**操作系统大作业2**

提交截止日：12月18日零时

1. **总体要求**

在github上创建os-assignment2项目，项目包括2个题目的结果:

1. 虚存管理模拟程序
2. Linux内存管理实验程序

该目录下，同时存放1个pdf/word文件，作为实验报告。

注：附加题的分数单独计算，累加到正常分数上面，最后总分不超过100分。

**2. 虚存管理模拟程序，50分+10分（附加题）**

1. Chapter 10. Programming Projects: Designing a Virtual Memory Manager（OSC 10th ed.）
2. 保持为vm.c，使用如下测试脚本test.sh，进行地址转换测试，并和correct.txt比较

|  |
| --- |
| #!/bin/bash -e  echo "Compiling"  gcc vm.c -o vm  echo "Running vm"  ./vm BACKING\_STORE.bin addresses.txt > out.txt  echo "Comparing with correct.txt"  diff out.txt correct.txt |

注：本小题不要求实现页置换（Page Replacement），TLB用简单的FIFO策略。30分。

**编程思路：**  
include头文件

定义全局变量

声明函数

main函数

声明各种要用到的变量

读取BACKING\_STORE.bin

读取addresses.txt

初始化页表（使用-1初始化页表，-1代表空）

初始化TLB（使用-1初始化TLB， -1代表空）

while(读取addresses.txt) 循环一千次：

读取虚拟地址，计算页码和偏移

检查tlb（FIFO）

若tlb未命中

{

在页表项查找，若页表项无效，即读出的帧码为-1

{

调页，读取BACKING\_STORE.bin的一页到物理内存的一帧

}

若页表项有效

{

读取帧码

计算物理地址

读取物理内存

}

更新tlb

}

若tlb命中

{

读取帧码

计算物理地址

读取物理内存

}

输出存储的数值

格式为printf("%d\n", Value);

输出页错误数量，tlb命中数量，关闭文件指针（一开始打开的那个地址文件的）

printf("Page faults = %d\n",page\_fault\_counter)；

printf("TLB hits = %d\n",tlb\_hit);

关闭函数指针

return 0;

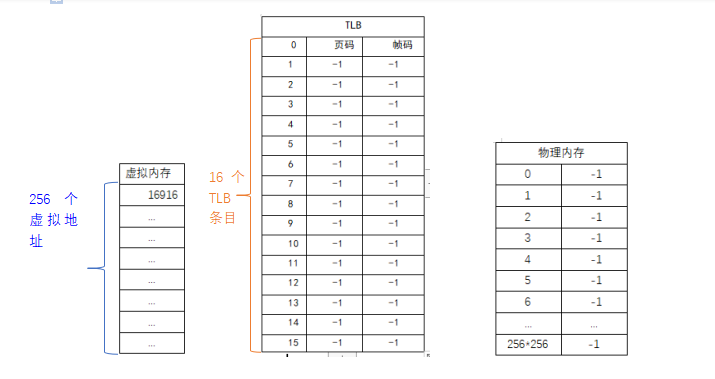
}

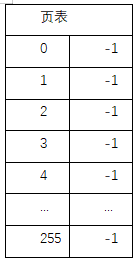
TLB[16][2]被初始化为-1，-1代表该TLB条目为空或无效

页表page\_table[256] 被初始化为-1，表示该缺页

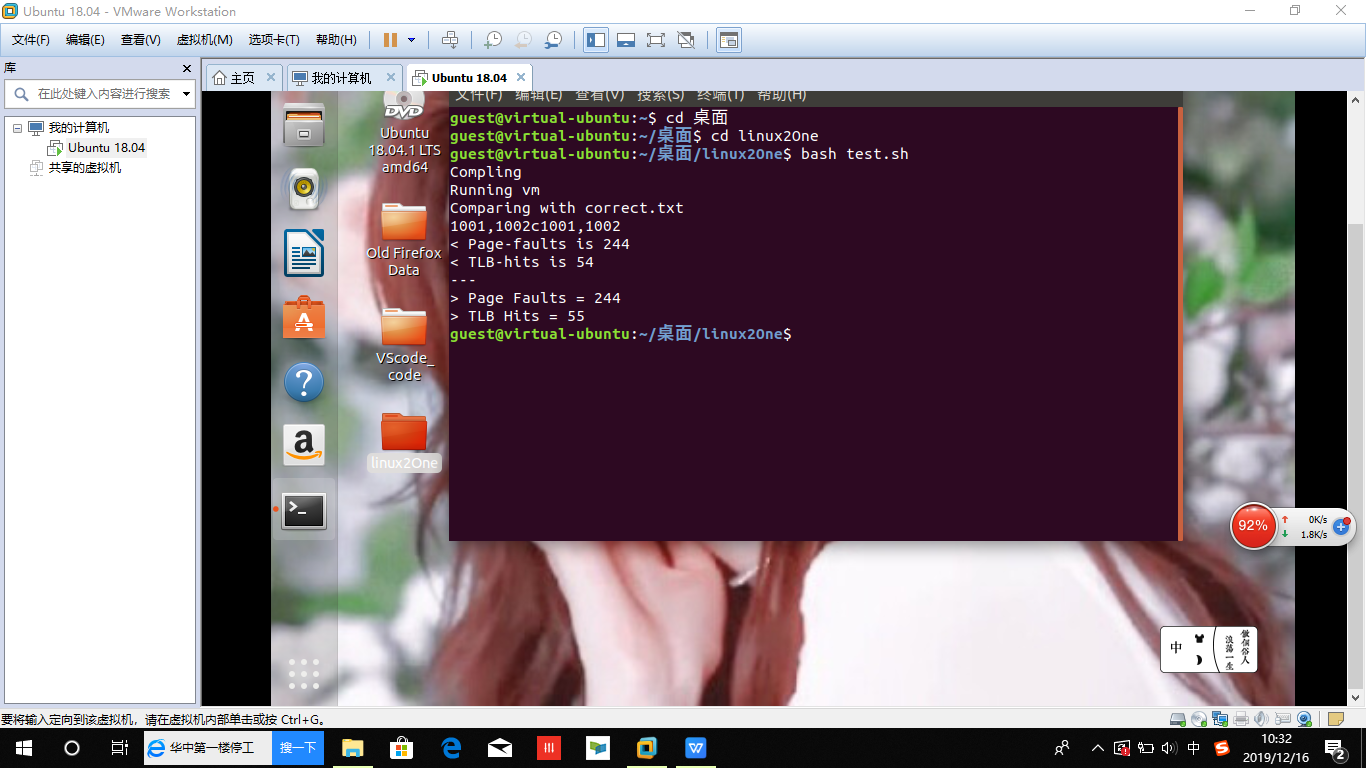
物理内存memory[256\*256]

