# 操作系统

# 内存管理：

## 0:内存存在的问题

###### 0-1:进程之间的隔离：

虚拟地址空间解决，进程运行在虚拟地址空间，通过MMU映射到物理内存空间



###### 0-2:内存有限：

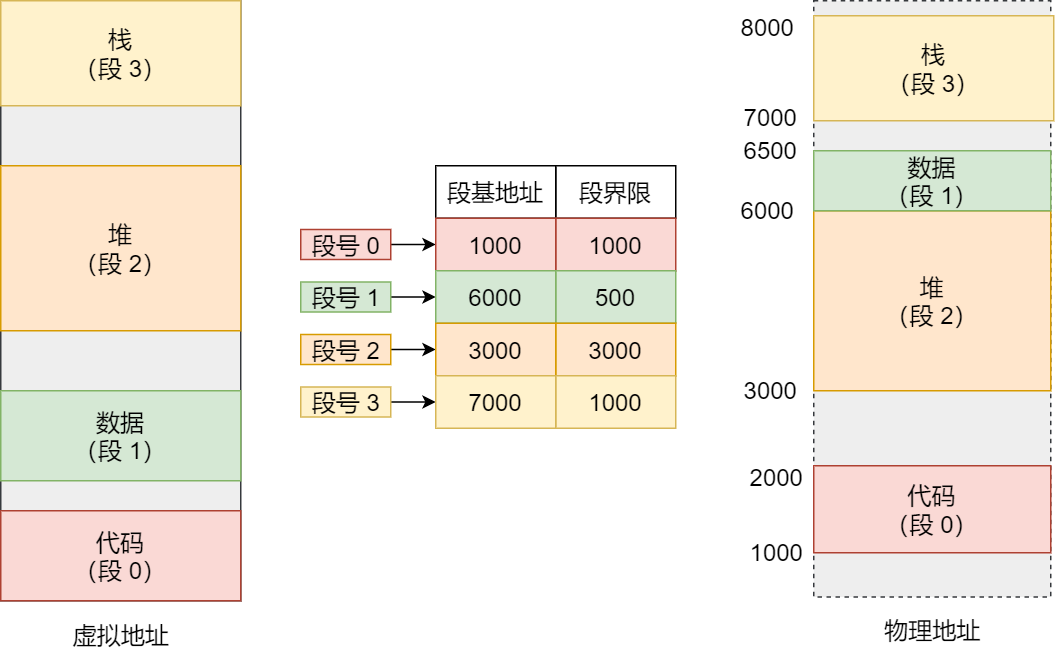
使用下面讲的段页式内存管理来解决

## 1:段式内存管理

###### 原理：

1:进程的虚拟地址空间段，每个段有段号

2:段表记录：虚拟段基地址 ->实际内存的段基地址映射



###### 优点：

可以按段控制访问权限

###### 问题：

按照段进程加载淘汰，存在内碎片

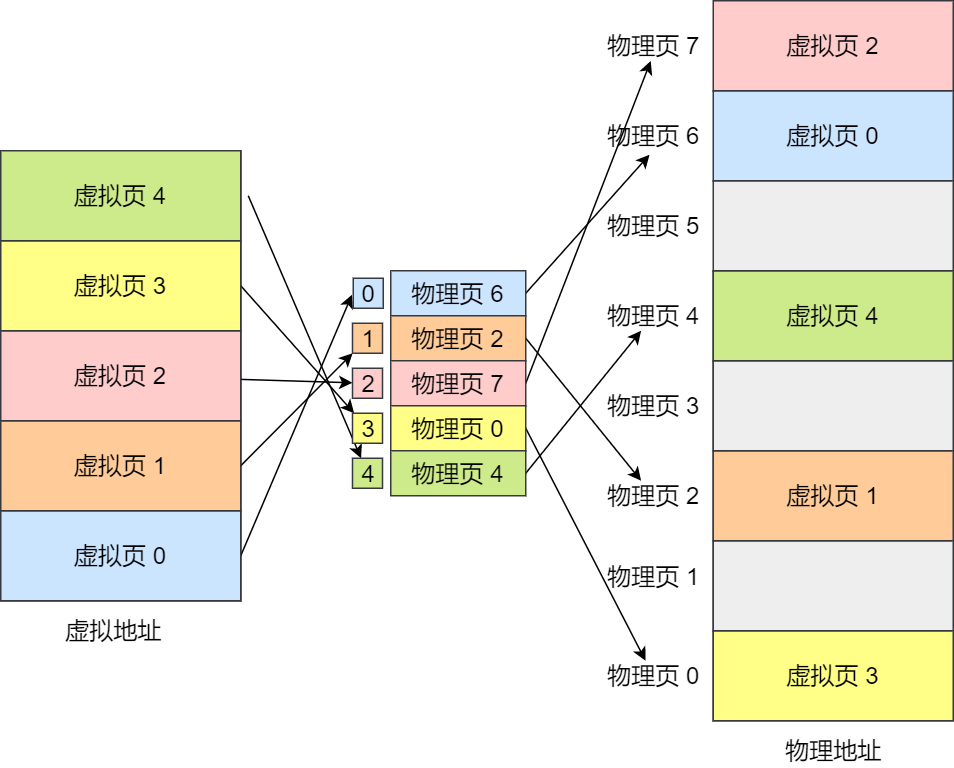
按照段进程加载淘汰，存在一次swap成本大

