**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。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  //数据结构 页式管理 FIFO 先进先出 单进程  #define MEMORY\_MAX\_BLOCKS 10//定义物理块最大数量  #define PROGRESS\_MAX\_PAGE 20 //定义虚拟页面的最大数量  #define PROGRESS\_MAX\_LENGTH 30 //定义进程的最大长度  //当前进程的相关信息  int curProPages;//当前进程虚拟页  int curMemoryBlocks;//当前进程物理块  int proLength;// 进程执行长度  int proSort[PROGRESS\_MAX\_LENGTH];//虚拟页执行顺序数组  typedef struct PageItem  {  int vpage;//虚拟页的编号  int mblock;//对应的物理号 默认不存在对应的内存块-1  int loadTime;//该页面在内存中存在的时间  } PageItem;  PageItem pageList[PROGRESS\_MAX\_PAGE];//页表：  int memoryAllocInfo[MEMORY\_MAX\_BLOCKS];//内存物理块分配信息，某个内存块放某个页面  //初始化函数  void init()  {  int i=0;  curProPages=5;//虚拟页面数量  curMemoryBlocks=3;//内存块数量  proLength=7;//虚拟页执行顺序数组长度  proSort[0]=2;//存放的是虚拟页的编号  proSort[1]=4;  proSort[2]=3;  proSort[3]=0;  proSort[4]=1;  proSort[5]=2;  proSort[6]=4;  for(i=0; i<curProPages; i++) //页表的初始化  {  pageList[i].vpage=i;  pageList[i].mblock=-1;  pageList[i].loadTime=0;  }  for(i=0; i<curMemoryBlocks; i++) //内存块的初始化  {  memoryAllocInfo[i]=-1;  }  }  void showInfo()  {  //页表信息打印  printf("-------------------------------------------------------------\n");  printf("页表信息：\n");  printf("页号\t页对应的内存块编号\t页面在内存中存活的时间\n");  int i;  for(i=0; i<curProPages; i++)  {  printf("%d\t\t%d\t\t\t%d\n",pageList[i].vpage,pageList[i].mblock,pageList[i].loadTime);  }  printf("\n");  printf("内存块信息表:\n");  printf("内存块地址\t存放在此的页面\n");  for(i=0; i<curMemoryBlocks; i++)  {  printf("%d\t\t%d\n",i,memoryAllocInfo[i]);  }  printf("-------------------------------------------------------------\n\n\n");  }  //页表中已经存在于内存的虚拟页的时间加一个点  void extnedTime()  {  int i=0;  for(i=0; i<curProPages; i++)  {  if(pageList[i].mblock!=-1)  {  pageList[i].loadTime+=1;  }  }  }  //检查物理内存中是否有空闲的块  int findFreeBlock()  {  int i=0;  for(i=0; i<curMemoryBlocks; i++)  {  if(memoryAllocInfo[i]==-1)  {  return i;  }  }  return -1;//返回-1表示不存在空闲内存块  }  //FIFO淘汰  //找到在内存块中存活时间最久的页面  int fifoPage()  {  int i=0;  int maxTime=-1;  int vpage=-1;  for(i=0; i<curProPages; i++)  {  if(pageList[i].mblock==-1)  {  continue;  }  if(pageList[i].loadTime>maxTime)  {  maxTime=pageList[i].loadTime;  vpage=i;  }  }  return vpage;  }  void fifo()  {  //开始程序执行，调用虚拟机  int index;  int vpage;  int fpage;  int freeIndex;  for(index=0; index<proLength; index++)  {  vpage=proSort[index];  //检查是否已经分配物理页  if(pageList[vpage].mblock!=-1)  {  //对于已经存在于内存的虚拟页都必须加一个时间点  extnedTime();  }  else  {  //说明虚拟页没有在内存中  //检查内存中是否有空闲块  freeIndex=findFreeBlock();  if(freeIndex!=-1)  {  memoryAllocInfo[freeIndex]=vpage;  pageList[vpage].mblock=freeIndex;  extnedTime();  }  else  {  //空闲区没有  //查页表：淘汰最长时间  fpage=fifoPage();  //找到在内存中所占有的物理块  freeIndex=pageList[fpage].mblock;  //更新淘汰的虚拟页的相关信息  pageList[fpage].loadTime=0;  pageList[fpage].mblock=-1;  //更新分配情况  memoryAllocInfo[freeIndex]=vpage;  //更新当前调到内存中的虚页的信息  pageList[vpage].mblock=freeIndex;  extnedTime();  }  }  showInfo();  }  }  int main()  {  //初始化(交互)  init();  //显示（）  showInfo();  //核心逻辑（）  fifo();  return 0;  } |

