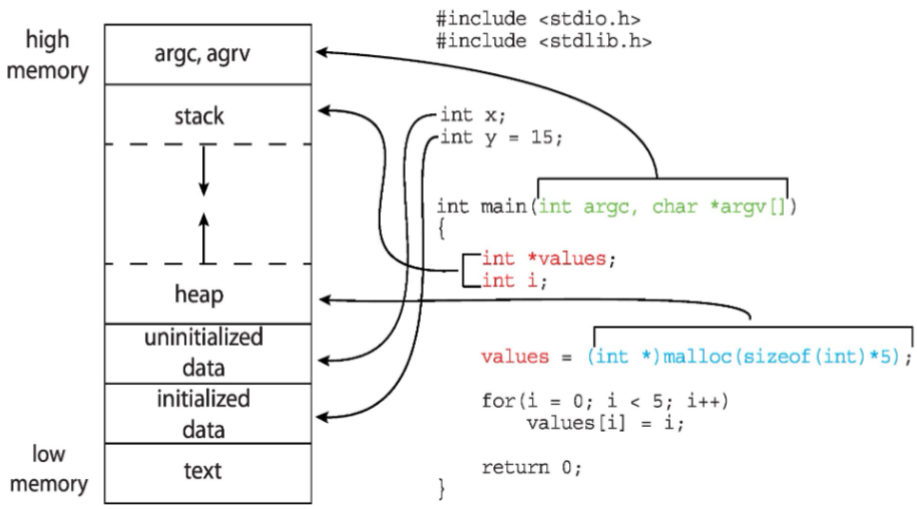


操作系统结构

- 简单结构
- 层次结构
- 单/宏内核结构
- 微内核结构
- 模块
- 混合系统
- 虚拟机

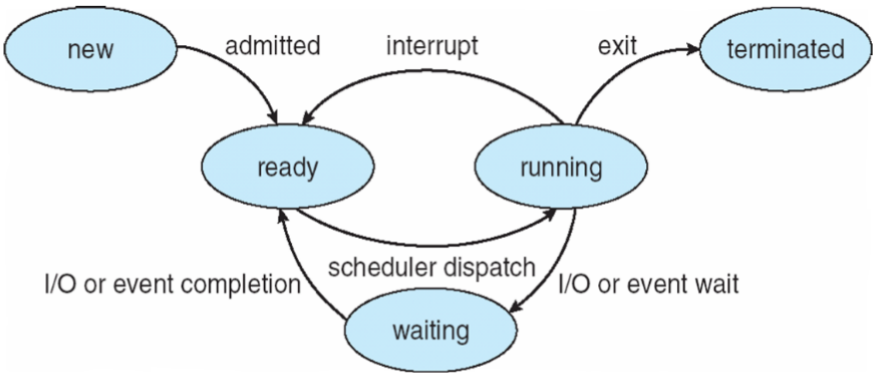
进程

> 进程是程序在处理机上执行的活动



进程状态

- new（新）：一个进程刚被创造出来
- **running**（运行、执行）：占用处理单元运行。与处理器的个数有关
- **ready**（就绪）：这个进程只需要CPU就可以运行。一般处于就绪状态的进程按照一定的算法排列，先来后到或者按照优先级进行排列
- **waiting**（等待，阻塞）：需要等待信号输入，如键盘、鼠标等。一般处于等待状态的进程也排列，一般按照等待的原因不同，等待队列也可以按时间分成几个队列。
- terminated（终止）：进程运行完毕



以上三个加粗状态为基本状态。其可能转换如下：

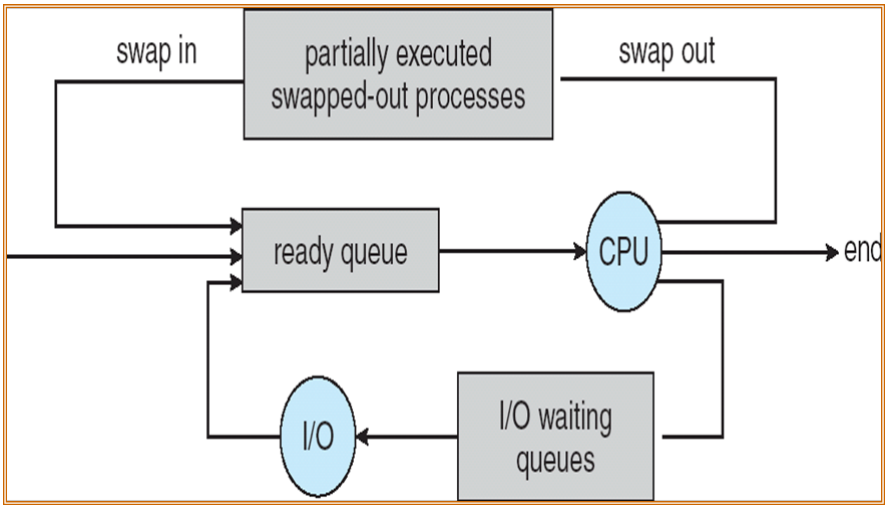
- 就绪-运行：当处理器空闲，就对就绪队列中的进程进行调度使一个进程执行
- 运行-等待：当进程运行到需要请求外部输入时转换为等待状态
- 等待-就绪：当等待的进程得到所需的输入，只需要CPU就可以运行
- 运行-就绪：正在运行的进程由于调度而使得暂停运行

PCB（进程控制块）

进程控制块包含一些相关信息，如进程状态，进程号，CPU调度信息，I/O状态信息等。

调度规则

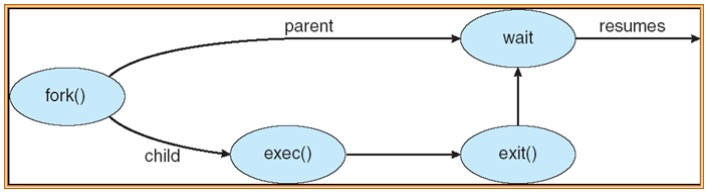
包括长程调度、中程调度、短程调度。不同类型的进程进行不同的调度规则。



上下文切换

当CPU进行任务转换时，需要保存旧进程的相关信息，这个信息称为上下文。上下文信息一般保存在PCB中。

进程操作

1. 进程创建。为每个创建的进程分配pid。由资源共享程度可以分为父子进程共享全部、子进程共享父进程部分与不共享三种类型。
   在unix系统中，fork用于创建子进程，从系统调用fork中返回时，两个进程除了返回值pid1不同外，具有完全一样的用户级上下文。在子进程中，pid1的值为0，父进程中，pid1的值为子进程的进程号。
2. 进程执行。exec系统调用进行子进程的执行。
   