## 2:页式内存管理

###### 原理：

1:进程虚拟地址空间分割成虚拟段；物理内存分割成物理段

2:页表记录：虚拟页表地址 -> 时机物理页表地址的映射



###### 优点：

内存碎片小，swap交换效率高

###### 问题：

1:进程的各个区间没有权限的划分

2:页表占空间（4G虚拟地址空间，假设一个页大小4KB，那就需要100W 个页，每个页项需要4B的话，一个进程就需要4MB来存页表）

ps：用哪个映射哪个不行么？非要全映射？

## 3:段页式内存管理

结合段式和页式的优点，引入段页式存储

###### 原理：

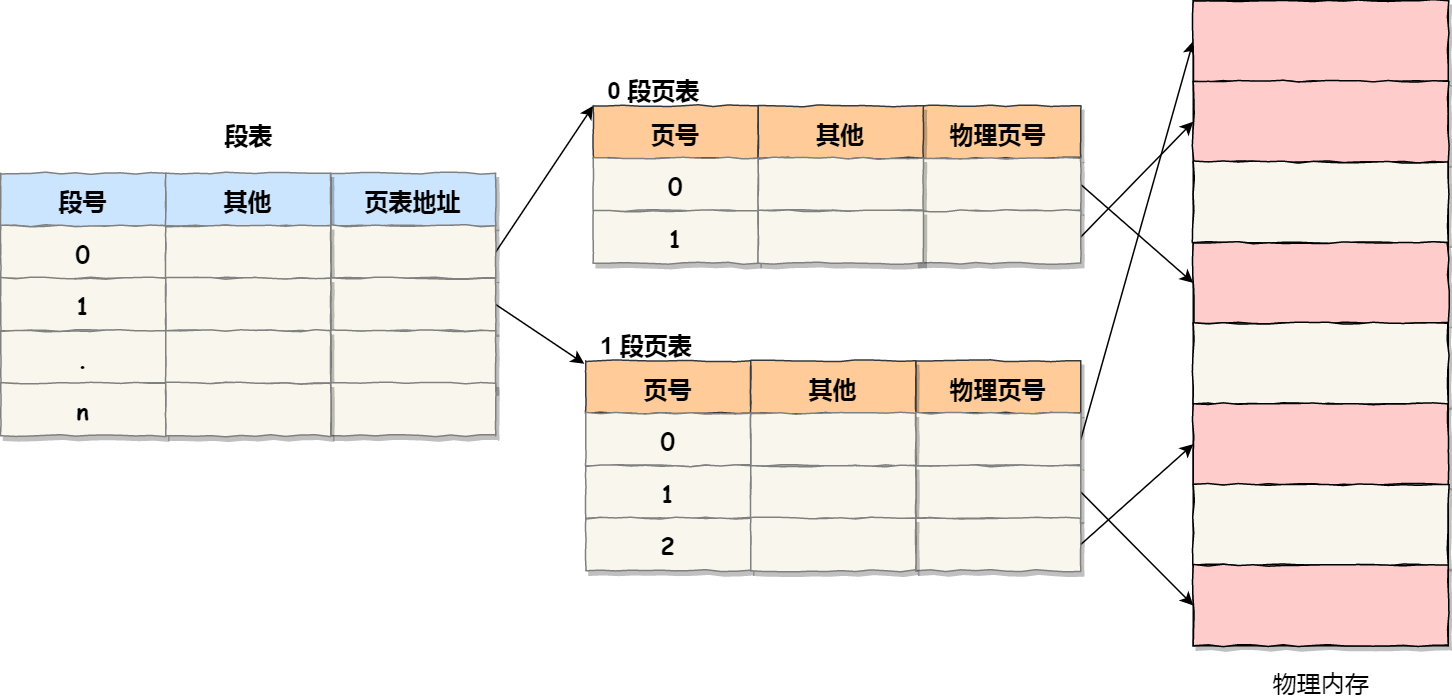
1:进程虚拟地址空间 **先分段，再分虚拟页**

2:实际物理内存 **分物理页**

