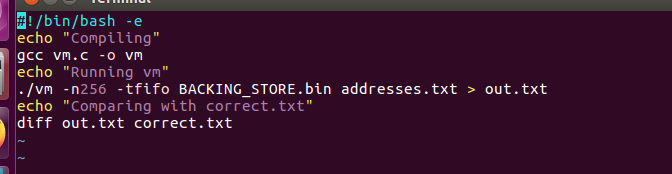
操作系统第二次大作业

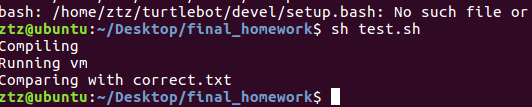
17363104 朱韬正

1. **虚存管理模拟程序**

**运行比较size=256, FIFO**

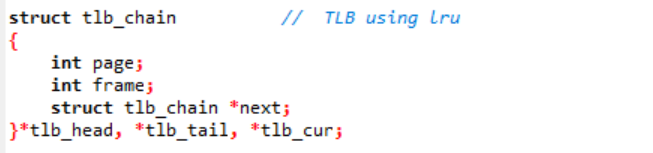
****

**运行比较，得出结果完全相同。**

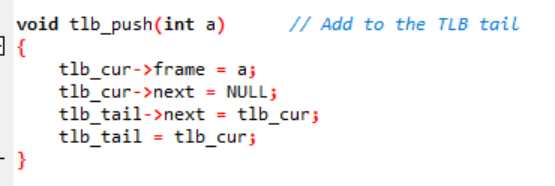
****

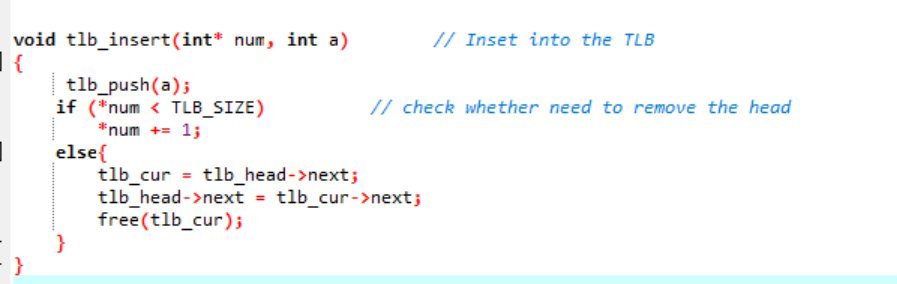
**2 LRU tlb的实现**

**为了实现last recent used 的效果，代码采取链表的形式，并声明当前指针，头指针，尾指针。**

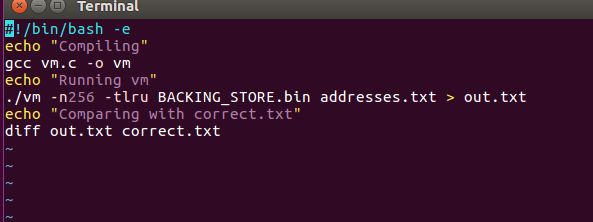
****

**采用类似栈的方式，删除头指针，在尾指针处加入**

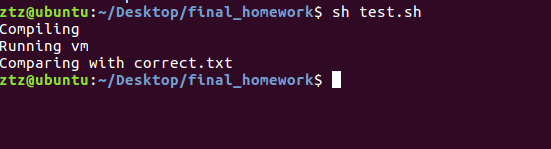