采用FIFO策略替换TLB条目



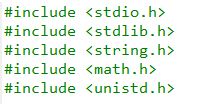


**程序运行结果截图：**

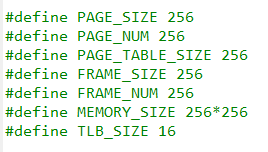


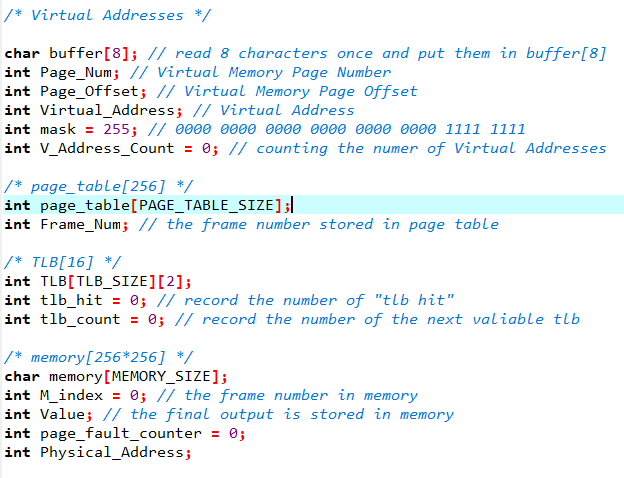
**代码部分：vm1.c**

include头文件：

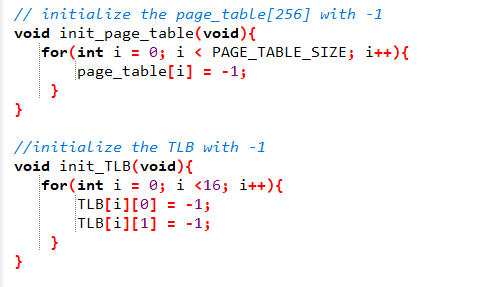


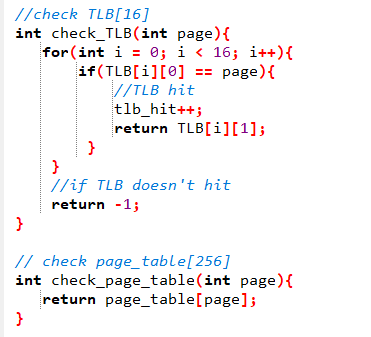
定义常量：

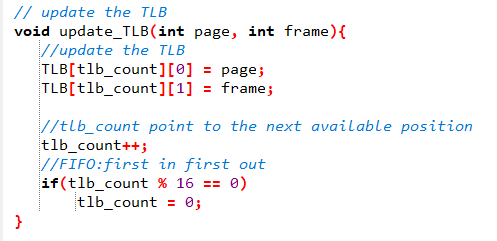


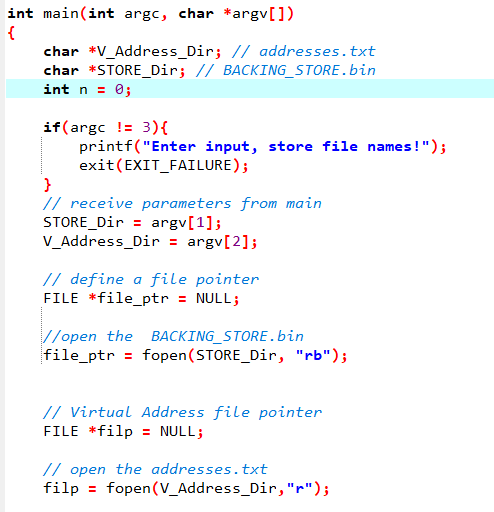
定义全局变量：  


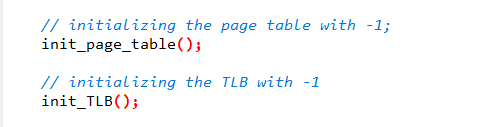
定义函数：

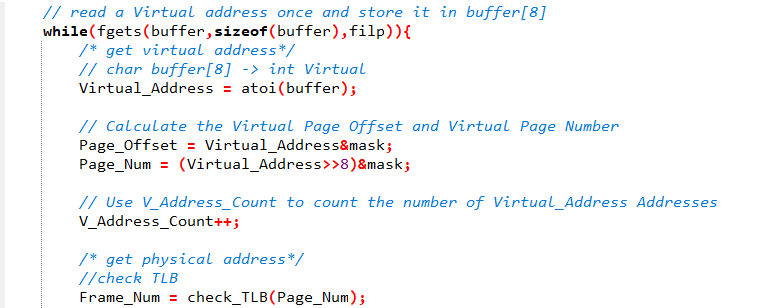


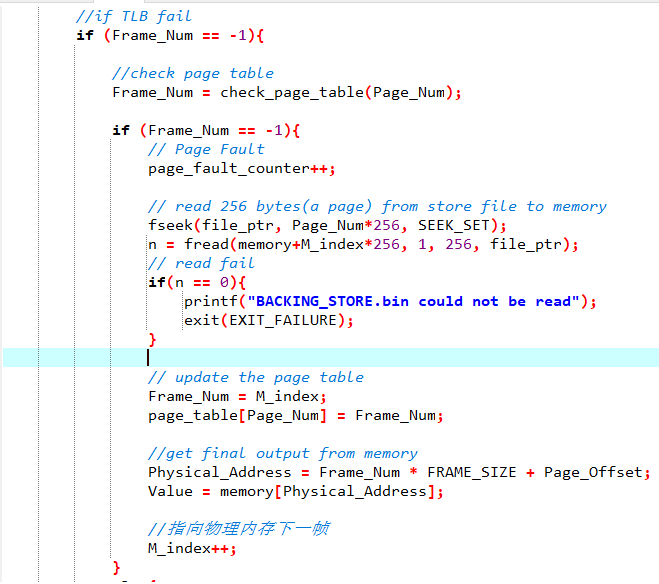


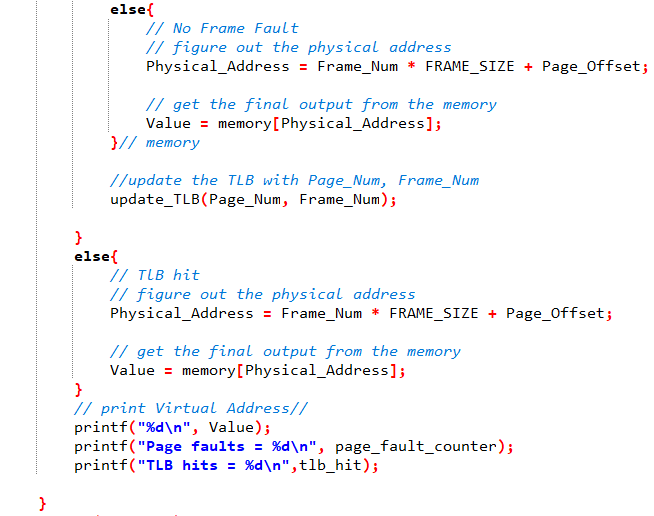


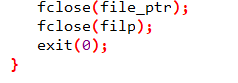
main函数：  












1. 实现LRU的TLB，8分。
2. 实现基于LRU的Page Replacement，8分。
3. 代码可读性，4分。
4. 使用FIFO和LRU分别运行vm（TLB和页置换统一策略），打印比较Page-fault rate和TLB hit rate，给出运行的截屏。

**编程思路：**

定义头文件

定义全局变量

声明函数

main函数

声明各种要用到的变量

接收BACKING\_STORE.bin

接收addresses.txt

接收内存大小

接收页置换策略

读取BACKING\_STORE.bin

读取addresses.txt

初始化页表（使用-1初始化页表，-1代表空）

初始化TLB（使用-1初始化TLB， -1代表空）

初始化time\_counter[256](这个数组每个元素是memory中每一页使用的时间计数器，采用LRU策略)

while(读取addresses.txt) 循环一千次：

读取虚拟地址，计算页码page和偏移offset

检查tlb

调用update\_time\_counter( page ), 使time\_counter[page]=0,而其他time\_counter[i]+1来标记page为最近使用的页

若tlb未命中

{

在页表项查找page，若页表项无效，即读出的帧码为-1

{

调页，读取BACKING\_STORE.bin的一页到物理内存的一帧

更新页表

计算物理地址

读取物理内存

若物理内存还有空闲帧，M\_index指向下一个空闲帧

若是无空闲帧{

若替换算法为LRU

调用lru\_replace\_page( )函数， 得到下一个的用于页置换的帧号M\_index

若替换算法为FIFO

M\_index = (M\_index + 1) % MEMORY\_SIZE;

如果是页置换，释放free对应页表项，使该被置换页表项的帧码为-1（使无效）

并使对应的time\_counter[page] = -1（使无效）

如果是页置换，释放free对应TLB项（使无效）

}

}

若页表项有效， 读取帧码

{

计算物理地址

读取物理内存

}

更新TLB（分为LRU和FIFO两种更新方法）

}

若TLB命中，读取帧码

{

计算物理地址

读取物理内存

}

输出存储的数值 printf("%d\n", Value);

输出页错误数量，tlb命中数量，关闭文件指针（一开始打开的那个地址文件的）

printf("Page faults = %d\n",page\_fault\_counter)；

printf("TLB hits = %d\n",tlb\_hit);

关闭函数指针

return 0;

}

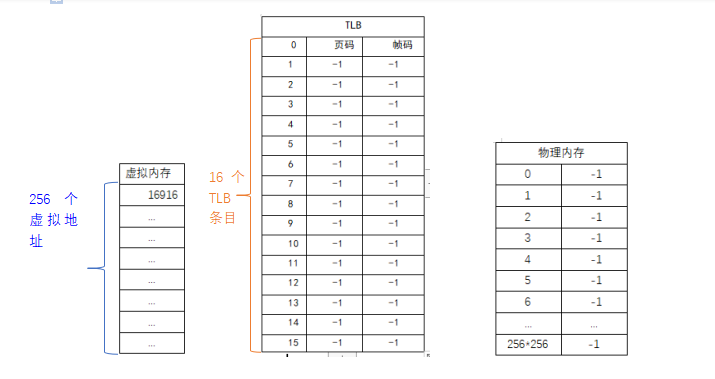
TLB[16][2]被初始化为-1，-1代表该TLB条目为空或无效。

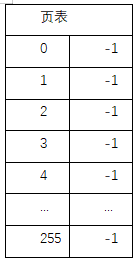
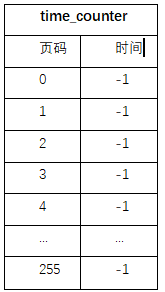
页表page\_table[256] 被初始化为-1，表示该缺页。

time\_counter[256]被初始化为-1，表示没有页面被读取。

当替换算法为LRU时，使用time\_counter[256]来记录每一页的使用情况。

物理内存定义为memory[256\*256]，当用户定义物理内存大小MEMORY\_SIZE为256时，使用全部的memory[256\*256]，当用户定义物理内存大小为128时，仅使用memory[256\*256]的前128\*256的空间，即模拟物理内存为memory[128\*256]。

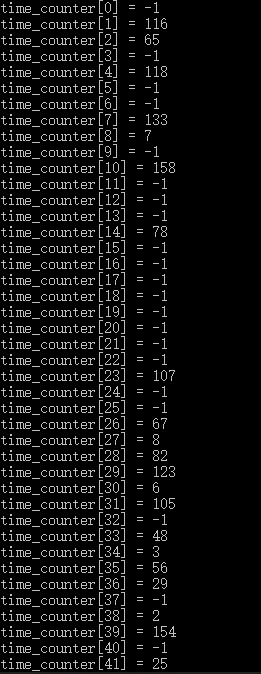
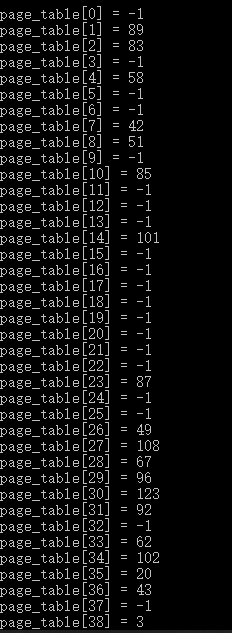


替换算法为LRU，内存大小为128时，time\_counter[256]最终结果输出部分截图和page\_table[256]的最终结果输出部分截图如下图所示：

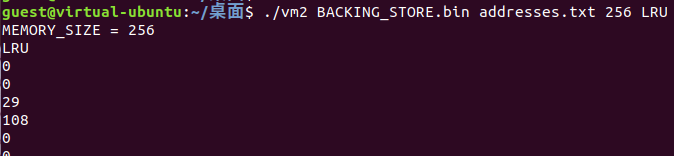
time\_counter数组中，time\_counter[ i ] = -1表示：第i页不在物理内存中； time\_counter数组中的最大值为最近最少读取的页； time\_counter数组中的time\_counter[ i ] = 0 表示最近读取的页。

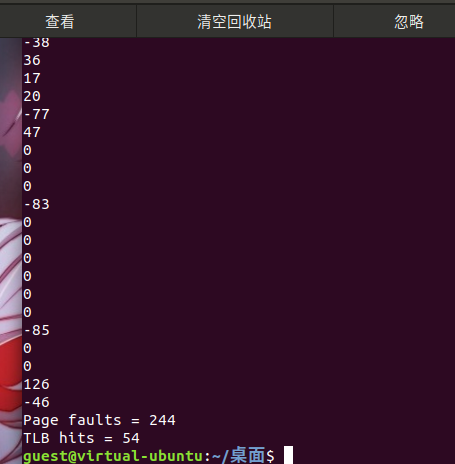
页表page\_table数组中，page\_table[ i ] = -1表示：第i页不在物理内存中。

