

32bitCPU可寻址4G线性空间，每个进程都有各自独立的4G逻辑地址。3-4G是内核空间；0-3G是用户态空间

静态区域：**文本代码段**（只读存储区（字符串常量）和文本区（程序的机器代码））

**Data数据**区（存储已初始化的全局变量和静态变量）；

**bss**（未初始化的全局变量和静态变量，程序运行main会统一清0）；

堆： malloc（内部是brk和mmp负责快中断）和new

映射区：存储动态链接库等文件映射、申请大内存（malloc时调用mmap会将物理地址映射成虚拟地址）

栈：使用栈空间存储函数的返回地址、参数、局部变量、返回值，从高地址向低地址增长。在创建进程时会有一个最大栈大小，Linux可以通过ulimit命令指定

MMU将cpu用到的逻辑地址翻译成成物理地址

操作系统基本功能

1. 操作系统包含了进程管理，内存管理，文件管理，I/O管理
2. 内核态：硬件/计时，中断，原子/系统管理
3. 从用户态进入内核态唯一途径是中断

整个切换过程：

1. 传参说明功能类型
2. 在用户态陷入中断
3. 进入内核态进程阻塞等待返回值
4. 内核态完成工作后，修改PCB放入就绪队列，用户态回会继续运行

------------------------------------------------------------------------------

**进程和线程**

**1.概念**

进程是系统资源分配的最小单元

线程是CPU调度的最小单元

一个进程可以包含多个线程，fork函数可以创建子进程

进程由PCB控制块管理（进程id，进程状态，切换时需要保存和恢复的cpu和寄存器，程序段）

2. 进程状态

创建，就绪，运行阻塞，结束

内存不够的时候回调度到磁盘，这时候叫挂起（就绪挂起和阻塞挂起）

就绪：万事具备只差cpu(时间片用完，I/O中断，优先进程)

阻塞：系统调用，IO操作，等待某事件发送

# 内核线程、用户线程、轻量级线程

内核线程就是内核的分身，一个分身可以处理一件特定事情。这在处理异步事件如异步IO时特别有用。内核线程是使用是廉价的，唯一使用的资源就是内核栈和上下文切时保存寄存器的空间。支持多线程的内核叫做多线程内核。

内核线程只在内核态运行，不受用户态上下文的拖累。

1. 在全系统范围内竞争处理器资源
2. 唯一使用的资源是内核栈和上下文切换时保持寄存器的空间
3. 调度的开销可能和进程自身差不多昂贵
4. 资源同步和数据共享比整个进程数据同步共享低一些

**1对1模型**

一般直接使用API或系统调用创建的线程均为一对一的线程。一个用户使用的线程就唯一对应一个内核使用的线程。

优点：用户线程具有了和内核线程一致的有点

缺点：1、由于许多操作系统限制了内核线程数量，因此一对一线程会让用户的线程数量受到限制/2、许多操作系统内核线程调度时，上下文切换的开销较大，导致用户线程的执行效率下降。

图片包含 形状

描述已自动生成

## 多对1模型

图示

描述已自动生成

多对一模型将多个用户线程映射到一个内核线程上，线程之间的切换由用户的代码来进行。

优点：因此相对于一对一模型，多对一模型的线程切换要快速许多。

缺点：多对一模型一大问题是，如果其中一个用户线程阻塞，那么所有的线程将都无法执行，因为此时内核里的线程也会随之阻塞。另外，在多处理器系统上，处理器的增多线程性能也不会有明显的帮助。但同时，多对一模型得到的好处是高效的上下文切换和几乎无限制的线程数量。

## 用户线程

用户线程时完全建立在用户空间的线程库，用户线程的创建、调度、同步和销毁全由库函数在用户空间完成，不需要内核的帮助。这种线程时及其低消耗和高效的。

处理器竞争：单纯的用户线程是建立在用户空间，其对内核是透明的，因此其所属进程单独1. 参与处理器的竞争，而进程的所有线程参与竞争该进程的资源。

2. 使用资源：与所属进程共享进程地址空间和系统资源。

3. 调度：由在用户空间实现的线程库，在所属进程内进行调度

图示

描述已自动生成

加强版用户进程

图示

描述已自动生成多对多

## 轻量级线程

轻量级进程(LWP)是建立在内核之上并由内核支持的用户线程，它是内核线程的高度抽象，每一个轻量级进程都与一个特定的内核线程关联。内核线程只能由内核管理并像普通进程一样被调度。