****

****

**初步测试，size=256，采用LRU策略。**

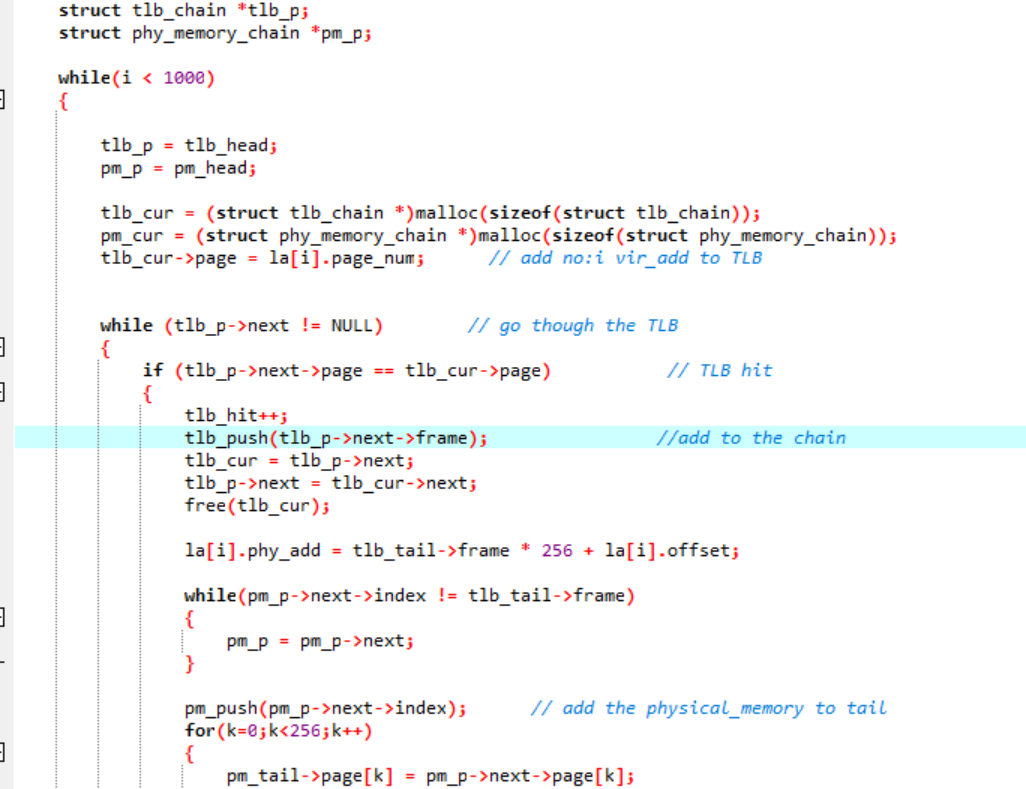
****

**可以看到结果也与correct完全相同。**

****

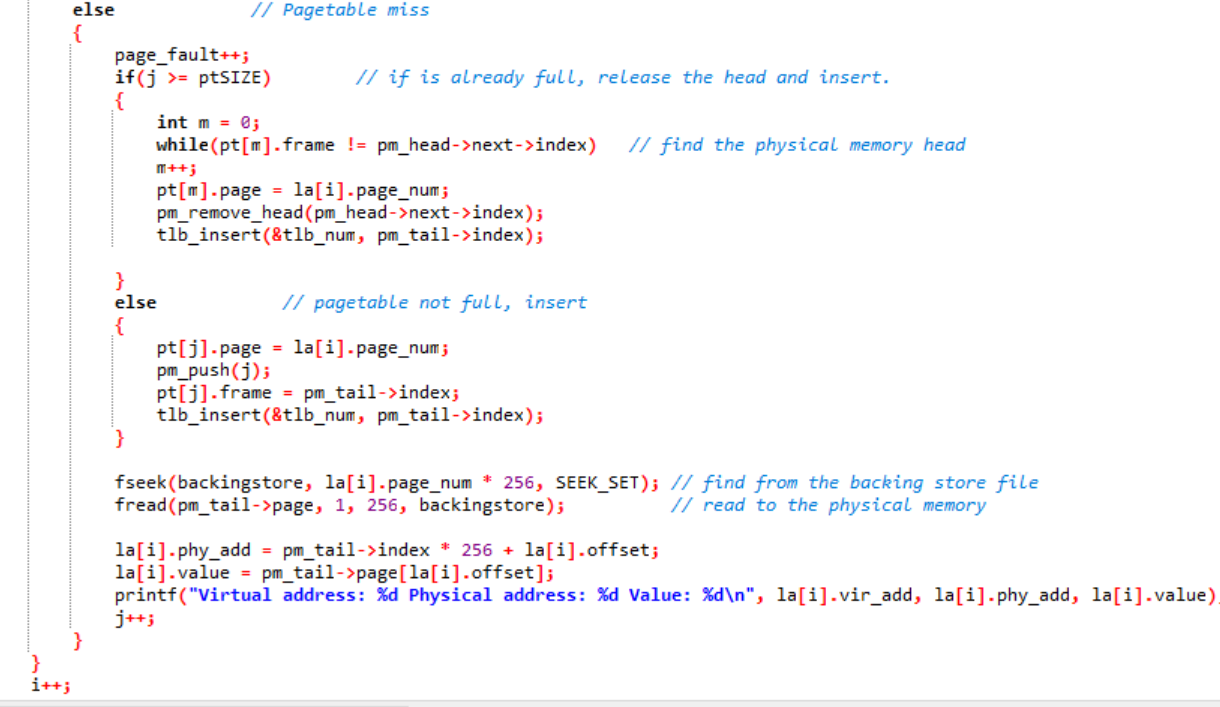
1. **lru 的换页**

**采用两个临时指针来遍历TLB与物理内存，pagetable 与FIFO共用数组。**

****

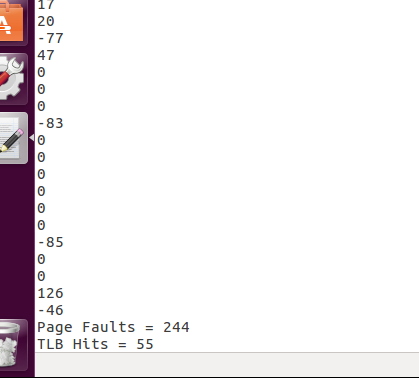
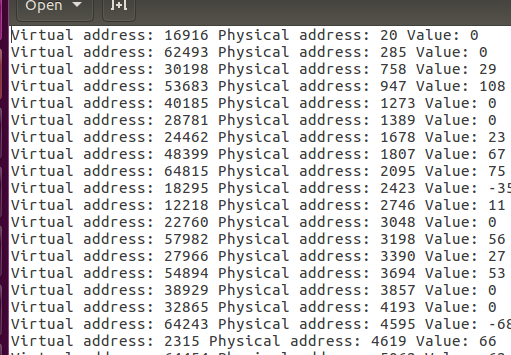
**如果命中则记录下来，并采取上面提到过的操作，将命中的设置为最新（插入表尾）。**

**若出现page fault,检查物理内存是否满了，并从后备文件调入数据。**

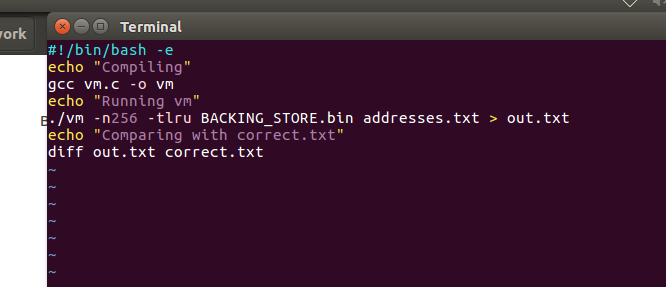
****

**3 TLB HIT 与Pagefault测试**

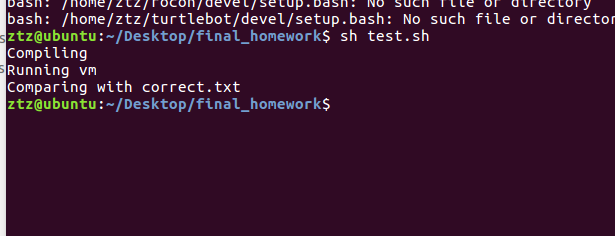
**教程的correct 格式与作业要求的不同，更改代码输出方式，进行比较。**