**代码运行结果截图：**

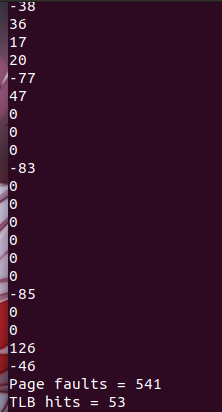
1、



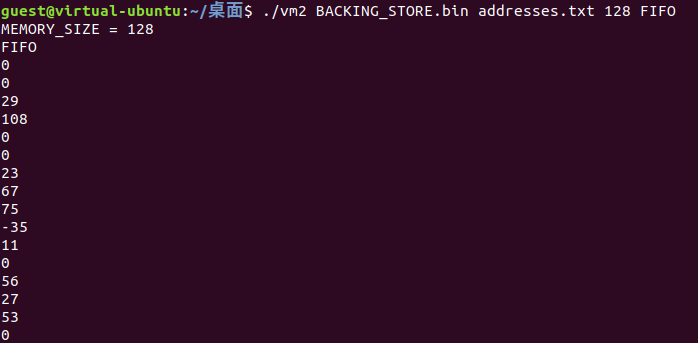


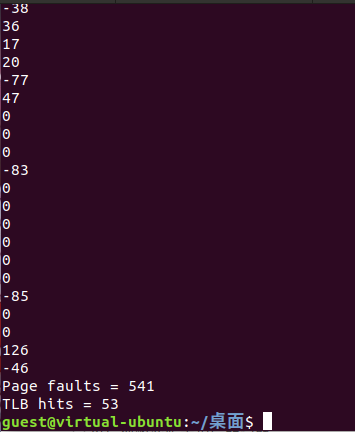
2、



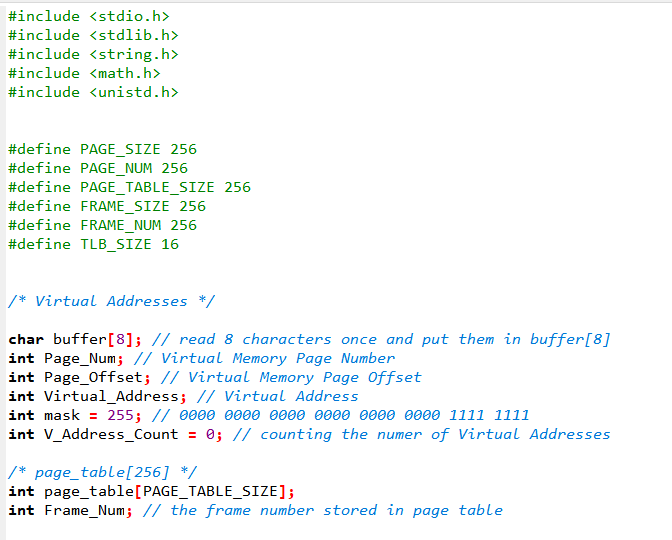


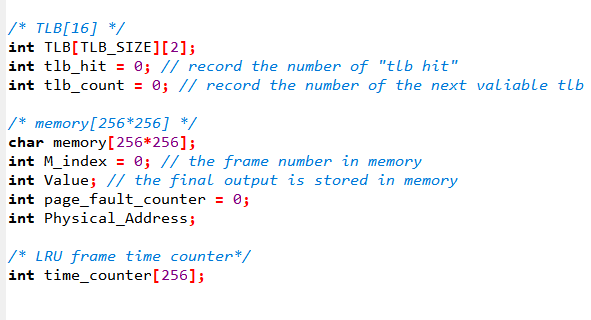
3、

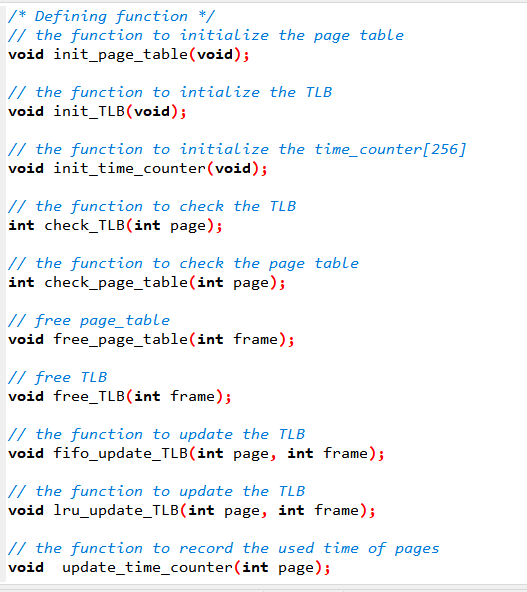




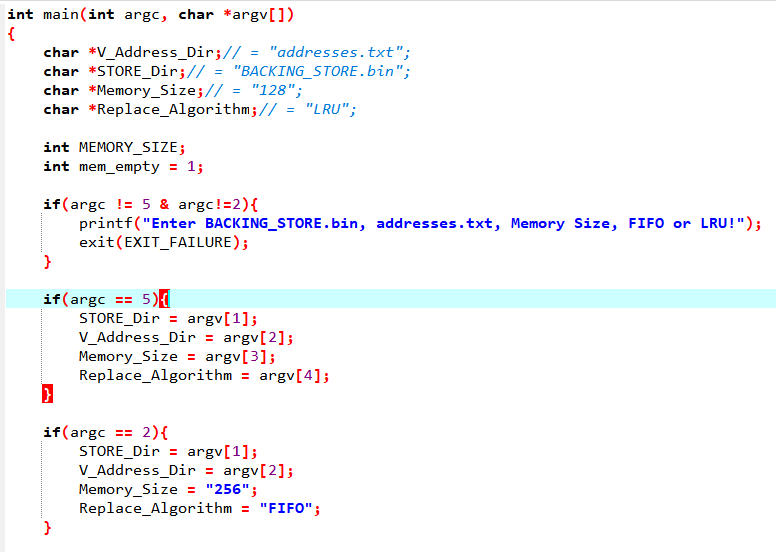
**程序代码截图：**

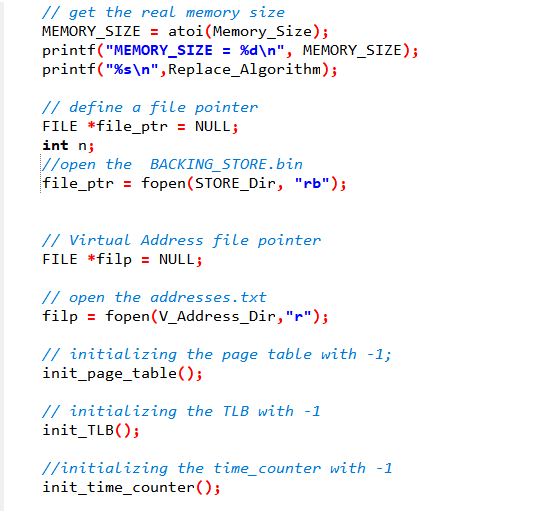


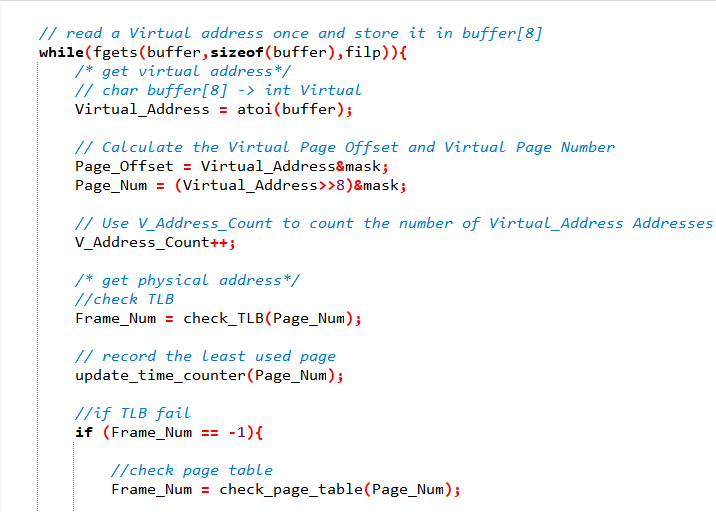


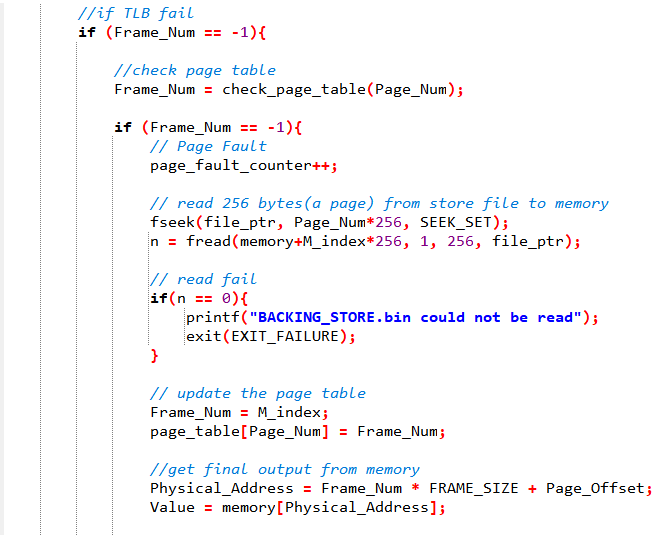


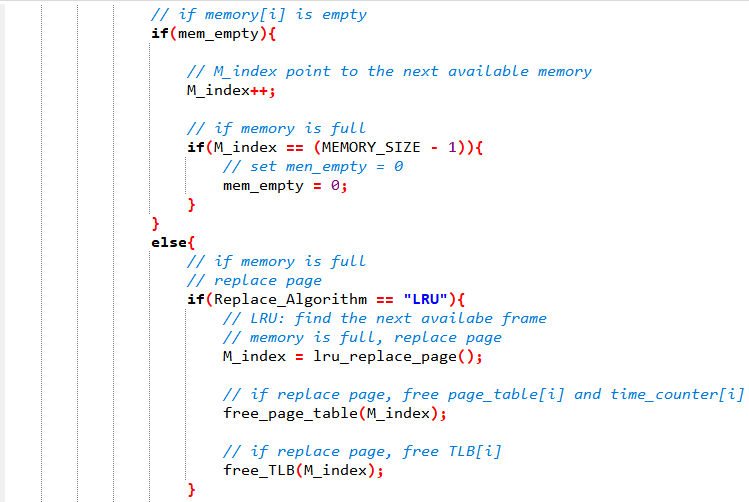


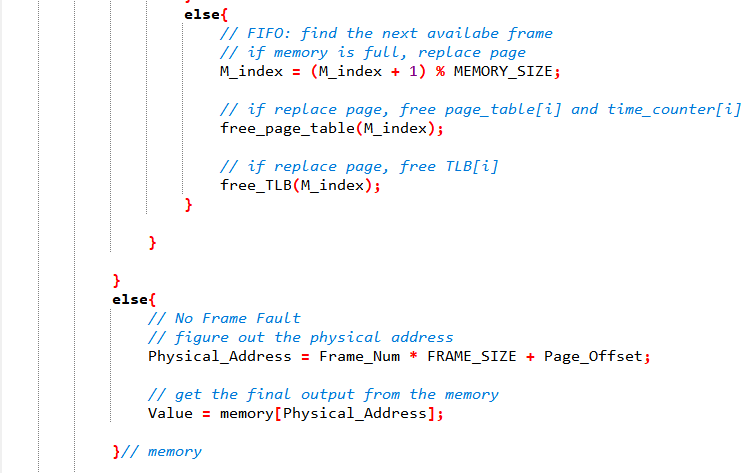


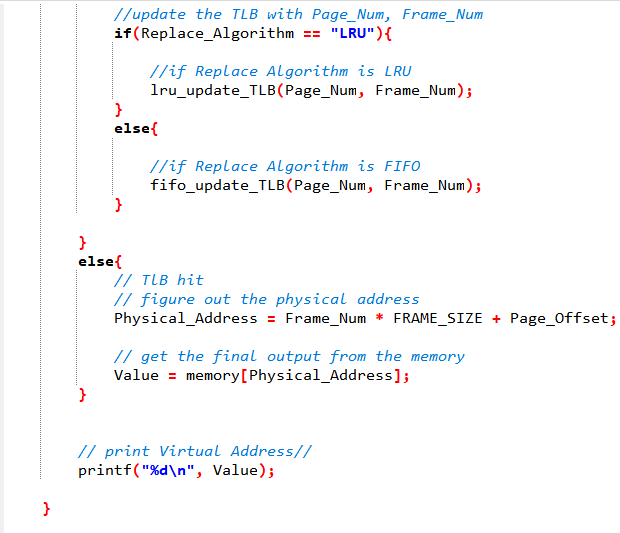


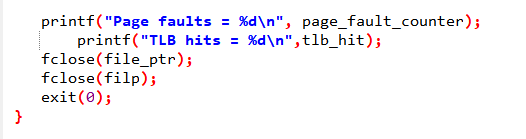


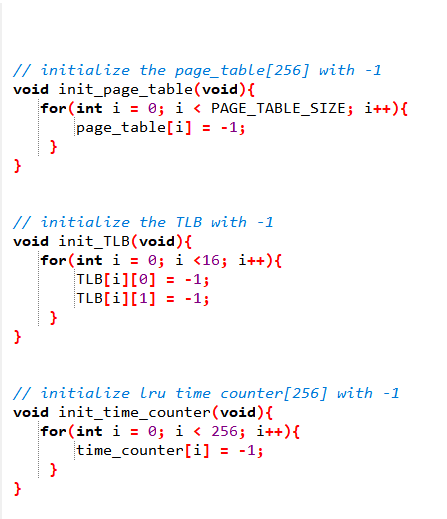


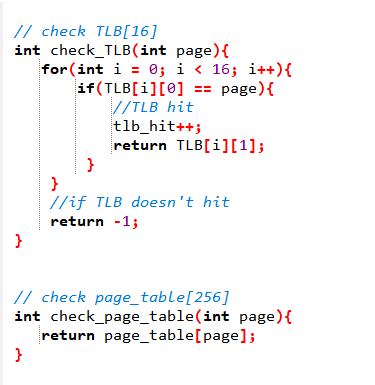


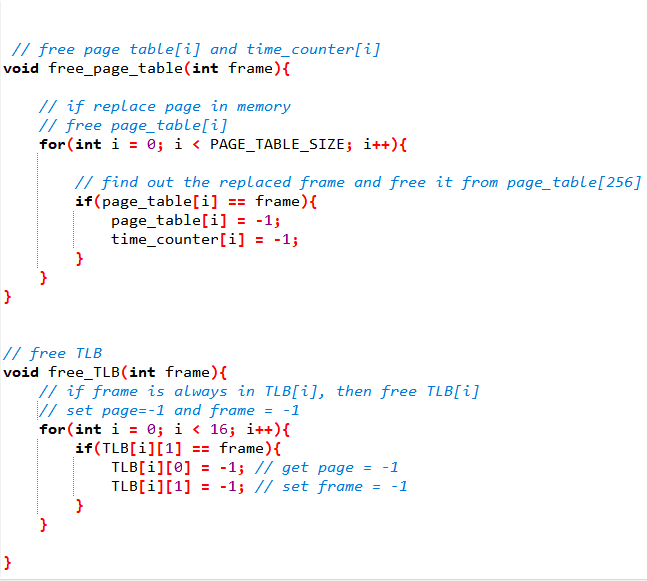


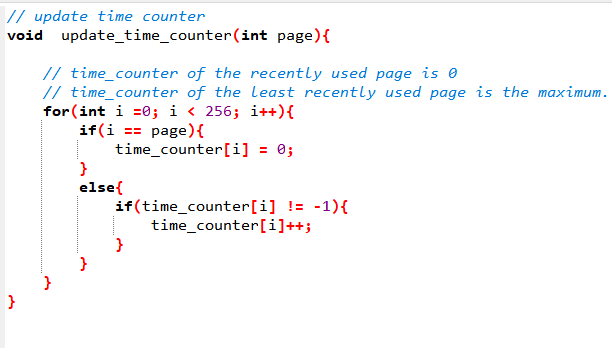


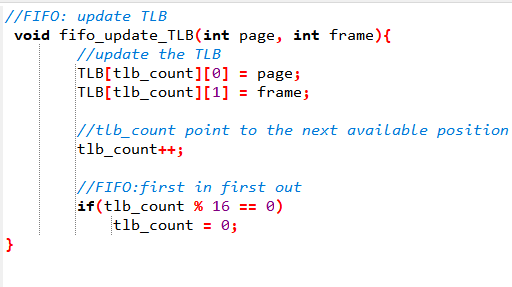


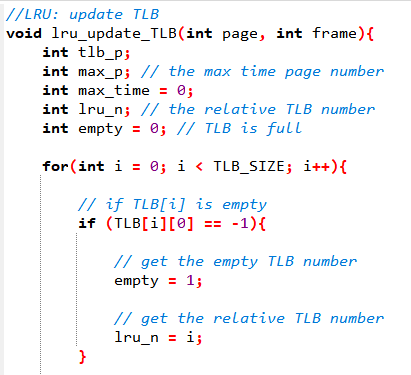


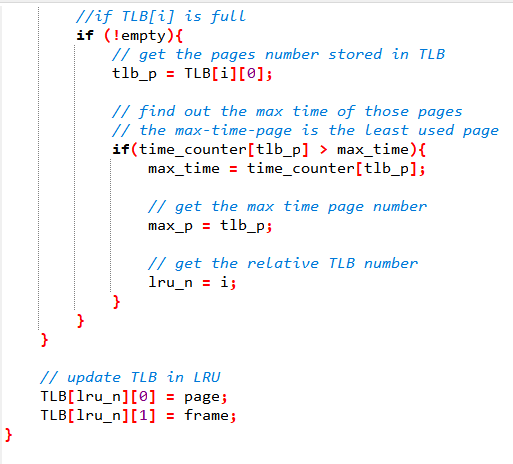


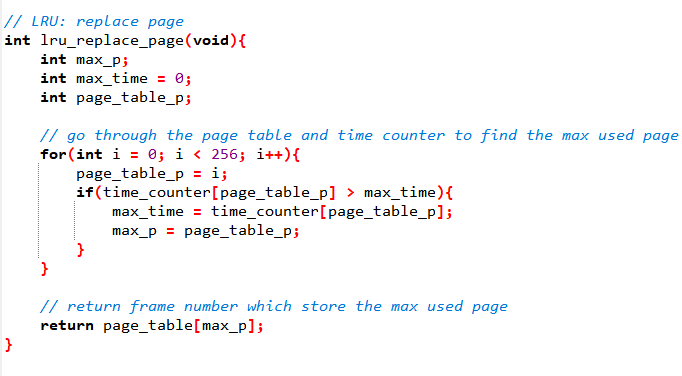












1. （**附加题10分**）编写一个简单trace生成器程序（可以用任意语言，报告里面作为附件提供），运行生成自己的addresses-locality.txt，包含1万条访问记录，体现内存访问的局部性（参考Figure 10.21, OSC 10th ed.），绘制类似图表，表现内存页的局部性访问轨迹。然后以该文件为参数运行vm，比较FIFO和LRU策略下的性能指标，最好用图对比。给出结果及分析。
2. **Linux内存管理实验，50分+10分（附加题）**

阅读Linux内存管理相关代码片段，提供程序和阅读报告，描述关键数据结构中和内存相关的成员的意义，以及指针指向关系。涉及的数据结构包括（但不限于）task\_struct，mm\_struct, vm\_area\_struct, vm\_operations\_struct, page等