图示

描述已自动生成1对多

## 进程上下文切换过程

PCB task\_struc

(1)标示符pid：描述本进程的唯一标识符，用来区别其他进程。

(2)状态：任务状态，退出代码，退出信号等。

(3)优先级：相对于其他进程的优先级。

以上是进程信息。

(4)程序计数器：程序中即将被执行的下一条指令的地址。

(5)内存指针：包括**程序代码**和进程相关数据的指针，还有和其他进程共享的内存块的指针。

(6)上下文数据：进程执行时处理器的寄存器中的数据。

(7) I／O状态信息：包括显示的I/O请求,分配给进程的I／O设备和被进程使用的文件列表。

(8) 记账信息：可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。

内核空间有进程队列，队列中放着pid

切换进程的时候，pid1出去，会告诉cpu下一个进来的pcb

Pcb进来之后，cpu就获得了虚拟内存空间

然后通过MMU（页表）虚拟地址映射到物理地址，只有在请页时才完成，这时要建立页表和更新页表（页表是动态建立的）。页表不可被换出，不记年龄，它们被内核中保留，只有在exit时清除。（https://blog.csdn.net/yhc1991/article/details/28900895）

图示

描述已自动生成

图示

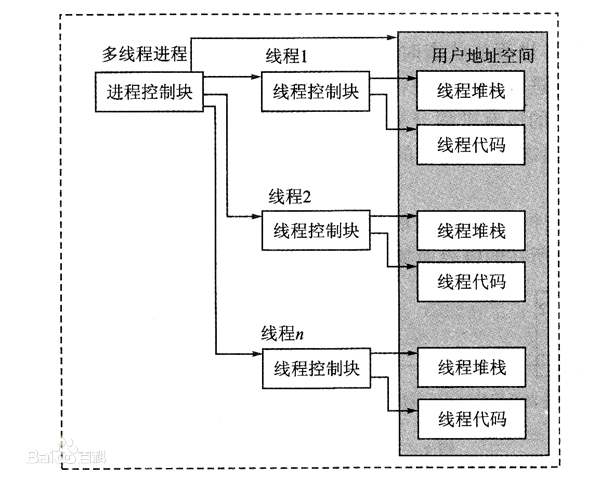
描述已自动生成

你想问的是页目录的地址还是页表的地址，页目录地址X86存储在cr3寄存器里面，ARM存储在cp15寄存器里面。

## 线程的上下文切换

Tcb是一种线程信息数据结构：线程标识符（线程信息）、（上下文信息）一组寄存器（通用，程序计数器。状态寄存器）

线程运行状态，线程专有存储区，堆栈指针，优先级别（调度信息）



线程是运行在进程上下文中的逻辑流，tcb保存内容少切换开销小

## 为什么说进程开销比线程高很多

进程切换：进程本身信息pid，调度信息优先级；最主要的是上下文信息

进程上下文信息包含：硬件上下文（寄存器信息），内核中栈信息，切换虚拟地址空间。

页表查找中的涉及到的：地址寄存器，TLB存储器（比内存快），进程的虚拟内存都需要切换

而线程的开销就只涉及：硬件上下文，和内核栈

### 在linux下进程和线程区别

线程共用进程中的全局变量，内存空间，代码段。即打开同一文件集

进程和线程的描述一致。独立的task\_struct进程描述符，独立的pid。只是文件集共用。

操作系统调度上都是在等待队列和双向链表中选取一个task\_struct切换到运行态。

同一进程下的线程之所有用getpid看到的pid是一样，是因为task\_struct中由tgid字段。

隶属于统一进程的线程该字段相同。

## 什么是协程？

在访问量非常的大的时候，如果采用线程存在以下问题：

* 1. 线程占用非常多的内存空间
  2. 线程切换占用cpu时间

协程没有增加线程数量，只是在线程的基础上通过分时复用的方式运行多个协程。

协程的切换在用户态完成，切换代价非常小。（所以协程是代码优化实现的）

协程不能中不能调用导致线程阻塞的操作。也就是说，协程只有和异步IO结合才行。

本质是将多线程变成单线程，通过分时复用原理实现代码优化。减少了内核中线程切换开销。

协程适用于大量I/O操作场景，但不能是阻塞IO的场景（线程一阻塞协程都阻塞了，阻塞IO如打印，读取文件）

所谓协程就是IO复用

## 线程之间通信

线程享有进程的堆数据区

所有可以通过全局变量，对象引用

以及java中notify（）唤醒wait（）所在的进程

synchronized(b) {...}；的意思是定义一个同步块,使用b作为资源锁。b.wait();的意思是临时释放锁，并阻塞当前线程,好让其他使用同一把锁的线程有机会执行,在这里要用同一把锁的就是b线程本身.这个线程在执行到一定地方后用notify()通知wait的线程,锁已经用完,待notify()所在的同步块运行完之后,wait所在的线程就可以继续执行.

## 进程之间通信IPC（inter\_process communication）都是内核对象，原子操作

管道pipe ：半双工，数据流，大小收到限制

1. 匿名管道

