**操作系统：**

//时钟振荡器发送方波信号,供CPU启动,

CPU启动的时候一个BOOT会将磁盘的信息复制到内存当中.

CPU到内存中执行,CPU在内存中没找到就会去磁盘中找,找到之后复制到内存中执行.

//操作系统是软件和硬件资源的管理者:操作系统也是一种软件,管理一些硬件的分配.?(CPU分配,内存分配,磁盘分配,外设分配)

//操作系统在硬件之上,用户的程序运行在操作系统之上;

**基本特征：**

//并发:是指宏观上能在一段时间内同时运行多个程序，而并行则指的是统一时刻能运行多个指令；并行需要硬件支持，如流水线、多核处理器或者分布式计算机系统；操作系统通过引入进程和线程，使得程序能够并发运行。

//共享:是指系统中的资源可以被多个并发进程共同使用;有两种共享方式:互斥共享和同时共享;互斥共享的资源称为临界资源,例如打印机,在同一时刻只允许一个进程访问,需要用同步机制来实现互斥共享.

//虚拟:虚拟技术把一个屋里实体转换成多个逻辑实体;主要有两种虚拟技术:时分复用技术和空分复用技术;多个进程能在同一个处理器上并发执行使用了时分复用技术,让每个进程轮流占用处理器,每次只执行一个小的时间片,并且快速切换;虚拟内存使用了空分复用技术,将物理地址抽象为地址空间,每个进程都有各自的地址空间,地址空间的页被映射到物理内存,地址空间的页并不需要全部在物理内存中,当使用一个没有在物理内存的页的时候,执行页面置换算法,将该页置换到内存中.

//异步：进程不是一次执行完毕，是走走停停，以不可知的速度向前推进

**基本功能：**

（1）进程管理

进程控制、进程同步、进程通信、死锁处理、处理机调度

（2）内存管理

内存分配、地址映射、内存保护与共享、虚拟内存、

（3）文件管理

文件存储空间管理、目录管理、文件读写和保护

（4）设备管理

完成用户的I/O请求，方便用户使用各种设备，提高设备利用率

包括：缓冲管理，设备分配，设备处理，虚拟设备

**系统调用：**

如果一个进程在用户态需要使用内核态的功能，就进行系统调用从而陷入内核，由操作系统代为完成。头文件都是　#include<unistd.h>



**进程控制函数：进程控制：进程是一个程序执行的过程，是动态的。**

**exec函数：**

exec():用于在进程中启动另一个程序执行的方法，运行新的可执行文件，取代原来的数据段，代码段和堆栈段，一般是在子进程中执行exec；有6个函数

**fork函数：**

fork()/vfork():从一个已经存在的进程中创建一个新的进程，fork函数执行一次返回两个值，子进程返回0，父进程返回子进程的PID，出错返回-1.

#include<sys/types.h>

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

pid\_t result;

result = fork();//创建一个进程

if(result == -1)

{

perror("fork");

exit(1);

} else if(result == 0)

{

printf("The return value is 0,In child process!! My PID is %d\n",getpid());

}

else

{

printf("The return value is %d, In father process !! My PID is %d\n", result, getpid());

}

}

在语句result = fork();之前，只有一个进程在执行这段代码，但是这条语句之后，就变成两个进程执行了，这两个进程几乎完全相同，

下一条执行的语句都是if(result == -1)。

fork函数的神奇就是他调用一次就有两个返回值，父进程执行的时候，返会的是子进程的PID，子进程就是返回0；

**wait函数**:

wait：用于使父进程阻塞，直到一个子进程结束或者该进程接收到一个指定信号位置。如果该进程没有子进程或者他的子进程已经结束，wait就立即返回。

waitpid，不一定要等待第一个终止的子进程，wait只是其一个特例。

**exit函数:** 进程无条件停止。

exit(0):先清理I/O缓冲后调用系统exit

\_exit(0):直接调用exit；

**进程通信函数:**

**pipe()函数：**

管道函数：把两个进程之间的标准输入输出连接起来。从而提供一种让多个进程间通信的方法。

当进程创建管道的时候，每次都需要提供两个文件描述来操作管道，其中一个对管道进行写操作，另外一个对管道进行读操作。对管道的读写与一般的IO系统函数一致。使用write写，read读。

1. 函数说明

pipe（建立管道）：

1) 头文件 #include<unistd.h>

2) 定义函数： int pipe(int filedes[2]);

3) 函数说明： pipe()会建立管道，并将文件描述词由参数filedes数组返回。

filedes[0]为管道里的读取端

filedes[1]则为管道的写入端。

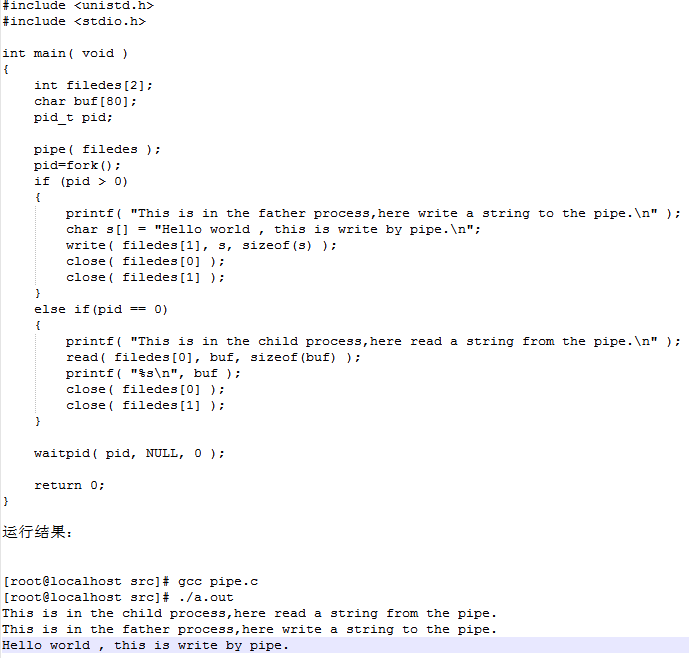
4) 返回值： 若成功则返回零，否则返回-1，错误原因存于errno中。

错误代码:

EMFILE 进程已用完文件描述词最大量

ENFILE 系统已无文件描述词可用。

EFAULT 参数 filedes 数组地址不合法。

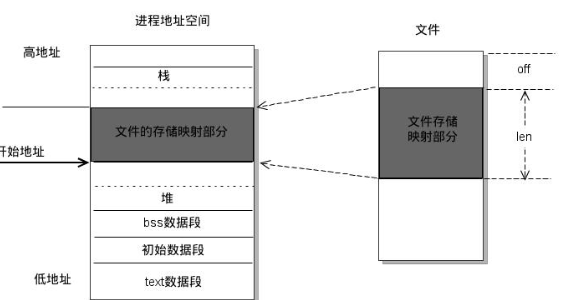


**Shmget函数**：进程间IPC共享内存

**Mmap函数**：

内存映射文件的函数。将一个文件或者对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的对应关系。

实现上述关系之后，进程可以采用指针的方式读写操作这一段内存，而系统会自动回写将页面对应的文件磁盘上，完成对文件的操作而不需要调用read和write函数。相反，内核空间对这段区域修改也直接反映用户空间，实现不同进程间的文件共享。



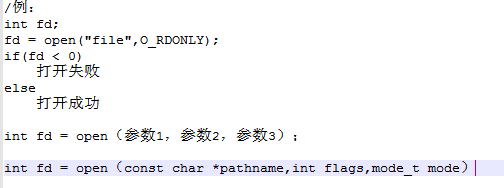
**文件操作函数**：

**Open函数**：

#include <sys/types.h>

#inlcude <sys/stat.h>

#inlcude <fcntl.h>



参数1是打开的文件路径；

参数2是打开方式的选择分为主类和副类

主类：O\_RDONLY 以只读方式打开 / O\_WRONLY 以只写方式打开 /O\_RDWR 以可读可写方式打开

三这是互斥的

副类:

O\_CREAT 如果文件不存在则创建该文件

O\_EXCL 如果使用O\_CREAT选项且文件存在，则返回错误消息

O\_NOCTTY 如果文件为终端，那么终端不可以调用open系统调用的那个进程的控制终端

O\_TRUNC 如果文件已经存在则删除文件中原有数据

O\_APPEND 以追加的方式打开主副可以配合使用，

参数三：如果文件被新建，指定权限为mode。

mode是八进制权限码，0777表示文件所有者，该文件组其他用户具有读写权限。

**Read函数：**

读取文件：磁盘一份，内存一份

ssize\_t count = read(参数1，参数2，参数3)

ssize\_t read(int filedes,void \*buf,size\_t nbytes);

count返回读到的字节数，若是文件末尾返回0，失败返回-1；

头文件：#include <unistd.h>

参数1 (filedes)

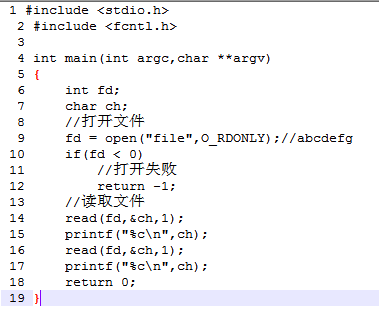
filedes:open函数的成功返回值

参数2 (buf)

内存地址（开辟空间 栈、堆）

参数3 (nbytes)

数据大小（空间大小）



**Write函数：**

将内存中的数据写入文件中

ssize\_t count = write(参数1，参数2，参数3)ssize\_t write(int filed,const void \*buf,size\_t nbytes);

1.count 写入的字节数返回值：返回实际写入的字节数，若出错则返回-1。2.使用write前需要先包含头文件

#include <unistd.h>

3.参数1（filed）open函数的成功返回值

4.参数2（buf）内存地址

5.参数3（nbytes）数据大小

**设备操作：**

**Ioctl函数：**

提供一个用于控制设备及其描述符行为和配置底层服务的接口。

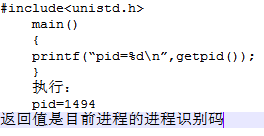
#include <unistd.h>

int ioctl(int fildes,int cmd,...);　　ioctl 对描述符 fildes 引用的对象执行 cmd 参数中给出的操作。

**信息维护：**

**Getpid函数：**

取得进程识别码；头文件：#include<unistd.h>



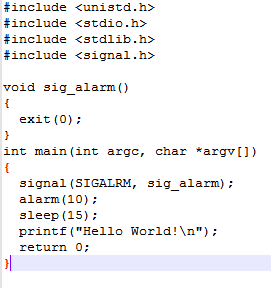
**Alarm函数：**

设置信号传送的闹钟函数；

unsigned int alarm(unsigned int seconds);