1. 分析图1（注：图1是2级页表，对应于IA-32位系统），解释图中**每一类方框**和箭头的含义，在代码树中寻找相关数据结构片段，做简单解释。30分。
2. 参考图2解释内核层不同内存分配接口的区别，包括\_\_get\_free\_pages，kmalloc，vmalloc等，3分。
3. 参考[Anatomy of a Program in Memory](https://manybutfinite.com/post/anatomy-of-a-program-in-memory/)和[User-Level Memory Management](https://linuxdevices.org/ldfiles/misc/Linux_Programming_by_Example_ch03.pdf)中例程，写一个实验程序mtest.c，生成可执行程序mtest；打印代码段、数据段、BSS，栈、堆等的相关地址；需要创建自己的例子，不允许简单照搬，8分。
4. 参考[How The Kernel Manages Your Memory](https://manybutfinite.com/post/how-the-kernel-manages-your-memory/)，通过/proc/pid\_number/maps，分析mtest各个内存段（参考[链接](https://blog.csdn.net/lijzheng/article/details/23618365)）。绘制图表，解释输出的每一段的各种属性，包括每一列的内容。为了让mtest程序驻留内存，可以在程序末尾加上长时睡眠，并将mtest在后台运行，即./mtest & 6分。
5. 参考[A Malloc Tutorial](https://wiki-prog.infoprepa.epita.fr/images/0/04/Malloc_tutorial.pdf)以及相关资料（如[链接](https://blog.csdn.net/gfgdsg/article/details/42709943)）回答以下问题：3分
6. 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？
7. 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？
8. 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？
9. （**附加题，10分**）模仿malloc接口，实现一对简单的函数，命名为myalloc/myfree，实现堆上的动态内存分配和释放，并提供测试函数。相关代码以myalloc.c文件提供在项目目录下面。在自己的机器上进行实验，观察随着malloc/free的行为，/proc/pid\_number/maps中如何反映堆内存的变化情况，给出截屏和解释。实现基本功能5分，在内存块管理方面进行专门优化5分。