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。
5. （**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。
6. **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. **物理内存管理**

①**页**：物理内存以页（page frame）为单位，一般为4K，如果是4G的内存那么，将会有一个大小为4G/4K=1M的数组mem\_map，每一项保存着每一页的地址。

②**Zone**：Zone是一些连续物理页的合集

|  |  |
| --- | --- |
| ZONE\_HIGHMEM | 物理内存高于896M的区域 |
| ZONE\_NORMAL | 常规内存区域，如果DMA可以在此区域做内存访问，也可以使用这块区域 |
| ZONE\_DMA | 0~nM，不同的架构和不同的芯片都可能不一样 |

数组mem\_map与物理页对应，因此mem\_map也被默认分为了上面三个区域

③**内存节点**

|  |  |
| --- | --- |
| UMA | 只有一个内存节点，CPU访问内存的内存的任何地址的速度是一样的。 |
| NUMA | 多个处理器，每个处理器都有自己的本地内存，通过总线通讯。访问本地内存比访问其他节点的内存快。 |

④物理页面分配器-**伙伴系统：**使用物理页为单位，记录哪些是空闲的物理页，那些是内核使用过的物理页，可以让系统分配一个或者多个连续的物理页，kernel启动后会映射物理页面直接映射区，但是ZONE\_HIGHMEM区域并没有被映射到内核的虚拟地址上。

⑤**slab分配器：**在低端内存区域分配空间，分配比一个页更小的空间

1. **虚拟内存管理**

**vmalloc**：获得连续的虚拟内存

1. **线性地址映射到高端内存**

① **Linux内核地址空间划分**：通常32位Linux内核地址空间划分0~3G为用户空间，3~4G为内核空间。

②**逻辑地址与物理地址对应的关系**：物理地址 = 逻辑地址 – 0xC0000000

③ **Linux内核高端内存的划分：**VMALLOC\_START~VMALLOC\_ENDKMAP\_BASE~FIXADDR\_START和FIXADDR\_START~4G。

④内核想访问高于896M的物理地址时，从0xF8000000~0xFFFFFFFF虚拟地址空间中取一部分与想要访问的物理内存建立映射即填充内核PTE页表（内核页表），访问完成之后内核释放0xF8000000~0xFFFFFFFF中的虚拟地址空间，以便其它进程访问（采用这128M的虚拟空间，建立临时地址映射，完成了对所有高于896M物理内存的访问）

1. **线性地址转物理地址**

线性地址是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。

1. **内存描述符：**

Linux 内核为每个进程， 分配一个task\_struct数据结构，并创建页表目录、页表。

一个进程的虚拟地址空间主要由两个数据结构来描述。一个是最高层次的：mm\_struct（定义在mm\_types.h中），一个是较高层次的：vm\_area\_structs。