3:os给每个进程分配固定大小的页个数+使用LRU算法

4:段表 -> 虚拟页表 –> 虚拟页号 –> 物理页号 做地址映射





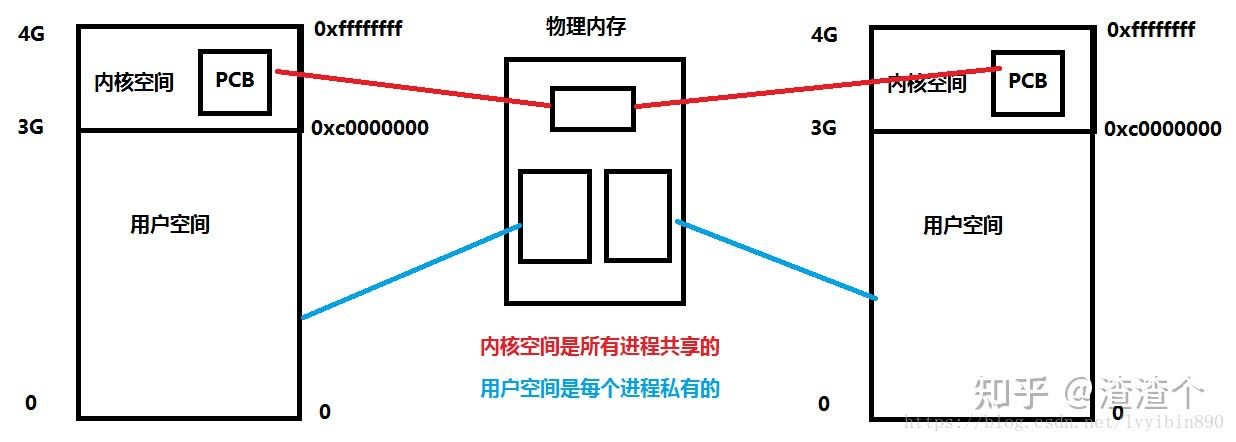
###### 优点：

碎片少，交换效率高，权限分段管理

## 4:进程的虚拟地址空间 与 进程用户态内核态

<https://www.cnblogs.com/sparkdev/p/8410350.html>

进程的虚拟地址空间主要分为两部分



###### 1:内核空间

背景：

CPU指令中有一些是非常危险的命令，如果交给用户程序直接执行很可能会出问题，所以提出了CPU的执行等级，执行在ring0等级，就需要切换到内核空间执行程序，此时程序处于内核态，执行相应的系统调用；