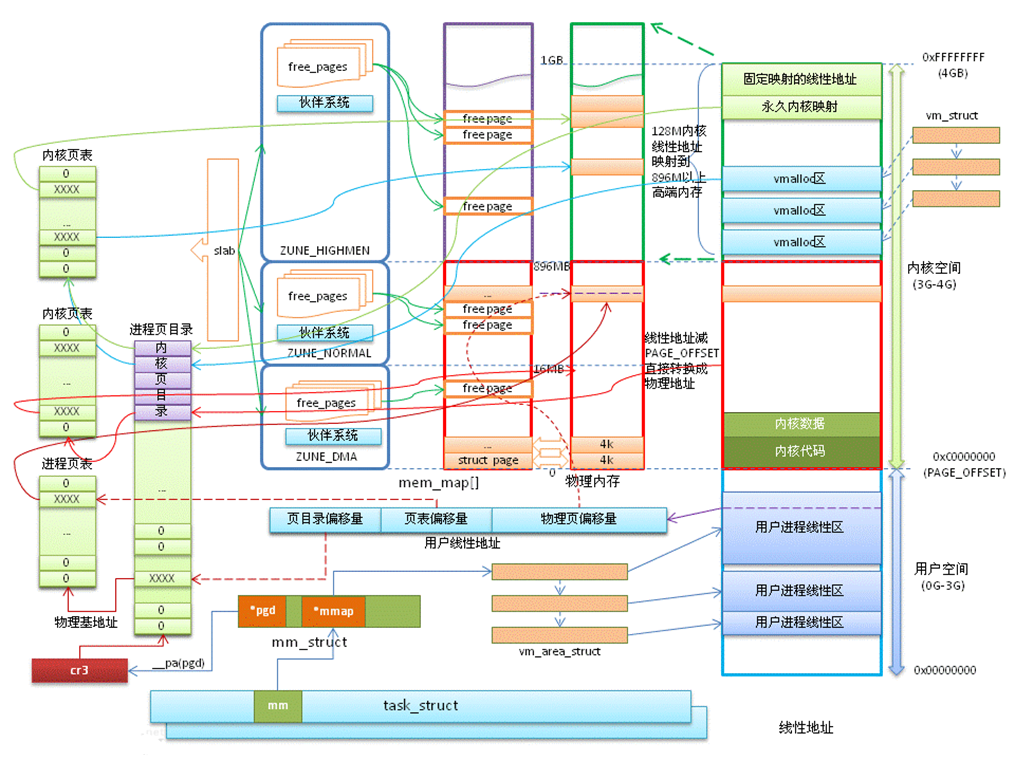
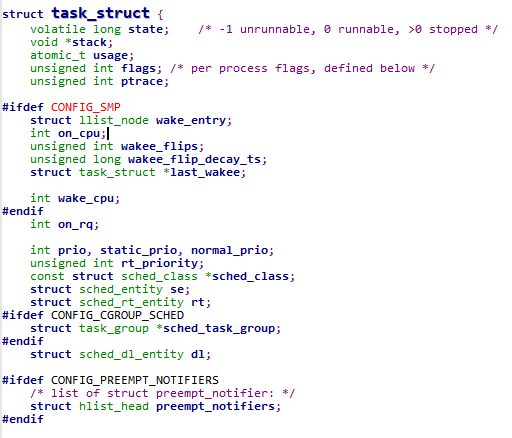
****

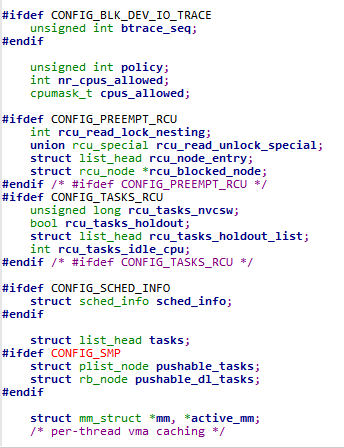
图1. Linux内核内存管理示意图（IA\_32）

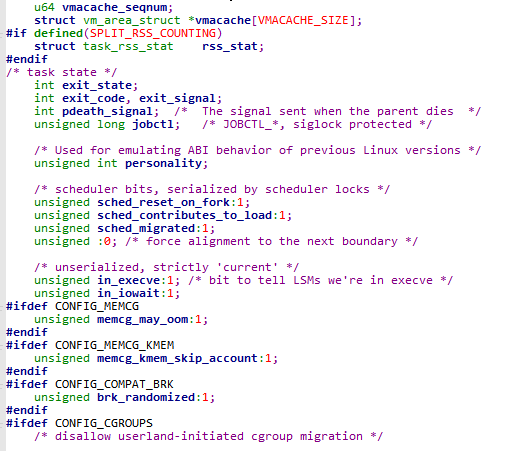
、

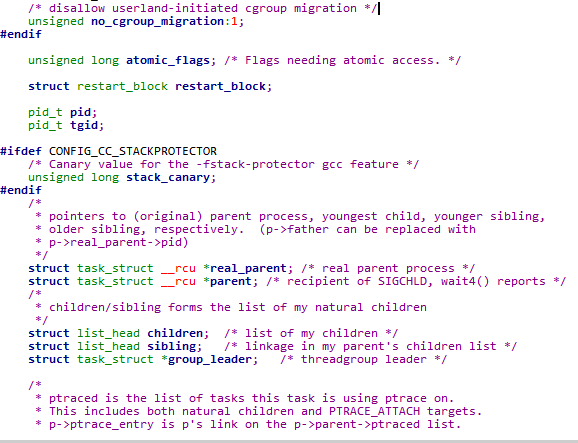
1、分析图1（注：图1是2级页表，对应于IA-32位系统），解释图中**每一类方框**和箭头的含义，在代码树中寻找相关数据结构片段，做简单解释。30分。

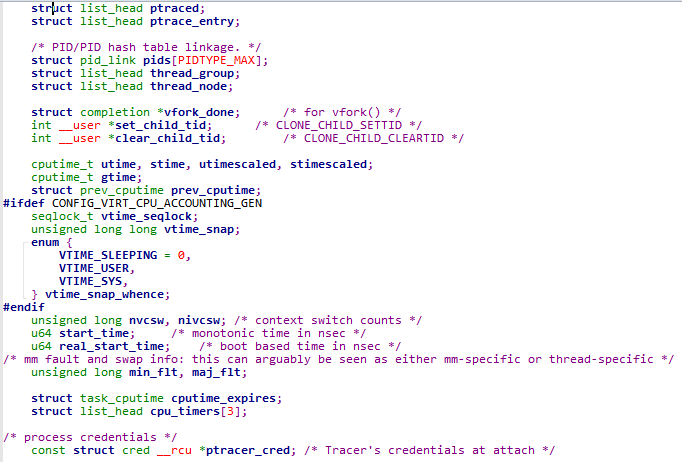
**task\_struct :** Linux 内核通过一个被称为进程描述符的task\_sruct结构体来管理进程，这个 task\_sruct结构体包含了一个进程所需的所有信息。

