**操作系统大作业2**

18364039 冷春江

前言：运行trace.py文件就会产生addresses-locality.txt

运行gcc vm.c -lm 就会产生out.txt

Diff out.txt correct.txt 无输出。

运行draw.py就会产生点阵图。

运行gcc vm\_LRU.c -lm 就会输出读入addresses-locality.txt 的TLBhit和pageFault。该代码采用了实现了LRU策略的TLB与页替换策略

运行vm\_FIFO.c同上

1. **第一题**

首先搞明白一个程序，要先明白，输入和输出。

输入是虚拟地址，输出是物理地址和对应的存放在物理地址上的值。

那么核心的问题就是如何进行思路的转换。

虚拟地址，有课本上给出的定义我们知道，虽然他是一个整数，但是你可以将他进行转化成为16位的二进制码，然后取前八位和后八位，就是我们想要的页码和偏移量，下文均称为page和offSet。

代码方面要注意，c语言是要对数组进行初始化的，否则会出问题

int temp = addresses[i];

int count = 0;

int binary[100];

int l;

for(l = 0; l < 100; l++){

binary[l] = 0;

}

while(temp != 0){

binary[count] = temp & 1;

//这里相当于temp对000000001按位和，那么只有第一位为1的时候，才会是1，正好是我们想要的结果。

temp = (temp>>1);

count++;

//右移一位，相当于不要了，这里最好注意一下运算符优先级

}

//我们得到了二进制的数组，但是是反过来的，不过刚刚好，因为我们运算回去，也是要反过来运算。

//前8位不用关心，我们直接从第8位开始算，思路上文已经说了

int page = 0;//页号

page = binary\_to\_int(binary,8,16);

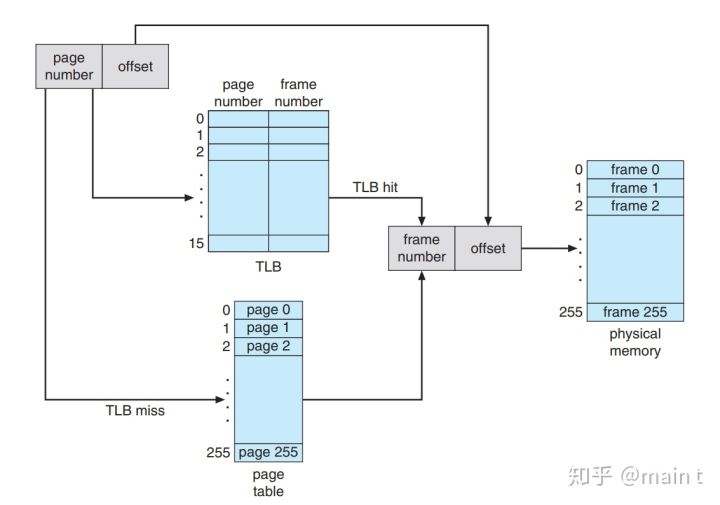
在上文中，我们获得了page和offSet，那么我们要得到frame和offSet，首先我们要知道在该题中，我们的offSet是不变的，那么我们解决了一半的问题。接下来就是page和frame这两者的关系。

**重点来了，在第一问中，我们的虚拟地址和物理地址也就是page对frame是256对256的，也就是说没有说内存条（物理内存）不够用，需要进行置换的情况存在。**

**那么我们就直接可以用一个数据结构，记录下目前已经构建好的page-frame的对应关系，**

**然后如果说我们得到一个的page进行处理，那么我们要先去看刚才的数据结构，也就是页表。有没有对应关系，（之前有没有出现过），**

1. **如果之前出现过，好，那么我们就得到了frame**
2. **如果没有出现过，好，我们就从上到下，把还没有分配的frame分配给他**



if(map[page] == -1){

//发生了缺页错误。我们要去二进制文件进行读取。

//以下是读取代码

FILE \*file;

char \*buffer;

int x;

file = fopen("BACKING\_STORE.bin", "rb");

fseek(file,256\*page,0);

buffer=(char \*)malloc(257);

fread(buffer,1,256,file);

for ( x = 0; x < 256; x++)

{

memory[firstFram][x] = buffer[x];

}

free(buffer);

fclose(file);

map[page] = firstFram;

frame = firstFram;

firstFram++;

}else{

//如果有的话，那我们就直接把对应的结果给他就好了，因为是256对256，所以不用担心，如果是10000对256的情况，就要引入页替换了

frame = map[page];

}

//现在我们得到了我们想要的数据，一个是偏移量，一个是对应的帧表数值，结果是要让我们返回二者和起来的数字。

//frame + offset 和起来的数值，以及对应的value

在这里我是用map【page】 = frame 作为页表的数据结构，但是理论上应该用map来做，但是c语言还要自己调红黑树的接口实现，所以如果想卷一下的话，可以考虑。

现在我们获得了frame和offSet的值，那么我们将其按规则组装就可以得到结果了。

那么老师要求的value怎么做的呢。

首先我们要确定一点，就是一开始所有的物理内存都是空的，那么我们只需要每次缺页的时候，把bin文件读到物理内存里面就好了。（为什么读bin文件，在课本的习题那里有要求）

然后最后的结果是value值，一开始我还是很疑惑，value值不应该是，每一页从0到offSet所有的字节吗，后来根据correct.txt文件倒推才发现，仅仅是对应的偏移的那个字符而已。

那其实到这里，就是c语言本身的问题了

代码的逻辑是，我们每次缺页时，就是在页表中找不到page对应的frame时，就会去bin文件读对应的256个字节放入内存条中，（我们只需要value，也要读256个字节，因为有可能后来，你遇到相同的页表，不同的偏移值，那样，只要读一次就好了）

这里我才用char 【256】【256】来处理。代码如下

FILE \*file;

char \*buffer;

int x;

file = fopen("BACKING\_STORE.bin", "rb");

fseek(file,256\*page,0);

buffer=(char \*)malloc(257);

fread(buffer,1,256,file);

for ( x = 0; x < 256; x++)