****

**Size = 256, LRU:**

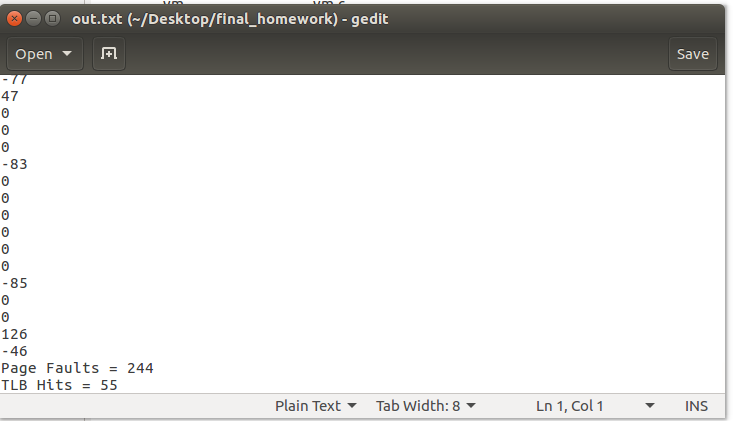
****

**Outcome:**

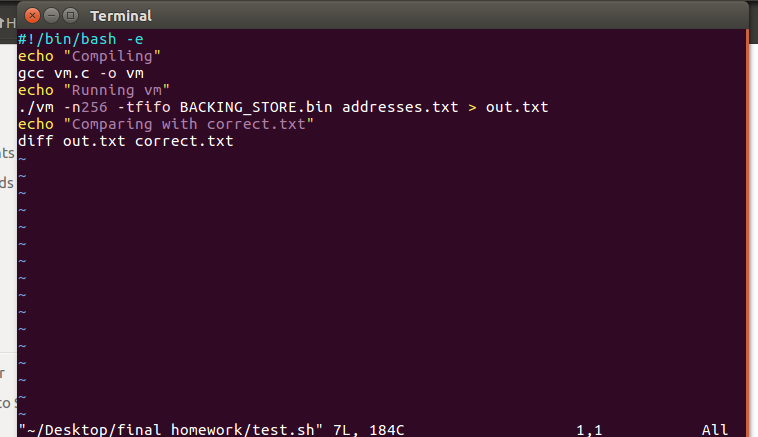
****

**完全一致，没有错误输出**

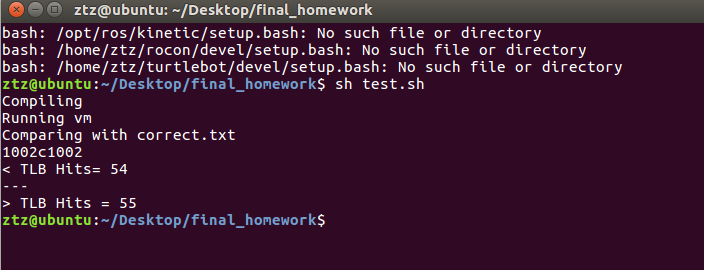
**Out.txt:**

****

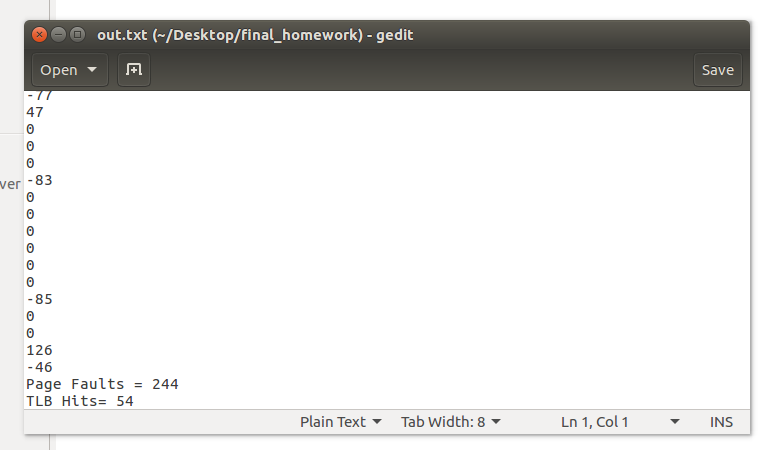
**Size = 256, FIFO:**

****

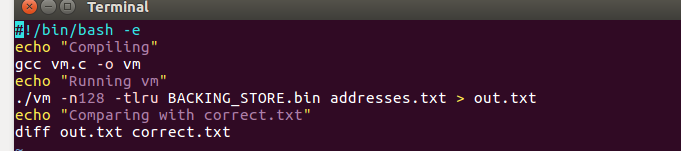
**运行：**

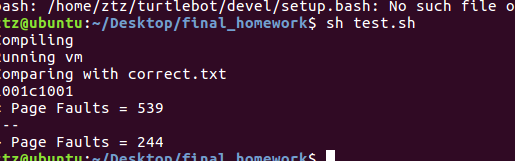
****

**Out.txt:**

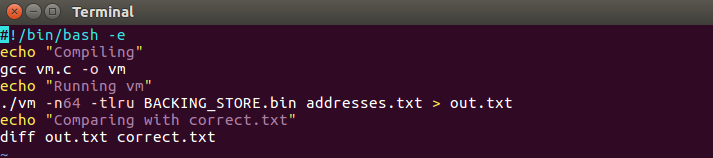
****

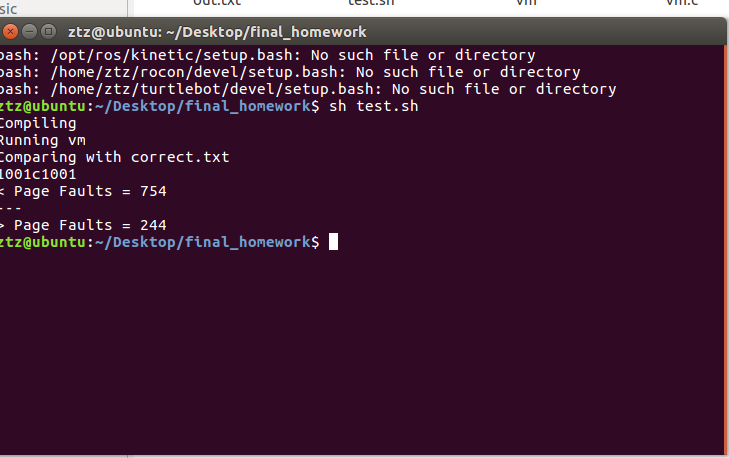
**Size = 128, LRU:**

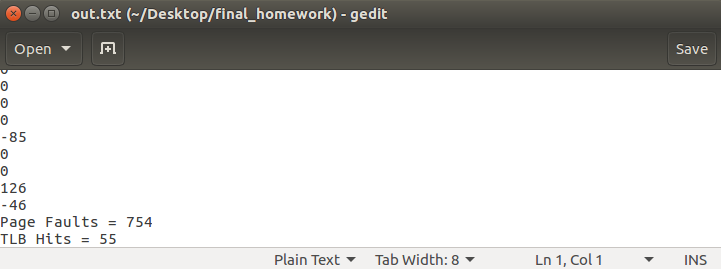
****

****

**Size = 64, LRU:**

****

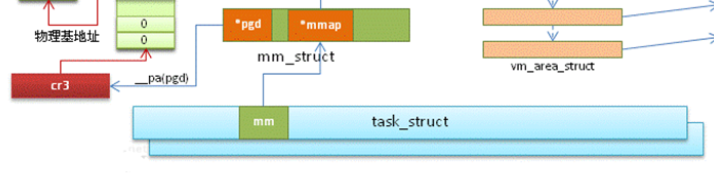
****

****

**明显page fault的概率大幅度上升了**

1. **Linux内存管理实验**

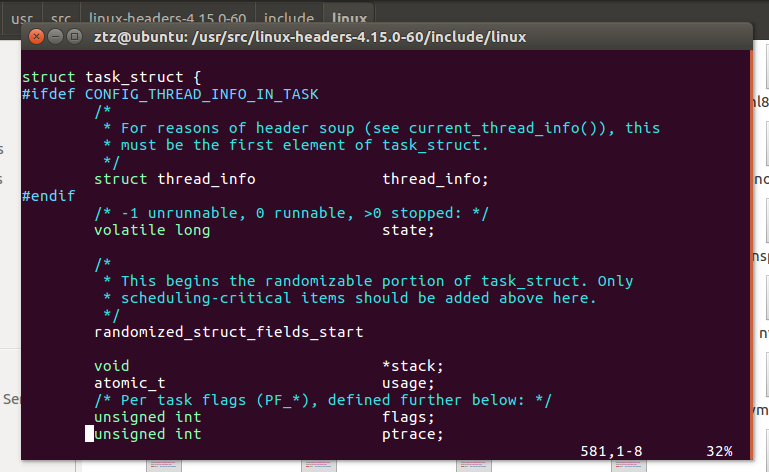
1分析图一

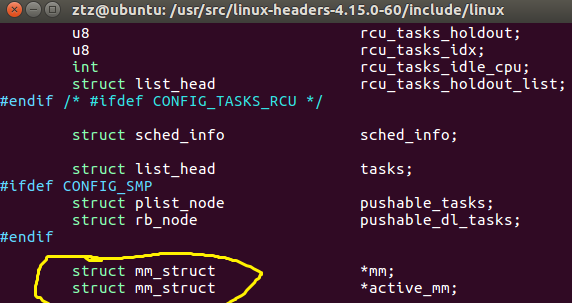


从最开始的task\_struct开始分析。

**Task\_struct**

在linux源码的sched.h文件中可以看到task\_struct的定义：



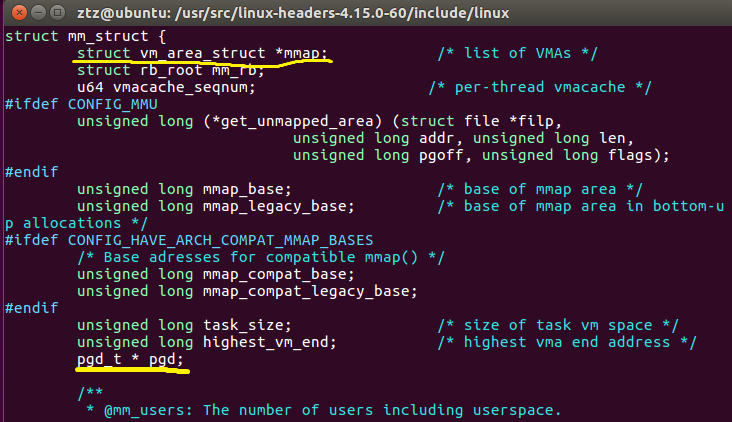


Task\_struct是描述进程的指针

可以发现和图一所示一样，有指向结构mm\_struct的指针。

**Mm\_struct**

它在mm\_types.h中可以找到，是进程的虚拟内存空间的最高描述结构，描述整个空间。每一个进程都有独立的mm\_struct。所谓的进程间共享，便是共享mm\_struct.