3. 进程终止。exit。引起进程终止的事件：正常结束、异常结束、外界干预。如果一个父进程终止，则一般情况所有子进程也终止，也可能挂载到另一个进程中。
   安卓系统中由于资源不足可能会经常终止一些进程，按照重要性从低到高进行终止：
   - 前端进程
   - 可视进程
   - 服务进程
   - 后台进程
   - 空进程

进程通信

合作进程需要IPC（interprocess communication）

IPC的两种类型：

- Shared memory,即共享一块内存，以某种规则对其进行存取。
- Message passing,即通过message queue来进行通信。

通信的两种类型：

- 直接通信
- 间接通信，即通过中介来进行通信

常用的通信机制：信号、共享存储区、管道、消息、套接字。

共享存储区中的IPC机制

有一个in指针，一个out指针，分别表示存取数据。当in=out时，说明空间是空的，当in=out+1时，说明空间是满的。而指针的移动也通过取余来实现循环。

IPC的例子：POSIX（UNIX）。LPC（Windows）

管道（pipes)

- ordinary pipes: 可以用来在父进程和子进程间进行通信，但不能被创建它的进程以外的进程调用。是单向的，有write-end,read-end.
- named pipes: 功能更强大，可以不依赖父子关系调用。是双向的。

Client-Server system

- socket：套接字。The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- 远过程调用。注意数据大小端存储问题。

线程

> 线程是执行的最小单元，又被称为轻量级进程。而进程被称为资源占有的最小单元，可以分成多个线程，线程定义为进程内一个执行单元或一个可调度实体。

线程的特点：

- 有执行状态
- 不运行时要保存上下文
- 有一个执行栈
- 有一些局部变量的静态存储
- 可存取所在进程的内存和其他资源
- 可以创建、撤销另一个线程
- 不拥有系统资源
- 可并发执行
- 系统开销小、切换快

同一进程中的线程共享数据区、文件系统、代码域，只是用的寄存器不同、栈不同。

线程的优点：

- 创建一个新线程花费时间少
- 线程的切换花费时间少
- 同一进程内的**线程共享内存和文件**，因此它们之间相互通信无须调用内核
- 适合多处理机系统

多核编程

挑战有：

- 线程之间的任务平衡。（木桶效应）
- 数据间的独立性/依赖性（耦合度要低）
- 测试和调试

多线程

- 用户级线程：不依赖于OS核心（内核不了解用户线程的存在），应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。（阻塞型，一个线程发起系统调用阻塞，其他线程都要等待）
- 内核级线程：依赖于OS内核，由内核的内部需求进行创建和撤销，用来执行一个指定的函数。一个线程发起系统调用而阻塞，不会影响其他线程。时间片分配给线程，所以多线程的进程获得更多

CPU时间。（非阻塞型）
- 两者结合

多线程模式

- **多对一**：多个用户级线程映射到单核上。
  优点：不需要OS支持，可以调整调度策略来满足用户级的需求。线程操作代价低。
  缺点：不能利用多处理器，即不是真正的并行。当一个线程阻塞，则整个进程阻塞。
- **一对一**：每个用户级线程映射到内核线程上。
  优点：每个内核级线程都可以并行执行。当一个线程阻塞，其他线程可以正常工作。
  缺点：线程操作代价高
- **多对多**：多个用户级线程映射到多个内核线程上。允许操作系统创建足够数量的内核级线程。
- Two-level模式：将两种模式结合起来。

线程库（Thread Libraries）

Pthreads

隐私多线程（Implicit Threading）

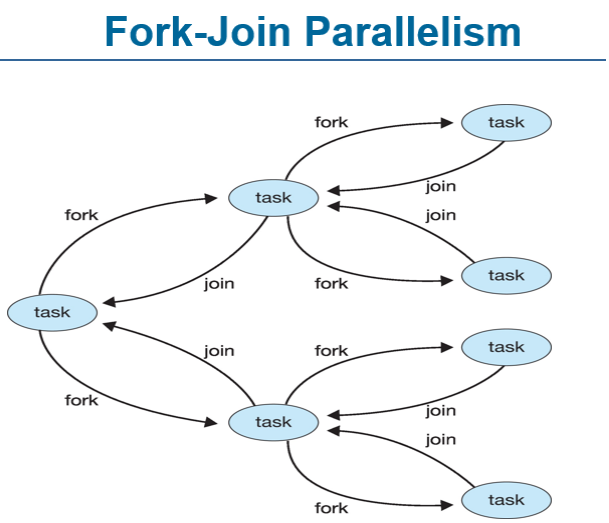
多核系统多线程编程，面临诸多挑战：

- 编程挑战：任务分解、任务的工作量平衡、数据分割、数据依赖、测试与调试
- 程序执行顺序的正确性问题：同步、互斥

策略：隐私线程Implicit Threading，当前一种流行趋势。即将线程的创建与管理交给编译器和运行时库来完成。

几种隐私线程的设计方法：

- 线程池
- Fork Join
- OpenMP
- GCD，大中央调度
- TBB，intel开发的多线程库，一个可移植的C++库
- java



CPU 调度

> CPU调度=处理器调度=进程调度

CPU调度一般为短程调度。内存调度一般为中程调度。admission调度（进程调度）一般为长程调度。

调度的时机：

- 当该进程从运行转到waiting/ready状态
- 当该进程从waiting转到ready状态
- 当该进程执行完毕

调度方式：

- 非抢占式，如调度时机1、4
- 抢占式，如调度时机2、3。依据的原则有优先权原则、短进程优先原则、时间片原则。

> 要尽量减小调度延迟

调度算法的选择准则和评价

- 面向用户的准则
    - 周转时间（Turnaround time）：进程从提交到完成所经历的时间。
    - 响应时间（Response time）：进程提出请求到首次被响应的时间。
    - 等待时间：进程在就绪队列中等待的时间总和。
    - 截止时间：开始截止时间 完成截止时间——实时系统。
    - 公平性、优先级。
- 面向系统的准则
    - 吞吐量（Throughput）：单位时间内完成的进程数——批处理系统。
    - 处理器利用率：使CPU尽可能忙碌。
    - 各设备的均衡利用。
- 算法本身的准则
    - 易于实现。
    - 执行开销小。

调度算法

- First-Come, First-Served (FCFS) Scheduling先来先服务调度
- Shortest-Job-First (SJF) Scheduling　短作业优先调度
- Priority Scheduling　　优先权调度
- Round Robin (RR)　　时间片轮转调度
- Multilevel Queue Scheduling　　多级队列调度
- Multilevel Feedback Queue Scheduling多级反馈队列调度

高响应比优先调度算法 Highest Response Ratio Next(HRRN)

