1. 虚拟内存 可以达到扩展内存的目的

将部分程序存入内存，执行完再存入下一部分程序(当内存空间不够时)

每个进程有自己的地址空间，这个空间被分成好多块，每一块称为一个页面。每个页面都有连续的地址，这些页被映射到物理内存。当程序引用到一部分在物理内存中的地址空间时，由硬件立刻执行必要的映射。当程序引用到一部分不再物理内存中的地址空间时，由操作系统（缺页异常）负责将缺失的部分装入物理内存并重新执行失败的指令。页面置换算法

按需调页：需要时才调入相应的页

写时复制：父子进程开始共享同一个页面，只有任何一个进程对页进行写操作时，就创建一个共享页的副本

页面置换：1、查找所需页在磁盘的位置 2、找一个空闲页。如果找不到空闲页，就使用页面置换算法牺牲一个页，并将牺牲页存入磁盘中

常见页面置换算法：

1. FIFO ：最先进入的页被置换 存在Belady异常：因为可能最早进入的页是程序经常调用的部分，所以这些页可能被频繁的换入换出，导致缺页率增加。
2. 最优页置换 : 置换最长时间不会使用的页(预测将来不会使用的，很难实现)
3. LRU :最近最少使用，

帧分配：每个进程分配多少内存？：平均分配、按比例分配（按照进程大小和优先级等）

全局分配:允许进程从所有帧中选择置换帧；局部分配：只能从进程自己的帧中选择置换帧。

内核内存分配的方法：

1. Buddy系统分配

<https://www.cnblogs.com/longchang/p/10749392.html>

1. slab分配

<https://blog.csdn.net/liuhangtiant/article/details/81259293>

之所以提出slab分配器，是因为buddy system只能按page对齐来分配内存。然而大多数情况下，需要的内存size都不是按page对齐的，如果直接通过buddy system分配内存，就会出现很大的内存碎片，内存碎片即分配了却没有使用，也无法再被分配的内存。

<https://www.cnblogs.com/ygj0930/p/6539590.html>

1. 内存管理的问题(也就是使用虚拟内存如何解决的)：
2. 进程地址空间隔离
3. 内存使用率低
4. 程序运行的地址不能确定 ，每次运行程序分配的内存是不同的，所以需要重定位
5. 中断和异常的区别

CPU对系统发生某事件时的一种响应：保留现场，执行中断处理程序。

外中断：外部设备事件引起的中断，如打印机中断、IO冲断、硬件故障等 由外因引起

内中断：也就是异常，如程序出错(非法指令、地址越界等)、系统调用、页故障等 由cpu本身原因引起

由于系统调用引起的中断叫做异常

1. Fork系统调用

一次调用，两次返回，返回值是0，就是子进程；返回值>0，就是父进程，返回值代表子进程的进程号。

整个父进程空间会全部复制到子进程，包括指令，变量值，程序调用栈，环境变量，缓冲区，等等

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main(void)

{

   int i;

   for(i=0; i<2; i++)

{

      fork();

      printf("-");

 }

   return 0;

}

解释：以上函数会输出8个’-’。由于有buffer

Unix 的块设备:一般都有缓存 程序遇到\n,EOF，缓冲区满，主动flush，数据会刷出缓冲区。

字符设备：一般没有缓存

以下程序一共有4个进程

#include "stdio.h"

#include "sys/types.h"

#include "unistd.h"

int main()

{

pid\_t pid1;

pid\_t pid2;

pid1 = fork();

pid2 = fork();

printf("pid1:%d, pid2:%d\n", pid1, pid2);

}

linux中有一个叫进程表的结构用来存储当前正在运行的进程。可以使用“ps aux”命令查看所有正在运行的进程。

请补足横线，使之输出welcome xiyoulinux

if(\_\_\_\_\_)

printf(“xiyoulinux ”);

else

printf(“welcome”);

应该填 fork()&&wait() 如果只填fork，那么输出的顺序是不确定的，

系统调用exit,使当前进程退出，但也仅仅限于将一个正常的进程变成一个僵尸进程，并不能将其完全销毁。不会占用内存，但linux中进程数量是有限的，太多僵尸进程，会影响新进程的产生。