{

memory[firstFram][x] = buffer[x];

}

free(buffer);

fclose(file);

而后我们想要的到value就很方便

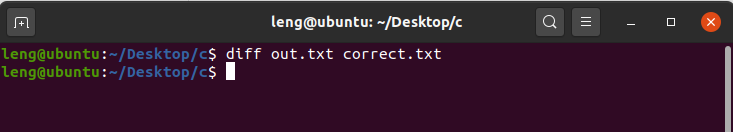
char value = memory[frame][off\_set];

然后就是输出文件了。

以上就是第一题的全部内容

将addresses文件放进来是可以直接gcc的但是要加-lm。

gcc vm.c -lm;



详细代码见vm.c

**根据老师的题意，我觉得第一题更核心的的是能够实现输出正确txt，tlb与页替换我在第二题中进行了彻底的完成。两种策略的源码分别在vm\_FIFO.c 和vm\_LRU.c中。**

首先搞明白输入page（虚拟地址的页表）和输出frame（物理地址的帧表），在第二问中，我们并不想要得到value，而是关注于输入和输出转化的过程。

首先，我们引入了缺页率，就是我们得到一个输入的page在页表中找不到的概率。（存疑）

还是按之前的流程，我们在一开始，内存条（物理地址）还没有分配满的时候，我们出现缺页，直接一个一个按顺序给就好了。

但是**在第二问中，出现了问题，如果说我们的内存条满了，页表还是找不到，怎么办。**

**因为是256-128的情况，所以会出现这种问题，这个时候我们就要进行处理，把内存条拿出来一个，把新的放进去。**

**拿哪一个？这个时候就是大问题了，拿最长时间没用过的LRU,拿最先进来的FIFO，两种策略，两个文件。**

我们先说FIFO的，那就是满了，我们就放第0个帧嘛，frame = 0 ； 然后我们再接着++嘛。

代码如下，

//搜查页表

if(frame == -1){

int q;

for(q = 0; q < 128;q++ ){

if(map[q] == page){

//找到拉

frame = q;

}

}

//进行页替换

if(frame == -1){

//没找到

//实现先进先出的页替换，这里用队列比较好。

printf("出现了一次缺页！ ");

page\_default++;

if(point == 128){

point = 1;

}

map[point] = page;

frame = point;

point++;

}

//更新TLB

其他的内容就和第一问差不多，而且我们不需要搞value了。

但是问题来了,老师还要我们搞TLB，这又是什么呢，我的理解是，他是相当于一个更高级的缓存，我们先去查询TLB，没有再去查询页表，在得到结果之后。在进行对TLB进行更新。

int page = 0;//页号

page = binary\_to\_int(binary,8,16);

int m = 0;

int frame = -1;

//实现map很麻烦，这里我们就反过来使用我们之前的map[]数组，key用来存放帧表，value放页表，搜查我们使用便利

//在实际中，应该是使用map，rb\_tree来实现这一对应的功能，但是老师说语言不是问题，我觉得完成任务，这二者是没有差别的

//搜查TLB

int w;

for(w = 0; w < 16; w++){

if(TLB[w].value == page ){

TLBHitTimes++;

printf("出现一次tlb命中 ");

frame = TLB[w].key;

}

}

//搜查页表

if(frame == -1){

int q;

for(q = 0; q < 128;q++ ){

if(map[q] == page){

//找到拉

frame = q;

}

}

//进行页替换

if(frame == -1){

//没找到

//实现先进先出的页替换，这里用队列比较好。

printf("出现了一次缺页！ ");

page\_default++;

if(point == 128){

point = 1;

}

map[point] = page;

frame = point;

point++;

}

//更新TLB

if(point\_TLB == 16){

point\_TLB = 0;

}

TLB[point\_TLB].value = page;

TLB[point\_TLB].key = frame;

point\_TLB++;

}

//现在我们得到了我们想要的数据，一个是偏移量，一个是对应的帧表数值，结果是要让我们返回二者和起来的数字。

那么老师要求的TLBhit和缺页率也就很好求了，我们在现有的代码上，如果TLB找到啦，就++，如果没找到，页表也没找到，就pageDefault++

最后进行总计就好了。以上

**总体代码在vm\_FIFO.c**

以上是第一种情况，第二种情况改变的就是我们如何进行替换，用到LRU方法

大致意思就是说，我们用map来存储东西，但是每次get和put就相当于我们用了一个数据，那就把这个数据放到开头，然后在创建的时候，会有一个size值限定大小，超过了这个值，我们就直接把最后面（最久没用的数据扔掉）。

然后，我们只需要将代码进行改进就好了

int main(void) {

FILE \*fp;

int len;

int addresses[1500];//开一个足bai够大的数组。

int i = 0;

char address\_txt[100] = "addresses.txt";

if((fp = fopen(address\_txt,"r"))== NULL){

return 0;

}

while(fscanf(fp, "%d", &addresses[i]) != EOF) {

i++;

}

fclose(fp);//关闭文件

//理论上说，为了代码的可阅读性，应该多放几个函数,但是c语言的函数传递，指针调用有点复杂，在这里就不调用了，其实俺觉得这题用别的编程语言也可以

//以下是对每一个数据的处理。

//处理的主要核心思路是，得到每一个int数字的二进制字符串，并且，从右往左数，0-7原封不动的给我们的物理地址

//随后先来先得，将帧玛从0-255依次分配，并且记录下来，这里我觉得可以采用int数组的形式来存取，不需要使用map

LRUCache \*page\_map;

LRUCache \*page\_TLB;

page\_TLB = lRUCacheCreate(16);

page\_map = lRUCacheCreate(128);

int TLBHitTimes = 0;

int page\_default = 0;

//严格意义上说，这个就是页表

//设定这个-1为我们未存取的情况

int point\_frame = 0;

int point\_TLB = 0;

for(i = 0; i <1000; i++){

int temp = addresses[i];

int count = 0;

int binary[100];

int l;

for(l = 0; l < 100; l++){

binary[l] = 0;