设置信号经过参数多少秒发送给目前的进程，如果为设置信号的处理函数，那么alarm默认处理终止进程。

函数的返回值：如果在seconds秒内再次调用了函数设置新的闹钟，则后面的定时器的设置将覆盖前面的设置；当参数为0的时候，之前设置的闹钟取消，将剩下的时间返回。



在文件test1.c中，定义了一个时钟alarm(10)，它的作用是让信号SIGALRM在经过10秒后传送给目前main()所在进程；接着又定义了sleep(15)，它的作用是让执行挂起15秒的时间。所以当main()程序挂起10秒钟时，signal函数调用SIGALRM信号的处理函数sig\_alarm，并且sig\_alarm执行exit(0)使得程序直接退出。因此，printf("Hello World!\n")语句是没有被执行的。

**Sleep函数：**

程序挂起一段时间；在windos中以ms为单位，在linux中以s为单位。

**安全：**

**Chmod函数：**

相关函数：fchmod, stat, open, chown

头文件：#include <sys/types.h> #include <sys/stat.h>

定义函数：int chmod(const char \* path, mode\_t mode);

函数会以来mode权限来更改参数path指定文件权限。

**Umask函数：**

umask设置了用户创建文件的默认 权限，它与chmod的效果刚好相反，umask设置的是权限“补码”，而chmod设置的是文件权限码。一般在/etc/profile、$ [HOME]/.bash\_profile或$[HOME]/.profile中设置umask值。

计算umask的值：

**Chown函数：**

利用 chown 将指定文件的拥有者改为指定的用户或组，用户可以是用户名或者用户ID；组可以是组名或者组ID；文件是以空格分开的要改变权限的文件列表，支持通配符。

**内核和微小内核**：

大内核：

将操作系统功能作为一个紧密结合的整体放倒内核中，

由于各个模块之间共享消息，具有很高的性能

微内核：

将一部分操作系统功能移出内核，降低内核的复杂性。移出的部分根据分层原则划分成若干个服务，相互独立。

操作系统被划分成小的，定义良好的模块，只有微内核这一个模块运行在内核态，其余模块运行在用户态。

需要频繁的在内核态和用户态之间切换，性能具有损失。

**中断分类：**

（1）外部中断

有CPU执行指令之外的事件引起，如I/O中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。还包括时间中断，控制台中断

（2）异常

CPU执行指令内部事件引起，非法操作码，地址越界，算术溢出

（3）陷入

用户程序中使用系统调用。

**Linux进程的基本概念：**

进程是一个程序一次执行的过程，程序是静态的，进程是动态的，包括创建，调试和销往的整个过程。

（1）进程控制块

进程是LINUX系统基本调度单位，系统通过进程控制块描述并表示其变化

进程控制块包含：进程的描述信息，控制信息及资源信息，是进程的一个静态描述。

静态控制块没一项都是一个task\_struct结构，在include/linux/sched.h头文件中定义

（2）进程的标识：

进程号(PID)和父进程号(PPID):在linux中可以通过系统调用函数getpid和getppid得到。

（3）进程运行的状态：执行态-就绪态-阻塞态

**进程结构：**

（1）包含三个段：数据段，代码段，堆栈段

数据段：存放全局变量，常数，及动态分配的数据空间

代码段：程序代码

堆栈段：子程序的返回地址，子程序的参数及程序的局部变量

进程的执行模式分为：用户模式和内核模式

如果用户程序执行过程中出现了系统调用或者是发生中断事件，就要运行操作系统程序，变成内核模式

**进程管理：**

1、进程启动和调度进程

（1）启动进程

//手工启动：由用户输入命令直接启动进程

1）前台启动：用户键入一个命令例如"ls-l"， 就已经启动了一个进程，并且是一个前台进程。

2）后台进程：在进程非常耗时的时候，且用户不着急需要结果的时候启动的，比如格式化文本文件的进程

（2）调度启动：费时且占用资源的维护工作，在深夜无人值守的时候进行，用户可以实现安排调度工作，指定任务运行的时间和场合。使用调度启动进程有几个常用的命令，如at命令在指定时刻执行相关进程，cron命令可以自动周期性的执行相关进程

2、调度进程

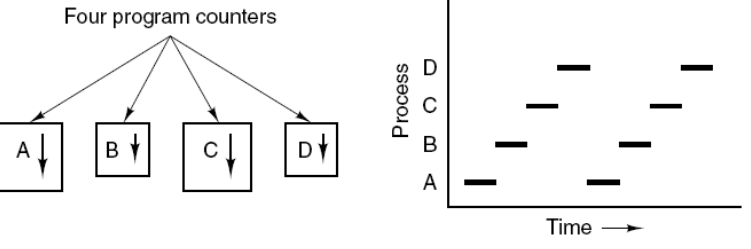
包括对进程的中断操作，改变优先级，查看进程状态。ps|top|nice|renice|kill|crontab|bg

**进程管理-CPU资源分配**:

进程是资源分配的基本单位

进程控制块（PCB）描述进程的基本信息和运行状态，所谓的创建进程和撤销进程，都是指对PCB的操作。

下面显示4个程序创建了4个进程，4个进程可以并发执行



**进程的切换:**

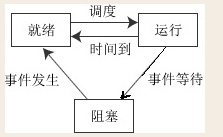
进程的三个状态：

（1）就绪状态：等待被调度；

（2）运行状态：

（3）阻塞状态：等待资源；

只有就绪状态和运行状态可以相互切换，其他的都是单向转换。就绪状态的进程通过调度算法获得CPU时间，转为运行状态；运行状态的进程，在分配给他的CPU时间用完之后就转为就绪状态，等待下一次调度；阻塞状态是缺少需要的资源从而由运行状态转换而来，但是该资源不包括CPU时间，缺少CPU时间会从运行状态转为就绪状态。



（1）就绪状态 -> 运行状态：处于就绪状态的进程被调度后，获得处理机资源（分派处理机时间片），于是进程由就绪状态转换为运行状态。

（2）运行状态 -> 就绪状态：处于运行状态的进程在时间片用完后，不得不让出处理机，从而进程由运行状态转换为就绪状态。此外，在可剥夺的操作系统中，当有更高优先级的进程就绪时，调度程度将正执行的进程转换为就绪状态，让更高优先级的进程执行。

（3）运行状态 -> 阻塞状态：当进程请求某一资源（如外设）的使用和分配或等待某一事件的发生（如I/O操作的完成）时，它就从运行状态转换为阻塞状态。进程以系统调用的形式请求操作系统提供服务，这是一种特殊的、由运行用户态程序调用操作系统内核过程的形式。

（4）阻塞状态 -> 就绪状态：当进程等待的事件到来时，如I/O操作结束或中断结束时，中断处理程序必须把相应进程的状态由阻塞状态转换为就绪状态。

**进程调度算法**:

（3）最高响应比优先法（HRRN）

对FCFS,SJF的一种综合平衡。只考虑每个作业的等待时间而未考虑执行时间的长短，FJF只考虑执行的时间未考虑等待时间的长短。

原理：响应比R=(W+T)/T=1+W/T

其中T是该作业的预估时间，W为作业在后备状态对列中等待时间。每当要进行作业调度的时候，系统计算每个作业的调度比，选择R最大的先执行

优点：由于长作业也有机会执行，在同一时间内处理的作业数显然要少于SJF法，吞吐量小于SJF。

缺点：由于调度前要计算响应比，所以系统开销较大.

**批处理系统:没有太多的用户操作，调度算法的目标是保证吞吐量和周转时间（提交到终止的时间）**  
（1）先来先服务(FCFS)first come first served

非抢占式的FCFS，进程按照请求CPU的顺序使用CPU，排队，先来的先执行，执行过程中也不能被中断。中途退出又得到队尾排队

优点：易于理解简单，只需要FIFO对列，相当公平

缺点：有利于长进程，不利于短进程，有利于CPU繁忙的进程，不利于I/O繁忙的进程

（2）最短作业优先：（SJF）短进程优先

目标是减少平均周转时间。

原理：对预计时间短的进程优先分配处理机。通常后来的短进程不抢占先正在执行进程。

优点：相比于FCFS，平均周转时间和平均带权周转时间，缩短进程的等待时间，提高系统的吞吐量

缺点：对长进程不利，可能长时间得不到执行，且未根依据进程紧迫性来划分优先级，很难准确估计执行时间，影响调度性能。

（3）最短剩余时间优先（SRTN）

最短作业优先的抢占式，按照剩余运行时间的顺序进行调度，当一个新的作业到达的时候，其整个运行时间与当前进程的剩余时间进行比较。如果新进程的时间需要小，则挂起当前进程，运行新的进程，否则新的进程进入等待。

**交互式系统**: 交互式系统具有大量的用户交互操作，以快速响应为目的

（1）时间轮转算法（RR）

采用剥夺策略，最简单公平的方法。每个进程分配一定的时间片。

原理：让就绪进程与FCFS的方式按照时间片轮流使用CPU调度。在一个时间片结束时，发生时间中断，调度程序据此暂停当前进程的执行，将其送到就绪对列的队尾，并且通过上下文切换执行当前对列的队首进程，进程可以为使用完一个时间片，就出让CPU（阻塞）。

优点：简单易行，平均响应时间短

缺点：不利于紧急作业，时间片的大小对系统性能影响较大

时间片的确定：

1）系统对响应时间的要求

2）就绪对列中进程的数目

3）系统的处理能力

（2）多级反馈对列（MFQ）

CPU处理机调度算法。

1）进程在进入待调度的对列等待时，首先进入优先级最高的Q1等待

2）首先调度优先级高的对列中的进程，若高优先级中对列中已没有调度的进程，则调度次优先级的进程。

3）对于同一个优先级的进程，按照时间片轮转的调度。

4）在低优先级对列中的进程运行时，又有新到达的作业，那么在运行完这个时间片后，CPU马上分配新到大的作业（抢占式）.