①**task\_struct**

|  |
| --- |
| struct task\_struct {  volatile long state; //说明了该进程是否可以执行,还是可中断等信息  unsigned long flags; //Flage 是进程号,在调用fork()时给出  int sigpending; //进程上是否有待处理的信号  mm\_segment\_t addr\_limit; //进程地址空间,区分内核进程与普通进程在内存存放的位置不同  //0-0xBFFFFFFF for user-thead  //0-0xFFFFFFFF for kernel-thread  //调度标志,表示该进程是否需要重新调度,若非0,则当从内核态返回到用户态,会发生调度  volatile long need\_resched;  int lock\_depth; //锁深度  long nice; //进程的基本时间片  //进程的调度策略,有三种,实时进程:SCHED\_FIFO,SCHED\_RR, 分时进程:SCHED\_OTHER  unsigned long policy;  struct mm\_struct \*mm; //进程内存管理信息  int processor;  //若进程不在任何CPU上运行, cpus\_runnable 的值是0，否则是1 这个值在运行队列被锁时更新  unsigned long cpus\_runnable, cpus\_allowed;  struct list\_head run\_list; //指向运行队列的指针  unsigned long sleep\_time; //进程的睡眠时间  //用于将系统中所有的进程连成一个双向循环链表, 其根是init\_task  struct task\_struct \*next\_task, \*prev\_task;  struct mm\_struct \*active\_mm;  struct list\_head local\_pages; //指向本地页面  unsigned int allocation\_order, nr\_local\_pages;  struct linux\_binfmt \*binfmt; //进程所运行的可执行文件的格式  int exit\_code, exit\_signal;  int pdeath\_signal; //父进程终止是向子进程发送的信号  unsigned long personality;  //Linux可以运行由其他UNIX操作系统生成的符合iBCS2标准的程序  int did\_exec:1;  pid\_t pid; //进程标识符,用来代表一个进程  pid\_t pgrp; //进程组标识,表示进程所属的进程组  pid\_t tty\_old\_pgrp; //进程控制终端所在的组标识  pid\_t session; //进程的会话标识  pid\_t tgid;  int leader; //表示进程是否为会话主管  struct task\_struct \*p\_opptr,\*p\_pptr,\*p\_cptr,\*p\_ysptr,\*p\_osptr;  struct list\_head thread\_group; //线程链表  struct task\_struct \*pidhash\_next; //用于将进程链入HASH表  struct task\_struct \*\*pidhash\_pprev;  wait\_queue\_head\_t wait\_chldexit; //供wait4()使用  struct completion \*vfork\_done; //供vfork() 使用  unsigned long rt\_priority; //实时优先级，用它计算实时进程调度时的weight值  //it\_real\_value，it\_real\_incr用于REAL定时器，单位为jiffies, 系统根据it\_real\_value  //设置定时器的第一个终止时间. 在定时器到期时，向进程发送SIGALRM信号，同时根据  //it\_real\_incr重置终止时间，it\_prof\_value，it\_prof\_incr用于Profile定时器，单位为jiffies。  //当进程运行时，不管在何种状态下，每个tick都使it\_prof\_value值减一，当减到0时，向进程发送  //信号SIGPROF，并根据it\_prof\_incr重置时间.  //it\_virt\_value，it\_virt\_value用于Virtual定时器，单位为jiffies。当进程运行时，不管在何种  //状态下，每个tick都使it\_virt\_value值减一当减到0时，向进程发送信号SIGVTALRM，根据  //it\_virt\_incr重置初值。  unsigned long it\_real\_value, it\_prof\_value, it\_virt\_value;  unsigned long it\_real\_incr, it\_prof\_incr, it\_virt\_value;  struct timer\_list real\_timer; //指向实时定时器的指针  struct tms times; //记录进程消耗的时间  unsigned long start\_time; //进程创建的时间  //记录进程在每个CPU上所消耗的用户态时间和核心态时间  long per\_cpu\_utime[NR\_CPUS], per\_cpu\_stime[NR\_CPUS];  //内存缺页和交换信息:  //min\_flt, maj\_flt累计进程的次缺页数（Copy on　Write页和匿名页）和主缺页数（从映射文件或交换  //设备读入的页面数）； nswap记录进程累计换出的页面数，即写到交换设备上的页面数。  //cmin\_flt, cmaj\_flt, cnswap记录本进程为祖先的所有子孙进程的累计次缺页数，主缺页数和换出页面数。  //在父进程回收终止的子进程时，父进程会将子进程的这些信息累计到自己结构的这些域中  unsigned long min\_flt, maj\_flt, nswap, cmin\_flt, cmaj\_flt, cnswap;  int swappable:1; //表示进程的虚拟地址空间是否允许换出  //进程认证信息  //uid,gid为运行该进程的用户的用户标识符和组标识符，通常是进程创建者的uid，gid  //euid，egid为有效uid,gid  //fsuid，fsgid为文件系统uid,gid，这两个ID号通常与有效uid,gid相等，在检查对于文件  //系统的访问权限时使用他们。  //suid，sgid为备份uid,gid  uid\_t uid,euid,suid,fsuid;  gid\_t gid,egid,sgid,fsgid;  int ngroups; //记录进程在多少个用户组中  gid\_t groups[NGROUPS]; //记录进程所在的组  //进程的权能，分别是有效位集合，继承位集合，允许位集合  kernel\_cap\_t cap\_effective, cap\_inheritable, cap\_permitted;  int keep\_capabilities:1;  struct user\_struct \*user;  struct rlimit rlim[RLIM\_NLIMITS]; //与进程相关的资源限制信息  unsigned short used\_math; //是否使用FPU  char comm[16]; //进程正在运行的可执行文件名  //文件系统信息  int link\_count, total\_link\_count;  //NULL if no tty 进程所在的控制终端，如果不需要控制终端，则该指针为空  struct tty\_struct \*tty;  unsigned int locks;  //进程间通信信息  struct sem\_undo \*semundo; //进程在信号灯上的所有undo操作  struct sem\_queue \*semsleeping; //当进程因为信号灯操作而挂起时，他在该队列中记录等待的操作  //进程的CPU状态，切换时，要保存到停止进程的task\_struct中  struct thread\_struct thread;  //文件系统信息  struct fs\_struct \*fs;  //打开文件信息  struct files\_struct \*files;  //信号处理函数  spinlock\_t sigmask\_lock;  struct signal\_struct \*sig; //信号处理函数  sigset\_t blocked; //进程当前要阻塞的信号，每个信号对应一位  struct sigpending pending; //进程上是否有待处理的信号  unsigned long sas\_ss\_sp;  size\_t sas\_ss\_size;  int (\*notifier)(void \*priv);  void \*notifier\_data;  sigset\_t \*notifier\_mask;  u32 parent\_exec\_id;  u32 self\_exec\_id;  spinlock\_t alloc\_lock;  void \*journal\_info;  }; |