}

while(temp != 0){

binary[count] = temp & 1;

//这里相当于temp对000000001按位和，那么只有第一位为1的时候，才会是1，正好是我们想要的结果。

temp = (temp>>1);

count++;

//右移一位，相当于不要了，这里最好注意一下运算符优先级

}

//我们得到了二进制的数组，但是是反过来的，不过刚刚好，因为我们运算回去，也是要反过来运算。

//前8位不用关心，我们直接从第8位开始算，思路上文已经说了

int page = 0;//页号

page = binary\_to\_int(binary,8,16);

int frame = -1;

//实现map很麻烦，这里我们就反过来使用我们之前的map[]数组，key用来存放帧表，value放页表，搜查我们使用便利

//在实际中，应该是使用map，rb\_tree来实现这一对应的功能，但是老师说语言不是问题，我觉得完成任务，这二者是没有差别的

//搜查TLB

if(lRUCacheGet(page\_TLB,page) == -1){

//TLB命中失败

printf("TLB命中失败 ");

}else{

frame = lRUCacheGet(page\_TLB,page);

TLBHitTimes++;

printf("TLB命中成功！ TLBHitTimes : %d ",TLBHitTimes);

}

//搜查页表

if(frame == -1){

if(lRUCacheGet(page\_map,page) == -1){

page\_default++;

printf("出现了一次缺页！ page\_default : %d", page\_default);

//这时我们要将最久没用的 page-frame 置换出去，那首先我们要得到对应的frame值，

//作为我们的输入对象，这与传统的lru数据结构是有一点小小的偏差的注意。

frame = lRUCacheGetoldest(page\_map);

if(frame == -1){

//说明根本就没有满，那我们就直接分配就好了

if(point\_frame == 128){

point\_frame = 0;

//理论上来说这里是不需要的，因为页表以及满了

}

lRUCachePut(page\_map,page,point\_frame);

frame = point\_frame;

point\_frame++;

}else{

//满了哦

lRUCachePut(page\_map,page,frame);

}

}else{

//没有缺页，直接完成任务

frame = lRUCacheGet(page\_map,page);

}

//更新TLB，因为这个时候我们已经知道了page和frame，直接加就完事了，因为这个数据结构已经很完善了

lRUCachePut(page\_TLB,page,frame);

}

//现在我们得到了我们想要的数据，一个是偏移量，一个是对应的帧表数值，结果是要让我们返回二者和起来的数字。

//frame + offset 和起来的数值，以及对应的value

int phyAddress[16];

int y = 0;

for(y = 0; y < 8; y++){

phyAddress[y] = binary[y];

}

for ( y = 8; y < 16; y++)

{

phyAddress[y] = 0;

}

int phyCount = 8;

int temp\_f = frame;

while (temp\_f != 0)

{

phyAddress[phyCount] = temp\_f&1;

temp\_f= (temp\_f>>1);

phyCount++;

}

int physicalAddress = binary\_to\_int(phyAddress,0,16);

printf("TLBHitTimes %d page\_default %d \n", TLBHitTimes,page\_default);

}

double TLB\_p = (double)TLBHitTimes/1000;

double page = (double)page\_default/1000;

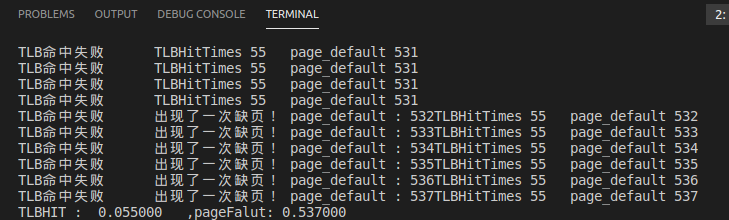
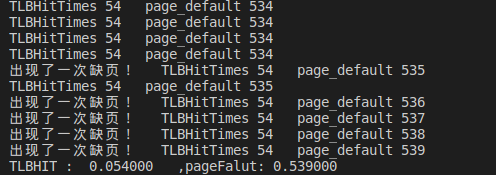
printf("TLBHIT : %lf ,pageFalut: %lf \n" ,TLB\_p ,page);

lRUCacheFree(page\_TLB);

lRUCacheFree(page\_map);

return 0;

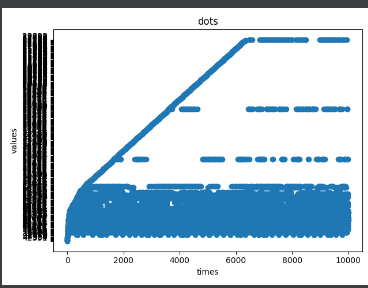
}

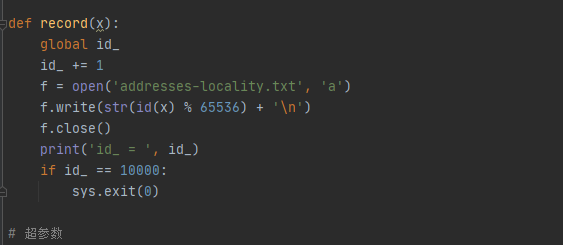


可以看到，这两个程序对随机的地址的结果是差不多一样的，符合预期。

1.2 自己实现trace

用py自带的plt包进行画点，我选择用x轴做1-10000，y轴做65535，我们可以看到6fbce575d7f84e634b1ef0a28547919说明存在很多情况时Y随着X增加而停留在一个区间的时候，是体现了内存的局部性。