响应比R = (**等待时间 + 要求执行时间**) / **要求执行时间**

FCFS算法

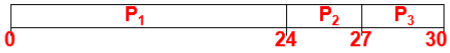
按照进程或作业提交顺序形成就绪状态的先后次序，分派CPU。当前进程或作业占用CPU，直到完成或阻塞，才让出CPU。

> 有利于长进程，不利于短进程。有利于CPU Bound，不利于I/O Bound。

FCFS Scheduling(Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- The Gantt Chart for the schedule is:



  - Waiting time for $P1$ = 0; $P2$ = 24; $P3$ = 27
  - Turnaround time for $P1$ = 24; $P2$ = 27; $P3$ = 30
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Turnaround time: (24 + 27 + 30)/3 = 27

```
turnaround time = termination time – arrival time.
```

```
Waiting time = turnaround time – burst time.
```

SJF算法

需要预计执行时间。执行时间短的进程优先执行。

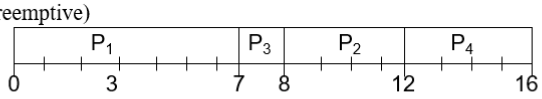
SJF的变型：

- 最短剩余时间优先SRTF：允许比当前进程剩余时间更短的进程抢占。
- 最高响应比优先HRRN： R越大，越先执行。

非抢断型：

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

- SJF (non-preemptive)



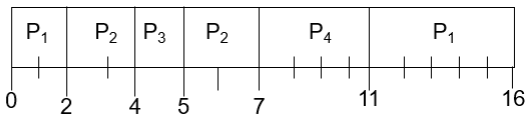
  - *Average Turnaround time*=(7+10+4+11)/4=8
  - *Average waiting time* = (0 + 6 + 3 + 7)/4 = 4

```
turnaround time = termination time – arrival time.
```

抢断型（SRTF）：

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

- SJF (preemptive)



  - *Average Turnaround time*=(16+5+1+6)/4=7
  - *Average waiting time* = (9 + 1 + 0 +2)/4 = 3

CPU运行时间预测方法：

1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define：

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

优先级算法

总把处理器分配给就绪队列中具有最高优先级的进程。有静态优先权或动态优先权。

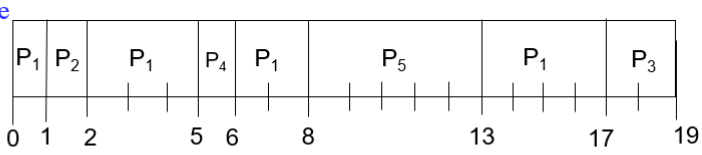
假定：优先级越高，优先级数值越小。

也分为抢占式、非抢占式。

抢占式：

Process	Arrival Time	Burst Time	Priority
P_1	0	10	3
P_2	1	1	1
P_3	4	2	4
P_4	5	1	2
P_5	8	5	2

- preemptive



- *Average Turnaround time*=(17+1+15+1+5)/5=7.8
- *Average waiting time* = (7+0+13 +0 +0)/5 = 4

这种调度算法可能会造成进程饥饿，即一个低优先级进程可能长期不被调度。

时间片轮转调度算法（RR）

> 基本思路：通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率。

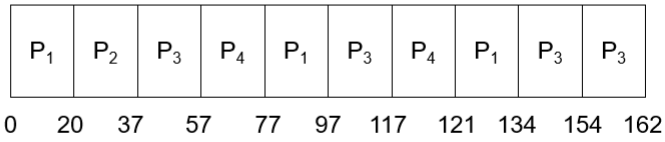
RR算法：

将系统中所有就绪的进程按照FCFS原则排成一个队列。
每次调度时将CPU分派给队首进程，让其执行一个时间片。
时间片结束，发生时钟中断。调度程序据此暂停当前执行的进程，将其送到就绪队列的末尾，然后上下文切换到当前的队首进程进行执行。

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- ***Average Turnaround time*=(134+37+162+121)/4=113.5**
- ***Average waiting time* = (81+20+94 +97)/4= 70.5**

- Typically, higher average turnaround than SJF, but better *response*.

> Q的确定：80%的进程的burst time要小于Q

多级队列调度

根据进程的性质或类型不同，将就绪队列分为若干个子队列。不同的队列有不同的调度方式。

每个作业固定归入一个队列。

> 一般而言，前台进程（交互性）使用RR调度，后台进程（批处理）使用FCFS调度。

此时，不仅不同的队列需要调度，队列间也需要调度。

队列间的调度方式：**多级反馈队列**

多级反馈队列：

- 设置多个就绪队列，分别赋予不同的优先级，如逐级降低。队列1的优先级最高。每个队列执行时间片的长度也不同，优先级越低则时间片越长。
- 新进程进入内存后，先放在队列1的末尾。按FCFS调度。若在队列1的一个时间片内未执行完，则降低到队列2的末尾。如此不断执行。
- 先完成优先级高的队列。优先级高的队列进程可以抢占CPU。

几点说明

- **I/O型进程**：让其进入最高优先级队列，以及时响应I/O交互。通常执行一个小时间片，要求可处理完一次I/O请求的数据，然后转入到阻塞队列。
- **计算型进程**：每次执行完时间片，进入更低优先级级队列。最终采用最大时间片来执行，减少调度次数。
- **I/O次数不多**，而主要是CPU处理的进程：在I/O完成后，优先放回I/O请求时离开的队列，以免每次都回到最高优先级队列后再逐次下降。
- 为适应一个进程在不同时间段的运行特点，I/O完成时，提高优先级；时间片用完时，降低优先级；

多处理器调度

多处理器调度存在一些问题：处理器间的优先级、数据的依赖性等

线程调度

线程调度库

进程同步

> 原子操作：执行该操作时不可以被中断

要达到的目的：将对共享的变量进行修改的操作变成原子操作。

进程之间竞争资源面临的三个控制问题：

- 互斥：多个进程不能同时使用同一个资源。
- 死锁：多个进程都不能得到足够的资源。永远得不到资源。
- 饥饿：一个进程长期得不到资源。资源分配不公平。

Critical-Section Problem

临界区：当一个程序在它的临界区执行时，不可以被打断。

临界资源：一次只允许一个进程访问的资源。

临界区的三个满足条件：

- 互斥：进入临界区时，不允许其他进程进入它自己的临界区。
- 空闲让进：如果没有进程在临界区，选择一个需要进入临界区的进程进入临界区。
- 让权等待：

Peterson's Solution

> 使load和store操作都是原子的

使用两个共享变量来控制：turn，表示轮到哪个进程进入临界区。flag，表示哪个进程准备好进入临界区了。

Algorithm 3(Peterson)