Ps：不仅是系统调用（软中断），硬中断处理程序也会跑在内核态

主要存储内容：

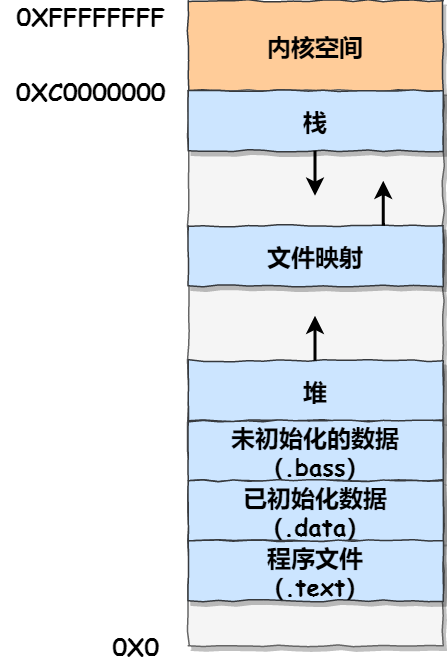
1:进程的PCB

2:内核栈

###### 2:用户空间

普通进程运行的空间，运行在ring3级别，用户态

其结构如下：



1：程序文件

2：初始化数据，包括静态变量

3：未初始化数据，包括未初始化静态变量

4 : 堆（malloc mmap数据）

5: 栈:线程栈

###### 总结：

1:运行在内核地址空间就是内核态，运行在用户地址空间就是用户态

2:进程的用户态和内核态有各自的 堆栈空间，运行在内核态的时候，肯定用的就是内核地址空间的堆栈。

3：CPU在任意时间点，肯定是以下状态之一：

* **运行于用户空间，执行用户进程。**
* **运行于内核空间，处于进程上下文，代表某个特定的进程执行。**
* **运行于内核空间，处于中断上下文，与任何进程无关，处理某个特定的中断。**
* 说明：Linux 系统的中断服务程序不在进程的上下文中执行，它们在一个与所有进程都无关的、专门的中断上下文中执行。之所以存在一个专门的执行环境，就是为了保证中断服务程序能够在第一时间响应和处理中断请求，然后快速地退出。