其中rb\_root 为红黑树的根指针。

其中的\*pgd,\* mmap两个指针在图一中特别标出。

**\*Pgd**

****

Pgd即Page global Directory指向Pgd\_t结构，指向了全局页表的目录。

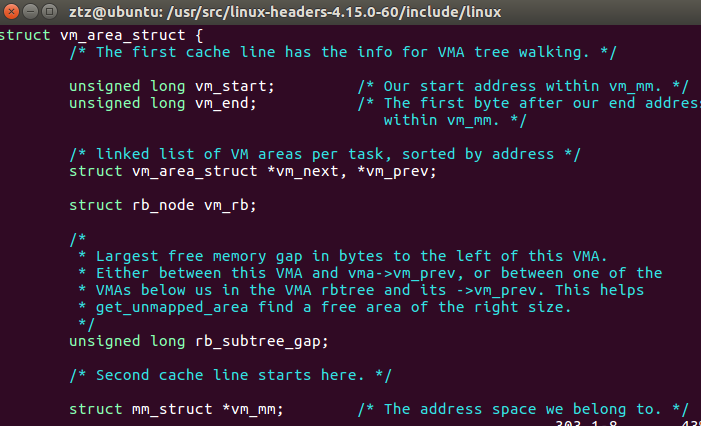
此外还有pud,pmd,pte，他们一起实现四级（或两级）页表结构。它是一个长整型，系统通过将它加载到cr3控制寄存器(页目录基地址寄存器)便可载入页表。进程切换时重新加载页表。

**\*Mmap**

该指针指向了一个vm\_area\_struct结构。是指向线性区间的链表头。

**Vm\_area\_struct**

同样可以在mm\_types.h中找到vm\_area\_struct结构。



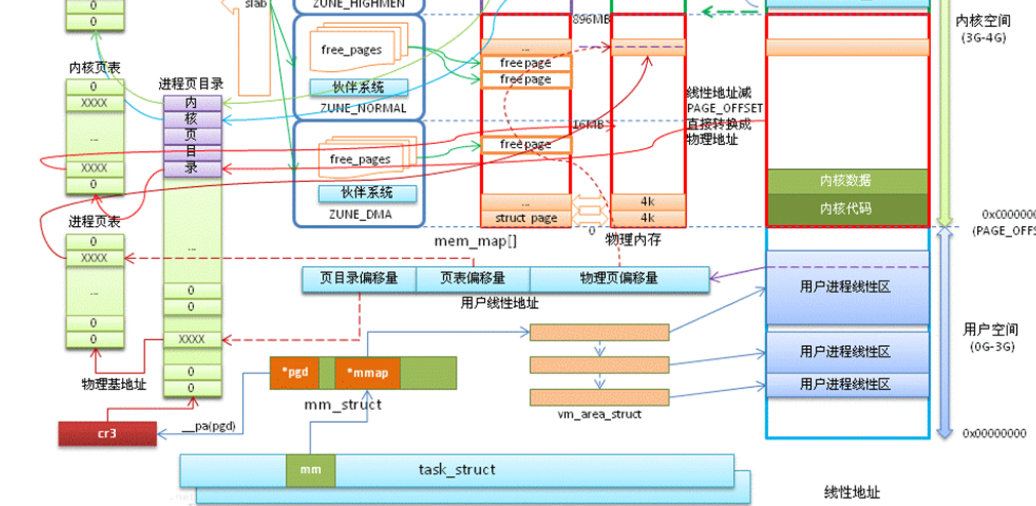
它描述了虚拟内存的一个区间，从 vm\_start 到 vm\_end(地址)

通过vm\_next 和 vm\_prev两个指针实现双向链表的功能,使得进程可以使用不连续的内存，并使得每块内存有自己的属性。当节点过多的时候还可以生成avl树来方便查找。

它构成了0-3G的用户内存。



于是，可以解释图1下半部分的内容



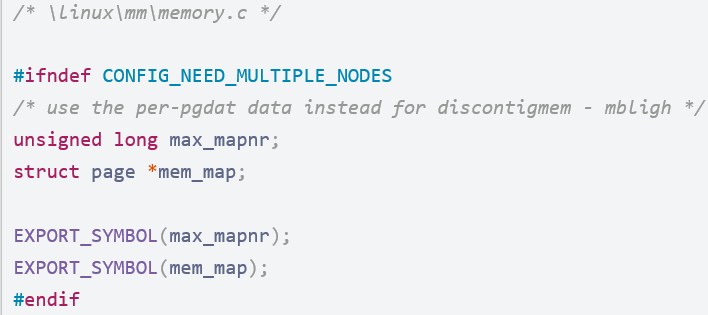
1. 对于一个进程，我们使用task\_struct中的mm\_struct,找到mmap指向的vm\_area\_struct链表，并从中提取用户线性地址，同时将pgd加载入cr3,获得对应的进程页目录。
2. 分解用户线性地址的页目录偏移量，定位进程页目录找到对应的进程页表，同样地用页表偏移量定位进程页表，找到物理内存，并定位到物理偏移量，就能获得物理地址了。

而上半部分则表示了内核地址的映射。

对于32位的系统，通常分配0~3G共3G作为用户空间，3~4G共一G作为内存空间。

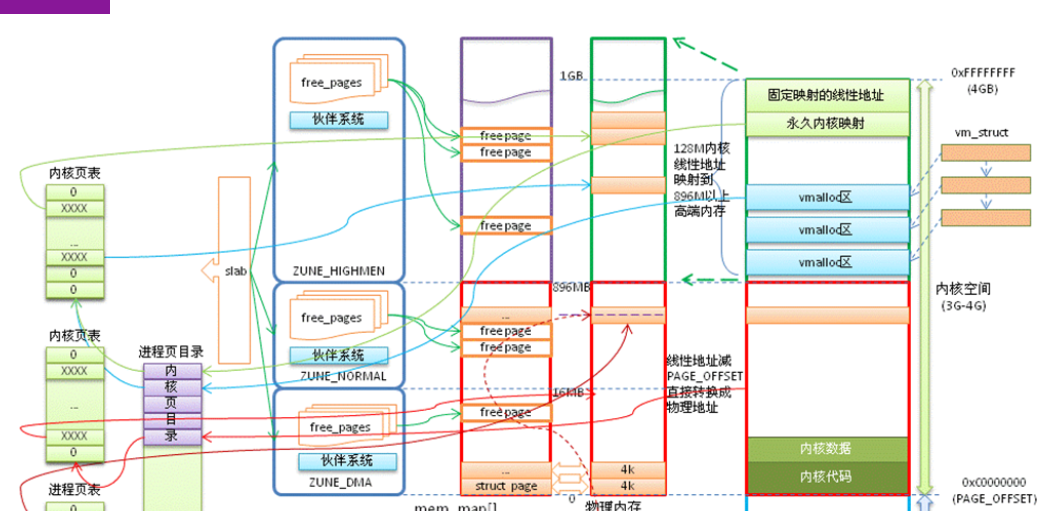
0~16M划分为ZONE\_DMA,一般供专门的I/O设备直接访问内存用

16~896M划分为ZONE\_NORMAL，该区域的物理内存可以被内核直接使用，一般pgd,mem\_map等放置在这个区域



ZONE\_NORMAL支持连续映射 ，线性 地址减去page\_offset就是物理地址。

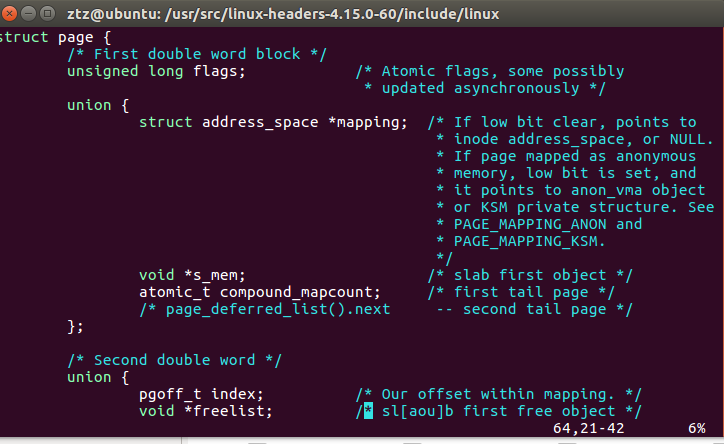
896~结束划分为ZONE\_HIGHMEN,是内核无法直接使用的高端内存。需要kmap()建立映射后才能使用。并且需要vmalloc进行分配后与用户空间相似地寻址，有独立的内核页表。