Two-Process Solutions

- Combined shared variables of algorithms 1 and 2.
- The two processes share two variables:
  - int turn;
  - boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!
- Process P_i

```
P0:
do {  flag[0]= true;
      turn = 1;
      while (flag[1] and turn = 1) ;
            critical section
      flag[0] = false;
          remainder section
} while (1);
```

```
P1:
do {  flag[1]= true;
      turn = 0;
      while (flag[0] and turn =0) ;
            critical section
      flag[1] = false;
          remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

面包房算法(bakery algorithm)

> 解决多处理器算法

每个进程都在进入critical section之前会分配一个数字，数字最小者、排序靠前者先执行。

硬件方法

> 设计特殊硬件与硬件指令来实现多进程算法

设计硬件同样也需要通用的指令。以下为两个指令：

- Test_and_set: 传入lock，将lock设为true，传出lock的原值。

```
Shared data:
    boolean lock = false;
while(1) {
    while (TestAndSet(lock)) ;
    critical section
        lock = false;
    remainder section
};
```

  这样当一个程序执行到该指令时，自己是false，但是将lock设置为true了，此时其他程序都无法进入临界区。

- Swap: 交换传入的a和b的值。

- 硬件方法的优点
  - 适用于任意数目的进程，在单处理器或多处理器上
  - 简单，容易验证其正确性
  - 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量
- 硬件方法的缺点
  - 等待要耗费CPU时间，不能实现"让权等待"
  - 可能“饥饿”：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上。

    > 解决方法：不要随机选择，设置一个waiting数组，表示要进入临界区的进程，然后按顺序选择。

  - 可能死锁：单CPU情况下,P1执行特殊指令(swap)进入临界区，这时拥有更高优先级P2执行并中断P1,如果P2又要使用P1占用的资源，按照资源分配规则拒绝P2对资源的要求，陷入忙等循环。然后P1也得不到CPU，因为P1比P2优先级低。

自旋锁：win、linux内核从来达到多处理器互斥的机制。当线程试图获得自旋锁时，处理器上的其他工作将终止。

信号量

> 用于同步互斥中

- 整形信号量
- **记录型信号量**
- AND型信号量，信号量集
- 二值信号量

一种数据类型，仅初始化、wait(P), signal(V)操作。

- wait(S)操作：

```
wait (S) {
    while S <= 0; // no-op
    S--;
}
```

- signal(S)操作:

```
signal (S) {
    S++;
}
```

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
         // Critical Section
     signal (mutex);
        // remainder section
} while (TRUE);
```

> 但是还是存在忙等待。

记录型信号量

> 一个waiting queue，一个值value。block()操作，将一个进程running变为waiting. wakeup()操作，将一个进程从waiting变为ready。

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

> 没有忙等待。

wait，signal操作讨论

通常用信号量表示资源或临界区，当信号量为正n，则表示有n个进程可以进入临界区。

信号量的物理含义：

- S.value > 0 表示有S.value个资源可用
- S.value = 0 表示无资源可用或表示不允许进程在进入临界区
- S.value < 0 有|S.value|个进程在等待队列。

wait(S)=P(S)=down(S): 申请一个资源

signal(S)=V(S)=up(S): 释放一个资源。

wait和signal必须成对出现。一般情况下，当为互斥操作时，两者处于同一进程，当为同步操作时，不在同一进程出现。

如果两个wait操作相邻，则他们的顺序很重要。如果一个同步wait与一个互斥wait相邻，一般同步wait操作在互斥wait操作之前。而signal操作的顺序无关紧要。

操作的优点：简单，表达能力强

操作的缺点：不够安全，使用不当会出现死锁，实现复杂。

通用的同步控制

先执行Pi的A，后执行Pj的B：

先将flag标志初始化为0，然后在Pi中，A执行完后进行signal(flag)操作，在Pj中，B的执行在wait(flag)之后进行，就可以达到目的。

存在的问题

同样也会产生死锁和饥饿问题。

信号量同步的缺点

1. 同步操作分散。信号量机制中同步操作分散在不同的进程中，操作不当就会产生死锁、饥饿等问题。
2. 易读性差。要了解某时刻某进程的信号量必须通读全程。
3. 不利于修改和维护。各模块耦合性过强。
4. 正确性难以保证。复杂系统容易出错。

经典同步问题

有限缓冲区问题（生产者-消费者问题）

> 一个有限共享缓存池，生产者存，消费者取

则信号量mutex初始化为1,用来控制不同进程的存取同步，信号量full初始化为0,用来控制存的限制，信号量empty初始化为n,用来控制取的限制。

读写者问题

> 对一个共享数据区，不同进程间的读写同步问题

则信号量mutex初始化为1，用来控制共享信号readcount的变化，信号量wrt初始化为1，用来控制写，共享信号readcount初始化为0，用来控制读。

哲学家就餐问题

> 一个圆桌围坐n个哲学家，人与人之间只有一只筷子，而人要吃饭需要左右两只筷子。即相邻的人不能同时吃饭，所以需要每个哲学家在eat和think之间做选择。（一个资源竞争问题）

则信号量chopsticks[n]初始化为1，用来表示n只筷子的占用。

- 解决方法1：

```
do {
    wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
        …
        eat
        …
        signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    …
        think
        …
} while (1);
```

这个方法会产生饥饿问题：每个人都拿一只筷子。

- 解决方法2：

  每次只让n-1个人竞争筷子。或者奇数编号先拿左边筷子，偶数编号先拿右边筷子。或者每次拿两只筷子，否则不拿。或者当哲学家拿起第一只筷子，在随机时间未拿到第二只，则放下第一只。

管程（Monitors）

> 用来解决同步问题的高级机制。

- 管程的定义：管程是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块。
- 管程的特征
  - 模块化：一个管程是一个基本程序单元，可以单独编译执行。
  - 抽象数据类型：管程是一种特殊数据类型，其中不仅有数据，还有对数据进行操作的代码。
  - 信息封装：管程是半透明的，有外部函数，但是具体实现不可见。
- 管程实现的要素
  - 管程中的共享变量在管程外部不可见。
  - 管程之间互斥进入。
  - 管程通常是用来管理资源的，因此在管程中应当有进程等待队列和相关等待与唤醒操作。

由于管程是互斥进入的，所以当一个进程要进入一个被占用的管程时，需要等待，所以在管程的入口处应该设置一个 **入口等待队列**。
而管程中如果进程不断唤醒其他进程，则管程中也存在进程等待的情况，故管程中也存在等待队列，称为 **紧急等待队列**，且其优先级高于入口等待队列。