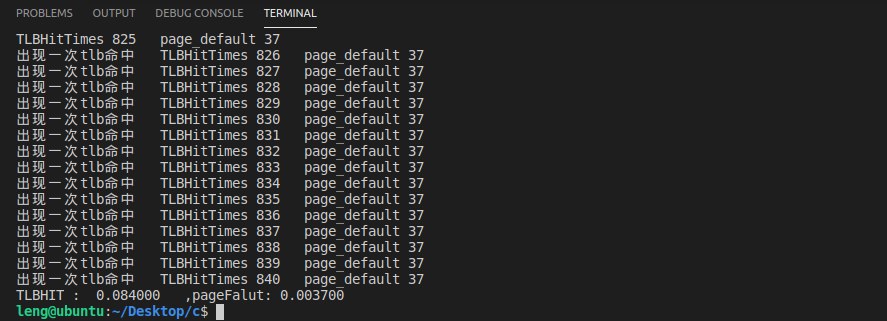




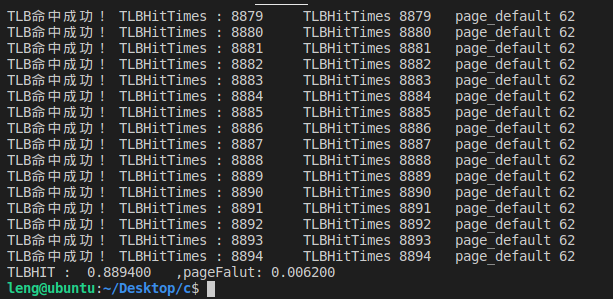
这样可以生成自己的addresses文件

然后，我去网上随便整了一个dqn控制倒立摆的小车pytorch代码来输出访问的内存地址。生成了对应的address-locality.txt

然后发现生成结果与随机生成的很不一样效果变得好了很多。

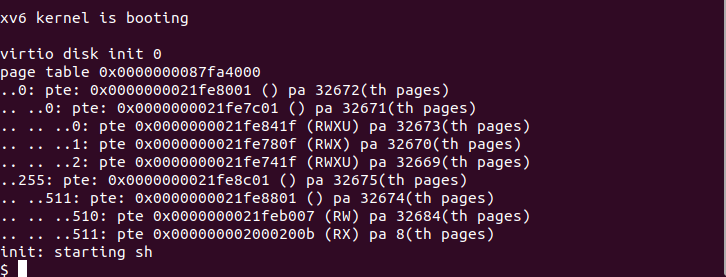


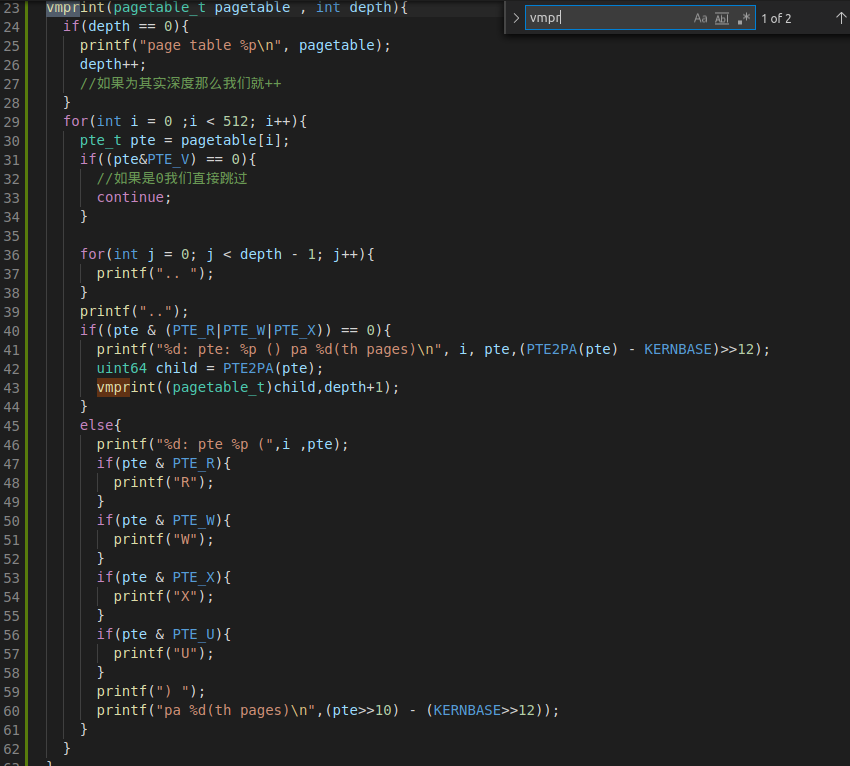
上图是先入先出的，可以看到比随机生成的效果要好很多

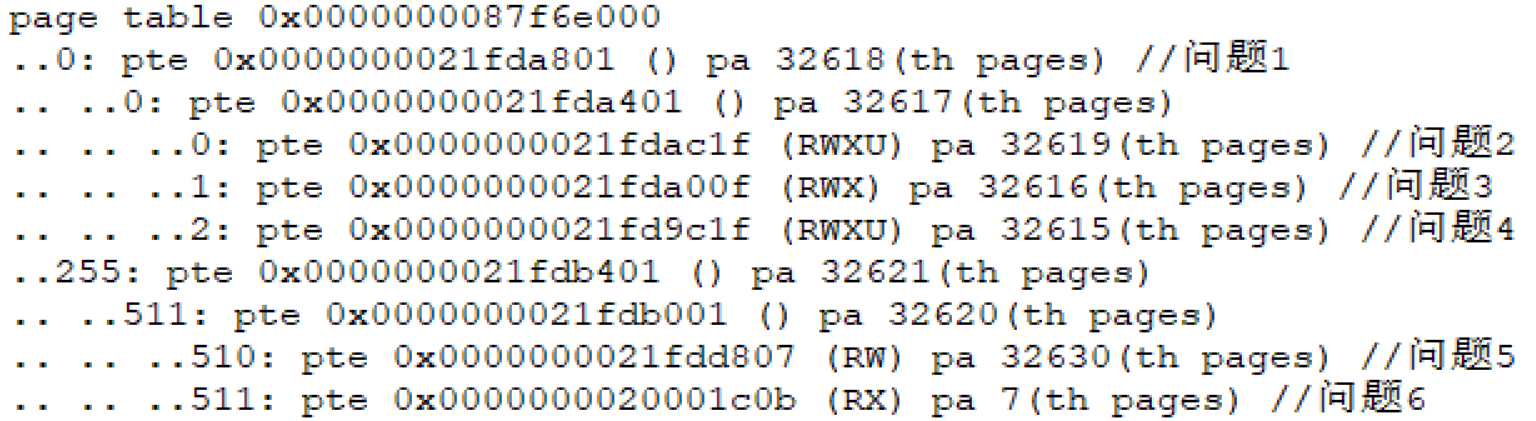


上图是LRU的，可以看出效果好了更多，并且远远好于FIFO算法。也充分解释了TLB快速缓存机制的必要性。

**第二题实验 page table**



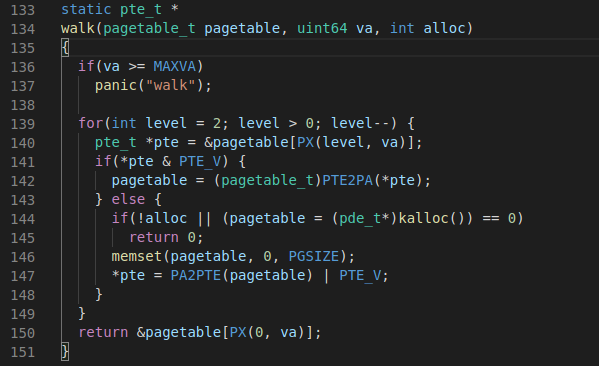




**问题1：为什么第一对括号为空？32618在物理内存的什么位置，为什么不从低地址开始？结合源代码内容进行解释。**

首先我没有切换分支，并且版本也不是最新的，所以在数值上有一定的差别。老师说不用在意。

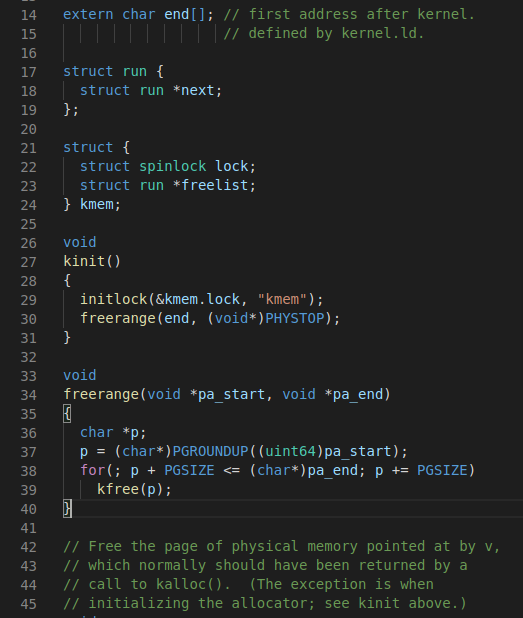
为什么二级页表的第一个括号为空？



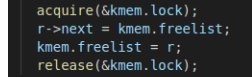