收集信息，结束僵尸进程的系统调用：wait和waitpid

pid\_t wait(int \*status)：进程调用wait，会阻塞自己，wait自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait就会一直阻塞在这里，直到有一个出现为止。参数status用来保存被收集进程退出时的一些状态，它是一个指向int类型的指针。但如果我们对这个子进程是如何死掉的毫不在意，只想把这个僵尸进程消灭掉，（事实上绝大多数情况下，我们都会这样想），我们就可以设定这个参数为NULL。如果成功，wait会返回被收集的子进程的进程ID，如果调用进程没有子进程，调用就会失败，此时wait返回-1

可以用于进程同步:

pid\_t waitpid(pid\_t pid,int \*status,int options)：pid>0时，只等待进程ID等于pid的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid就会一直等下去；pid=-1时，等待任何一个子进程退出，没有任何限制，此时waitpid和wait的作用一模一样；pid=0时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid不会对它做任何理睬。

exec系统调用：根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。如果一个进程想执行另一个程序，它就可以fork出一个新进程，然后调用任何一个exec，这样看起来就好像通过执行应用程序而产生了一个新进程一样。

1. 多进程通信的方法：

<https://www.cnblogs.com/zgq0/p/8780893.html>

1. 管道

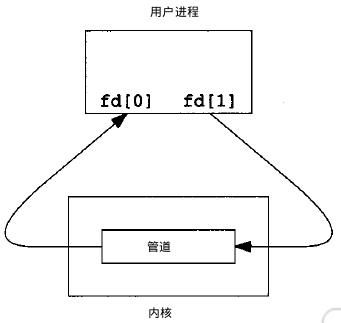
**、特点：**

1. 它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。
2. 它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。
3. 它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

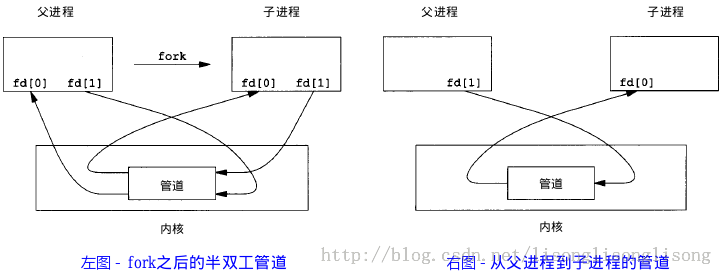
1 #include <unistd.h>

2 int pipe(int fd[2]); // 返回值：若成功返回0，失败返回-1

当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。如下图：



单个进程中的管道几乎没有任何用处。所以，通常调用 pipe 的进程接着调用 fork，这样就创建了父进程与子进程之间的 IPC 通道。如下图所示：



1. 命名管道

### 1、特点

1. FIFO可以在无关的进程之间交换数据，与无名管道不同。
2. FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。
3. 信号
4. 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

### 1、特点

1. 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。
2. 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。
3. 消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。
4. 共享内存
5. 信号量

### 特点

1. 信号量用于进程间同步，若要在进程间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值加 1 或减 1，而且可以加减任意正整数。
4. 支持信号量组。
5. 套接字

1.管道：速度慢，容量有限，只有父子进程能通讯

2.FIFO：任何进程间都能通讯，但速度慢

3.消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

4.信号量：不能传递复杂消息，只能用来同步

5.共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

1. 进程与线程的区别：

进程是系统进行资源分配和调度的一个独立单位

线程是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，可以与同一进程内的其他线程共享资源。

进程有独立的地址空间。一个进程崩溃后，不会对其他进程有影响，线程是一个进程中不同的执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

同一进程的线程共享哪些资源：

1. 堆，堆是在进程空间上开辟出来的。
2. 全局变量，与特定线程无关
3. 静态变量。存放的位置和全局变量一样
4. 文件等公用资源

独享的资源：

1. 栈
2. 寄存器

一个进程可以创建的线程数由可用虚拟空间和线程的栈的大小共同决定，只要虚拟空间足够，那么新线程的建立就会成功

1. Linux文件属性 共十位 文件类型标识符 用户权限位 组权限位 其他权限位

r 读 w 写 x执行

chmod修改权限

1. Makefile的作用：自动化编译，确定文件编译的顺序，类似shell脚本
2. 死锁产生的条件。预防死锁的方法：

条件：

1. 互斥：一个资源每次只能被一个进程使用
2. 不可抢占：进程已获得的资源，在未使用完之前，不能强行剥夺
3. 占有并等待：一个进程因请求资源而阻塞时，对已获得的资源保持不放
4. 环形等待：