同步操作原语

wait & signal。

x.wait表示将自己阻塞在x队列中。x.signal表示将x队列中的一个进程唤醒。

死锁

> 死锁：指多个进程因竞争共享资源而造成的一种僵局，若无外力作用，这些进程都将永远不能再向前推进

产生死锁的四个必要条件：

1. 互斥
2. 占有并等待
3. 不可抢占
4. 循环等待

资源分配图：

P：进程。 R：资源

请求边：$P_i \to R_j$

分配边：$R_j \to P_i$

如果资源分配图中存在环：如果每个资源只有一个实例，则一定死锁，否则不一定死锁。

死锁定理：S为死锁状态的充分条件是：当且仅当S状态的资源分配图是不可完全简化的。

处理死锁的策略

1. 死锁预防（虚拟化、可抢占）、避免（每个进程申明所需的最大资源数，由算法动态检测以避免死锁）
2. 死锁检测、解除
3. **鸵鸟方法**：假设不存在死锁，即忽略死锁。是大部分操作系统使用的方法。

下面重点讲讲避免算法：

在避免死锁算法中有一个定义：**安全状态。**是指在某种状态下系统可以按某种顺序($P_1, P_2, \ldots, P_n$)来为各进程分配其所需资源直至最大需求，使每个进程可以按顺序顺利执行完成，则这个顺序($P_1, P_2, \ldots, P_n$)被称为安全序列。如果系统不存在一个安全序列，则称系统处于不安全状态。

> 在避免算法中需要解决以下两类情况：单资源问题，多资源问题。

单实例资源（即每个资源只有一个实例可用）问题使用**资源分配图算法**，如果有需求请求，我们将资源分配图中的请求边换为分配边，看是否出现了环，如果没有环，则存在安全状态，允许分配。

多实例资源（即每个资源有多个实例可用）问题使用**银行家算法**（该算法由Dijikstra提出，用于银行贷款现金发放而得名），其数据结构如下（假设有n个进程，m种资源）：

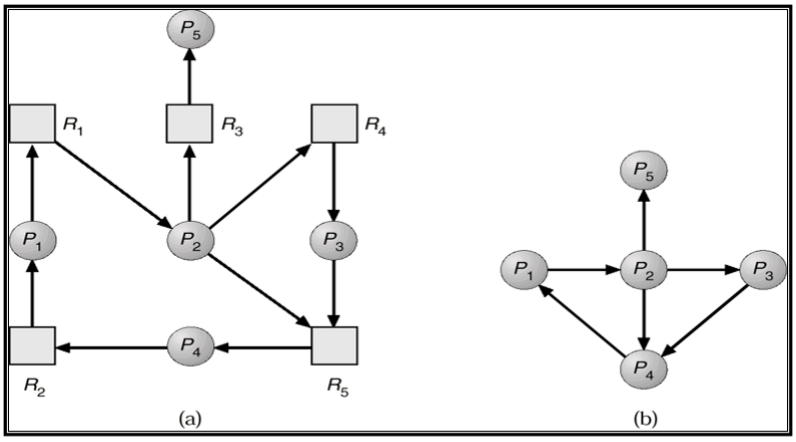
- Available: 长度为m的向量，表示各资源可分配的数量。
- Max: n*m的矩阵，表示各进程对各资源所需的最大数量。
- Allocation: n*m的矩阵，表示现已分配给各进程的各资源的数量。
- Need: n*m的矩阵，表示各进程现对各资源的需求数量。

在此基础上，有一个简单的算法叫**安全算法**：即检查每个进程的Need是否小于Available，如果是，则执行完该进程，并改变相应的值，如果否，则检查下一个。如果每个进程都执行完，则安全。

安全算法的升级版：**资源分配算法。**即如果一个进程请求小于它的Need并且Available资源数可满足，则先假装分配给它，然后使用安全算法检测，如果是安全状态，则分配。

下面再讲讲死锁检测算法：

这里需要引入另一个图：资源等待图。



Resource-Allocation Graph **Corresponding wait-for graph**

资源等待图中没有资源，只有消去了资源后进程之间的相互等待关系。

其数据结构如下：

- Available：长度为m的数组，表示m个资源是否可用。
- Allocation： n*m的矩阵，表示m个资源分别已分配给n个进程的数量。
- Request： n*m的矩阵，表示n个进程对m个资源分别请求的数量。

算法如下：

work = Available, 检查Allocation中的每个值，如果不等于0，则finish[i]=false.

分别检查，是否存在finish[i]==false, Request[i] <= work, 如果存在，则work+=Allocation[i]（即释放资源）, finish[i]=true（表示该任务完成），然后继续检查。如果不存在，则死锁。

其实死锁检测算法在避免算法中也用到了，在避免的时候其实是假装先分配，然后进行死锁检测。

下面再讲讲死锁恢复：

检测到死锁后采取的措施：

- 通知系统管理员
- 系统自己恢复

打破死锁的两种方法：

- 进程终止
- 抢占资源

Main Memory（内存）

> 本章主要内容
>
> - 简介
> - 交换(swap)
> - 连续内存分配
> - 分页(page)
> - 分段(segmentation)
> - 段页式
> - 例子：The Intel Pentium

简介

现代计算机架构为冯诺依曼架构，计算和存储分离。目前有研究类脑计算方向，目的为实现仿生类型的计算机。

逻辑地址/物理地址：逻辑地址即相对地址、虚地址。物理地址即绝对地址、实地址。

逻辑地址与物理地址之间存在映射关系，需要使用动态重定向。MMU: 内存管理单元。

交换（SWAP）

由于内存总是不能直接满足需要，所以刚运行过的进程会被暂时交换到硬件上的backing store, 需要的时候再交换回来。

在交换过程中，变量的内存地址会发生变动，比如被swap进来的进程占用了，这时需要动态重定位，需要用到MMU。

交换机制：交换需要消耗资源和算力，所以交换的频率越快，性能越低。一般而言不允许交换。如果剩余空间低于某个低阈值，则进行换出，直到高于某个高阈值。

> 移动系统不支持交换。IOS要求程序自愿放弃分配的内存，Android系统会主动终止某程序，然后将其写到闪存，便于快速重启。

连续内存分配

内存一般分为两块：操作系统空间、用户空间。用户空间中的各进程应该相互独立，各自具有基地址(Base register，保存最小物理地址)，限制地址(Limit register，最大的可访问逻辑地址)，内存管理单元(MMU)。

对于每个请求空间的进程，操作系统会进行**动态内存分配**，按照其请求的空间大小分配一块连续内存给它。这样做会产生一些间隔空间（当某进程执行完毕返回空间时）。那么当进行动态内存分配时，有以下三种策略：First-fit（速度最快）, Best-fit（效果最好）, Worst-fit（即找剩余空间最大的进行分配，这样我们认为分配后的剩余空间还可以再利用，而不是变成碎片）。