②**mm\_struct:**

mm\_struct结构体通过自身的mmlist域链接在一个双向链表上，该链表的首元素是init\_mm内存描述符，代表init进程的地址空间。

|  |
| --- |
|  struct mm\_struct {       struct vm\_area\_struct \* mmap;  /\* list of VMAs，指向线性区对象的链表头部 \*/       struct rb\_root mm\_rb;  /\* 指向线性区对象的红黑树\*/       struct vm\_area\_struct \* mmap\_cache;  /\* last find\_vma result 指向最近找到的虚拟区间 \*/   #ifdef CONFIG\_MMU  /\*用来在进程地址空间中搜索有效的进程地址空间的函数\*/         unsigned long (\*get\_unmapped\_area) (struct file \*filp,                   unsigned long addr, unsigned long len,                   unsigned long pgoff, unsigned long flags);   /\*释放线性区的调用方法\*/    void (\*unmap\_area) (struct mm\_struct \*mm, unsigned long addr);   #endif       unsigned long mmap\_base;  /\* base of mmap area ，内存映射区的基地址\*/       unsigned long task\_size;  /\* size of task vm space \*/       unsigned long cached\_hole\_size;  /\* if non-zero, the largest hole below free\_area\_cache \*/       unsigned long free\_area\_cache;  /\* first hole of size cached\_hole\_size or larger \*/       pgd\_t \* pgd;  /\* 页表目录指针\*/       atomic\_t mm\_users;  /\* How many users with user space?，共享进程的个数 \*/       atomic\_t mm\_count;  /\* How many references to "struct mm\_struct" (users count as 1)，主使用计数器，采用引用计数，描述有多少指针指向当前的mm\_struct \*/       int map\_count;  /\* number of VMAs ,线性区个数\*/       struct rw\_semaphore mmap\_sem;       spinlock\_t page\_table\_lock;  /\* Protects page tables and some counters，保护页表和引用计数的锁 （使用的自旋锁）\*/         struct list\_head mmlist;  /\* List of maybe swapped mm's.    These are globally strung                            \* together off init\_mm.mmlist, and are protected                            \* by mmlist\_lock                            \*/           unsigned long hiwater\_rss;  /\* High-watermark of RSS usage,进程拥有的最大页表数目 \*/       unsigned long hiwater\_vm;  /\* High-water virtual memory usage ,进程线性区的最大页表数目\*/         unsigned long total\_vm, locked\_vm, shared\_vm, exec\_vm;       unsigned long stack\_vm, reserved\_vm, def\_flags, nr\_ptes;       unsigned long start\_code, end\_code, start\_data, end\_data;  /\*维护代码区和数据区的字段\*/       unsigned long start\_brk, brk, start\_stack;  /\*维护堆区和栈区的字段\*/       unsigned long arg\_start, arg\_end, env\_start, env\_end;  /\*命令行参数的起始地址和尾地址，环境变量的起始地址和尾地址\*/         unsigned long saved\_auxv[AT\_VECTOR\_SIZE];  /\* for /proc/PID/auxv \*/         /\*        \* Special counters, in some configurations protected by the        \* page\_table\_lock, in other configurations by being atomic.        \*/       struct mm\_rss\_stat rss\_stat;         struct linux\_binfmt \*binfmt;         cpumask\_t cpu\_vm\_mask;         /\* Architecture-specific MM context \*/       mm\_context\_t context;         /\* Swap token stuff \*/       /\*        \* Last value of global fault stamp as seen by this process.        \* In other words, this value gives an indication of how long        \* it has been since this task got the token.        \* Look at mm/thrash.c        \*/       unsigned int faultstamp;       unsigned int token\_priority;       unsigned int last\_interval;         unsigned long flags;  /\* Must use atomic bitops to access the bits \*/         struct core\_state \*core\_state;  /\* coredumping support \*/   #ifdef CONFIG\_AIO       spinlock\_t        ioctx\_lock;       struct hlist\_head    ioctx\_list;   #endif   #ifdef CONFIG\_MM\_OWNER       /\*        \* "owner" points to a task that is regarded as the canonical        \* user/owner of this mm. All of the following must be true in        \* order for it to be changed:        \*        \* current == mm->owner        \* current->mm != mm        \* new\_owner->mm == mm        \* new\_owner->alloc\_lock is held        \*/       struct task\_struct \*owner;   #endif     #ifdef CONFIG\_PROC\_FS       /\* store ref to file /proc/<pid>/exe symlink points to \*/       struct file \*exe\_file;       unsigned long num\_exe\_file\_vmas;   #endif   #ifdef CONFIG\_MMU\_NOTIFIER       struct mmu\_notifier\_mm \*mmu\_notifier\_mm;   #endif   |