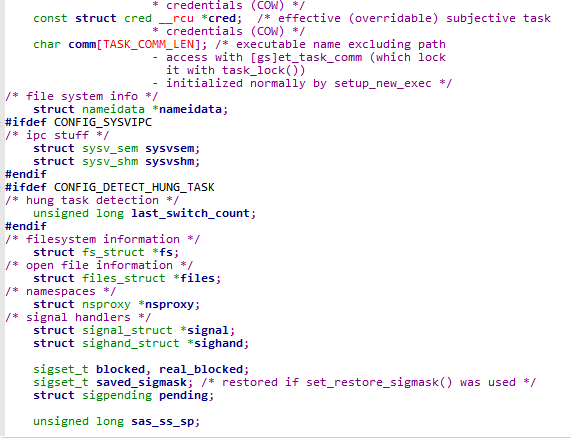


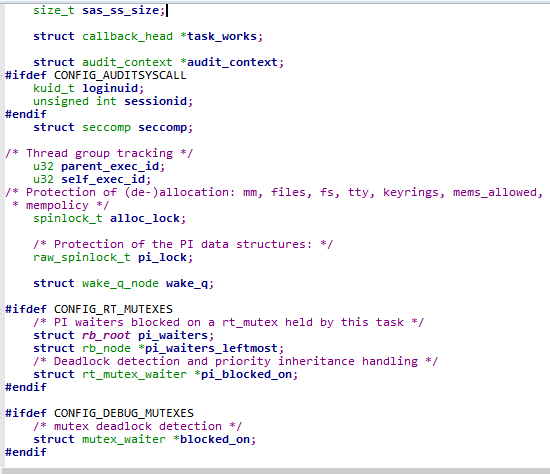


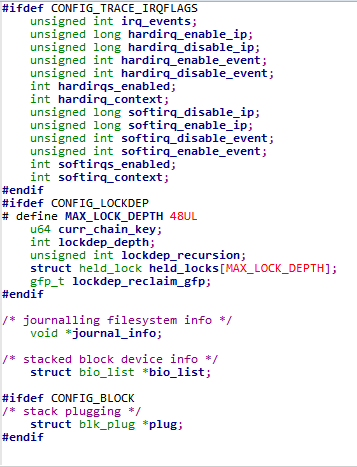


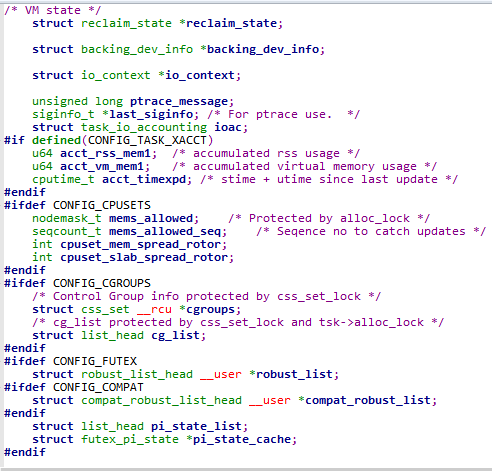


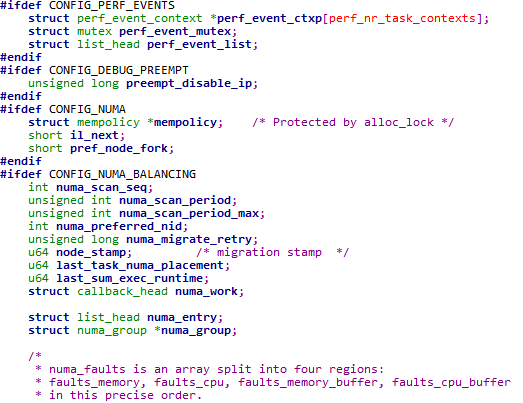


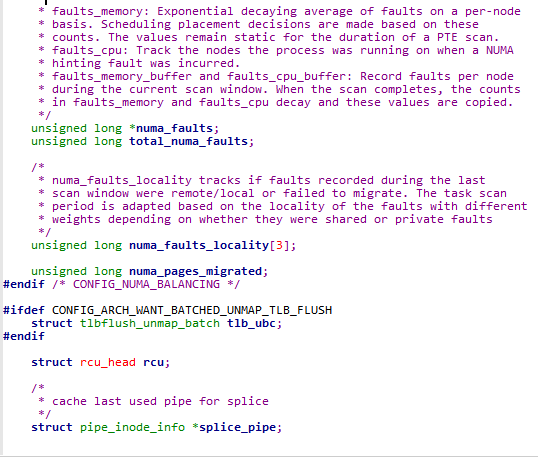


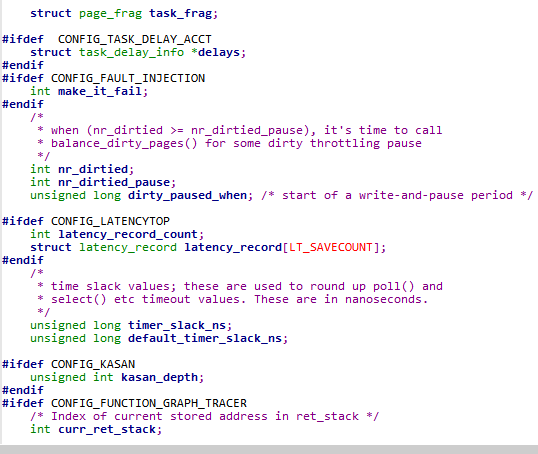


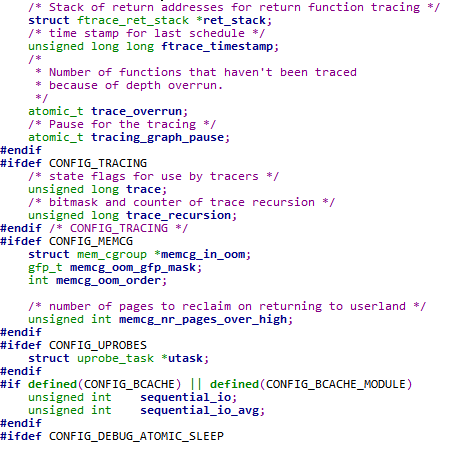


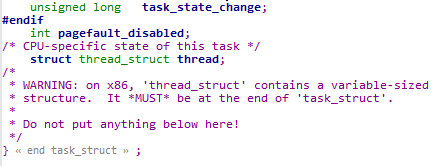








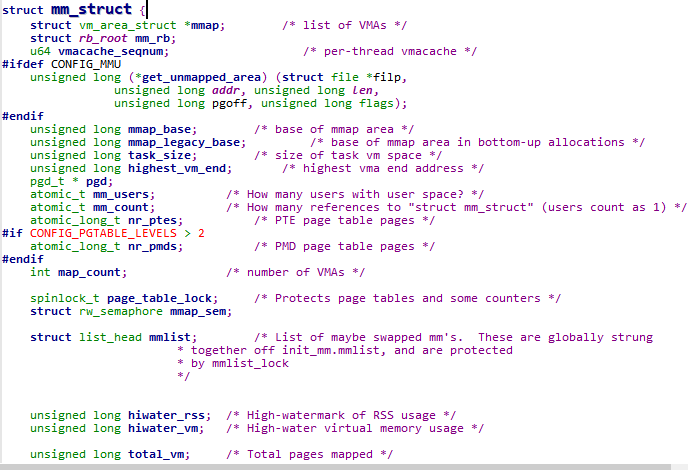


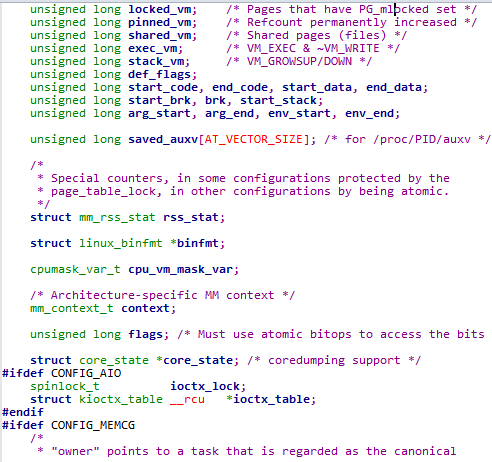


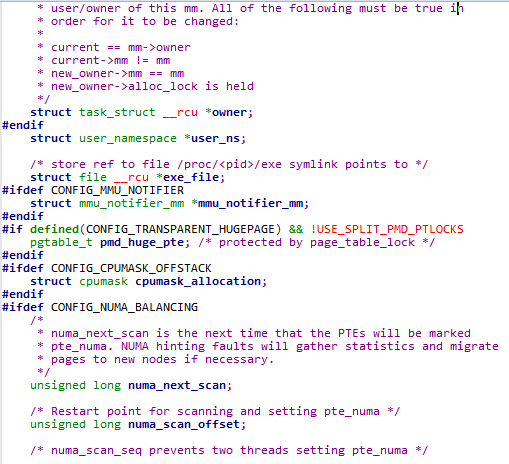
**mm :** 进程所拥有的用户空间内存描述符。

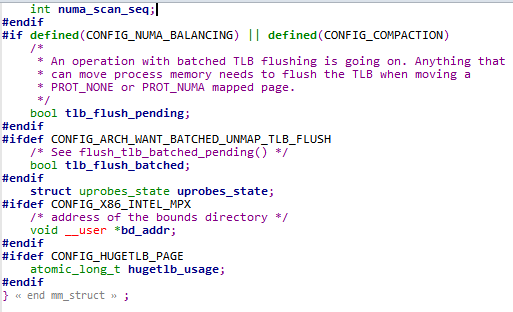


**mm\_struct :** Linux 内核通过一个被称为内存描述符的mm\_sruct结构体来管理进程，抽象描 述了linux视角下一个进程整个虚拟地址空间的所有信息。每个进程都拥有自己一 个mm\_sruct结构体。





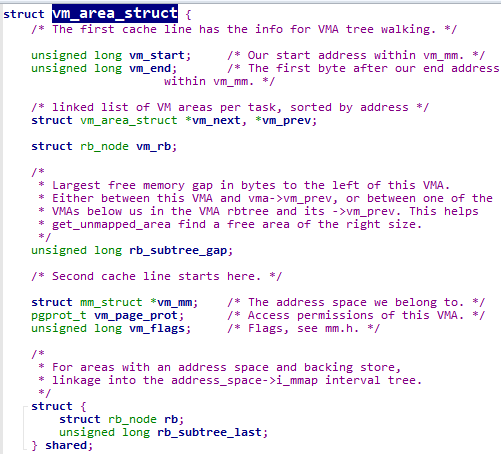


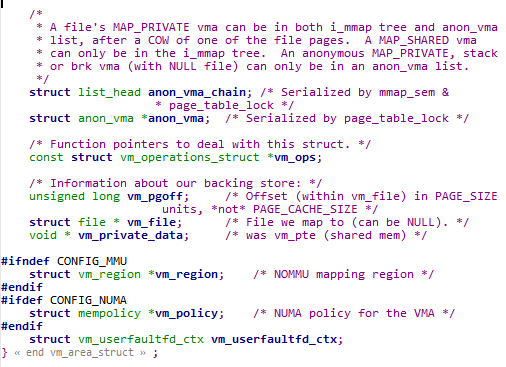


**\* mmap ：**指向虚拟区间的链表，来查找线性区



**vm\_area\_struct :** linux 通过vm\_area\_struct结构的对象实现线性区，每个线性区描述符表示一个线性地址区间。



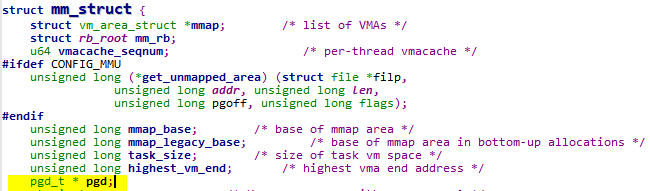


**用户进程线性区：**系统中进程的虚拟地址空间起始于地址0，延伸至TASK\_SIZE -1， 总的地址空间按3：1划分，内核分配1GB，各个用户空间进程可用的部分为3GB。用户空间从0G到3G, 内核空间从3G到4G。

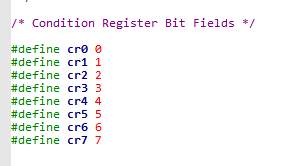
**mm -> \*mmap :** 进程描述符task\_struct数据结构中进程所拥有的用户空间内存描述符mm指向\*mmap，来查找线性区。

**vm\_area\_struct -> 用户进程线性区 :**linux 通过vm\_area\_struct结构的对象实现线性区，每个线性区描述符表示一个线性地址区间, vm\_start包含区域的第一个线性地址，vm\_end表示区域之外的第一个线性地址，vm\_end-vm\_start和线性区的长度。所有线性区通过简单的链表链接在一起，链表按内存地址升序排序。

**\* pgd :** 页表目录指针



**cr3 :** 是X86的一个寄存器用来存放进程页目录的物理地址



**进程页目录 ：**进程页目录中存储的是页表的物理地址。

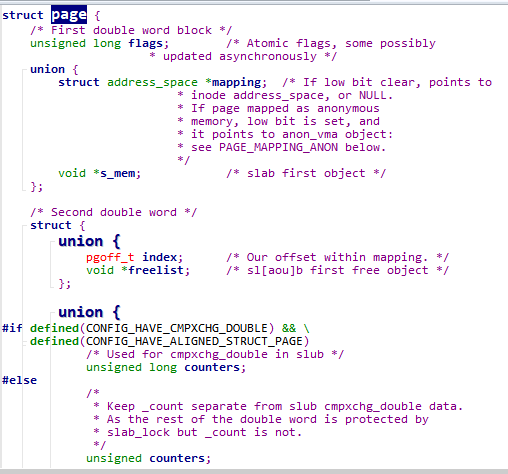
**\*pgd -> cr3 - > 进程页目录 ：** mm\_sturct中的页表目录指针\*pgd指向cr3寄存器，cr3寄存器中存放进程页目录的物理地址，进而指向进程页目录。

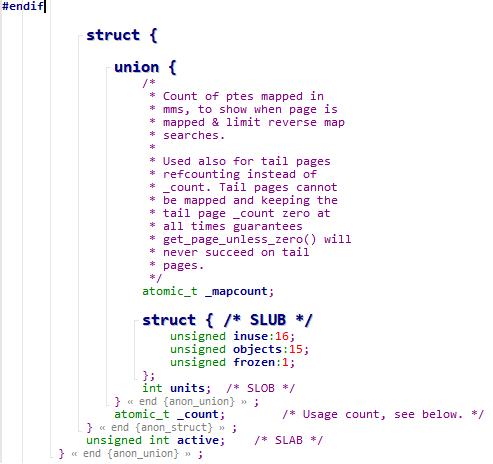
**用户线性地址 ：** 被分为三个部分：页目录表偏移量、页表偏移量、物理页内的字节偏移量。通过页目录表偏移量可在进程页目录中找到对应进程的页表的物理地址，通过页表偏移量可在对应进程的页表地址中找到最终物理页的物理起始地址，物理页基地址加上线性地址中的偏移量，CPU就找到了线性地址最终对应的物理内存单元。