创建一个管道会生成两个文件描述符发f[0] (读)f[1]（写），关闭管道只需关闭两个文件描述符即可。

int main()

{

Int fd[2];//两个文件描述符

Pid\_t pid;

Char buff[20];

if(pipie(fd)<0) 创建失败

if(pid=fork()<0) 创建子进程失败

else if(pid>0) //父进程要做的

{

Close(fd[0]); //关闭读端

Write(fd[1],”dad write\n”);

}

Else

{

Close(fd[1]);//关闭写端

Read(fd[0],buff,20);

Printf(“%s”,buff);

}

Return 0;  
}

//从代码可以看出

用户态buff[20], 拷贝到内核态，在内核创建了内存，然后操作；拷贝出到用户态

1. **命名管道FIFO （这个我用过）**

特殊设备文件形式存在于文件系统中，有路径名与之关联

**是什么数据结构？**

**可用于无关系进程；管道都是半双工**

**按顺序**

mkfifo(const char \*pathname, mode\_t mode); // 返回值：成功返回0，出错返回-1

(fd = open("fifo1", O\_RDONLY))

fd = open("fifo1", O\_WRONLY))

write(fd, buf, n+1)

1. 消息队列

内核中树链接表，一个消息队列用一个队列id标识

全双工，无关系队列，传输数据。由大小限制

由数据类型

可以不按顺序读取

进程结束依旧存在

1. #include **<sys**/msg.h**>**
2. // 创建或打开消息队列：成功返回队列ID，失败返回-1
3. int msgget(key\_t key, int flag);
4. // 添加消息：成功返回0，失败返回-1
5. int msgsnd(int msqid, const void \*ptr, size\_t size, int flag);
6. // 读取消息：成功返回消息数据的长度，失败返回-1
7. int msgrcv(int msqid, void \*ptr, size\_t size, long type,int flag);
8. // 控制消息队列：成功返回0，失败返回-1
9. int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);
10. 共享内存 IPC中最快的因为读写只要2次，直接对内存进行操作，不用切换到内核
11. #include **<sys**/shm.h**>**
12. // 创建或获取一个共享内存：成功返回共享内存ID，失败返回-1
13. int shmget(key\_t key, size\_t size, int flag);
14. // 连接共享内存到当前进程的地址空间：成功返回指向共享内存的指针，失败返回-1
15. void \*shmat(int shm\_id, const void \*addr, int flag);
16. // 断开与共享内存的连接：成功返回0，失败返回-1
17. int shmdt(void \*addr);
18. // 控制共享内存的相关信息：成功返回0，失败返回-1
19. int shmctl(int shm\_id, int cmd, struct shmid\_ds \*buf);

信号量用于同步和互斥读写共享内存

1. 信号量实现互斥访问和同步（互斥访问也有swap法，但标志法，对临界资源的访问）

PV操作，singnal wait

1. 信号（signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一致的
2. Socket

TCP，udp全双工

### 什么是守护进程

个人理解守护进程和普通进程区别是指：**将后台程序变成一种服务**，比如说，用命令行输入启动程序，如果不是守护进程的话，一旦命令行窗口关闭，程序就终止了；而如果启动守护进程，则退出命令行窗口之后，服务一直处于运行状态。该服务**停止**（挂起但没终止）后变成**孤儿进程**由init收养。孤儿进程是父进程**已经终止**。

