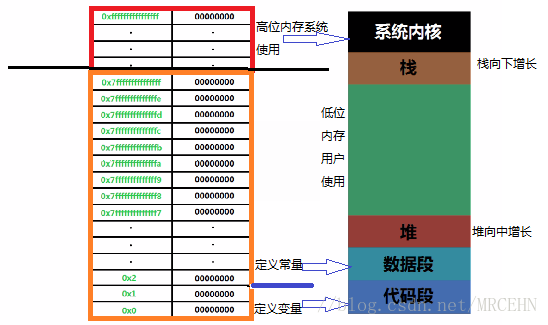
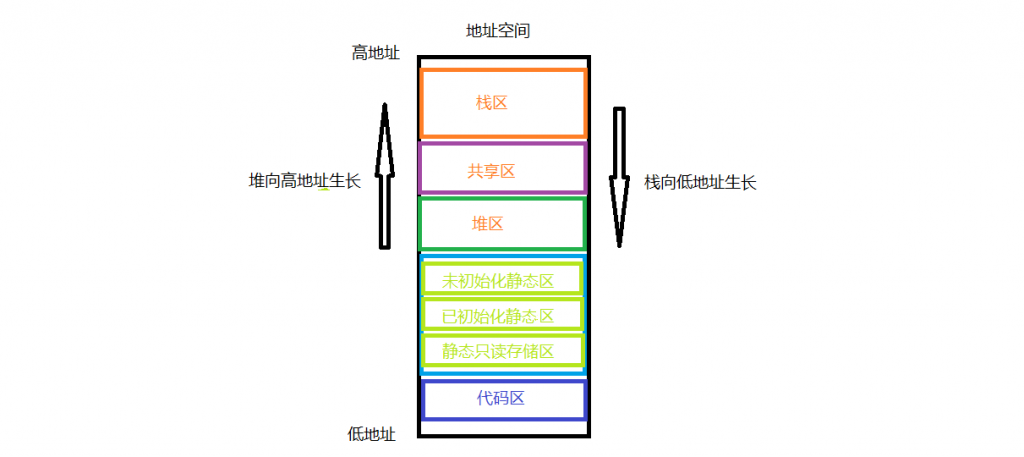
1. 程序的内存结构