预防死锁：使上述b、c、d条件之一不成立

* 1. 破坏c) :也即在不允许进程在已获得某种资源的情况下，去申请其他资源。

方法一：创建进程时，一次性申请所有资源。

方法二：进程每次需要申请新的资源前，释放它所占有的资源

* 1. 破坏b)：允许对资源进行抢夺

方法一：如果某个进程进一步申请资源被拒绝时，就要释放它所占有的资源

方法二：根据优先级，释放资源，优先级高的抢占优先级低的资源。

* 1. 破坏d):对资源编号，进程申请资源的顺序是按照编号的顺序

避免死锁：银行家算法

加锁时限，线程尝试在加锁时加上一定的时限，超时则释放自己占有的锁

检测死锁：

1. 5种IO模型：

阻塞式IO

非阻塞式IO 阻塞非阻塞是指在等待内核数据时是否阻塞，两种方式在内核数据拷贝到用户态时都是阻塞的

信号IO

IO多路转接

异步IO

1. 进程通信的方式： IPC(进程间通信)

https://www.cnblogs.com/zgq0/p/8780893.html

1. 管道
2. 是半双工的，即数据只能在一个方向上流动，有固定的读端和写端
3. 只能在父子进程或者兄弟进程中通信
4. 它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

int pipe(int fd[2]); // 返回值：若成功返回0，失败返回-1

当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。要关闭管道只需将这两个文件描述符关闭即可。

可以这样理解，一个进程把数据写入管道，然后另一个进程从管道中读数据

1. FIFO 命名管道

1、FIFO可以在无关的进程之间交换数据，与无名管道不同。

2、FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

在数据读出时，FIFO管道中同时清除数据，并且“先进先出”，类似于文件传输数据。

1. 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

1. 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。
2. 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。
3. 消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取
4. 信号量

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数据。信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

当semget创建新的信号量集合时，必须指定集合中信号量的个数（即num\_sems），通常为1； 如果是引用一个现有的集合，则将num\_sems指定为 0

1. 共享内存 两个或多个进程共享一个给定的存储区

信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

1. 多线程经典问题:

<https://wenku.baidu.com/view/ca54848fba4cf7ec4afe04a1b0717fd5360cb2e3.html>

1. 生产者-消费者问题:

该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

要解决该问题，就必须让生产者在缓冲区满时休眠（要么干脆就放弃数据），等到下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。同样，也可以让消费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。通常采用进程间通信的方法解决该问题。

会出现死锁问题

需要加两个锁，一个判断缓冲区是否为空(为满)，一个判断是否有资源对缓冲区进行操作。

1. 读者写者问题：有一个写者很多读者，多个读者可以同时读文件，但写者在写文件时不允许有读者在读文件，同样有读者在读文件时写者也不去能写文件。

等待的情况：

第一．写者要等到没有读者时才能去写文件。

第二．所有读者要等待写者完成写文件后才能去读文件。

可以设置一个变量来记录正在读文件的读者个数，第一个开始读文件的读者要负责将关闭允许写者进入的标志，最后一个结束读文件的读者要负责打开允许写者进入的标志。这样第一种“等待”情况就解决了。第二种“等待”情况是有写者进入时所以读者不能进入，使用一个事件就可以完成这个任务了——所有读者都要等待这个事件而写者负责触发事件和设置事件为未触发。详细见代码中注释：

1. 哲学家进餐问题: 会发生死锁，链状循环？

每个哲学家拿到两个叉子之后才会进食，如果一个线程拿到左面的叉子后，等待右面的叉子

1. 内碎片与外碎片：
2. 虚拟内存地址翻译的过程：

首先通过页表基址寄存器找到页表的地址，然后找到页表中的某一项，页表中的一项包括：标记位，记录是否有物理地址（也就是数据是否在内存中，或者是高速缓存中），如果有，则找到对应的物理地址返回，没有就产生中断（缺页异常处理程序），将相应的在磁盘中的内容拷贝到物理内存中（用到置换算法），中断结束后，会重新进行一次寻址。因为页表在物理内存中，每次CPU都会产生一个虚拟地址，MMU就要从物理内存中查询一次PTE，可以将PTE缓存起来，这样加速查找的过程（用到了局部性原理，分为时间局部性：最近访问的数据可能会经常访问；空间局部性:访问某一地址的数据，可能会继续访问该地址附近的数据）

多级页表：只有第一级页表放在内存中，其他级的页表放在磁盘中，节省了磁盘空间