### 什么是僵尸进程

僵尸进程是指一个已经终止、但是其父进程尚未对其进行善后处理获取终止进程的有关信息的进程，这个进程被称为“僵尸进程”(zombie)。

在Linux进程的状态中，僵尸进程是非常特殊的一种，它已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其他进程收集。除此之外，僵尸进程不再占有任何内存空间。它需要它的父进程来为它收尸，如果他的父进程没安装 **SIGCHLD** 信号处理函数调用wait或waitpid()等待子进程结束，又没有显式忽略该信号，那么它就一直保持僵尸状态，如果这时父进程结束了， 那么init进程自动会接手这个子进程，为它收尸，它还是能被清除的。但是如果如果父进程是一个循环，不会结束，那么子进程就会一直保持僵尸状态，这就是 为什么系统中有时会有很多的僵尸进程。

查看进程：

Ps命令：父进程为1是孤儿进程

状态是Z的是僵尸进程

怎么处理僵尸进程

* 1. 父类中设置SIGchild信号函数，放置wait（）或者waitpid给子进程收尸
  2. 杀死父进程，init接管处理

### 线程锁

**自旋锁(spinlock)**，非IO阻塞。回一直询问（开销小，效率更高，适合大量短时间访问）

**互斥锁(mutex lock)**，访问对象无法访问回阻塞，直到完成被唤起（开销比较大）

**条件锁(condition lock)**

条件锁就是所谓的条件变量，某一个线程因为某个条件未满足时可以使用条件变量使该程序处于阻塞状态。一旦条件满足以“信号量”的方式唤醒一个因为该条件而被阻塞的线程。最为常见就是在线程池中，起初没有任务时任务队列为空，此时线程池中的线程因为“任务队列为空”这个条件处于阻塞状态。一旦有任务进来，就会以信号量的方式唤醒一个线程来处理这个任务。

**读写锁**

（unique \_lock与shared\_lock配合使用）

读者-写者模型：

计算机中某些数据被多个进程共享，对数据库的操作有两种：一种是读操作，就是从数据库中读取数据不会修改数据库中内容；另一种就是写操作，写操作会修改数据库中存放的数据。因此可以得到我们允许在数据库上同时执行**多个“读”**操作，但是某一时刻只能在数据库上有**一个“写”**操作来更新数据。这就是一个简单的读者-写者模型。

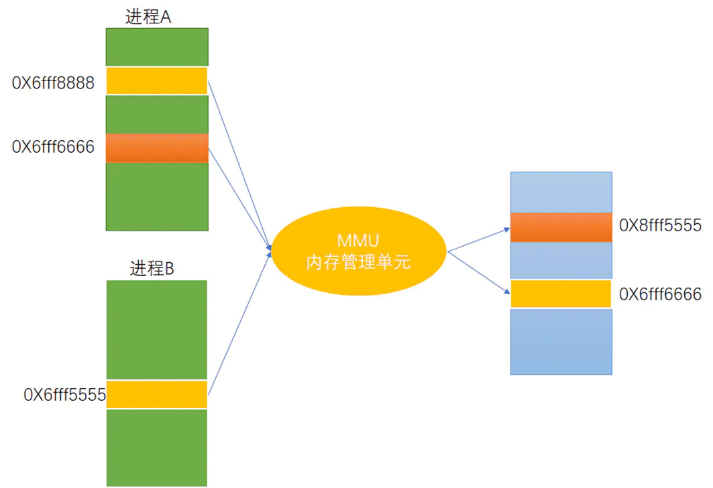
# 信号量和锁的区别

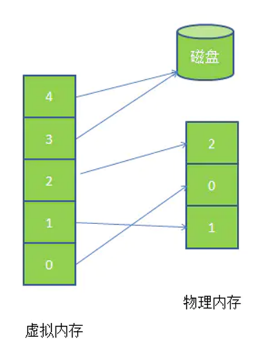
信号量是实现多线程多任务的同步。

mutex是semaphore的一种特殊情况（n=1时）。也就是说，完全可以用后者替代前者。**但是，因为mutex较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计**。

# 虚拟内存寻址实现过程

每个进程都有一个虚拟内存（4G），





平时操作都是操作虚拟地址内存单元，通过MMU实现物理地址访问。

一个新进程建立都有自己独立的4G内存空间；新进程建立时候，回建立自己的内存空间，此进程数据，代码等从磁盘拷贝到自己对应的物理空间。数据对应的虚拟内存地址在task\_struct中记录（链表记录）。