③**vm\_area\_struct**

|  |
| --- |
|  /\*    \* This struct defines a memory VMM memory area. There is one of these    \* per VM-area/task.  A VM area is any part of the process virtual memory    \* space that has a special rule for the page-fault handlers (ie a shared    \* library, the executable area etc).    \*/   struct vm\_area\_struct {       struct mm\_struct \* vm\_mm;    /\* The address space we belong to. \*/       unsigned long vm\_start;        /\* Our start address within vm\_mm. \*/       unsigned long vm\_end;        /\* The first byte after our end address                          within vm\_mm. \*/         /\* linked list of VM areas per task, sorted by address \*/       struct vm\_area\_struct \*vm\_next;         pgprot\_t vm\_page\_prot;        /\* Access permissions of this VMA. \*/       unsigned long vm\_flags;        /\* Flags, see mm.h. \*/         struct rb\_node vm\_rb;         /\*        \* For areas with an address space and backing store,        \* linkage into the address\_space->i\_mmap prio tree, or        \* linkage to the list of like vmas hanging off its node, or        \* linkage of vma in the address\_space->i\_mmap\_nonlinear list.        \*/       union {           struct {               struct list\_head list;               void \*parent;    /\* aligns with prio\_tree\_node parent \*/               struct vm\_area\_struct \*head;           } vm\_set;             struct raw\_prio\_tree\_node prio\_tree\_node;       } shared;         /\*        \* A file's MAP\_PRIVATE vma can be in both i\_mmap tree and anon\_vma        \* list, after a COW of one of the file pages.    A MAP\_SHARED vma        \* can only be in the i\_mmap tree.  An anonymous MAP\_PRIVATE, stack        \* or brk vma (with NULL file) can only be in an anon\_vma list.        \*/       struct list\_head anon\_vma\_chain; /\* Serialized by mmap\_sem &                         \* page\_table\_lock \*/       struct anon\_vma \*anon\_vma;    /\* Serialized by page\_table\_lock \*/         /\* Function pointers to deal with this struct. \*/       const struct vm\_operations\_struct \*vm\_ops;         /\* Information about our backing store: \*/       unsigned long vm\_pgoff;        /\* Offset (within vm\_file) in PAGE\_SIZE                          units, \*not\* PAGE\_CACHE\_SIZE \*/       struct file \* vm\_file;        /\* File we map to (can be NULL). \*/       void \* vm\_private\_data;        /\* was vm\_pte (shared mem) \*/       unsigned long vm\_truncate\_count;/\* truncate\_count or restart\_addr \*/     #ifndef CONFIG\_MMU       struct vm\_region \*vm\_region;    /\* NOMMU mapping region \*/   #endif   #ifdef CONFIG\_NUMA       struct mempolicy \*vm\_policy;    /\* NUMA policy for the VMA \*/   #endif   \*/ |