可以看到如果level = 2 的话，我们的页表的项都只有valid被取值，其他关键位并不会被取值。

那么为什么会优先从高位赋值呢？

通过上网查询相关知识，我得知：



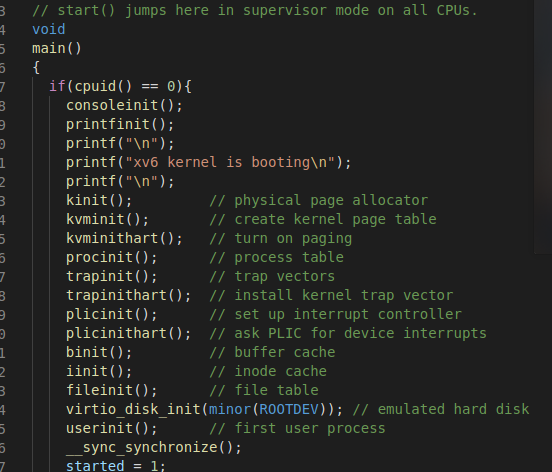
这里首先kinit()先initlock()锁住，然后再通过freerange()进行从end到PHYSTOP的内存清洗。是kfree()函数



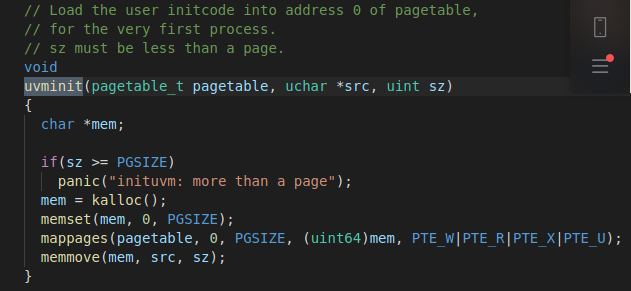
最关键的就是这两句，我们将清空的区域接在了freelist上。所以最先被分配的反而是高地址的。

**问题2：这是什么页？装载的什么内容？结合源代码内容进行解释。**

答：在main.c调用的创建第一个进程的函数userinit()中调用了分配用户页表的uvminit函数

****

而这个userinit也就是用户态的意思，他又回去调用vm.c的uvminit()的函数

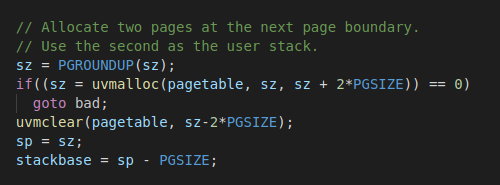
****

他将user initcode装载到虚拟地址对应的物理页，user initcode即exec init函数

问题3：这是什么页，有何功能？为什么没有U标志位？

上面那一页我们执行了userinit()函数执行完成后，通过scheduler()进程调度器，第一个调度的就回事init进程，它执行的就会使kernel/exec.c中，会进行如下的操作