每次调用进程不能进行完全拷贝到物理内存，物理内存有限。用到部分放入内存

如何找到对应物理地址？页表

4G的虚拟地址对应2^20页块。每个页表项需要4Byte存储，那么整个映射表就要4M

也就是1024个页块区存储。

问题：没有连续的1024个页块，一张表全部导入内存太大，有些地方用不到

解决方法：二级页表 把4M分成1024个4K表。通过另一个表去索引虽然整个表大小没有减少。但是表可以不用连续放置。也可以部分加载。节省内存 。

万一访问的地址数据不在物理内存怎么办？

当进程访问某个虚拟地址，去查看页表的时候，发现对应数据不咋物理内存，就发生缺页中断。

缺页处理过程如下：

把进程所需的数据从磁盘拷贝到物理内存中如果内存已经满了，没有空地方了，就找一个页覆盖，当然如果被覆盖的页曾经被修改过，需要将此页写回磁盘。

页表存放在MMU（memory manager unite）内存管理单元中。

可以理解成一个方法，接收虚拟地址返回物理地址。

优点：

每个进程创建加载时候，内核只是为进程创建了虚拟内存布局，具体就是初始化PCB表中内存相关的链表。实际上没立刻把虚拟内存对应位置的程序数据和代码拷贝到物理内存，只是建立好虚拟内存和磁盘文件之间的映射（也就是代码存在了磁盘）。等运行相应程序时才会通过缺页中断，MMU更新页表，把数据加载到相应的物理内存。

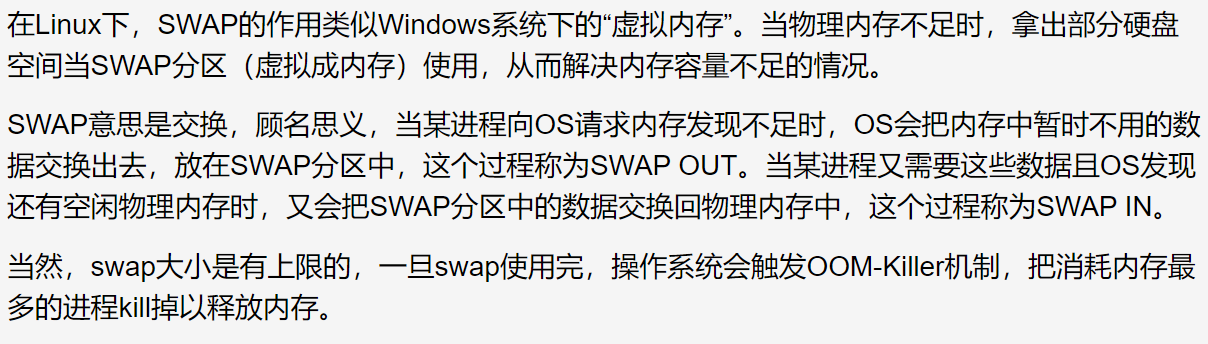
节省内存，充分利用内存资源

每个进程内存空间一致固定，链接器在连接可执行文件时，可以设定内存地址，不公管理实际地址

不同进程使用相同代码，如库文件中的代码，物理内存只需要存储一份，不同进程只需把自己虚拟内存映射到同一个位置即可。

充分利用碎片，虚拟内存连续，实际物理地址不连续

Swap区就是虚拟内存，内存不足的时候把硬盘一部分空间虚拟成内存使用，从而解决内u才能容量不足。



## 线程同步的方式有哪些？

线程同步的概念：所谓同步，就是并发的线程在一些关键点上可能需要互相等待与互通信息，这种相互制约的等待与互通信息称为线程同步。

互斥量：采用互斥对象机制，只有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，所以可以保证公共资源不会被多个线程同时访问。

信号量：它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。

事件（信号）：通过通知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。

## 进程死锁的原因

资源不足（可使用资源比进程资源少）

访问顺序不当

多个进程持有部分资源同时等待对方释放已占有的资源，从而陷入无限循环等待。

## 死锁的四个必要条件

互斥量

请求与保持

不可剥夺

循环等待

死锁的解决