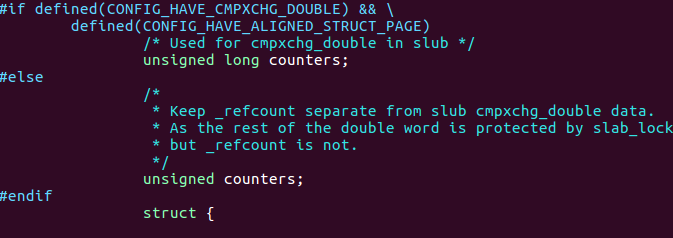


**伙伴系统**

伙伴系统会分配2的幂次个页框，让大小相同且物理连续的块复合，创造大块，有效解决了外部碎片的问题。

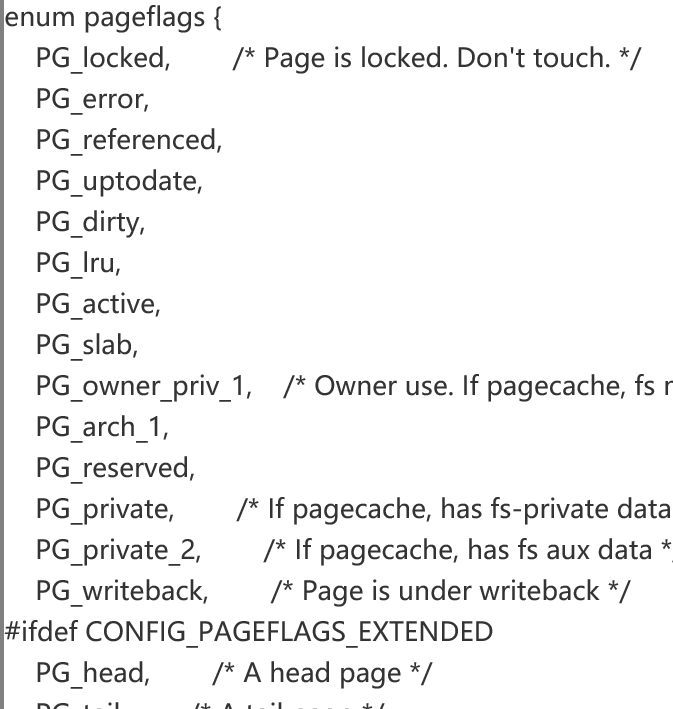
Page结构在mm\_types.h中定义如下：





使用flag表示状态比如是否为脏页，counters表示引用数。

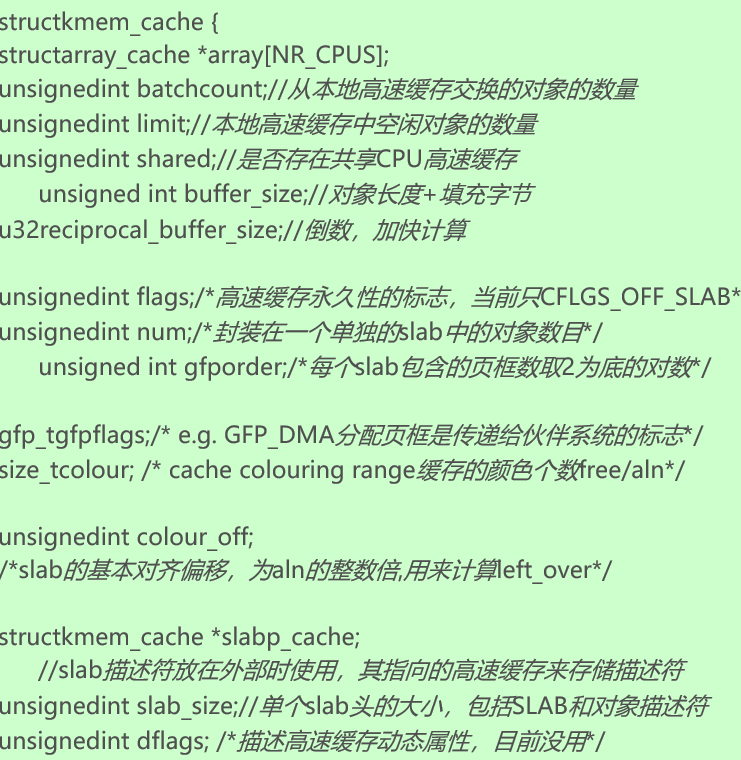
Flag的可能状态：



**Slab机制**

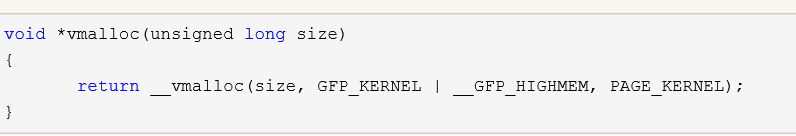
建立在伙伴机制之上，将常用小对象缓存进内存，防止污染伙伴系统。提高了效率，也防止了内部碎片。