1. **mem\_map：**

mem\_map是一个struct page的数组，管理着系统中所有的物理内存页面。在系统启动的过程中，创建和分配mem\_map的内存区域。

1. 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。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | API名称 | 物理连续 | 大小限制 | 单位 | 场景 |
| slab | vmalloc/vfree | 虚拟连续  物理不定 | vmalloc区大小限制 | 页  VMALLOC区域 | 可能睡眠，不能从中断上下文中调用，或其他不允许阻塞情况下调用。  VMALLOC区域vmalloc\_start~vmalloc\_end之间，vmalloc比kmalloc慢，适用于分配大内存。 |
| kmalloc/kcalloc/krealloc/kfree | 物理连续 | 64B-4MB(随slab而变) | 2^order字节  Normal区域 | 大小有限，不如vmalloc/malloc大。  最大/小值由KMALLOC\_MIN\_SIZE/KMALLOC\_SHIFT\_MAX，对应64B/4MB。  从/proc/slabinfo中的kmalloc-xxxx中分配，建立在kmem\_cache\_create基础之上。 |
|  | kmem\_cache\_create | 物理连续 | 64B-4MB | 字节大小，需对齐  Normal区域 | 便于固定大小数据的频繁分配和释放，分配时从缓存池中获取地址，释放时也不一定真正释放内存。通过slab进行管理。 |
| 伙伴系统 | \_\_get\_free\_page/\_\_get\_free\_pages | 物理连续 | 4MB(1024页) | 页  Normal区域 | \_\_get\_free\_pages基于alloc\_pages，但是限定不能使用HIGHMEM。 |
| alloc\_page/alloc\_pages/free\_pages | 物理连续 | 4MB | 页  Normal/Vmalloc都可 | CONFIG\_FORCE\_MAX\_ZONEORDER定义了最大页面数2^11，一次能分配到的最大页面数是1024。 |

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。
2. 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。
3. 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
4. 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？

|  |  |  |
| --- | --- | --- |
| 系统 | 区别 | 使用场景 |
| brk/sbrk | 1、brk是将数据段(.data)的最高地址指针\_edata往高地址推  2、brk分配的内存需要等到高地址内存释放以后才能释放 | 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) |
| mmap | 1、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  2、mmap分配的内存可以单独释放 | 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) |

1. 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？

①库函数调用是系统无关的，因此可移植性好。

②由于缓冲区技术，读写文件通常是大量的数据（这种大量是相对于底层驱动的系统调用所实现的数据操作单位而言），这时，使用库函数就可以大大减少系统调用的次数。

③ 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。malloc基本上都是C函数库实现的,跟OS无关。C函数库内部通过一定的结构来保存当前有多少可用内存。如果程序 malloc的大小超出了库里所留存的空间,那么将首先调用brk系统调用来增加可用空间,然后再分配空间。