32位：4G（3G用户空间；1G系统空间）





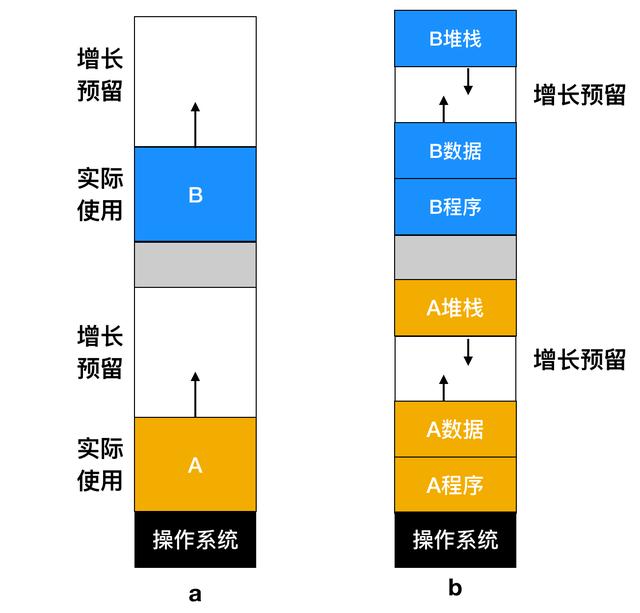
1. 代码段：指令
2. 数据段：

.data:初始化数据（全局变量、静态变量）

.rodata:常量

.bss:未初始化数据（未初始化全局变量、静态变量）

1. 堆：.heap
2. 栈：.stack
3. 程序运行加载到内存中（没有虚拟内存）



1. 加载程序

空闲区由操作系统管理（链表存储等）

运行程序->向操作系统申请内存加载程序->操作系统根据空闲区分配给程序

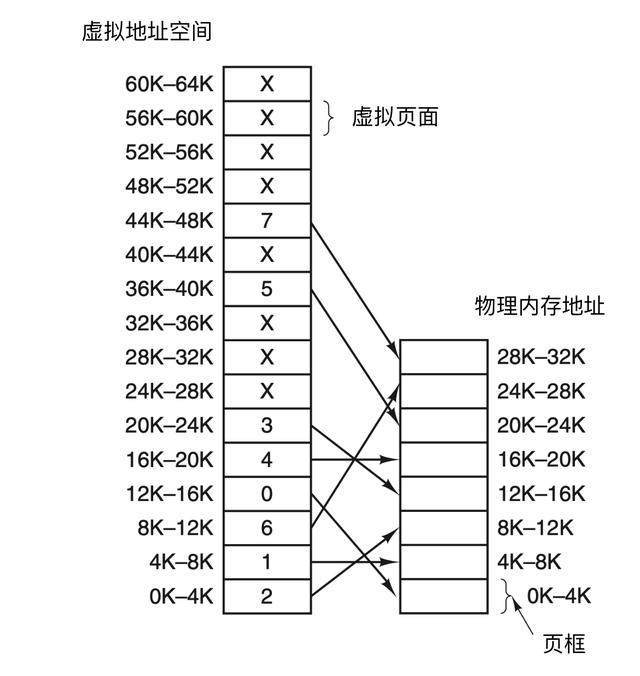
1. 操作系统匹配空闲区算法：

最佳适配、首次适配、最坏适配、下次适配

1. 问题：

产生空闲碎片；程序大小可能很大不可能都加载到内存中

1. 程序运行加载到内存中（有虚拟内存）

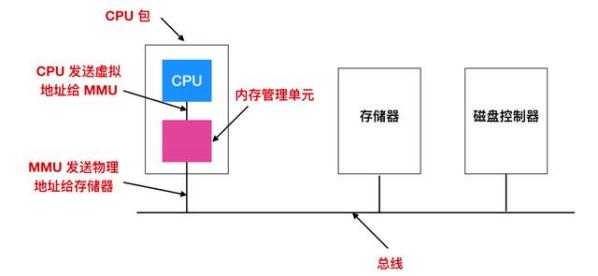


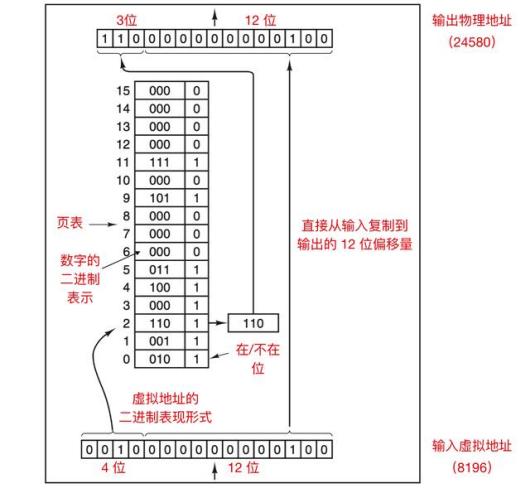
1. 程序以页为单位，物理内存也以页为单位

程序需要的页加载到内存中，不需要的暂时不加载，等CPU执行指令 发现缺页后在加载。

这样程序的大小就可以很大，也不用全部加载到内存中。

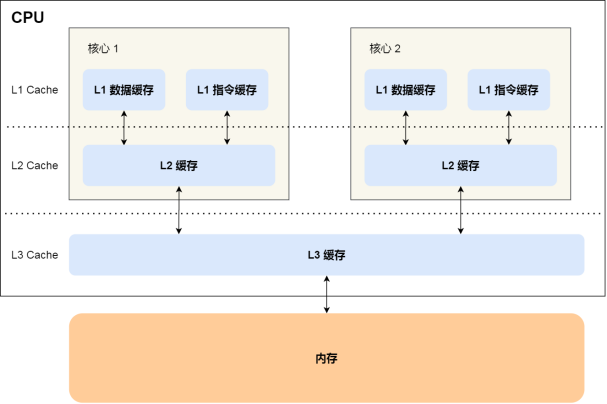
（2）地址映射MMU（将虚拟内存地址映射为物理内存地址）





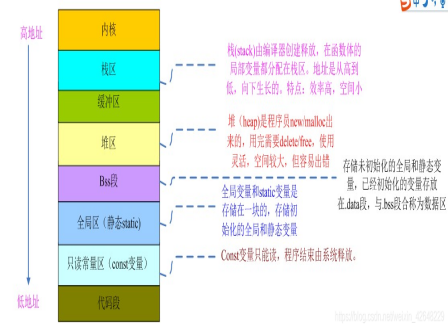
1. CPU访问内存流程

先看看是否命中缓存，再看内存，否则从磁盘中载入



产生的问题：

一致性问题，即可能改了缓存数据，但是主存和磁盘没改；改了主存，磁盘没改



1. 上下文切换

（1）上下文切换

定义：从当前任务切换到另外一个任务执行的过程。

动作：为了使得下一次任务从正确位置开始执行，需要保存上一个任务状态。

（2）线程与进程上下文切换区别

线程上下文切换：页表不变（虚拟内存与物理内存关系）

进程上下文切换：页表会变化

（3）线程上下文切换

需要保存的状态：程序计数器（PC）与CPU寄存器（Register）

（4）进程上下文切换

需要保存PCB：

a）进程ID。

b）进程状态。

c）CPU上下文。

d）I / O状态信息。

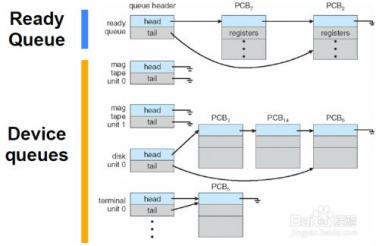