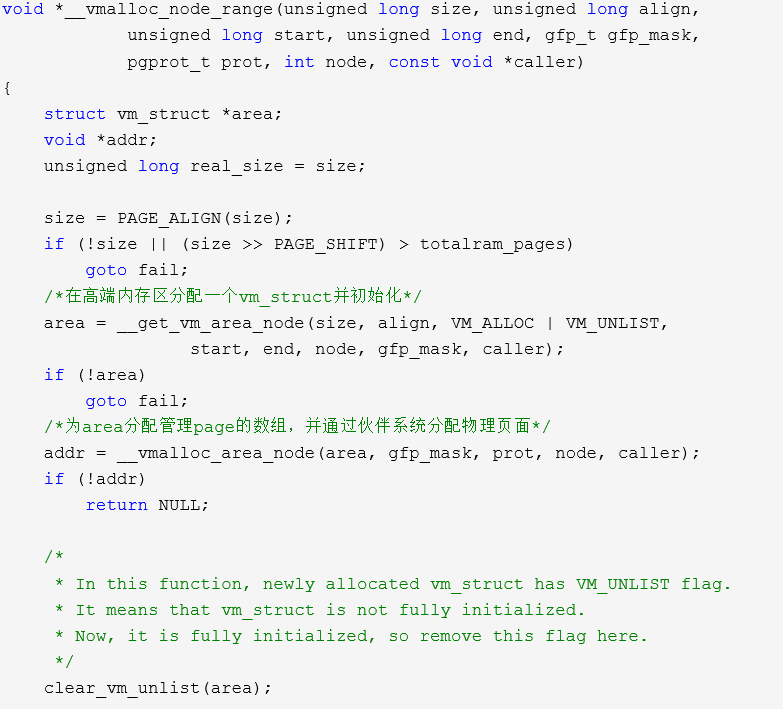
在slab.c中的缓存定义如下



1. 分析图二

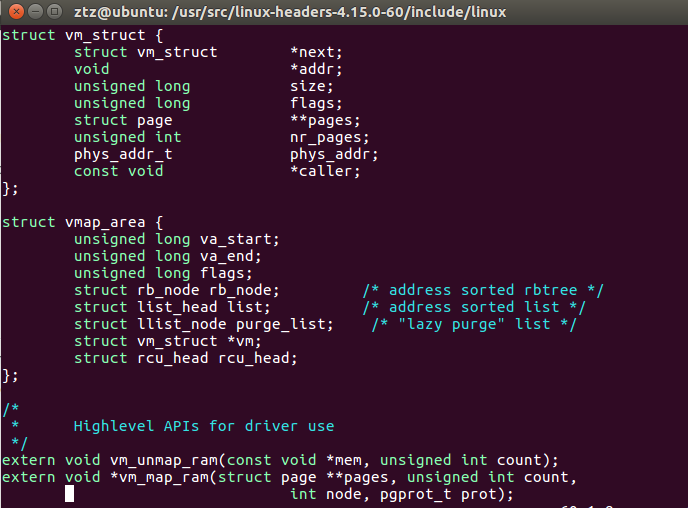
**Vmalloc**





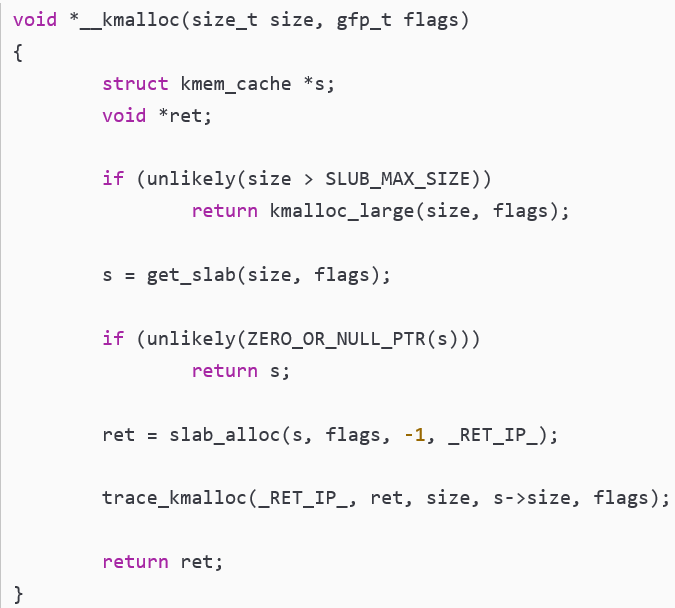
Vmalloc封装了\_\_vmalloc\_node,而\_\_vmalloc\_nod封装了\_\_vmalloc\_node\_range函数。首先对齐size，然后分配vm\_struct。

Vmalloc返回大于size的内存块指针，该内存块在逻辑上连续。



Vmap将vm\_struct挂在链表与红黑树上,维护映射关系。

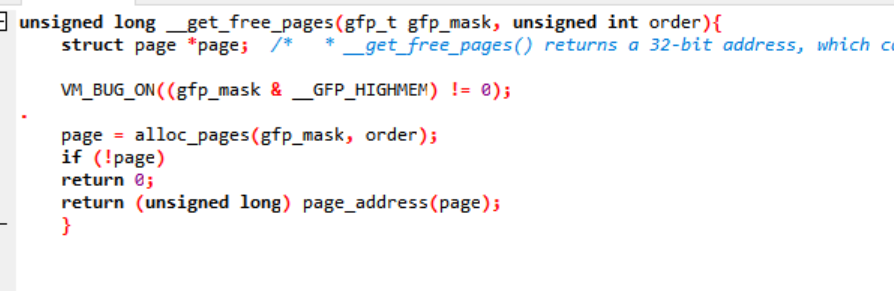
**Kmalloc**



kmalloc基于slab实现，返回指向内存块的指针ret,其大小大于size，在物理内存中连续.受到slab机制的限制，其大小最大为128K.可以修改slab.c中对cache\_size的定义以获得更大页面。

flags可能为GFP\_USER或GFP\_KERNEL等，用于分辨是否会休眠，属于内核还是用户空间.

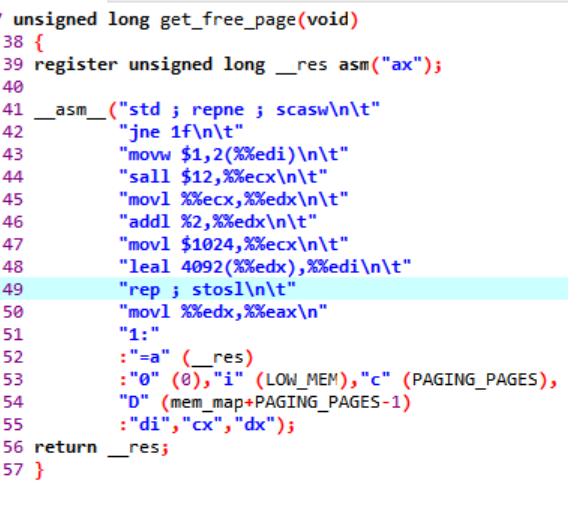
**get\_free\_pages**

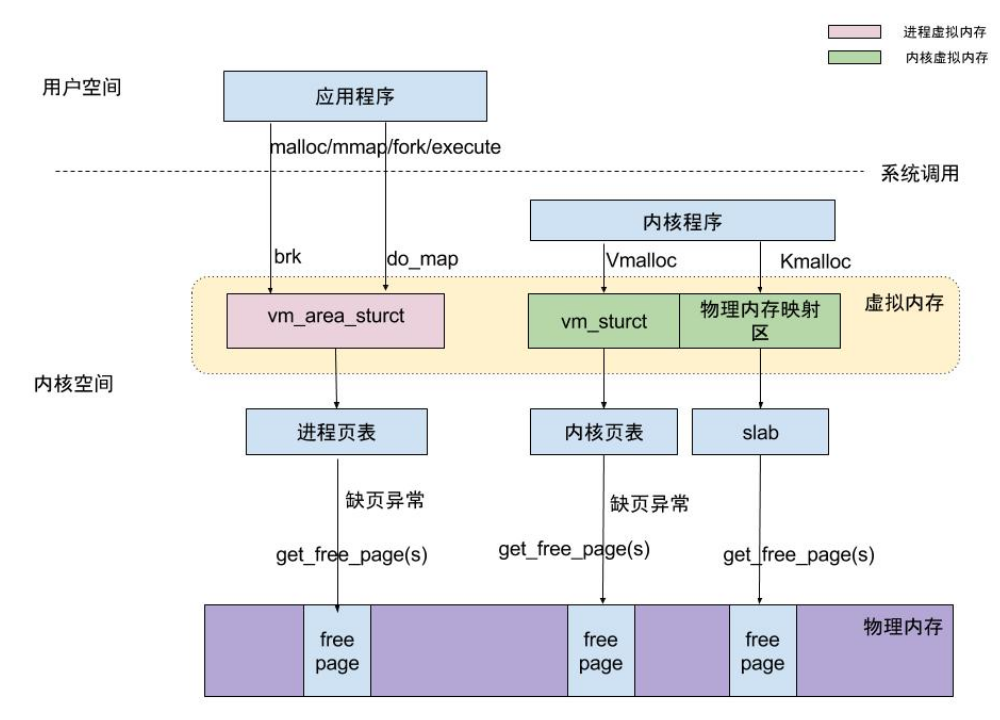
****

Order表示申请2的order次方个页，gfp\_mask 即是kmalloc中的flag,指定了分配方式。调用alloc\_page分配页并返回页地址。

**Get\_free\_page**

该函数描述了如何取得单个页：从mem\_map中寻找空闲页，并返回地址。它申请的是一整页，大小通常为128k.