**实时性系统:**

要求在一个确定的时间内得到响应

（1）硬实时：必须满足绝对的截止时间

（2）软实时：允许一定的超时

**进程同步:**

**临界区:**

临界区（被一个以上的进程或者线程公用的数据）

对临界资源进行访问的那段代码成为临界区，临界区资源一次只允许一个进程使用共享资源。

如果有若干进程要求进入空闲的临界区，一次仅允许一个进程进入。任何时候，处于临界区内的进程不可多于一个。如已有进程进入自己的临界区，则其它所有试图进入临界区的进程必须等待。进入临界区的进程要在有限时间内退出，以便其它进程能及时进入自己的临界区。如果进程不能进入自己的临界区，则应让出CPU，避免进程出现“忙等”现象。

**同步与互斥:**

同步：多个进程因为合作产生的直接制约关系，使得进程有一定的先后执行关系。

互斥：多个进程在同一个时刻只有一个进程能够进入临界区。

**信号量:**

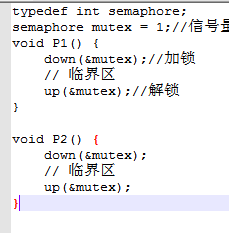
信号量：是一个整形的变量，对其执行down和up操作，也就是常见的P和V操作。

down：如果信号量大于0，执行-1操作，如果信号量等于0，进程睡眠，等待信号量大于0；

up：对信号量执行+1操作，唤醒睡眠的进程让其完成down操作。

down和up操作需要被设计成原语（执行过程过程不允许被中断，连续执行），不可分割，通常是在执行这些操作的时候屏蔽中断。

如果信号量只能取0或1，那么就成为了互斥量（Mutex），0表示临界区已经加锁，1表示临界区解锁。

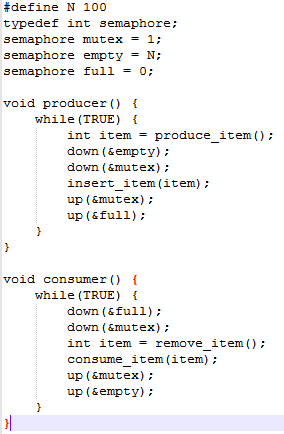


**使用信号量解决生产者-消费者问题**

问题：使用一个缓冲区来保存物品，只要缓冲区未满，生产者才能放入物品；只有缓冲区不为空，才能取走物品。

因为缓冲区属于临界资源，因此需要一个互斥量来控制对缓冲区的互斥访问。为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：empty 记录空缓冲区的数量，full 记录满缓冲区的数量。其中，empty 信号量是在生产者进程中使用，当 empty 不为 0 时，生产者才可以放入物品；full 信号量是在消费者进程中使用，当 full 信号量不为 0 时，消费者才可以取走物品。

注意，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 down(mutex) 再执行 down(empty)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 down(empty) 操作，发现 empty = 0，此时生产者睡眠。消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 up(empty) 操作，empty 永远都为 0，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。



**管程:**

使用信号量机制实现生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用容易。

C语言不支持管程。

特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用管程，否则其他进程将永远无法使用管程。

管程引入条件变量以及相关操作：wait和signal实现同步操作。对条件变量执行wait操作会导致调用进程阻塞，把管程让给另一个进程持有。signal操作用于唤醒被阻塞的进程。

也可以使用管程解决生产者-消费者问题。

**经典同步问题:**

**哲学家进餐;**

五个哲学家围着一张圆桌，每个哲学家面前放着食物。哲学家的生活有两种交替活动：吃饭以及思考。当一个哲学家吃饭时，需要先拿起自己左右两边的两根筷子，并且一次只能拿起一根筷子。

如果所有的都同时拿起左边的筷子，然后等待其他哲学家放下筷子，那么就不会有人放下，因为没有人拿到了两根筷子，就形成了死锁。

为了防止死锁：

（1）必须同时拿起左右两根筷子

（2）只有在两个邻居没有进餐的情况下才允许进餐

**读者-写者问题:**

允许多个进程同时对数据进行读操作，但是不允许读和写以及写与写操作同时发生。

一个整形变量count记录在对数据进行读操作的进程数量，一个互斥量count\_mutex对count进行加锁，一个互斥量data\_mutex用于对读写的数据加锁。

**进程通信:**

**管道:**

（1）管道通过调用pipe函数创建，fd[0]用于读,fd[1]用于写

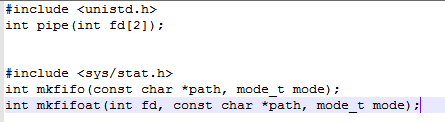
限制：

1）只支持半双工通信

2）只能在父子进程或者兄弟进程中使用

一个父进程里面两次fork两个子进程，就是兄弟进程。

（2）FIFO 命名管道:去除了管道只能在父子进程中使用的限制. FIFO常用于客户-服务器应用程序中，FIFO用做汇聚点，在客户端和服务器进程之间的传递数据



**消息对列:**

优点：

（1）可以独立于读写进程的存在，从而避免了FIFO中同步管道的打开或者关闭时可能产生的困难；

（2）避免了FIFO的同步阻塞的问题，不需要进程自己提供同步方法

（3）读进程可以根据消息的类型有选择的接受消息，而不像FIFO那样只能默认的接受。

**信号量:**

只是一个计数器，用于多个进程提供对共享数据对象的访问

**共享存储**:

允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以是最快的一种IPC。

需要使用信号量用来同步对共享内存的访问

多个进程可以将同一个文件映射到他们的地址空间实现共享内存。XSI共享不是使用文件，是使用内存的匿名段。

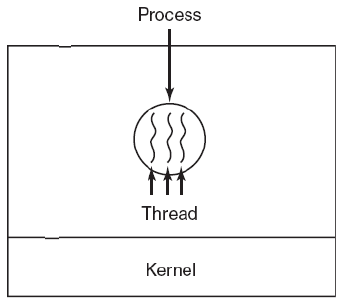
**套接字:** **可以用于不同机器间的进程通信。**

**线程:**

线程是CPU调度的基本单位

一个进程可以有多个线程，他们共享进程资源

qq和浏览器是两个进程，浏览器里面有很多线程，例如http的请求，时间响应线程等，线程的并发执行使得在浏览器中点击一个新连接从而发起http请求，浏览器还可以响应用户其他事件。



**进程和线程的区别:**

（1）资源：

进程是资源分配的基本单位，线程不拥有资源，线程可以访问隶属进程的资源

（2）调度：

线程是调度独立的基本单元，在同一个进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程的切换。

（3）系统开销

由于创建或者撤销进程的时候，系统都要位置分配和回收资源，如内存空间,I/O设备等，所付出的开销远大于创建或者撤销线程时候的开销。进程切换时，设计当前进程CPU环境的保存以及新调度进程CPU的环境设置，而线程切换时只需要保存和设置少量寄存器内容，开销很小

（4）通信方面

线程间可以通过直接读写通一进程中的数据进行同行，但是进程通信需要借助IPC。

**死锁:**

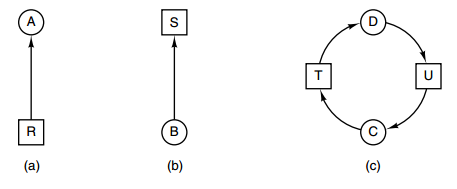
**必要条件:**

（1）互斥条件：每个资源要么已经分配给一个进程，要么就可用的

（2）请求与等待条件：已经得到某个资源的进程可以再请求新的资源

（3）不可剥夺：已经分配给一个进程的资源不能被强制性的抢占。

（4）环路等待：有两个或者两个以上的进程组成一个环路，该环路中的每个进程都在等待下一个进程所占有的资源



**解决办法:**

（1）鸵鸟策略

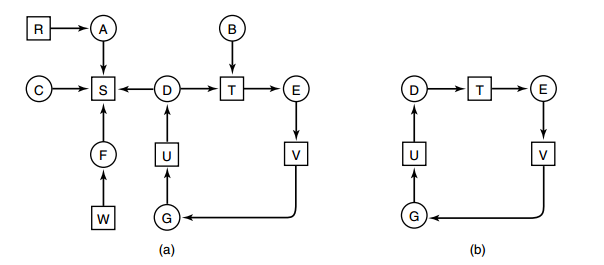
假装什么都没发生

因为解决死锁的代价很高，所以采用这种措施会获得更高的性能，当发生死锁不会对用户造成多大影响的时候，或者发生死锁的概率低的时候，采用鸵鸟策略。

（2）死锁检测与死锁恢复

不试图阻止死锁，而是当死锁发生的时候，采取措施进行恢复

1）每种类型一个资源的死锁检测



上图为资源分配图，方框代表资源，圆圈代表进程。资源指向进程，表示资源已经分配给了进程，进程指向资源表示进程请求资源。

图b满足了环路等待条件会形成死锁；通过检测有向图是否存在环来实现死锁检测，从一个节点触发进行深度优先搜索，对访问的节点进行标记，如果访问过已经标记的节点，就表示有环。