## 上下文切换内容：

###### 1:进程上下文切换

虚拟地址空间的切换

CPU寄存器+PC计数器（保存在进程PCB中）

###### 2:线程上下文切换

线程的CPU寄存器+PC计数器（保存在进程PCB中）

###### 3:系统调用上下文切换

用户态程序的CPU寄存器+PC计数器（保存在进程PCB中）

# 进程管理：

## 进程PCB：

进程的身份证，保存进程的**元信息**

**1:pid**

**2:进程类型**

**3:进程状态**

**4:进程优先级**

**5:进程调度策略**

**6:现场保护区（pc，寄存器现场信息，虚拟地址空间表）**

**7:资源需求，分配控制信息（打开文件数量，创建进程数量，mmap文件数）**

**8:进程的实体信息，是在物理内存中还是swap中**

**9:指向程序正文，数据，堆栈的内存区域的指针**

**10:打开的文件描述符表**

## 进程类别

###### 1：僵尸进程

使用ps –ef|grep defunc查看

说明：进程已经死掉了，但是PCB没有被销毁，PCB的销毁一般由父进程销毁，如果没有销毁，就会导致僵尸进程的存在

###### 2：孤儿进程

说明：进程的父进程已经退出了，那么此时子进程的父进程就是init进程，或者UI进程

## 进程调度：

###### 1:非抢占式

只要进程不让出CPU，就一直执行，完全FIFO

###### 2:抢占式

实时进程：

1:SCHED\_FIFO：先来先执行，除非自己让出CPU，或者更高优先级的抢走

2:SHCED\_RR：当两个进程优先级一样，使用时间片轮训

普通进程：

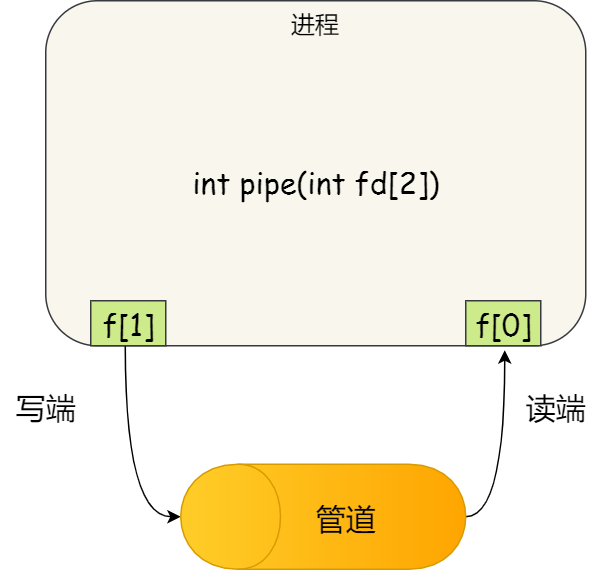
1:CFS（Completely Fair Scheduler）：完全公平调度器，按照优先级分配时间片比例，并且记录每个进程运行的时间片长度，如果某个进程一直处在IO阶段，没有占用CPU话，会优先照顾该类进程。