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？

虚拟内存。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。

1. （**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。

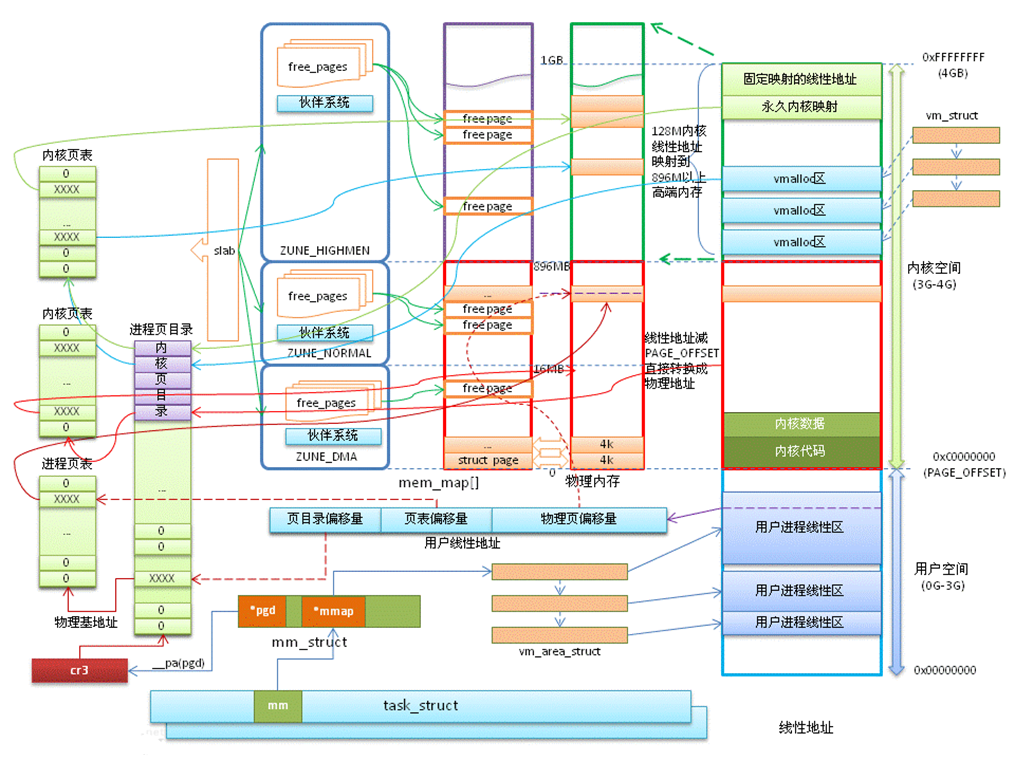
****

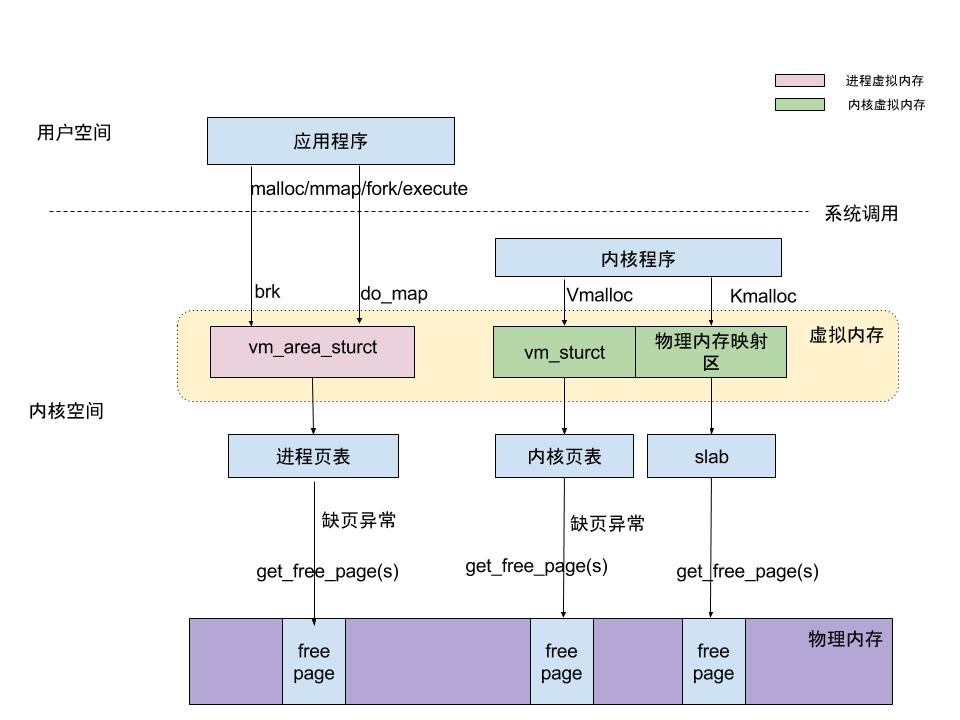
图1. Linux内核内存管理示意图（IA\_32）

图2. 用户空间和内核空间内存分配示意图