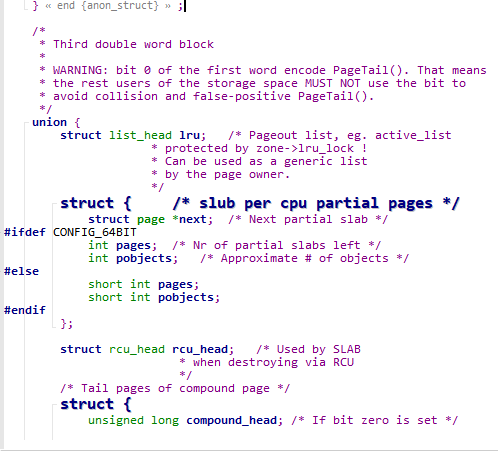
**内核空间：**内核空间是由内核负责映射，是固定的。内核空间地址有自己对应的内核页表。

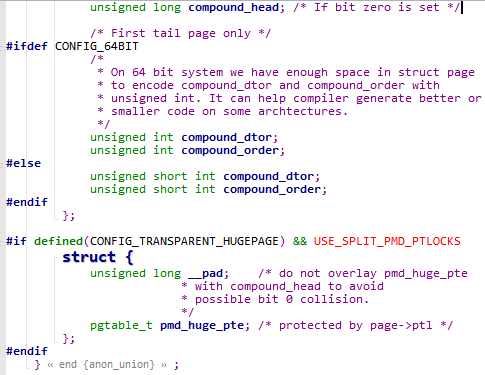
**内核空间线性地址 -> 进程页目录 -> 内核页表 -> 物理内存：**由内核空间线性地址得到进程页目录存放的内核页表的物理地址，进而找到内核页表得到物理内存的地址。

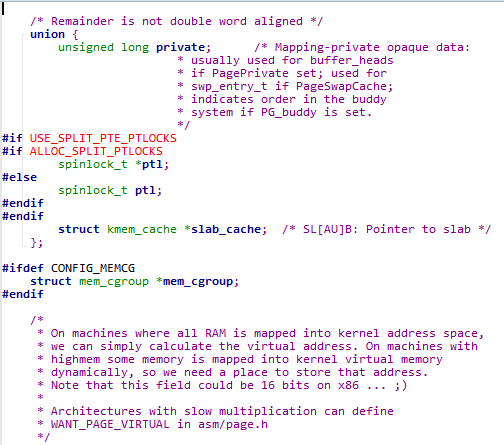
**struct\_page :** 内核用struct\_page结构体表示系统每一个物理页。flags存放页的状态，如改页是不是脏页，\_count域表示该页的使用计数。\_mapcout表示在页表中存在多少个指向该页的项。lru是一个表头，用于在各种链表上维护该页，mapping指定了页帧所在的地址空间，private 是一个指向私有数据的指针，virtual 用于高端内存区的页。

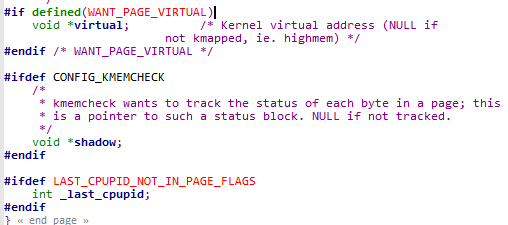




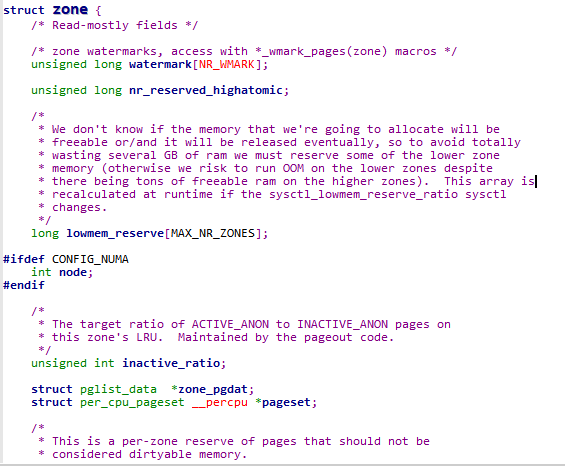


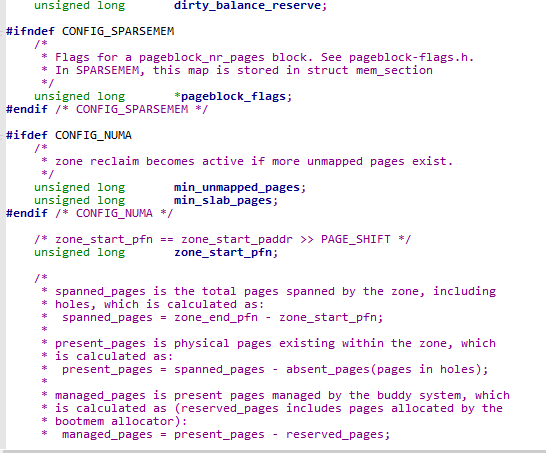


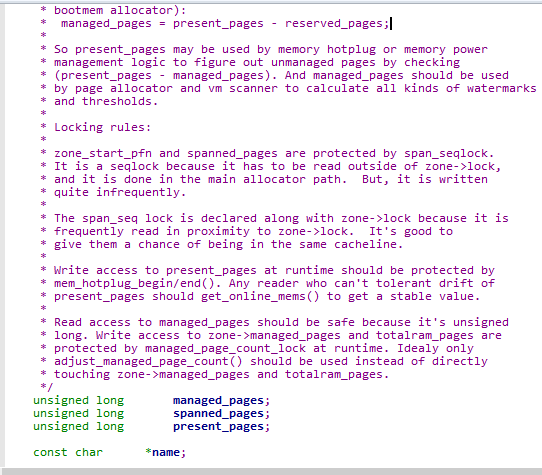


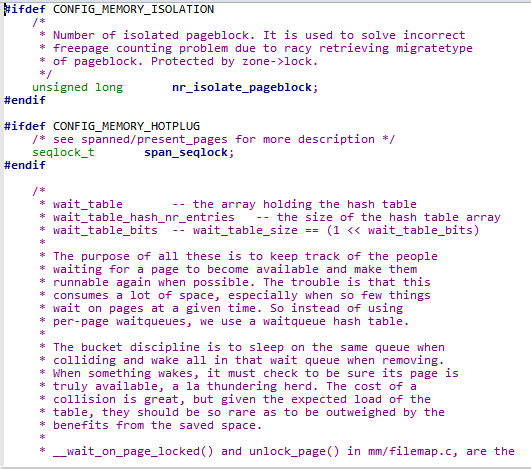


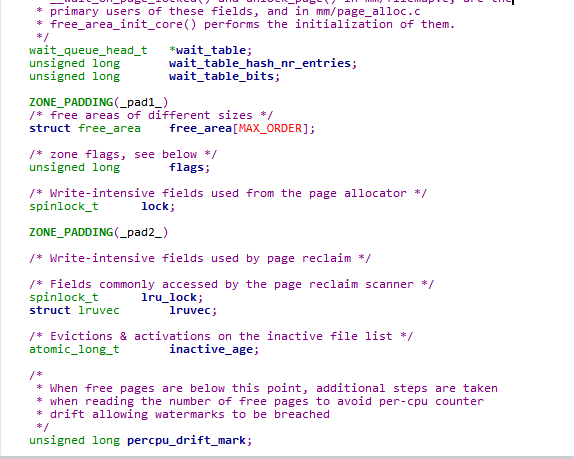
**区：**区是逻辑上分组的概念，在X86体系结构中主要分为3个区：ZONE\_DMA, ZONE\_NORMAL, ZONE\_HIGHMEM。ZONE\_DMA区中的页用来进行DMA 时使用，ZONE\_HIGHMEM是高端地址，其中的页没有虚拟地址。剩余的内存就属于ZONE\_NORMAL区。

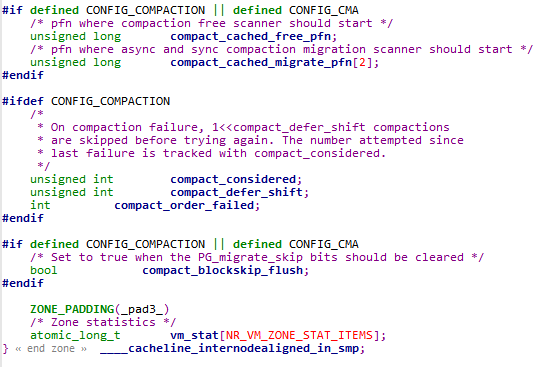












**vmalloc ：** vmalloc分配的是虚拟地址连续，物理地址不一定连续的一片区域，一般为高端内存。需要释放该段内存是使用vfree函数。

**伙伴系统：**内存初始化完成后，内存管理的工作由伙伴系统算法承担，伙伴算法采用叶框作为基本内存区，伙伴系统的调用既是为了获得存放新内存区所需的额外叶框，也是为了释放不再包含内存区的叶框。

**slab :** slab层用于解决频繁分配和释放数据结构的问题。物理内存中有多个高速缓存，每个高速缓存中会有一个或多个slab, slab通常为一页，其中存放着数据结构类型的实例化对象。通过建立slab缓冲，内核能储备一些对象，供后续使用。slab分配器将释放的内存块保存在一个内部列表中，而不是立刻返回给伙伴系统，这样内核不必使用伙伴系统算法，缩短了处理时间。