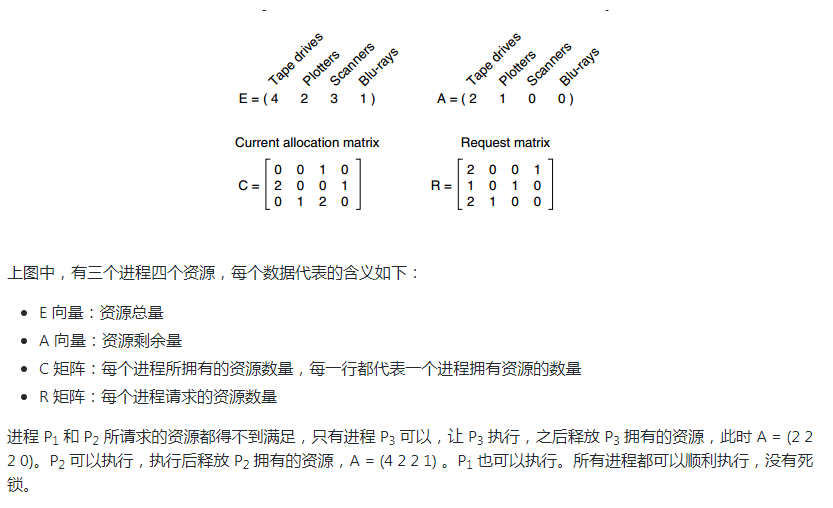
2）每种类型多个资源的死锁检测

每个进程最开始的时候不被标记，执行过程有可能被标记，当算法结束的时候，任何没有被标记的进程都是死锁进程。

寻找一个没有被标记的进程，他请求的资源小于等于A；

如果找到了一个这样的进程,那么将C矩阵的第I行向量加入A中，标记该进程，并转回上一步；

如果没有这样一个进程，算法终止



3）死锁恢复

利用抢占式恢复；

利用回滚恢复

通过杀死进程恢复

**死锁预防：**

在进程运行之前预防

（1）破坏互斥条件：例如脱机打印机被多个进程同时输出，唯一中请求物理打印机的进程是打印机的守护进程

（2）破坏请求保持条件：规定所有进程在开始执行前请求所需要的全部资源

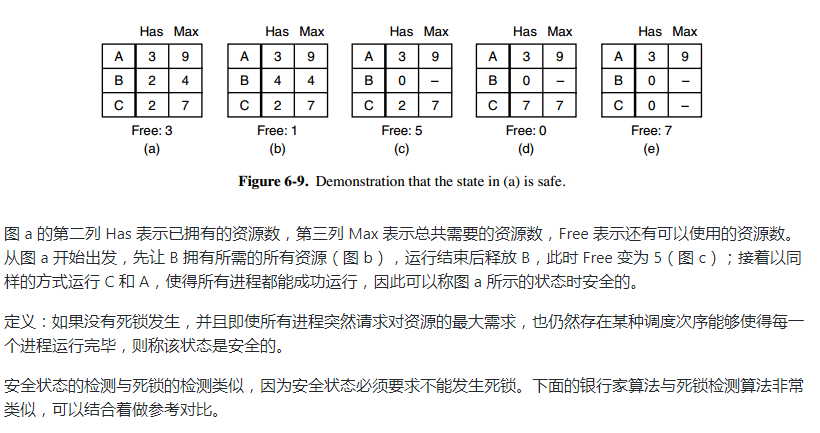
（3）破坏不可剥夺条件

（4）破坏环路等待条件：资源有序分配

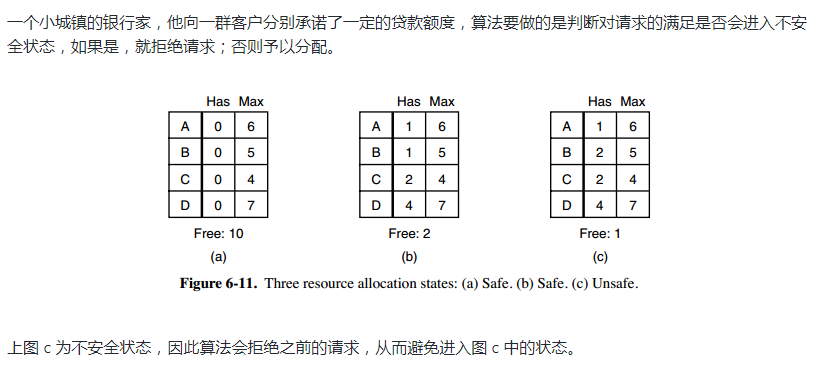
**死锁避免：**

在程序运行时避免

（1）安全状态



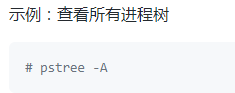
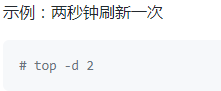
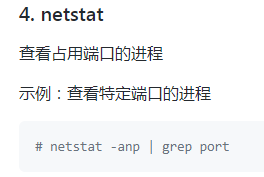
（2）单个资源银行家算法



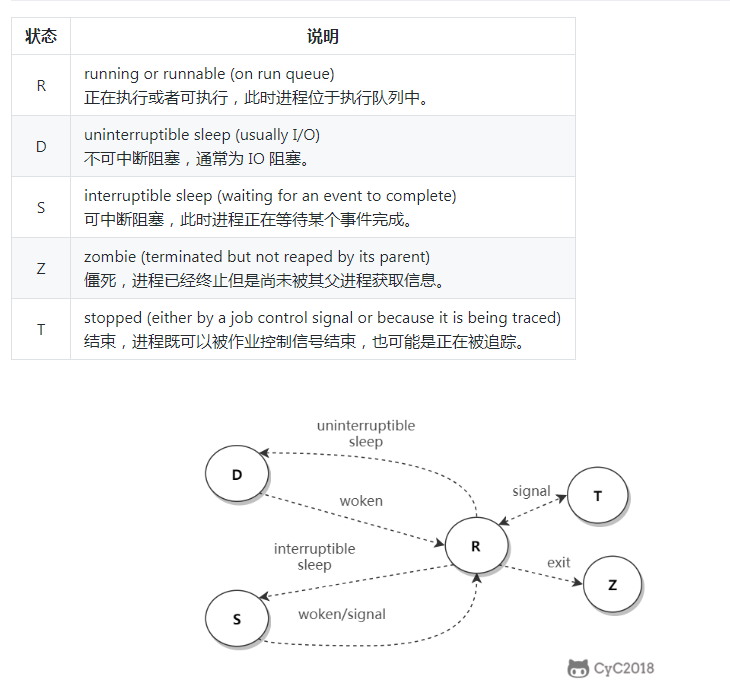
（3）多个资源银行家算法



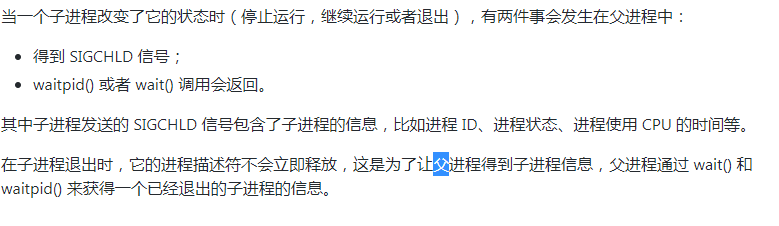
**Linux：**

(1)用PS查看某个时间点的进程  
  
  
(2)pstree:查看进程树  
  
  
(3)top:实时显示进程消息  
  
  
(4)netstat:查看占用端口和进程  


**进程运行状态**：



**SIGCHLD：**



**WAIT：**



**孤儿进程：**

一个父进程退出,而他的一个或者多个子进程还在运行.

孤儿进程将被init进程(进程号为1)所收养,并由init进程对他们完成状态手机工作

由于孤儿进程被init进程收养,孤儿进程不会对系统造成危害

**僵尸进程：**

一个子进程的进程描述符在子进程退出时不会释放,只有当父进程通过wait或者waitpid获取了子进程信息后才会释放.如果子进程退出,而父进程并没有调用wait和waitpid,那么子进程的进程描述符仍然保存在系统中.就是僵尸进程

僵尸进程通过ps查看显示状态为Z;

系统所能使用的进程号是有限的,如果产生了大量的僵尸进程,将因为没有可用进程号导致系统不能产生新的进程.

要消灭僵尸进程,只需要将其父进程杀死,此时僵尸进程就变成了孤儿进程,然后被init收养,init进程就会释放僵尸进城的资源,结束僵尸进程.

**内存管理：**

**虚拟内存**：

目的是：为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存

为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。这些页被映射到物理内存中，但是不需要映射到连续的物理内存，也不需要所有的页都必须在物理内存中。当程序引用到不在物理内存中的页时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令。

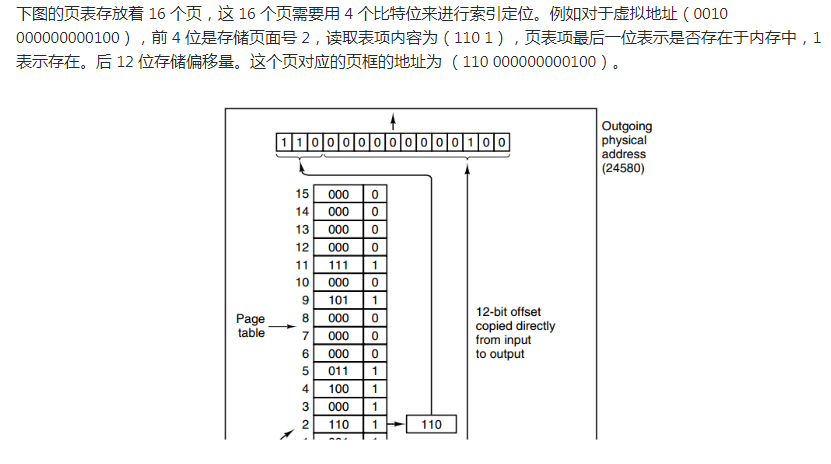
虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。

例如一台计算机可以产生16位的地址，那么一个程序的地址空间范围就是0~64k。该计算机只有32KB的物理内存，虚拟内存技术允许该计算机运行一个64k大小的内存。

**分页系统地址映射：**

内存管理单元(MMU)管理着地址空间和物理内存的转换，其中的页表存储着页（程序地址空间）和页框（物理内存空间）的映射表。

一个虚拟地址分为两个部分，一部分存储页面号，一部分存储偏移量。



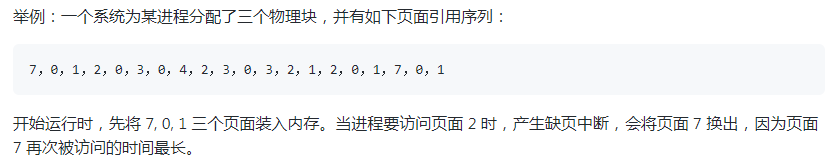
**页面置换算法：**

在程序运行过程中，如果访问的页面不在内存中，就发生缺页中断从而将该页调入内存中。此时如果内存已经没有空闲的空间，系统必须从内存中调出一个页面到磁盘对换区中来腾出空间。

页面置换算法和缓存淘汰策略类似，可以将内存看成磁盘的缓存。在缓存系统中，缓存的大小有限，当有新的缓存到达的时候，需要淘汰一部分已经存在的缓存。

页面置换算法的主要目的是使页面置换频率最低（缺页率最低）

（1）最佳OPT

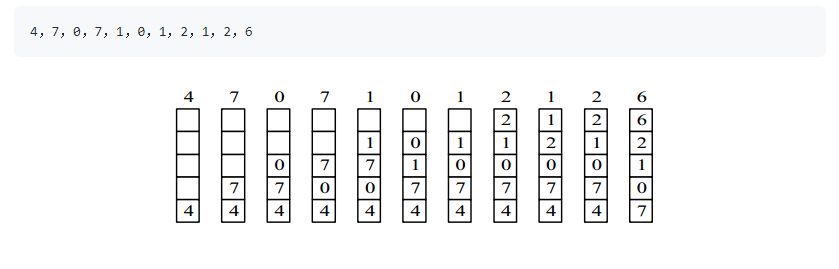
所选择的被换出的页面是最长时间不再被访问，通常可以保证获得最低的缺页率。理论方法，无法知道一个页面多长时间不再被访问。

（2）最近最久未使用LRU

LRU机制就是将最近最久未使用的页面换出，

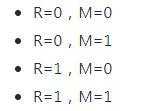
为了实现LRU，需要在内存中维护一个所有页面的链表，当一个页面被访问的时候，将这个页面移到链表的表头。这样就能保证链表表尾的页面是最久未被访问的，