只截取了一部分的关键代码



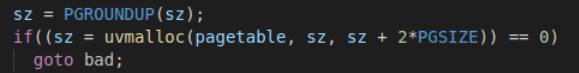
可以看到他分配了两页，并且取第二页在用户态下压入栈。

之后调用了uvmclear()，

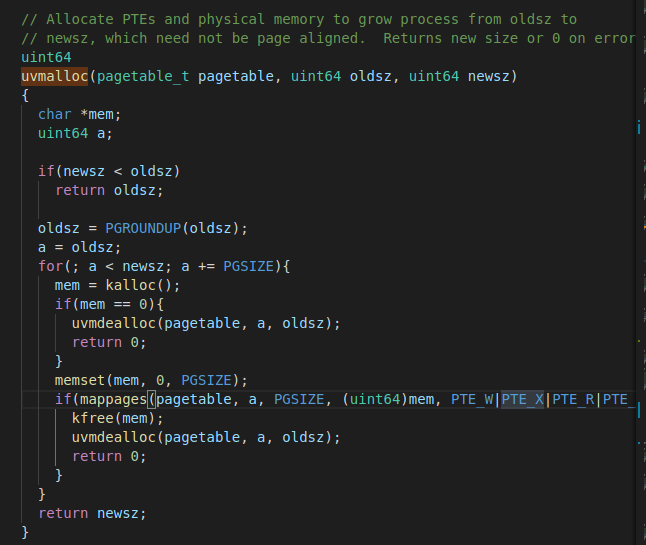
也就是说，这是一个user stack guard page页，从网上相关的知识，我们可以了解到，这个guard page能够去主动检测到用户堆栈溢出分配的堆栈内存，如果该堆栈溢出，进程尝试使用堆栈下的地址，硬件将生成页错误异常。并且不设置U标志位，即不允许用户进行

问题4：这是什么页？装载的什么内容？指出源代码初始化该页的位置。

该页的分配在上文的代码中以及提及



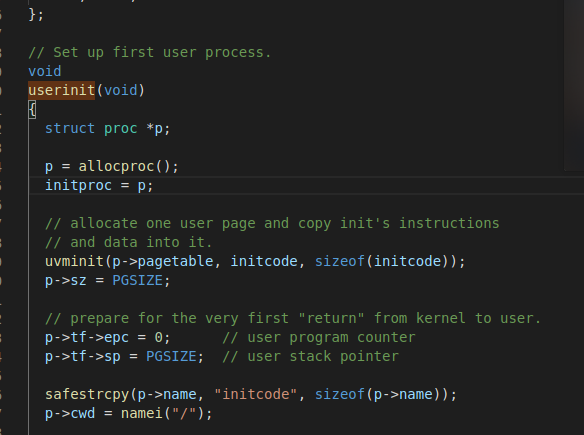
然后可以知道接下来的关键是uvmalloc来分配PTES和物理内存。



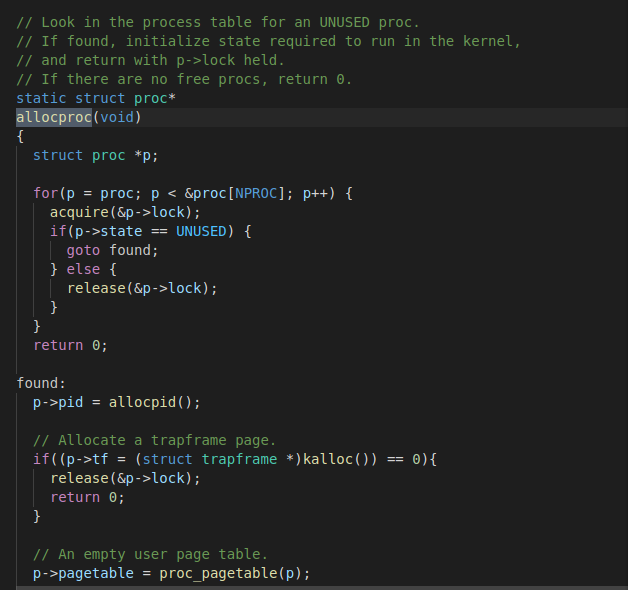
输入的oldsz比newsz小了两个PGSIZE即分配了两页，并且权限相同。并且我们装载的内容是输入的参数字符串argv，并且准备user stack中还装载了argv的指针