**固定映射的线性地址：**它所对应的物理地址不是通过简单的线性转换得到的，而是人为强制指定的。每个固定线性地址都映射到任何一页物理内存。

**永久内核映射**：允许内核建立高端页框到内核地址空间的长期映射。

2、参考图2解释内核层不同内存分配接口的区别，包括\_\_get\_free\_pages，kmalloc，vmalloc等，3分。

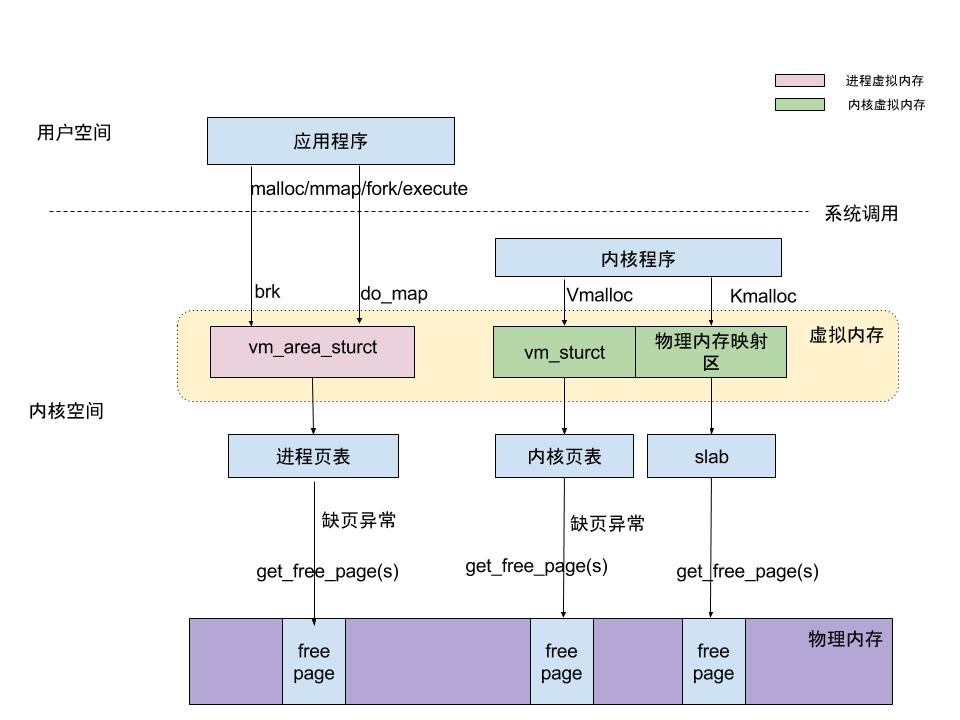


图2. 用户空间和内核空间内存分配示意图

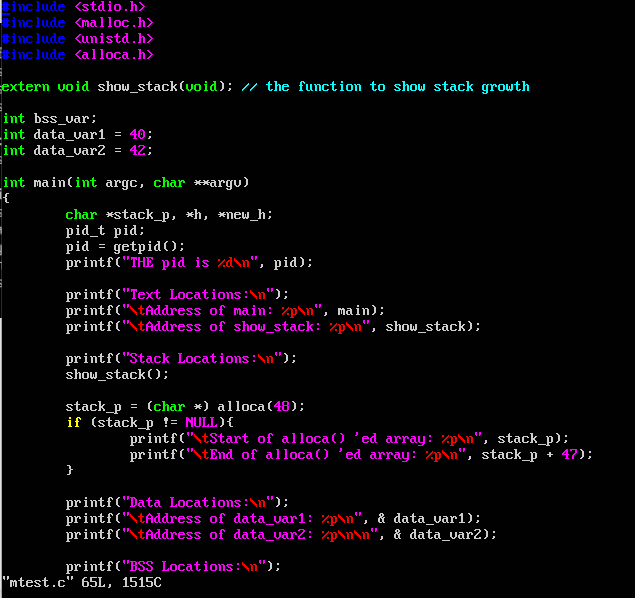
**vmalloc :** 在内存空间的vmalloc区（VMALLOC\_START~4GB）中申请内存，并建立虚拟地址到物理地址映射，此时的虚拟地址连续而物理地址可能是不连续的，因为在申请内存时每个页面都是单独申请的，建立映射过程也是每个页面单独建立映射。在内核空间中调用vmalloc()分配非物理连续空间，分配的地址成为内核虚拟地址。在分配过程中必须更新内核页表。

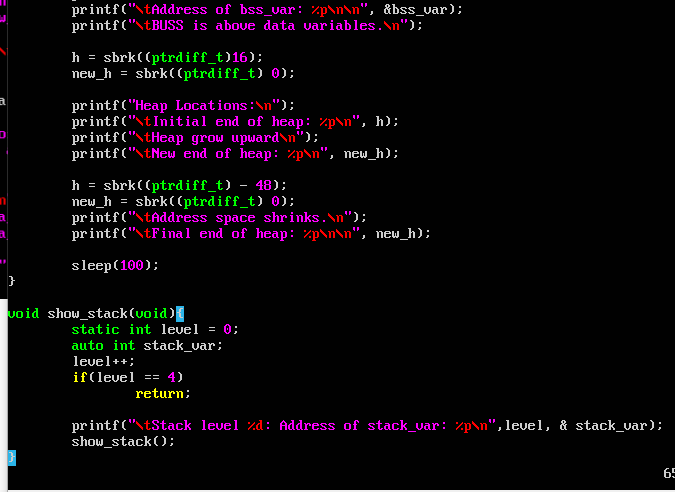
**kmalloc :** 分配的内核内存处于处于3GB~high\_memory之间，这段内核空间与物理内存的映射一一对应，虚拟地址连续，物理地址也连续。在内核空间中调用kmalloc()分配连续物理空间，分配的地址成为内核逻辑地址。kmalloc分配内存是基于slab。

**malloc :** 在用户内存进行分配，返回所分配内存块的虚拟地址。

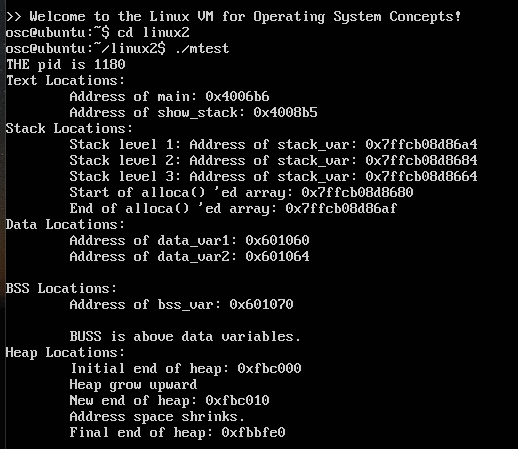
**\_get\_free\_pages :** 是最原始的内存分配方式，直接从伙伴系统中获取原始页框，返回值为第一个页框的起始地址。

1. 参考[Anatomy of a Program in Memory](https://manybutfinite.com/post/anatomy-of-a-program-in-memory/)和[User-Level Memory Management](https://linuxdevices.org/ldfiles/misc/Linux_Programming_by_Example_ch03.pdf)中例程，写一个实验程序mtest.c，生成可执行程序mtest；打印代码段、数据段、BSS，栈、堆等的相关地址；需要创建自己的例子，不允许简单照搬，8分。

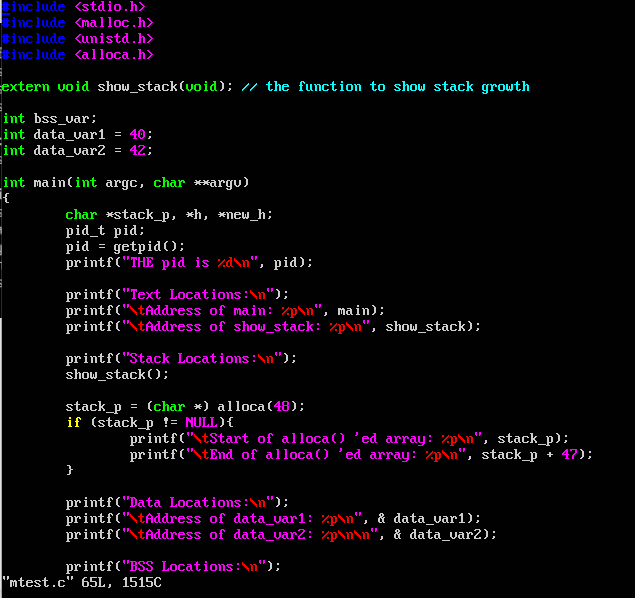


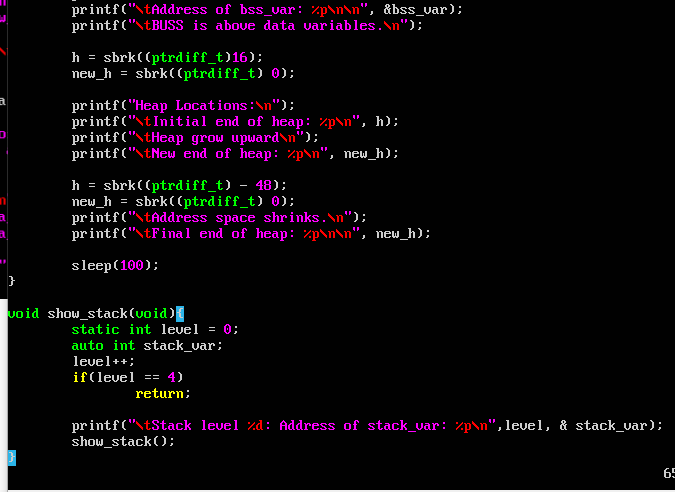


运行结果：

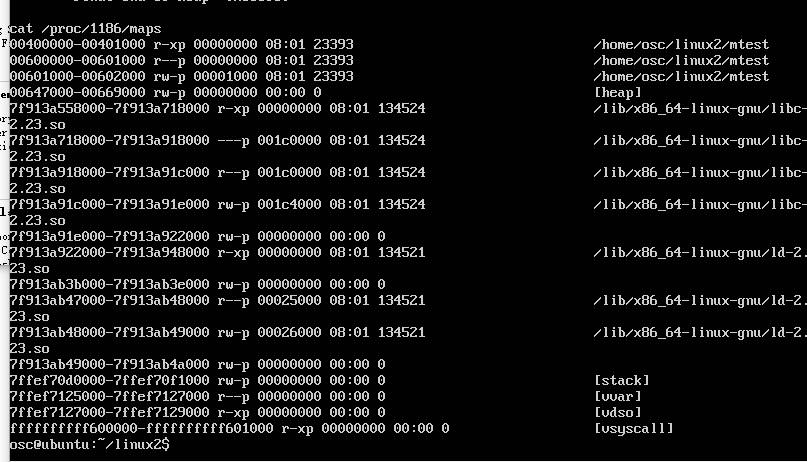
****

1. 参考[How The Kernel Manages Your Memory](https://manybutfinite.com/post/how-the-kernel-manages-your-memory/)，通过/proc/pid\_number/maps，分析mtest各个内存段（参考[链接](https://blog.csdn.net/lijzheng/article/details/23618365)）。绘制图表，解释输出的每一段的各种属性，包括每一列的内容。为了让mtest程序驻留内存，可以在程序末尾加上长时睡眠，并将mtest在后台运行，即./mtest & 6分。





运行结果：

****

分析表格在附件里

5、参考[A Malloc Tutorial](https://wiki-prog.infoprepa.epita.fr/images/0/04/Malloc_tutorial.pdf)以及相关资料（如[链接](https://blog.csdn.net/gfgdsg/article/details/42709943)）回答以下问题：3分

（1）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？

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk和mmap（不考虑共享内存）。

1、brk是将数据段(.data)的最高地址指针\_edata往高地址推；

2、mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

     这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk，mmap，munmap这些系统调用实现的。

情况一、当malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)

**情况二、malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)**

（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？

mmap系统调用实现了更有用的动态内存分配功能，可以将一个磁盘文件的全部或部分内容映射到用户空间中，进程读写文件的操作变成了读写内存的操作。在 linux/mm/mmap.c文件的do\_mmap\_pgoff()函数，是mmap系统调用实现的核心。do\_mmap\_pgoff()的代码，只是新建了一个vm\_area\_struct结构，并把file结构的参数赋值给其成员变量m\_file，并没有把文件内容实际装入内存。

Linux内存管理的基本思想之一，是只有在真正访问一个地址的时候才建立这个地址的物理映射。

（3）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？

虚拟内存，通过地址映射。