每次访问都需要更新链表，LUR代价很高



（3）最近未使用NRU

每个页面都有两个状态位：R和M，当页面被访问时设置页面的R为1，当页面被修改时设置M=1,。其中R位会定时被清零。将页面分成四类



当缺页发生的时候，NRU机制随机从类编号最小的非空类中挑选一个页面将他换出来。

NRU优先换出了已经被修改的脏页面(R=0，M=1)，而不是被频繁使用的干净页面(R=1,M=1)。

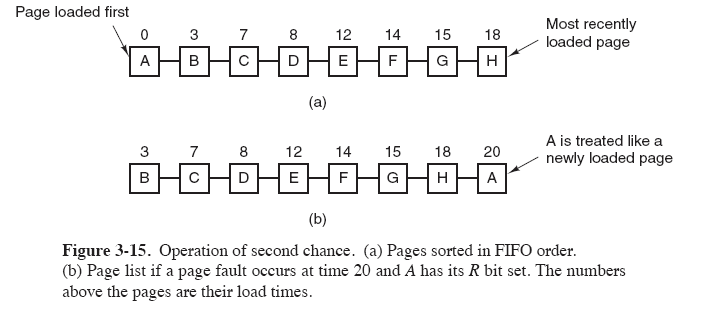
（4）先进先出FIFO

换出的页面是最先进入的页面，会将经常访问的页面换出，导致缺页率升高

（5）第二次机会算法

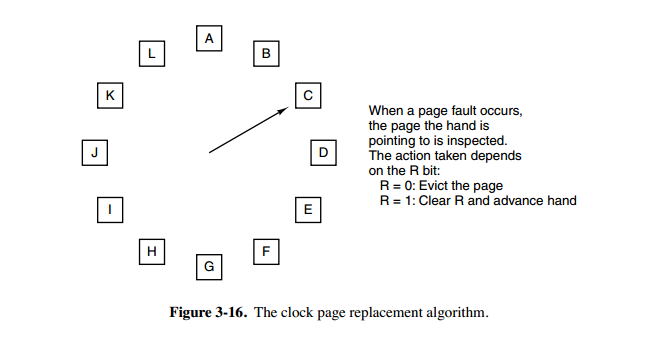
针对FIFO可能把经常使用的页面置换出去，对FIFO 进行简单的修改

当页面被访问时设置该页面的R=1。需要替换的时候，检查最老压面的R位。如果R是0，那么这个页面又老又没有被使用，可以立即置换掉。如果是1，就将R清0，并把该页面放到链表的尾端，修改它的装入时间使他像刚装入的一样，然后继续从链表头部开始搜索。



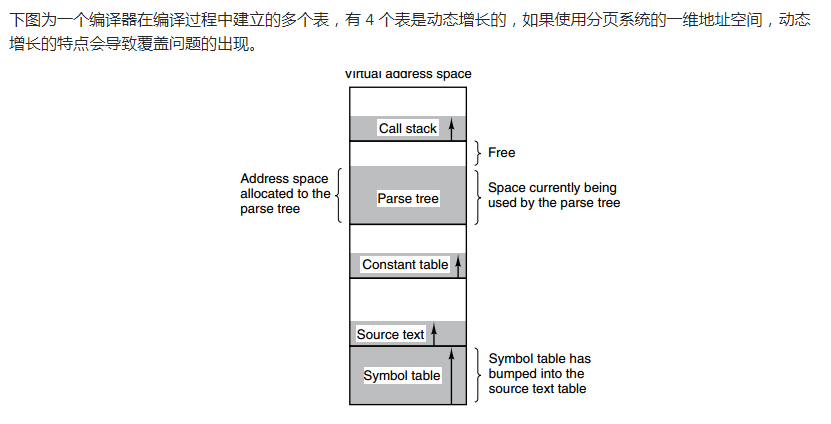
（6）时钟

第二次机会算法需要在链表中移动页面，降低了效率。时钟算法使用环型链表将页面链接起来，再使用一个指针指向最老的页面。

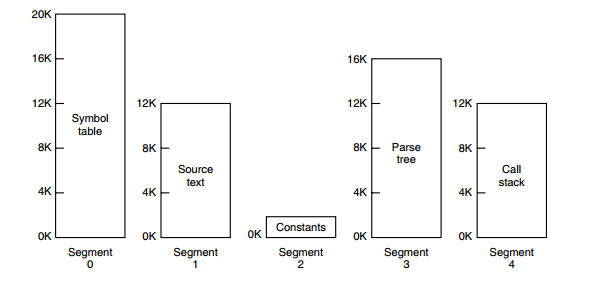


**分段：**

虚拟内存采用的是分页技术，就是将地址空间划分成固定大小的页，每一页与内存进行映射。



分段就是把表分成段，一个段构成一个独立的地址空间，每个段的长度也不同，并且可以动态增长。



**段页式：**

程序的地址空间划分成多个拥有独立地址空间的段，每个段上的地址空间划分成大小相同的页。这样既拥有分段系统的共享和保护，又拥有分页系统的虚拟内存功能。

**分页和分段比较**：

（1）对程序员的透明性：分页透明，但是分段需要程序员显示划分每个段

（2）地址空间的维度：分页是一维地址空间，分段是二维

（3）大小是否可以改变：页的大小是不可变的，段的大小是可以动态改变的。

（4）出现的原因：分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使得程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护。

**文件管理(磁盘管理)：**

**磁盘结构**：

盘面：一个磁盘有多个盘面

磁道：盘面上的圆形带状区域，一个盘可以有多个磁道

扇区：磁道上的一个弧段，一个磁道可以有多个扇区，他是最小的物理存储单位，目前主要有512bytes与4k两种大小。

磁头：与盘面非常接近，能够将盘面上的磁场转换成电信号（读），或者将电信号转换成磁场（写）；

制动手臂：用于在磁道之间移动磁头

主轴：使整个盘面转动。

**磁盘调度算法:**

读写一个磁盘块的时间影响因素：

（1）旋转时间（主轴转动盘面，使得磁头移动到合适的扇区）

（2）寻道时间（制动手臂移动，使得磁头移动到适当的磁道上）

（3）实际的数据传输时间

其中，寻道时间最长，因此磁盘调度的主要目的是使得磁盘的平均寻道时间最短。

算法：

（1）先来先服务FCFS

按照磁盘请求的顺序进行调度：

有点是公平简单。缺点是，没有对寻道进行优化，使得平均寻道时间可能较长。

（2）最短寻道时间优先SSTF

优先调度当前磁头所在磁道距离最近的磁道

虽然平均寻道时间比较低，但是不公平。如果新到达的磁道请求总是比一个等待的磁道请求近，那么等待的磁道请求就会一直等待，出现饥饿线现象。两端的磁道请求更容易出现饥饿现象。

（3）电梯算法SCAN

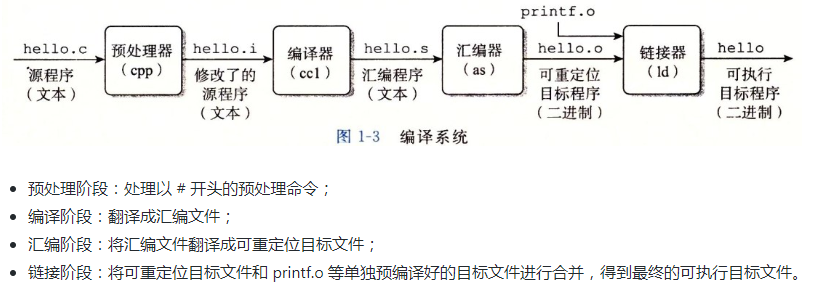
电梯总是保持一个方向运行，直到该方向没有请求位置，然后改变方向。

电梯算法（扫描算法）和电梯的运行过程类似，总是按照一个方向来进行磁盘调度，直到该方向上没有未完成的磁道请求，然后改变方向。

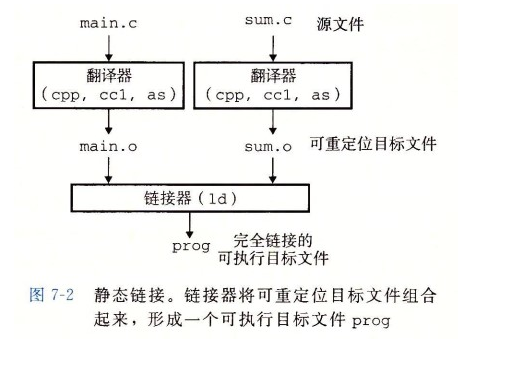
因为考虑了移动方向，因此所有的磁道请求都会被满足，解决了SSTF的饥饿问题

**链接**:

**编译系统:** **编译器把源文件变成目标文件的过程**



**静态链接:**



静态链接器以一组可重定位目标文件为输入，生成一个完全链接的可执行目标文件作为输出。链接器主要完成以下两个任务：

（1）符号解析：每个符号对应一个函数、一个全局变量、一个静态变量，符号解析的目的是将每个符号引用与一个符号定义关联起来

（2）重定位：链接器 通过每个符号定义与一个内存位置关联起来，然后修改所有对这些符号的引用，使得他们指向这个内存位置

**目标文件:**

（1）可执行目标文件：可以直接在内存中执行

（2）可重定位目标文件：可与其他可重定位目标文件链接阶段合并，创建一个可以执行的目标文件

（3）共享目标文件：特殊的可重定位目标文件，可以在运行的时候被动态的加载进内存并链接

**动态链接:**

静态库的两个问题：

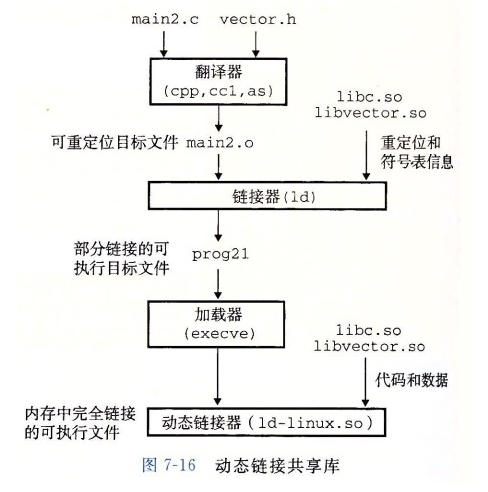
（1）当静态库更新整个程序都需要重新链接

（2）对于printf这种标准的函数库，如果每个程序都有代码，会造成极大的浪费

共享库就是为了解决静态库这两个问题而设计的，在linux中用.SO后缀表示，windos中成为DLL文件，具有以下特点：

（1）在给定的文件系统库中只有一个文件，所有引用该库的可执行目标文件都共享这个文件，他不会被复制到引用它的可执行文件中

（2）在内存中，一个共享库.text节（已经编译程序的机器代码）的一个副本可以被不同的正在运行的进程共享。



**select函数:**

select函数在使用前，先将需要监控的描述符对应的bit位置1，然后将其传给selsct，当有任何一个事件发生，select将会返回多有的描述符，需要在应用程序中自己去遍历去检查那个描述符上有事件发生，效率很低，并且其不断在内核态和用户态进行描述符的拷贝，开销很大.