从这个图我们便能看出这三者的异同与关系：

kmalloc是申请较小的物理连续的内存（与真正的物理地址只差一个偏移），kmalloc通常使用slab机制，并通过调用get\_free\_pages作为底层实现。而vmalloc申请的是连续的虚拟内存，通常较大。相应的也有较大的开销，需要内核页表的映射，效率较低。当发生缺页错误时也需要调用get\_free\_pages进行换页操作。

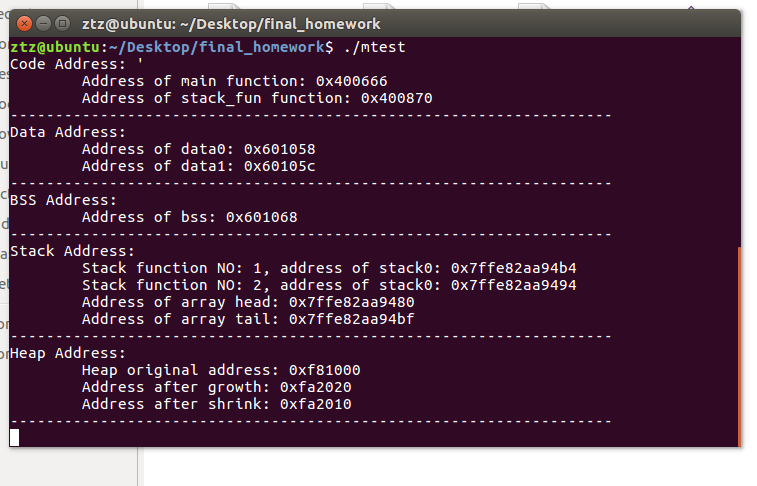
而普通的应用程序发生缺页时调用的也是get\_free-page.

Get\_free\_page调入整页，而\_\_get\_free\_pages课调入2的n次方个页。

Kmalloc可指定是否能够休眠，而vmalloc默认可以休眠。

1. mtest.c

输出如下：

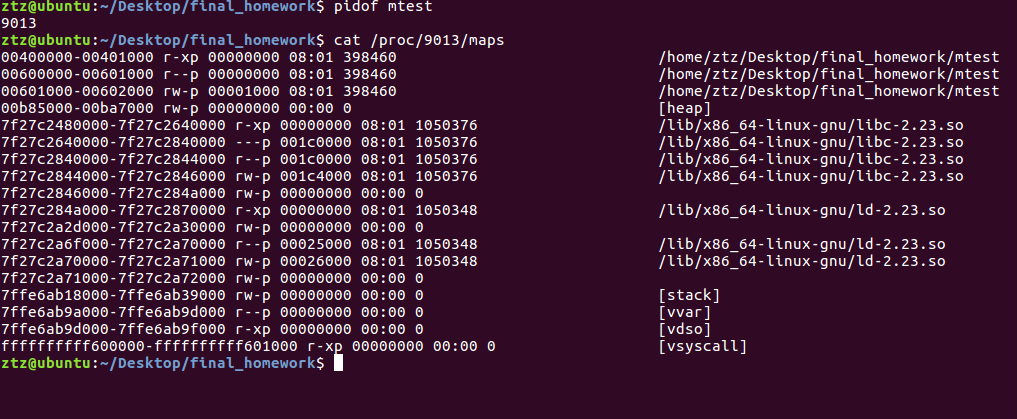


其中BSS为未初始化的数据内存空间。

Data为已经初始化了的数据内存空间。

1. Mtest内存段分析

运行mtest,如下图打印其内存：



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 行  列 | 虚拟地址空间范围  (vm\_start--  vm\_end) | vm\_flags  权限 | vm\_pgoff  偏移 | 设备号 | inode | 文件名 | 该行含义 |
| 1 | 00400000-00401000 | r -xp | 00000000 | 08:01 | 398460 | mtest | Mtest Code  Segment |
| 2 | 00600000-00601000 | r --p | 00000000 | 08:01 | 398460 | mtest | Mtest Data Segment |
| 3 | 00601000-00602000 | rw-p | 00001000 | 08:01 | 398460 | mtest | Mtest BSS Segment |
| 4 | 00b85000-00ba7000 | rw-p | 00000000 | 00:00 | 0 | heap | Heap Segment |
| 5 | 7f27c2480000-7f27c2640000 | r-xp | 00000000 | 08:01 | 1050376 | libc-2.23.so | libc为共享库，此段为  内存映射段。  用于分配公享资源 |
| 6 | 7f27c2640000-7f27c2840000 | ---p | 001c0000 | 08:01 | 1050376 | libc-2.23.so |
| 7 | 7f27c2840000-7f27c2844000 | r--p | 001c0000 | 08:01 | 1050376 | libc-2.23.so |
| 8 | 7f27c2844000-7f27c2846000 | rw-p | 001c4000 | 08:01 | 1050376 | libc-2.23.so |
| 9 | 7f27c2846000-7f27c284a000 | rw-p | 00000000 | 00:00 | 0 |  |
| 10 | 7f27c284a000-7f27c2870000 | r-xp | 00000000 | 08:01 | 1050348 | ld-2.23.so |
| 11 | 7f27c2a2d000-7f27c2a30000 | rw-p | 00000000 | 00:00 | 0 |  |
| 12 | 7f27c2a6f000-7f27c2a70000 | r--p | 00025000 | 08:01 | 1050348 | ld-2.23.so |
| 13 | 7f27c2a70000-7f27c2a71000 | rw-p | 00026000 | 08:01 | 1050348 | ld-2.23.so |
| 14 | 7f27c2a71000-7f27c2a72000 | rw-p | 00000000 | 00:00 | 0 |  |
| 15 | 7ffe6ab18000-7ffe6ab39000 | rw-p | 00000000 | 00:00 | 0 | stack | Stack Segment |
| 16 | 7ffe6ab9a000-7ffe6ab9d000 | r--p | 00000000 | 00:00 | 0 | vvar | 库函数数据映射区 |
| 17 | 7ffe6ab9d000-7ffe6ab9f000 | r-xp | 00000000 | 00:00 | 0 | vdso | 库函数映射区 |
| 18 | ffffffffff600000-ffffffffff601000 | r-xp | 00000000 | 00:00 | 0 | vsyscall | 固定地址映射区 |

可通过cat/proc/device查看设备情况。

1. **问答题**
2. 用户程序的内存分配涉及brk/sbrk和mmap两个系统调用，这两种方式的区别是什么，什么时候用brk/sbrk，什么时候用mmap？

当malloc函数族发现空间不够时，会自动调用sbrk函数，将数据段下界移动扩大空间。也可以传递负的参数缩小空间。malloc族函数的底层实现一般由brk完成，通过设置数据段的末端位置分配内存。

void \*mmap(void \*start, size\_t length, int prot,int flags,int fd, off\_t offset);

而mmap为调用者在虚拟内存中映射一块连续区域。它分配的虚拟地址与磁盘空间是一一对应的。它所分配的是vm\_area\_struct，由上文的分析可知它同时用链表与红黑树两种逻辑维护。它用文件指针fd

找到它的文件描述符，并加入到struct file中。这样，文件地址与虚拟内存和磁盘空间的映射关系就建立好了。

2.应用程序开发时，为什么需要用标准库里的malloc而不是直接用这些系统调用接口？malloc额外做了哪些工作？

若申请内存小于128K(小于一页)，malloc函数会使用brk分配内存，设置栈空间的高位。而若申请内存大于128K,则调用mmap函数。因为mmap函数的内存映射到整页上，可以调用free()独立释放，不会产生内碎片。