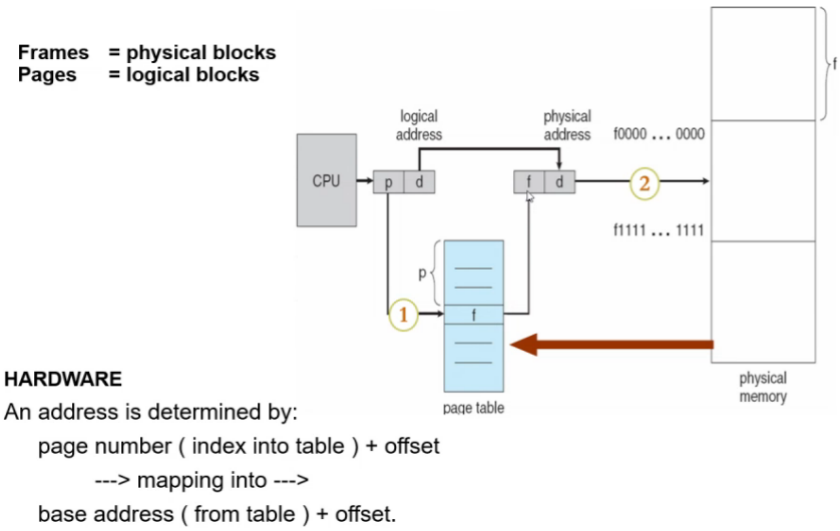
但是动态内存分配已经不再使用了。

现代内存分配的思想是，不再为进程分配完整的一块内存，而是分隔开，一个进程分成几块，这样可以利用间隔空间，然后用某种方式将这几块连起来。这就是**分页(page)**

分页（page）

将内存分页，在windows系统和linux系统中大小都固定为4K。将进程按页大小分割，然后存储在不同的页中，并用**页表**来存储对应关系。

在页表中进行地址变换，存储着页号(Page number)和偏移(page offset)，用来查找某页中的某个block。所以页大小有n bytes， offset有$log_2 n$位。



进程中使用的地址（虚拟地址）前一部分p在页表中找到对应的物理页地址，物理页地址加上后一部分的偏移量在物理的页中找到偏移了m行的block。

为了加快寻找的速度，我们有TLB表，本质是一个cache，但是这个cache专门用来存放页表信息，但是可存储量比较小，当miss时就需要到原页表中查找。

页表结构

- 分级页表

  windows分两层页表，linux分四层页表。分层的原因是如果只有一层页表，则为了表示所有物理地址（每一条信息表达4KB），则这个页表变得太大，会占用太多连续空间。所以这时候我们分两层，存储数据为p_1p_2d, p1在一级页表中找到地址，通过它找到二级页表，然后用p2在二级页表中找到物理页，然后通过偏移d来找。

- 哈希页表

  像是多路cache一样，我们通过p和某标志找到某一行，然后遍历，找到标志位存放的物理页地址，然后通过偏移d来找。

- 反向页表

  为每个页地址分配唯一的pid，然后通过pid来找到物理页地址，然后通过偏移d来找。

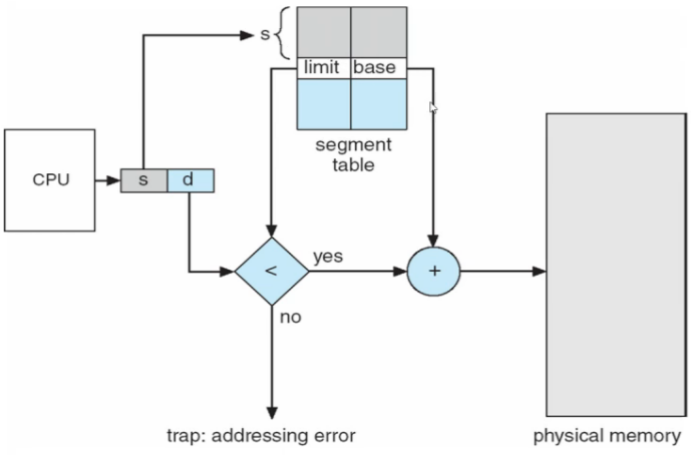
分段（segment）

对于分页来说，我们为每个内存分配的大小是统一的，全是4KB。那么我们可以对每种不同的需求分配不同的大小，比如对于栈、堆、主程序、变量等分配不同大小的页，这就是分段。

其结构为

段号	段内地址

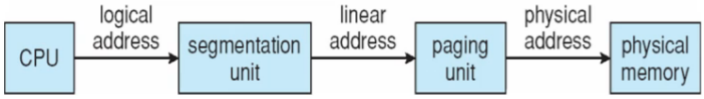
其存储有base地址和偏移地址。还有一个valid位。

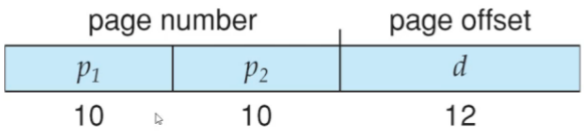


通过段号s找到一个limit和base，base就是物理地址的基地址，limit用来检测偏移量d是否合法，如果d<limit，则加上base进行查找，如果大于，则说明访问了非法地址。

段页式

通过MMU来结合分段和分页。





先用分段找到一个地址，在该地址通过分页找到物理页地址。

虚拟内存

> 本章内容：
>
> - Background
> - **Demand Paging**
> - Copy-on-Write
> - **Page Replacement**
> - Allocation of Frames
> - Thrashing
> - Memory-Mapped Files
> - Allocating Kernel Memory
> - Other Considerations
> - Operating-System Examples
> - Summary

Background

一般只有运行进程的一部分需要加载到内存中执行，这样做我们可以虚拟地扩大内存容量。

虚拟内存可以通过以下两种方式实现：

- Demand Paging（请求调页，按需调页，**请求页式管理**）
- Demand segmentation（**请求段管理**）

局部性原理：指程序在执行过程中一个较短的时期，所执行的指令地址和指令的操作数地址，分别局限在一定的范围内，有**时间局部性**（即刚访问过的内存很可能再次访问）和**空间局部性**（即刚访问过的地址其相邻地址很可能再次访问）。

Demand Paging

> 当某页被需要时才将其放入内存中，这叫作Lazy swapper

更完整的页表：每个页表项包括：

物理块号	状态位P	访问字段A	修改位R/W	外存地址

状态位表示该内容是否在内存中。

访问字段表示近期被访问的次数。

修改位表示调入内存后是否被修改过。

外存地址表示物理地址所在。

我们在页表中有一位v-i表示valid(in memory)，invalid(not in memory), 如果是i，则我们称为**Page fault**（缺页）.