**fork函数:**

创建一个当前进程映像一样的进程可以通过fork（）系统调用

成功调用fork函数之后，会创建一个新的进程，其几乎与调用fork的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork调用会返回0；在父进程中fork()返回子进程的pid。如果出现错误返回一个负值。

最常见的fork用法是创建一个新的进程，然后用exec()载入二进制映像，替换当前进程的映像。这种情况下，派生了新的进程，而这个子进程会执行一个新的二进制可执行的二进制文件的映像。

早期，当调用fork函数的时候，内核会把所有的数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页复制到子进程的地址空间中。从内核的角度讲，逐页复制方式十分耗时。现代的都是采用写时复制的方法，而不是对父进程空间整体复制。

**fork函数与vfork函数:**

fork:创建一个和当前进程映像一样的进程可以通过fork( )系统调用

fork和vfork的区别：

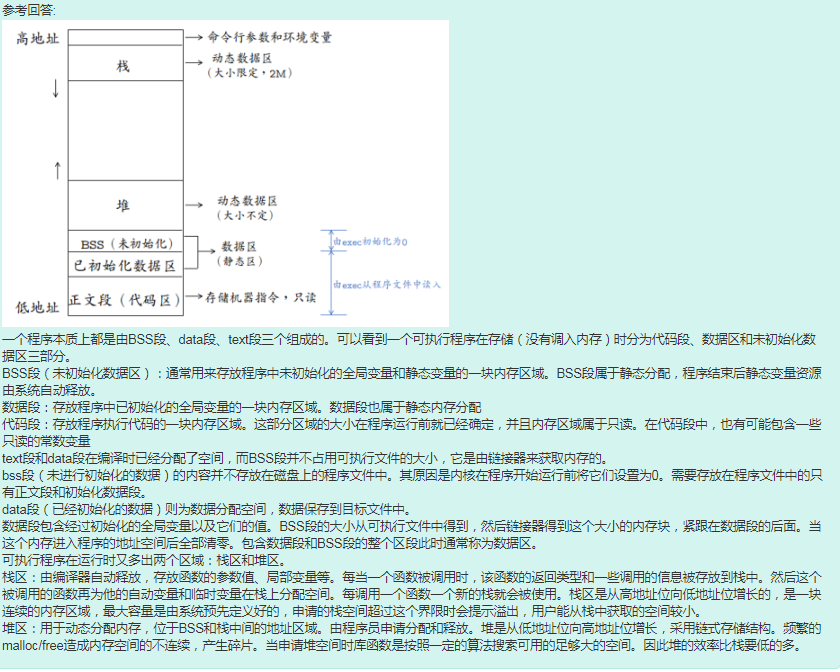
1. fork( )的子进程拷贝父进程的数据段和代码段；vfork( )的子进程与父进程共享数据段

2. fork( )的父子进程的执行次序不确定；vfork( )保证子进程先运行，在调用exec或exit之前与父进程数据是共享的，在它调用exec或exit之后父进程才可能被调度运行。

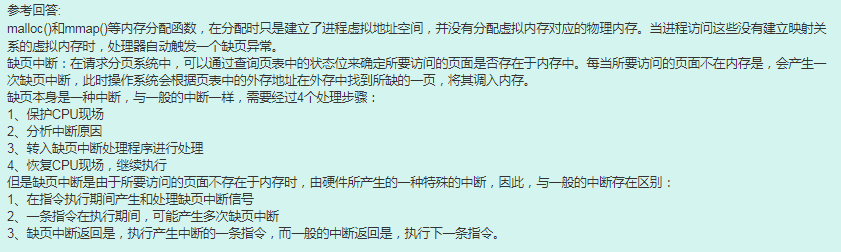
3. vfork( )保证子进程先运行，在它调用exec或exit之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

4.当需要改变共享数据段中变量的值，则拷贝父进程。

**操作系统的内存结构:**



**操作系统的缺页中断:**



**如何修改文件最大句柄数:**

linux默认最大文件句柄数是1024个，在linux服务器文件并发量比较大的情况下，系统会报"too many open files"的错误。故在linux服务器高并发调优时，往往需要预先调优Linux参数，修改Linux最大文件句柄数。有两种方法：

1. ulimit -n <可以同时打开的文件数>，将当前进程的最大句柄数修改为指定的参数（注：该方法只针对当前进程有效，重新打开一个shell或者重新开启一个进程，参数还是之前的值）

2. 对所有进程都有效的方法，修改Linux系统参数

vi /etc/security/limits.conf 添加

\* soft nofile 65536

\* hard nofile 65536

将最大句柄数改为65536

修改以后保存，注销当前用户，重新登录，修改后的参数就生效了.

**怎么实现线程池:**

（1）设置一个生产者消费对列，作为临界资源

（2）初始化n个线程，并让其运行起来，加锁去列取任务运行

（3）当任务对列为空的时候，所有线程阻塞

（4）当生产者对列来一个任务后，先对对列加锁，把任务挂到对列上，然后使用条件变量去通知阻塞中的一个线程。

**两个进程访问临界资源，会不会出现都获得自旋锁的情况:**

单核CPU，并且开了抢占可以造成这种情况.

**操作系统为甚要分内核态和用户态:**

为了安全

在CPU有些指令中，如果指令出错，将导致整个系统崩溃。

分了之后，用户需要操作这些指令，内核为其提供API，可以通过系统调用内核去执行。

用户态和内核态是操作系统的两种运行级别，两者最大的区别就是特权级不同。用户态拥有最低的特权级，内核态拥有较高的特权级。运行在用户态的程序不能直接访问操作系统内核数据结构和程序。内核态和用户态之间的转换方式主要包括：系统调用，异常和中断。

**有进程为什么还需要有线程:**

线程产生的原因：进程可以使得多个程序并发执行，提高资源的利用率和吞吐量，但是其具有一些缺点

进程在某个时刻只能干一件事，进程执行时如果遇到阻塞，整个进程就会挂起；因此引入比进程更小的线程，作为并发执行的基本单位，从而减少程序在并发时所付出的时空开销，提高并发性。

**死锁发生的条件及解决方式:**

死锁是指两个或者两个以上进程在执行的过程中，因为争夺资源造成的互相等待的现象。死锁发生有四个必要条件

解决办法：破坏四个条件之一

（1）资源一次性分配，从而剥夺请求和保持条件

（2）可剥夺资源，即进程新的资源未得到满足时，释放已经占有的资源，从而破坏不可剥夺的条件

（3）资源有序分配法：系统给每个资源赋予 一个序号，每个进程按照编号递增的请求资源，释放相反，从而破坏环路等待条件

**多线程同步锁机制:**

同步的时候使用一个互斥量，在访问公共资源前对互斥量进行加锁；在访问完后释放互斥量的锁；

对互斥量进行加锁之后，任何其他试图再次对互斥量加锁的线程将被阻塞直到当前线程释放互斥锁。如果互斥锁是有多个线程阻塞都会变成可运行状态，第一个变的线程可以对互斥锁加锁，其他线程将看到的是互斥锁被锁住，智能继续等待。每次只有一个线程可以向前执行。

**进程的状态切换:**

进程有五种状态

（1）创建状态：进程正在被创建

（2）就绪状态：进程被加入到就绪对列中等待CPU调度

（3）执行状态：进程正在被运行

（4）等待阻塞状态：因为某种原因，等待设备而不能运行

（5）终止状态：进程运行完毕

**就绪状态下的进程在等待什么:**

被调度使用CPU的运行权.

**怎么唤醒被阻塞的socket**: 给阻塞时候缺少的资源

**大小端及怎么判断**:

大端指的是低字节存储在高地址，小端是低字节存储在低字节；

根据联合体判断，因为联合体总是从低地址存储。

**怎么确定当前线程是繁忙还是阻塞:**

用PS命令查看

**多进程和多线程:**

进程是资源分配的最小单位，线程是cpu调度的最小单位。

多线程之间共享一个进程的地址空间，线程通信简单，同步复杂，线程创建，销毁和切换简单，速度快，占用内存少，适用于多核分布式系统。但是线程之间会相互影响，一个线程一位终止会导致同一个进程的其他线程也终止，程序可靠性弱。

多进程间拥有各自独立的运行地址空间，进程间不会相互影响，程序可靠性强，但是进程创建，销毁和切换复杂，占用内存多，进程间通信复杂，但是同步简单，适用于多核，多级分布

**多进程和多线程的使用场景:**

多进程模型的优势是CPU，适用于CPU密集型

多线程模型的主要优势为线程切换代价小，使用I/O密集型的工作场景，因为经常会频繁切换线程。多线程使用与单级多核分布场景。易于多机拓展

**进程间的通信:**

进程间的通信：

管道：无名管道和命名管道，管道可以用于具有亲缘关系的父子进程间的通信，有名管道处理具有管道的基本功能，还允许无亲缘关系进程间的通信

//普通管道PIPE：半双工；只能用于具有亲缘关系的进程之间的通信；

//命名管道FIFO：可以无关的进程之间交换数据；FIFO有路径名与之关联，以一种特殊设备文件形式存在于文件系统中

//消息对列：消息的链接表，存放在内核中，一个消息对列由一个ID标识，克服了信号传递信息少等特点，具有写权限的进程可以按照一定的规则向消息对列中添加新消息；对消息对列有读权限的进程则可以从消息对垒中读取消息

消息对列面向记录，具有特定的格式和优先级，独立于发送与接收进程，进程终止时，消息对列及其内容也不会被删除；消息对列可以实现消息的随机查询，消息一定要以先进先出的次序读取，也可以按照消息的类型读取

//信号量：知识一个计数器，可以控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信的数据

1）信号量用于进程间的同步，如果需要传递数据，需要结合共享内存

2）信号量基于操作系统的PV操作，程序对信号量的操作都是原子操作

3）每次对信号量的PV操作不仅限于对信号量的值加1减1，可以加减任意正整数

3）支持信号量组

//信号：用于通知接收进程某个事件已经发生

//共享内存：使得多个进程可以访问同一块空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。需要依靠某种同步操作，如互斥锁和信号量

1）共享内存是最快的IPC，因为进程是直接对内存IPC的存取

2）因为多个进程可以同时操作，所以需要进行同步

3）信号量+共享内存，信号量用来同步对共享内存的访问

（3）套接字socket：一种进程通信的机制，可以用于不同主机间的通信

**线程间的通信:**

临界区：通过多线程的串行化来访问公共资源或者一段代码，快速，适合控制数据的访问

互斥量：采用互斥对象机制，只拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以可以保证公共资源不会被多个线程同时访问