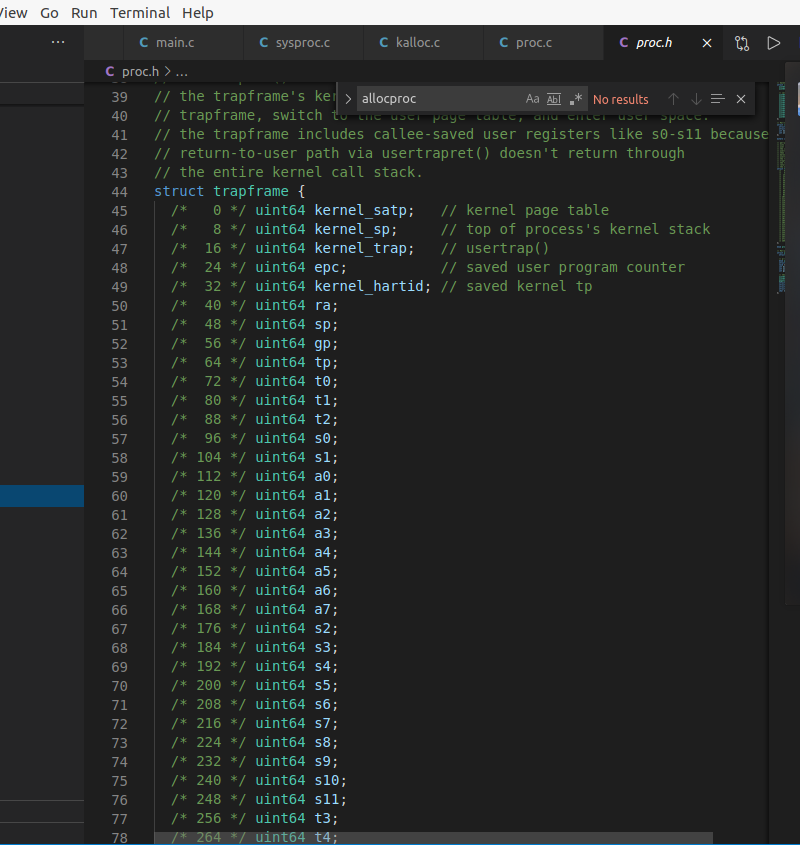
**问题5：这是什么页，为何没有X标志位？**

由图可知，该页的物理帧数最大，并且分配的最早，我们在初始化第一个进程的函数userinit中可以看到，

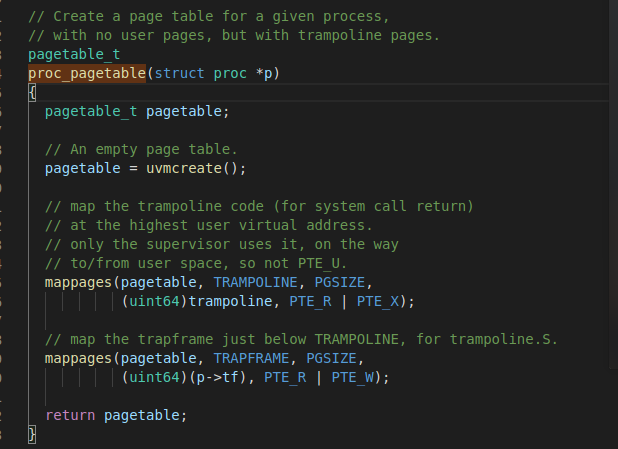


最开始调用了相关的allocproc()函数。



我们可以看到，allocproc()最先为p->trapframe分配了物理内存，这也是该pte的物理页帧 (pa) 最大的原因。然后，在该函数下，又调用了proc\_pagetable()函数来进行页表的操作.那么也就是说这一页是trapframe页，该页在虚拟内存中的映射始终是TRAPFRAME，是不变的。它存储的主要内容在/proc.h存储了内核的页表和各类寄存器的信息，这些都不是指令性质的内容，因此该页没有X标志位。

**是什么页，为何没有W标志位？装载的内容是什么？为何这里的物理页号处于低地址区域（第7页）？结合源代码对应的操作进行解释。**

最后这页是trampoline它的初始化是通过proc\_pagetable()来完成的

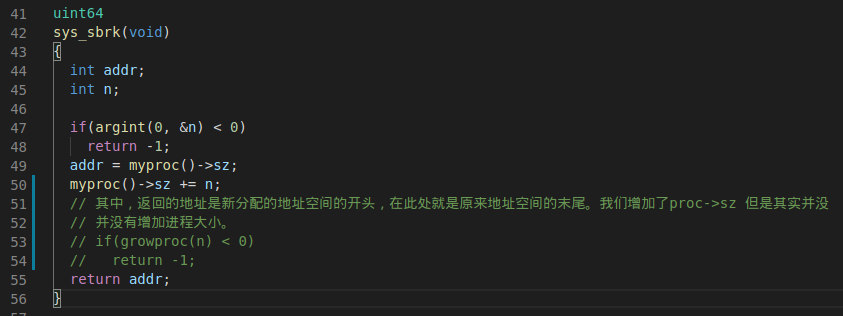
我们可以看出来，这一页并没有设置W标志位，因为这页是专门用来切换模式的，并且因为只有内核才能使用，所以U W都没有设置。

它装载的内容从右方可以看出，它成功的映射到了相同的虚拟地址，使得这一页在用户和内核空间切换页表时，仍然继续工作。

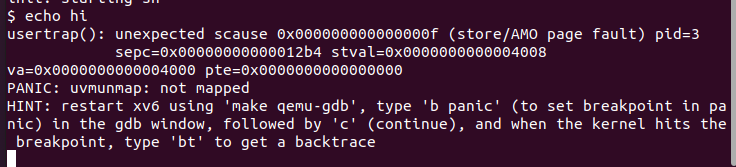
# 3.**lazy page allocation**

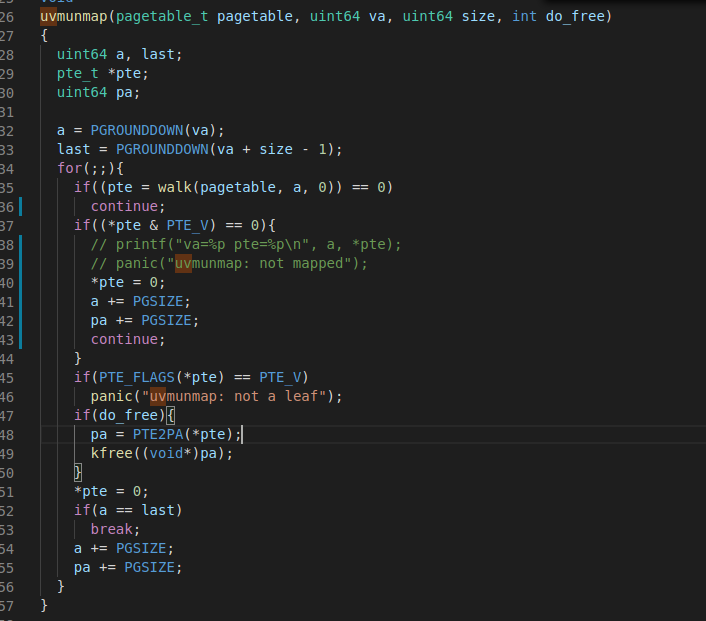
懒加载的核心就是当我们程序想要分配内存的时候，我们并不会直接将其分配，二十等待进程真正使用的时候才真正的进行分配。