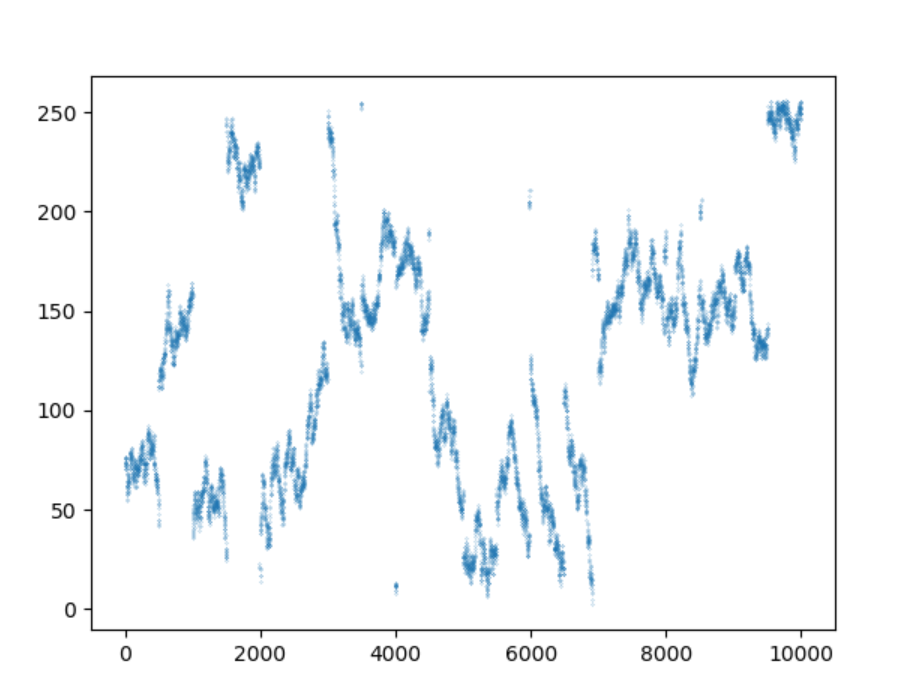
而brk则要等整个页一起释放了才能释放，会产生内部碎片。

1. malloc的内存分配，是分配的虚拟内存还是物理内存？两者之间如何转换？

Malloc分配的是虚拟内存。通过建立页表，使用MMU，进行映射转换。一般使用四级或两级的页表。

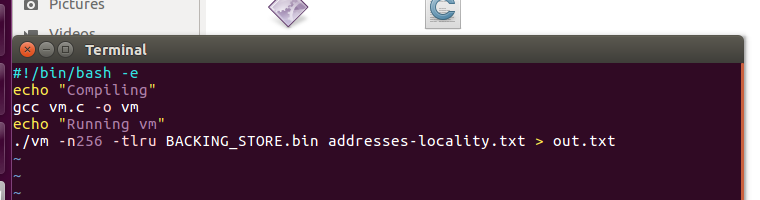
附加题 tracer与局部性验证

产生图片明显具有局部性，在一段时间内在一个范围内波动。



导入addresses-locality.txt进行观察

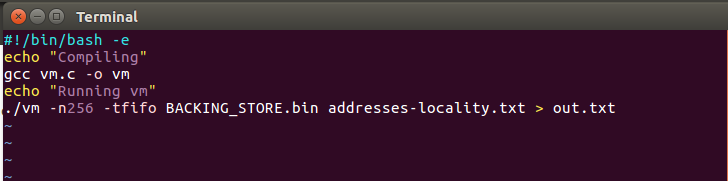
LRU:

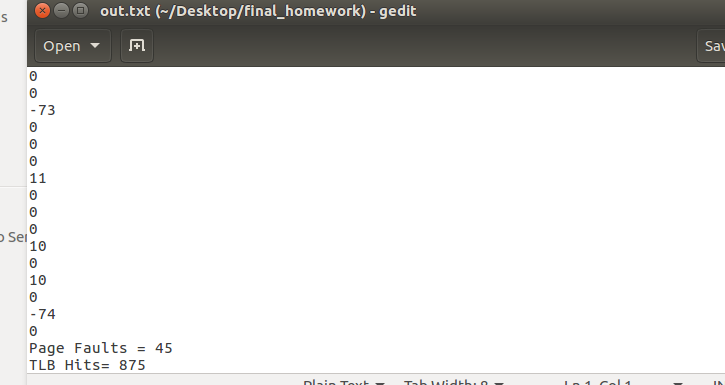




Page fault =45, TLB HIT =896

FIFO:





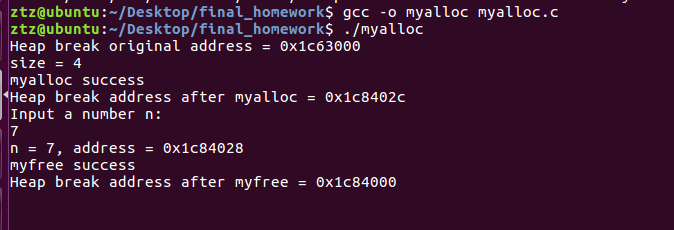
Page Fault =45, TLB HIT =875

(文件夹里附有fifo\_out与lru\_out)

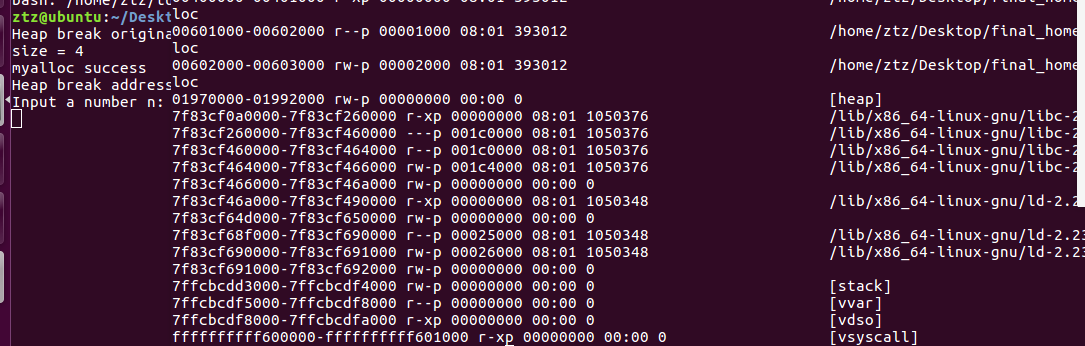
总结起来lru的表现要强与FIFO，局部性的优势得以体现。

附加题 myalloc

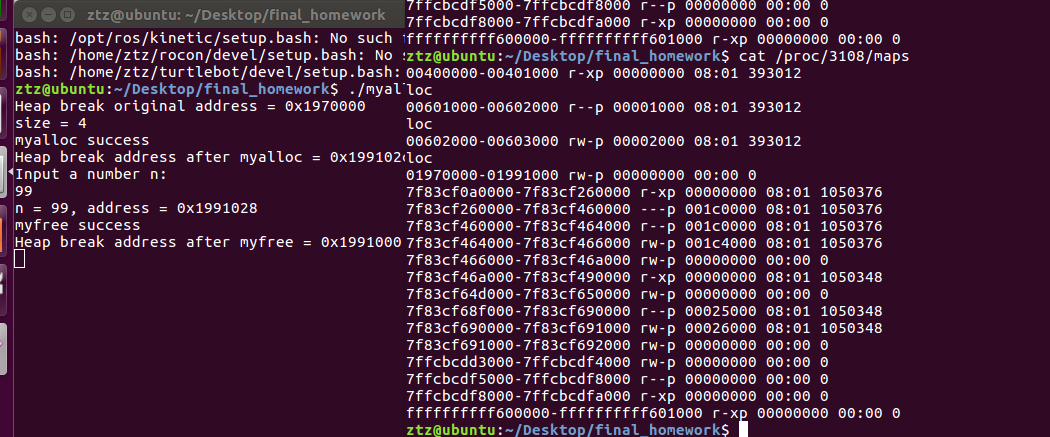
运行：



在输入n之前：



在输入n之后：



可以发现堆内存（heap）从01970000-01992000变为0197000-1991000这是因为调用了myfree函数，释放了部分的内存。