信号量：为控制具有有限数量的用户资源设计的，允许多个线程在同一时刻去访问同一个资源，但是一般需要限制统一时刻访问此资源的最大线程数目

事件：通过同志操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作

**进程与线程区别:**

（1）进程是CPU资源的最小单位，线程是CPU调度的最小单位

（2）进程具有独立的系统资源，同一进程内线程共享进程的大部分资源，包括堆，代码段，数据段，每个线程只拥有一些在运行中必要的私有属性，比如tcb，等

（3）一个进程崩溃，不会对其他进程影响；但是一个线程崩溃，会让同一个进程内的其他线程也死掉

（4）进程在创建，切换和销毁的时候开销比较大，线程相反；进程在创建的时候分配资源，销毁的时候释放系统资源，进程切换需要两步：切换页目录，刷新TLB以使用新的空间；切换内核和硬件上下文；同一进程逻辑地址空间是一样的，不需要切换页目录，刷新TLB;

（5）进程间通信复杂，同一进程中线程由于共享代码段和数据段，所以通信简单。

**游戏应该分配多个进程:** **分配线程会相互影响.**

**死锁产生的必要条件:**

（1）互斥条件：一个资源每次只能被一个进程使用

（2）请求与保持条件：一个进程因请求资源而阻塞时，对已经获得的资源保持不放

（3）不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺

（4）循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系

**操作系统的页表寻址**:

页式内存管理，内存分成固定长度的一个个页片。操作系统为每一个进程维护了一个从虚拟地址到物理地址的映射关系的数据结构，叫页表，页表的内容就是该进程的虚拟地址到物理地址的一个映射。页表中的每一项都记录了这个页的基地址。通过页表，由逻辑地址的高位部分先找到逻辑地址对应的页基地址，再由页基地址偏移一定长度就得到最后的物理地址，偏移的长度由逻辑地址的低位部分决定。一般情况下，这个过程都可以由硬件完成，所以效率还是比较高的。页式内存管理的优点就是比较灵活，内存管理以较小的页为单位，方便内存换入换出和扩充地址空间。

**线程需要保存那些上下文，SP,PC,ESX寄存器是什么:**

线程在切换的过程中需要保存当前线程Id、线程状态、堆栈、寄存器状态等信息。其中寄存器主要包括SP PC EAX等寄存器，其主要功能如下：SP:堆栈指针，指向当前栈的栈顶地址

PC:程序计数器，存储下一条将要执行的指令

EAX:累加寄存器，用于加法乘法的缺省寄存器

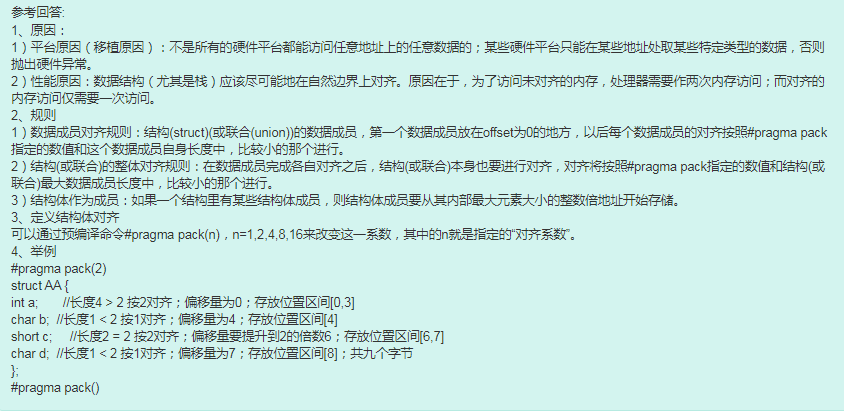
**操作系统缺页置换算法:**

当访问一个内存中不存在的页，并且内存已满，则需要从内存中调出一个页或将数据送至磁盘对换区，替换一个页，这种现象叫做缺页置换。当前操作系统最常采用的缺页置换算法如下：先进先出(FIFO)算法：置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。

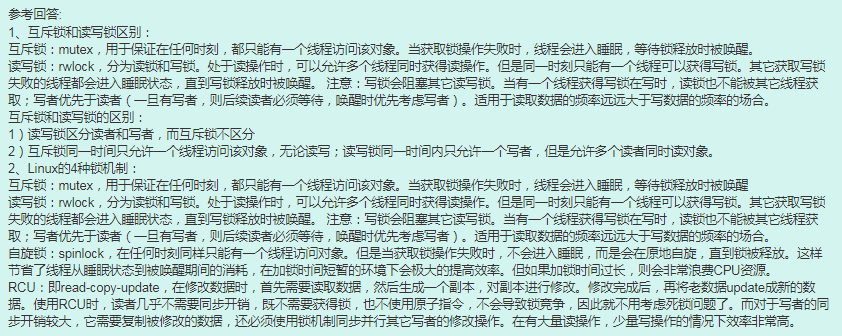
最近最少使用（LRU）算法: 置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。

当前最常采用的就是LRU算法。

**操作系统中结构体对齐，字节对齐:**



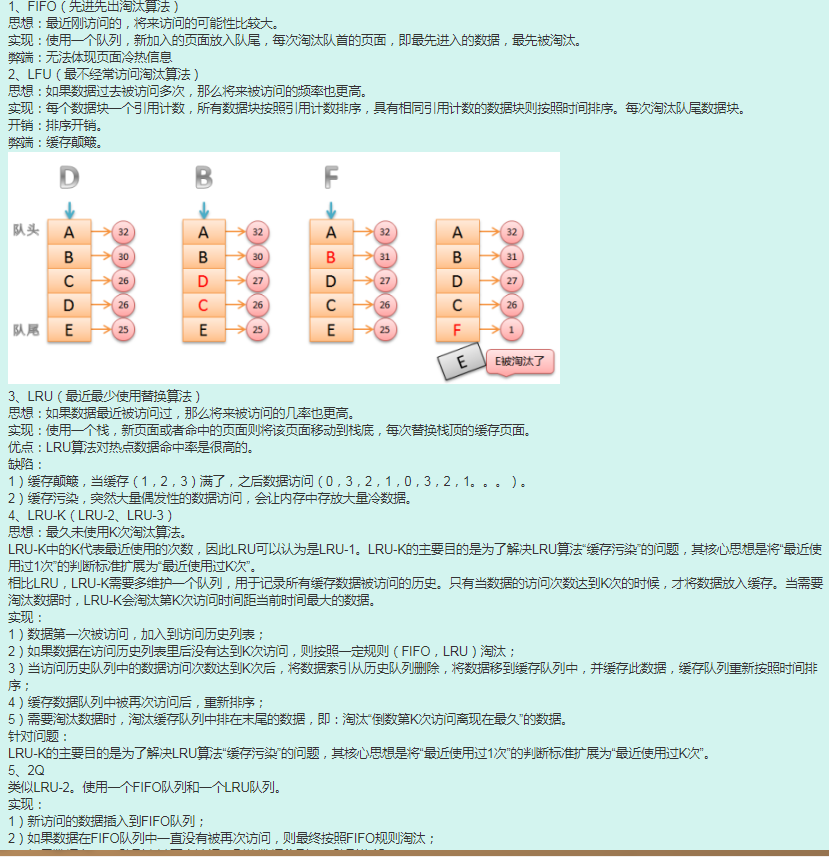
**互斥锁 与读写锁:**



**软连接与硬连接:**

为了解决文件共享问题，Linux引入了软链接和硬链接。除了为Linux解决文件共享使用，还带来了隐藏文件路径、增加权限安全及节省存储等好处。若1个inode号对应多个文件名，则为硬链接，即硬链接就是同一个文件使用了不同的别名,使用ln创建。若文件用户数据块中存放的内容是另一个文件的路径名指向，则该文件是软连接。软连接是一个普通文件，有自己独立的inode,但是其数据块内容比较特殊**。**

**虚拟内存置换的方式:**



**epoll原理:**

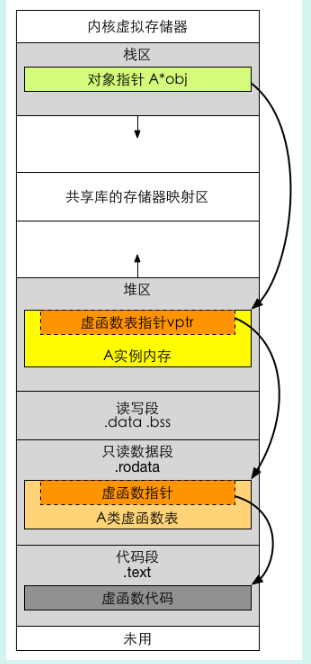
首先创建一个epoll对象，然后使用epoll\_ctl对这个对象进行操作，把需要监控的描述添加进去，这些描述如果将会以epoll\_evebt结构体的形式组成一颗红黑树，接着阻塞epoll\_wait，进入大的循环，当 某个fd有事件发生的时候，内核将会把其对应的结构体放到一个链表中，返回有事件发生的链表.

**fork，wait，exec函数:**

父进程产生子进程使用fork拷贝出来一个父进程的副本，此时只拷贝了父进程的页表，两个进程都读同一块内存，当有进程写的时候使用写实拷贝机制分配内存，exec函数可以加载一个elf文件去替换父进程，从此父进程和子进程就可以运行不同的程序了。

fork从父进程返回子进程的pid，从子进程返回0.调用了wait的父进程将会发生阻塞，直到有子进程状态改变，执行成功返回0，错误返回-1.exec执行成功则子进程从新的程序开始运行，无返回值，执行失败返回-1。

**内存:**



**并发和并行:**

并发：宏观上看起来两个程序同时运行，比如单核CPU上的多任务，但是微观上是程序的指令交替运行。这种并发不能提告计算机的性能，只能提高效率。

并行：严格物理意义的同时运行，多核CPU，两个程序分别运行在两个核上，互不影响。

**操作系统的中断**:

中断是指CPU对系统发生的某个事件做出的一种反应，CPU暂停正在执行的程序，保存现场后自动去执行相应的处理程序，处理完该事件后再返回中断处继续执行原来的程序。中断一般三类，一种是由CPU外部引起的，如I/O中断、时钟中断，一种是来自CPU内部事件或程序执行中引起的中断，例如程序非法操作，地址越界、浮点溢出），最后一种是在程序中使用了系统调用引起的。而中断处理一般分为中断响应和中断处理两个步骤，中断响应由硬件实施，中断处理主要由软件实施。

**如何设计server，能够接收多个客户端请求:** **多线程，线程池，io复用**

**死循环+来连接时新建线程的方法效率低:**

提前创建好一个线程池，用生产者消费者模型，创建一个任务队列，队列作为临界资源，有了新连接，就挂在到任务队列上，队列为空所有线程睡眠。改进死循环：使用select epoll这样的技术