缺页处理：

1. 操作系统通过另外一个表中来判断是非法地址访问还是只是不在内存中。
2. 获取空frame。
3. 将访问的物理页交换到空frame中。
4. 重置页表。
5. 设置valid位为v。
6. 重新执行该访问。

由于缺页很耗时，所以我们需要对它进行评估（EAT）。我们令p表示缺页率，范围[0.,1.]，通常p应该很小。

EAT = (1-p)*memory access time + p*average page-fault service time.

process creation(Copy-on-Write)

写时拷贝：父子进程共享的内存，当任一进程要修改时，我们先将该内存复制一块，存储在其他地方，然后修改这块内存。

Page Replacement（页面置换）

> 如果页表全被占满，而进程请求新的页，此时需要进行页置换

先按照一定的算法找到可以被交换出去的页信息，然后看它是否被修改过(dirty bit),如果没有被修改过，则直接覆盖即可，如果被修改过，则要先写出物理内存，然后再覆盖。

替换算法：要找到一个可以被交换出去的页信息，我们希望替换这个可以使得之后的page fault rate尽可能小。

评估替换算法我们使用“引用串”来表示之后会引用的页，然后通过它计算page fault rate.

替换算法有以下几种：

- FIFO算法
  最先进来的最先被替换出去
- OPT最佳页面算法
  “不再使用的”或“离当前最远位置上出现的（即当前要进入的位置与表中某页在引用串中距离最远的）”页被置换。
- LRU最近最久算法
  或者我们建立一个双向链表。每次一个页被访问，将它重新交换到头。每次要进行页替换，只需要替换链表尾的页。
- 基于计数的算法
  这个标志也可以是一个计数器。每个页分配一个计数器，每次被访问，计数器增加。对于这个计数器，我们有两种算法：
  LFU:最不经常使用算法：将计数器最小的换出。即使用频率最小，不一定是最久使用的换出。
  MFU:最常使用算法：将计数器最大的换出，即使用次数最多的换出。
- 近似LRU算法
  额外应用位算法：新增一个标志位，表示每个页最近使用过的时间。将用过最久的页换出。
  这个标志可以是8bits，然后每次使用，都将最高位置1.每过一定时间，这个标志右移一位。所以每次只取标志最小的页换出即可。
  二次机会算法：近似FIFO算法，当按照FIFO选择了一个算法后，观察其引用位，如果是1，则给它第二次机会，继续查找下一个。
  增强二次机会算法：近似二次机会算法，只是不仅要观察引用位，还要观察修改位。当两者都是0，则是最佳修改页面。
- ...

页面缓冲算法

用FIFO算法选择被置换页，把被置换的页面放到两个链表之一：如果页面未被修改，放到空闲链表末尾。否则放到已修改页面链表。
需要调入新的页面时，将新页面内容读入到空闲页面链表第一项所指的页面，然后将第一项删除。
空闲页面和已修改页面仍停留在内存中一段时间，如果这些页面被再次访问，这些页面仍在内存中。
当已修改页面达到一定数目后，将它们一起调出到外存，然后将它们归入空闲页面链表。

Allocation of Frames(帧分配)

分配给每个进程最小的页数。

分配策略：

- 固定式分配
  - 平均分配法
    将所有页表平均分配给n个进程。
  - 比例分配法
    按照n个进程的大小分配页表数。
- 优先级分配
  如果某进程发生了page fault，给它分配最小优先级的进程的帧。

置换策略：

- 全局置换：进程从所有帧中进行置换。
- 局部置换：进程只在自己分配的页中进行置换。

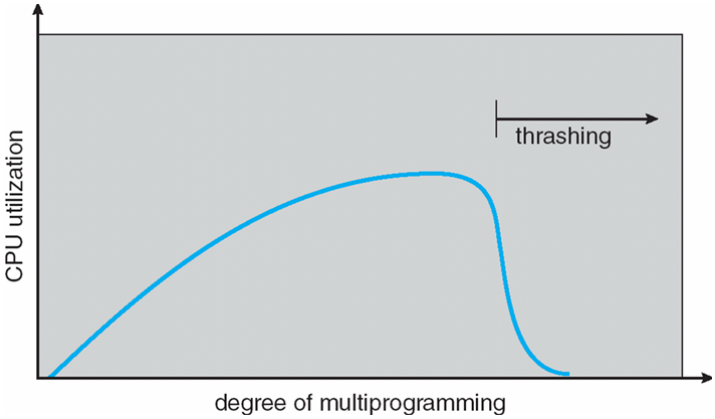
分配策略与置换策略组合成三种策略：

- 固定分配局部置换策略

- 可变分配全局置换策略
- 可变分配局部置换策略

Thrashing（颠簸）

当进程中频繁发生页面置换时，CPU利用率很快下降，耗费大量时间在进行页面的置换上。



如何解决thrashing?

Working-Set（工作集模型）

在最近的Δ次页面访问中所有被访问到的page的集合。

Δ：工作集窗口（一个固定的页面访问次数/一个固定的指令数）.

如果窗口太小，则无法执行完一个locality，效率低。

如果窗口太大，则会引入多个locality，可能会造成thrashing。

WSS_i: working set size of process p 工作集大小

$D = \sum WSS_i$= total demand frames; m = total frames.

if D > m, thrashing。此时我们需要不断挂起进程。

缺页频率

缺页频率应该有一个合适的范围，不能太高或太低。

如果太低，说明每次给进程分配的帧都过多地剩余，则帧使用率太低。

如果太高，则在页置换中消耗太多资源。

系统内存分配

给系统内存分配空间，我们有两种方式：

从上而下：Buddy System，我们找到一个空间，不断对它缩减一半大小，如果刚好符合，就把系统某块放到这里。

从下而上：Slab Allocator。

其他考虑

- 预调页
  减少冷启动时的page fault。
- 页大小
  页的大小影响table size，I/O处理以及局部性。

- TLB范围
- 程序结构
  尽量使每次调用都完整地使用，尤其是在循环遍历中，要考虑页面置换的问题。
- I/O锁定

Ch 10 File System Interface

> 本章主要内容
>
> 1. File Concept
> 2. Access Methods
> 3. Directory Structure
> 4. File-System Mounting
> 5. File Sharing
> 6. Protection