e）内存管理信息。 （不包括过程内存映像）

f）CPU调度信息。

PCB存储位置：

在内核中

由于PCB是数据结构，因此每个PCB的头和尾指针可以相互指向队列的头部，形成链表或像循环链一样，队列可以存储链表的头地址和尾部地址。



1. mmap

（https://blog.csdn.net/qq\_33611327/article/details/81738195）



传统：为了提高读写效率和保护磁盘，使用了页缓存机制，需要先拷贝到缓存再拷贝到主存两次拷贝

mmap：发生缺页，就从磁盘拷贝到主存中仅仅一次拷贝。

mmap是将文件映射到内存中，因此对文件的操作不用再调用内核函数进行读写，只需要用户空间进行操作。

之后进程即可对这片主存进行读或者写的操作，如果写操作改变了其内容，一定时间后系统会自动回写脏页面到对应磁盘地址，也即完成了写入到文件的过程。修改过的脏页面并不会立即更新回文件中，而是有一段时间的延迟，可以调用msync()来强制同步, 这样所写的内容就能立即保存到文件里了。

总而言之，常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝。而mmap操控文件，只需要从磁盘到用户主存的一次数据拷贝过程。说白了，mmap的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此mmap效率更高。

优点：

1、对文件的读取操作跨过了页缓存，减少了数据的拷贝次数，用内存读写取代I/O读写，提高了文件读取效率。

2、实现了用户空间和内核空间的高效交互方式。两空间的各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉。

3、提供进程间共享内存及相互通信的方式。不管是父子进程还是无亲缘关系的进程，都可以将自身用户空间映射到同一个文件或匿名映射到同一片区域。从而通过各自对映射区域的改动，达到进程间通信和进程间共享的目的。

     同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经保存在内存中的文件数据。

4、可用于实现高效的大规模数据传输。内存空间不足，是制约大数据操作的一个方面，解决方案往往是借助硬盘空间协助操作，补充内存的不足。但是进一步会造成大量的文件I/O操作，极大影响效率。这个问题可以通过mmap映射很好的解决。换句话说，但凡是需要用磁盘空间代替内存的时候，mmap都可以发挥其功效。