## 进程间通信方式

###### 1:管道

mkfifo myPipe创建有名管道，或者ps –ef | grep 123的｜也是匿名管道

原理：底层使用文件承载



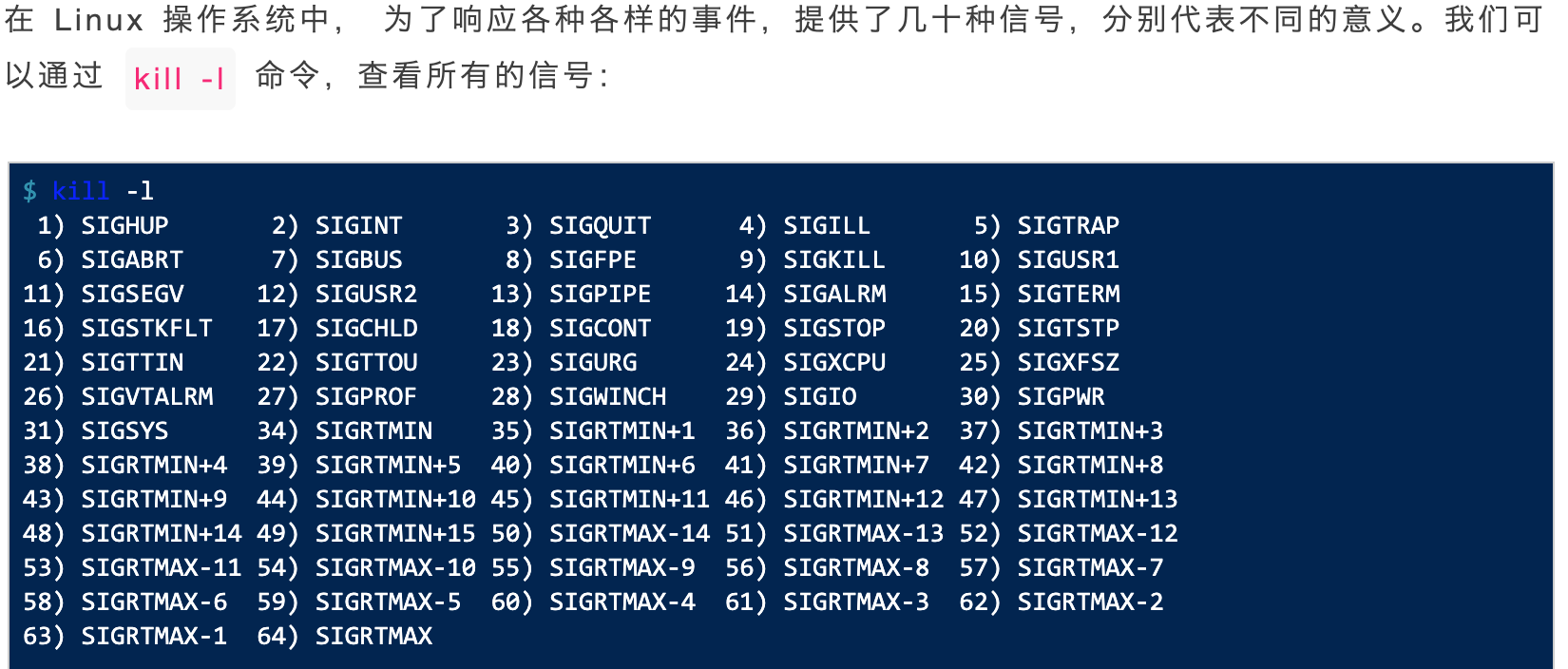
###### 2:消息队列

在内核中创建消息队列，因为内核空间是进程共享的，所以两个进程可以同时反问到。消息是放在内核中的。

###### 3:信号

进程直接接收各种信号，见下图

比如，我们在终端执行ctrl+c 就是给进程发送终止命令



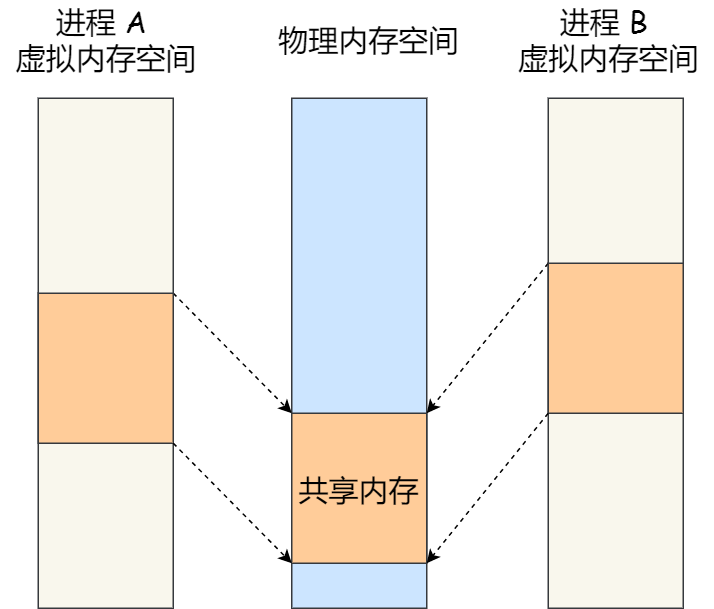
###### 4:信号量

jvm的synchronized底层就是使用二值信号量 mutex来实现；

信号量主要用于进程之间同步。

###### 5:共享内存

给两个进程的虚拟地址空间映射相同的物理内存，让两个进程访问相同的物理内存，配合信号量进行同步控制。比如使用mmap映射同一个文件的pagecache空间到 进程的虚拟地址空间。



###### 6:socket

通过网络协议通信

## 关于进程/线程/协程的思考

###### 进程：

os分配资源的最小单位，进程创建的子进程共享父进程创建的 物理内存，所有子进程有自己的虚拟地址空间，但是指向同一片物理内存；在jvm中，子进程会创一个独立的内存栈，大小最大1M

###### 线程：

os进程调度的最小单位，linux层面没有线程，只有LWP，轻量级进程，有自己的PCB(task\_struct)，其他空间共享父进程物理内存空间。

###### 协程：

用户级别线程，用来处理**大量的** **短任务**，因为使用线程来做的话，因为任务非常短，可能一个时间片都没有用到，就搞完了，那么线程切换就会显得非常耗时。如果想要在一个时间片内做很多任务，通过协程可以很好的做到，因为是用户级别的，不涉及线程的上下文切换，非常快。并且一个协程占用空间比线程小很多。