首先对系统调用的sbrk的sys\_sbrk()进行修改。只将进程的内存空间大小增加，而不进行实际的分配。实际代码如下

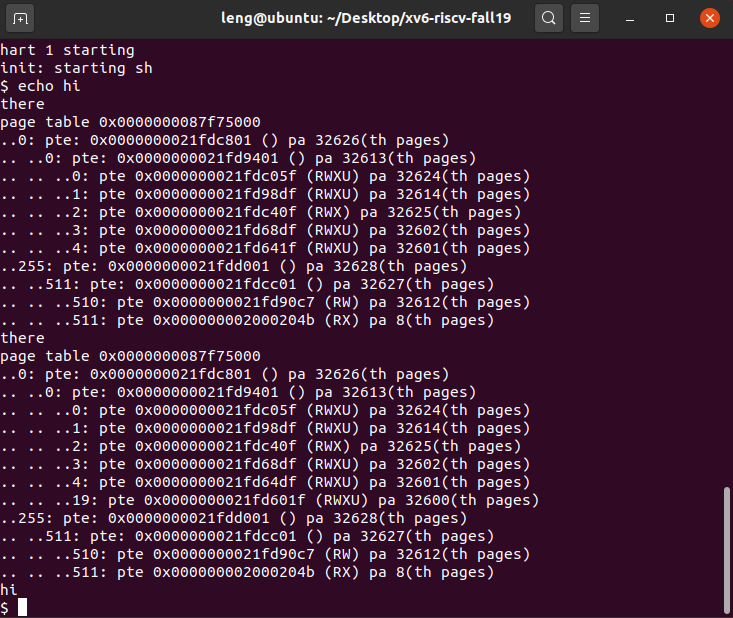
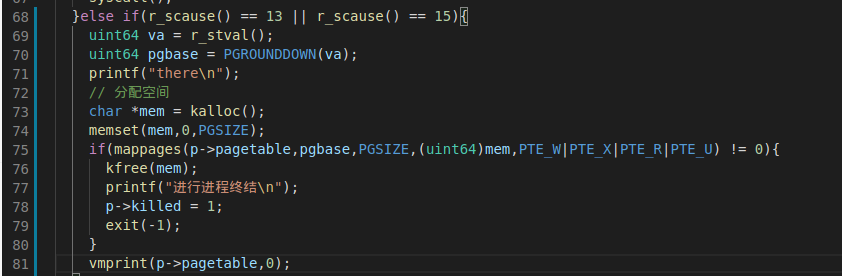


但是会报错



我们可以看到，他这个的意思是uvmunmap说你没有分配空间，不行。我们去修改这个函数，让他在我们的正常分支中直接结束流程而不进行报错

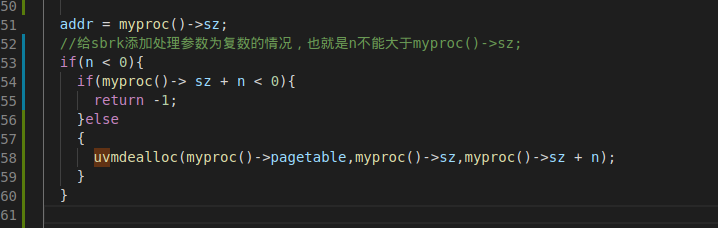
最后再修改一下trap就可以完成



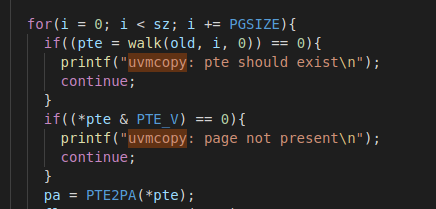
可以看到他成功的输出了相关的页表内容，并且也成功了返回了hi

3.2 Lazytests and Usertests

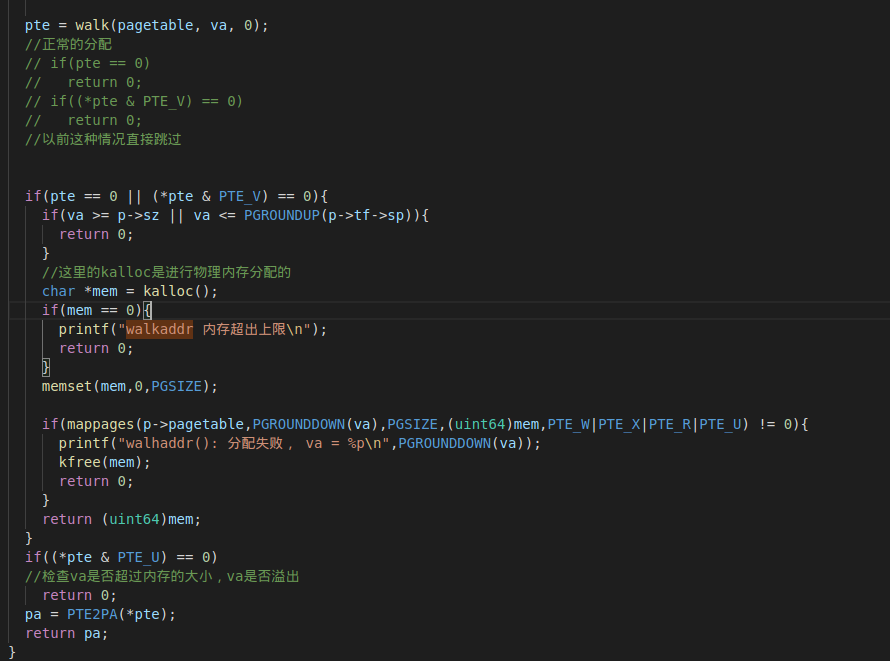
首先我们根据提示去给sbrk添加处理参数为负数的情况，就是dealloc相应的内存n，注意n不能大于p->sz

1. 

通过看官方视频和查询相关网络只是我们可以知道，为了完成懒加载我们还要去修改一下系统复制进程时的代码，还是蛮关键的，因为你fork是会经常用到的。



最后我们还要去修改一下核心的地址转换，就是说当我们想要得到地址时，但是得不到物理地址时，需要去添加相应的物理地址映射到page table 里，这里关键就是要找到系统调用。二这里老师上课也讲过相关的知识。时再walkaddr()里，我们再修改一下，代码里面有详细的注释



然后在官网以及对应的视频上也有相关的提示是最关键的usertrap函数