**协程:**



**微内核与宏内核:**

宏内核：除了最基本的进程、线程管理、内存管理外，将文件系统，驱动，网络协议等等都集成在内核里面，例如linux内核。优点：效率高。

缺点：稳定性差，开发过程中的bug经常会导致整个系统挂掉。

微内核：内核中只有最基本的调度、内存管理。驱动、文件系统等都是用户态的守护进程去实现的。

优点：稳定，驱动等的错误只会导致相应进程死掉，不会导致整个系统都崩溃

缺点：效率低。典型代表QNX，QNX的文件系统是跑在用户态的进程，称为resmgr的东西，是订阅发布机制，文件系统的错误只会导致这个守护进程挂掉。不过数据吞吐量就比较不乐观了。

**5种IO模型:**

1.阻塞IO:调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的去检查这个函数有没有返回，必须等这个函数返回才能进行下一步动作

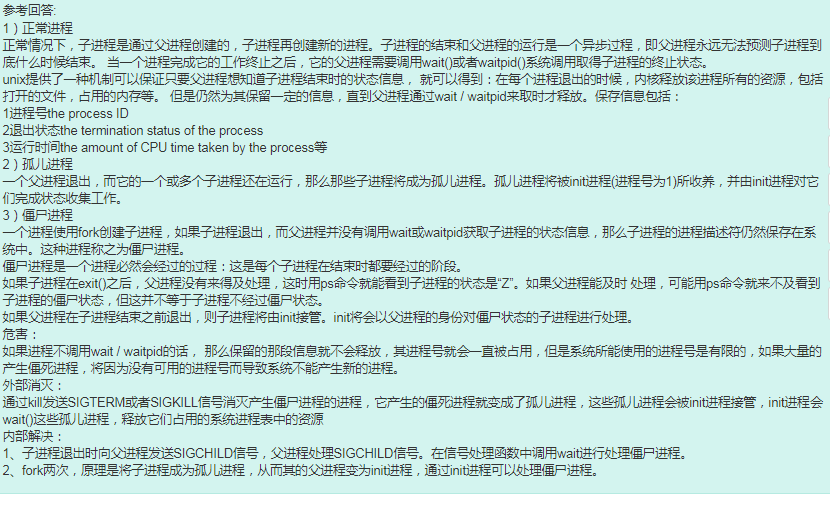
2.非阻塞IO:非阻塞等待，每隔一段时间就去检测IO事件是否就绪。没有就绪就可以做其他事。

3.信号驱动IO:信号驱动IO:linux用套接口进行信号驱动IO，安装一个信号处理函数，进程继续运行并不阻塞，当IO时间就绪，进程收到SIGIO信号。然后处理IO事件。

4.IO复用/多路转接IO:linux用select/poll函数实现IO复用模型，这两个函数也会使进程阻塞，但是和阻塞IO所不同的是这两个函数可以同时阻塞多个IO操作。而且可以同时对多个读操作、写操作的IO函数进行检测。知道有数据可读或可写时，才真正调用IO操作函数

5.异步IO:linux中，可以调用aio\_read函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。

**僵尸进程:**



守护进程：脱离于终端在后台执行的程序并周期性的执行某些任务，通常以d结尾，随系统启动，其父进程ppid通常是init进程；

**异步编程事件循环:**

事件循环就是不停循环等待时间的发生，然后将这个事件的所有处理器，以及他们订阅这个事件的时间顺序依次依次执行。当这个事件的所有处理器都被执行完毕之后，事件循环就会开始继续等待下一个事件的触发，不断往复。当同时并发地处理多个请求时，以上的概念也是正确的，可以这样理解：在单个的线程中，事件处理器是一个一个按顺序执行的。即如果某个事件绑定了两个处理器，那么第二个处理器会在第一个处理器执行完毕后，才开始执行。在这个事件的所有处理器都执行完毕之前，事件循环不会去检查是否有新的事件触发。在单个线程中，一切都是有顺序地一个一个地执行的.

**为什么要有page:**

加快从磁盘读取文件的速率。page cache中有一部分磁盘文件的缓存，因为从磁盘中读取文件比较慢，所以读取文件先去page cache中去查找，如果命中，则不需要去磁盘中读取，大大加快读取速度。在 Linux 内核中，文件的每个数据块最多只能对应一个 Page Cache 项，它通过两个数据结构来管理这些 Cache项，一个是radix tree，另一个是双向链表。Radix tree 是一种搜索树，Linux内核利用这个数据结构来通过文件内偏移快速定位Cache 项

**用户线程和内核线程：**

多个用户线程可以对应一个内核线程；

一个用户线程可以对应一个内核线程；

多个用户线程可以对用多个内核线程。

**池化技术：**

池化技术：实际应用中，分配内存、创建进程、创建线程都涉及系统调用，系统调用需要从用户态切换到内核态，非常耗时。因此使用池化技术来提前保存大量资源，以备不时之需和重复使用，提升程序的性能。

（1）内存池：

程序预先从操作系统中申请一块足够大内存；

此后程序需要申请内存时，不直接向操作系统申请，而是直接从内存池中获取；

释放内存时，并不真正将内存返回给操作系统，而是返回内存池；

当程序退出或特定时间时，内存池才将之前申请的内存释放。

（2）进程池：同线程池

（3）线程池：

先启动若干数量的线程，并让这些线程都处于睡眠状态；

当需要开辟线程去做具体任务时，则唤醒线程池中的某一睡眠线程；

任务完成后，线程又处于睡眠状态，而不是将线程摧毁。

**单核机器写多线程程序，是否要加锁：**

需要加线程锁。线程锁通常用来实现线程的同步和通信，在单核机器上写多线程程序，仍然存在线程同步的问题。因为在抢占式操作系统中，通常为每一个线程分配一个时间片，当某个线程时间片耗尽时，操作系统会将其挂起，然后运行另一个线程。如果这两个线程共享某些数据，不使用线程锁的前提下，可能会导致共享数据修改引起冲突。

**互斥锁、读写锁、自旋锁、RCU：**

互斥锁：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。

读写锁：rwlock，分为读锁和写锁。处于读操作时，可以允许有多个线程同时获得读操作，但是同一时刻只能有一个线程可以获得写锁，其他获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。

自旋锁：spinlock，在任何时刻同样只能有一个线程访问对象，但是当获取锁操作失败时，不会进入睡眠，而是在原地自旋，直到锁被释放。这样节省了线程从睡眠状态到被唤醒期间的消耗，在加锁时间短暂的环境下会极大地提高效率，但是如果加锁时间过长，则会非常浪费CPU资源。

RCU：即read-copy-update，在修改数据时，首先需要读取数据，然后生成一个副本，对副本进行修改，修改完后再将老数据update成新的数据。使用RCU时不需要考虑死锁问题。

**上下文切换：**

**上下文(context)：**一个进程存储在处理器各寄存器中的中间数据叫做进程的上下文

**上下文切换：**上下文切换（有时也称做进程切换或任务切换）是指 CPU 从一个进程或线程切换到另一个进程或线程**。**

进程上下文切换和线程上下文切换

上下文切换：切换CPU到另一个任务时，需要保存当前任务的上下文和加载新任务的上下文。只能发生在内核态

进程：1）切换页目录以使用新的地址空间；2）切换内核栈和硬件上下文。

线程：无需切换页目录，只需要切换内核栈和硬件上下文。

**进程的特征：**

动态性：进程是程序的一次执行过程，动态产生、动态消亡；

独立性：进程是一个能独立运行的基本单位，是系统进行资源分配和调度的基本单位；

异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进

并发性：进程可以同其它进程一起并发执行；

结构特征：进程由程序、数据和进程控制块三部分组成。

**进程同步机制遵循的原则：**

（1）空闲让进；

（2）忙则等待（保证对临界区的互斥访问）；

（3）有限等待（有限代表有限的时间，避免死等）；

（4）让权等待，（当进程不能进入自己的临界区时，应该释放处理机，以免陷入忙等状态）。

**进程调度的种类：**

高级调度：(High-Level Scheduling)又称为作业调度，它决定把后备作业调入内存运行

低级调度：(Low-Level Scheduling)又称为进程调度，它决定把就绪队列的某进程获得CPU

中级调度：(Intermediate-Level Scheduling)又称为在虚拟存储器中引入，在内、外存对换区进行进程对换

**抢占式调度和非抢占式调度：**

抢占式：操作系统将正在运行的进程强行暂停，由调度程序将CPU分配给其他就绪进程的调度方式

非抢占式：分派程序一旦把处理机分配给某进程后便让它一直运行下去，直到进程完成或发生进程调度进程调度某事件而阻塞时，才把处理机分配给另一个进程

**多线程的安全：**

当多个线程在共享同一个变量，做读写的时候，会由于其他线程的干扰，导致数据误差，就会出现线程安全问题。

synchronized关键字：将操作共享数据的语句加入synchronized关键字，在某一时段只会让一个线程执行完，在执行过程中，其他线程不能进来执行。

使用Lock锁：Lock使用起来比较灵活，但需要手动释放和开启。在并发量比较高的情况下，synchronized会让性能下降，此时使用Lock是个不错的方案。

使用Volatile修饰变量：保证可见性但不保证原子性

**多线程的三大特性：**

原子性：原子性就是在执行一个或者多个操作的过程中，要么全部执行完不被任何因素打断，要么不执行。

可见性：当多个线程访问同一个变量时，一个线程修改了变量的值，其他的线程能立即看到，这就是可见性。

有序性：就是程序的执行顺序会按照代码先后顺序进行执行，一般情况下，处理器由于要提高执行效率，对代码进行重排序，运行的顺序可能和代码先后顺序不同，但是结果一样。

**线程的状态：**

新建状态（New）：新创建了一个线程对象。

就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待 获取CPU的使用权。

运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

**sleep()/wait()/yield()/join()：**

join()：方法会使当前线程等待调用join()方法的线程结束后才能继续执行

wait()：调用该方法使持有该对象的线程把该对象的控制权交出去，然后处于等待状态，会释放当前的锁资源

notify()：调用该方法就会通知某个正在等待这个对象的控制权的线程可以继续运行

sleep()：在指定时间内让当前执行的线程暂停执行一段时间，进入阻塞状态，让其他线程有机会继续执行，但不会释放对象锁

yeild()：暂停当前正在执行的线程对象（及放弃当前拥有的cup资源），以允许具有相同优先级的其他线程获得运行机会。

**线程同步与互斥：**

线程同步：指多线程通过特定的设置（如互斥量，事件对象，临界区）来控制线程之间的执行顺序（即所谓的同步）也可以说是在线程之间通过同步建立起执行顺序的关系，如果没有同步，那线程之间是各自运行各自的！

线程互斥：指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步（下文统称为同步）