File Concept

文件是存储某种介质上的（如磁盘、光盘、SSD等）并具有文件名的一组相关信息的集合。

文件操作有很多，以下介绍一个重要的：

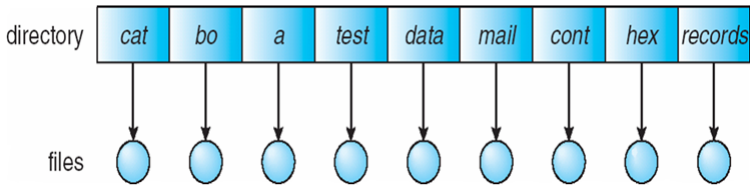
- Open file
  会有以下结构：
  file pointer: 指向打开的文件
  file-open count: 打开的文件计数器
  disk location of the file: 文件位置
  access rights: 文件访问权限

Access Methods

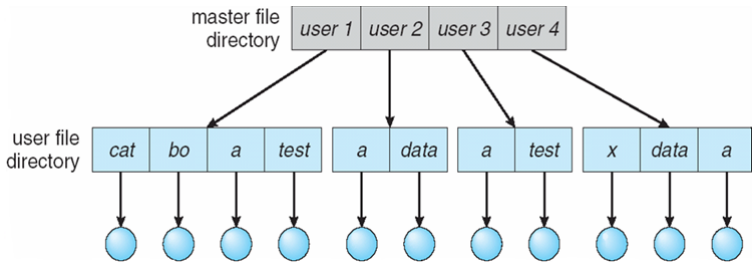
- 顺序存取
- 直接存取
- 索引顺序

Director Structure

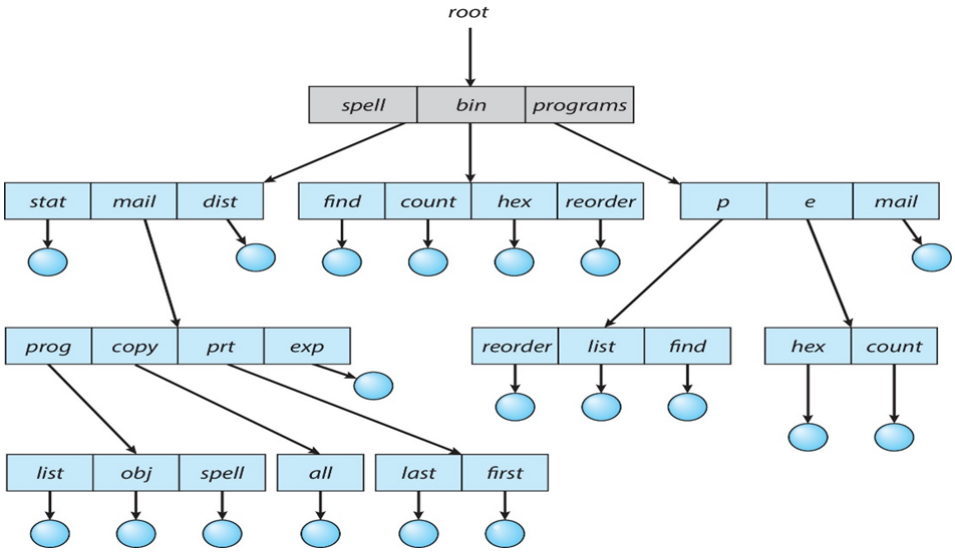
单级目录



二级目录



树型目录（搜索快、组织形式高效）



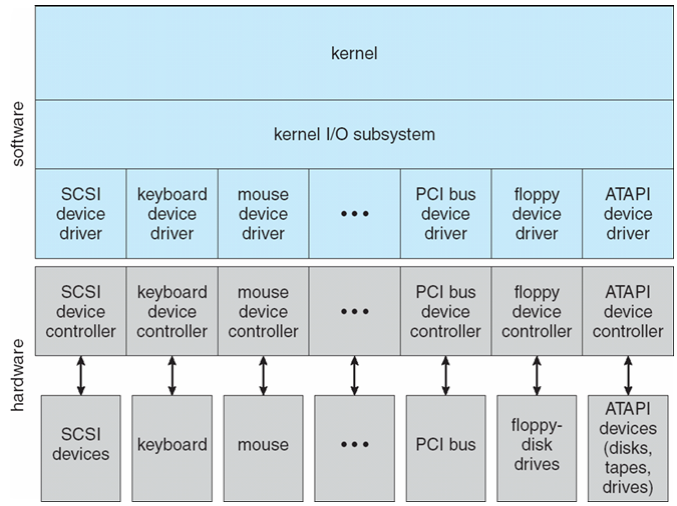
但是存在一个问题：如果将一个文件夹下的所有文件夹删除，则这个文件夹成了一个悬挂指针。

解决方式：添加一个反向指针。或者添加一个引用计数器，表示该文件指针被引用的数量。如果引用计数器变为0，则将其放到空闲列表中。

Ch13 I/O

- 轮询
- 中断
- DMA

I/O设备



- 缓冲Buffer

  用来保存在两设备之间或在设备和应用程序之间所传输数据的内存区域。

  解决设备速度不匹配，解决了设备传输块的大小不匹配。

- 假脱机技术SPOOLing

  用来保存设备输出的缓冲，这些设备如打印机不能接收交叉的数据流。

I/O请求到硬件操作

1. 确定保存文件的设备
2. 转换名字到设备的表示法
3. 把数据从磁盘读到缓冲区
4. 通知请求进程数据现在是有效的
5. 把控制权返回给进程

期末考试

3张A4纸

40选择+填空+3大题

要把知识点全部过一遍。

- 中断
  - 概念
  - 处理方法
- **系统调用**
  - 寄存器存储调用号
- 了解和区分系统内核结构
- **进程控制块PCB**
- **fork（创建子进程）**
  - 创建独立的空间
  - fork函数
- 线程的多种模型（内核、用户）
- **调度**
  - 了解评价指标及其关联性
  - **调度算法**
    - 甘特图
    - 评价指标的计算
- **进程同步**
  - 其中的逻辑
  - 临界区（不一致性问题）
  - **信号量**
    - 信号量的定义，什么样的变量可以定义为信号量（互斥的、可计数的）
    - 信号量的使用（P53页例）
- 死锁
  - 死锁定义（4个条件）
  - 死锁预防和避免
  - 死锁解决
- **内存！**
  - 内存分配（first-fit，best-fit，.....）
  - **分页！**

        - 分页机制
        - 页表转换
        - 快表TLB（计算EAT）
    - 分段
        - 分段机制
    - 缺页
    - **页面置换**
        - 页面置换算法
        - 计算页面置换次数
    - Thrashing
- 文件系统
    - 文件系统的保护（权限）
    - 文件分配方式（P46例）
    - 了解网络文件系统、虚拟文件系统
- 大容量存储
    - 磁盘的访问调度
- I/O
    - 实现方式