

OS ASSIGNMENT2

1. 虚拟内存管理模拟程序

1.1 (1) 地址转换

本题中需要编写一个程序，为大小为 $2^{16}=65536$ 字节的虚拟地址空间将逻辑地址转换到物理地址。这个程序将从包含逻辑地址的文件中读取，通过 TLB 和页表将每个逻辑地址转换到物理地址。这个程序将从包含逻辑地址的文件中读取，通过 TLB 和页表将每个逻辑地址转换为对应的物理地址，并且输出在转换的物理地址处存储的字节值。简要分析题目，程序可以分为以下几个步骤实现：

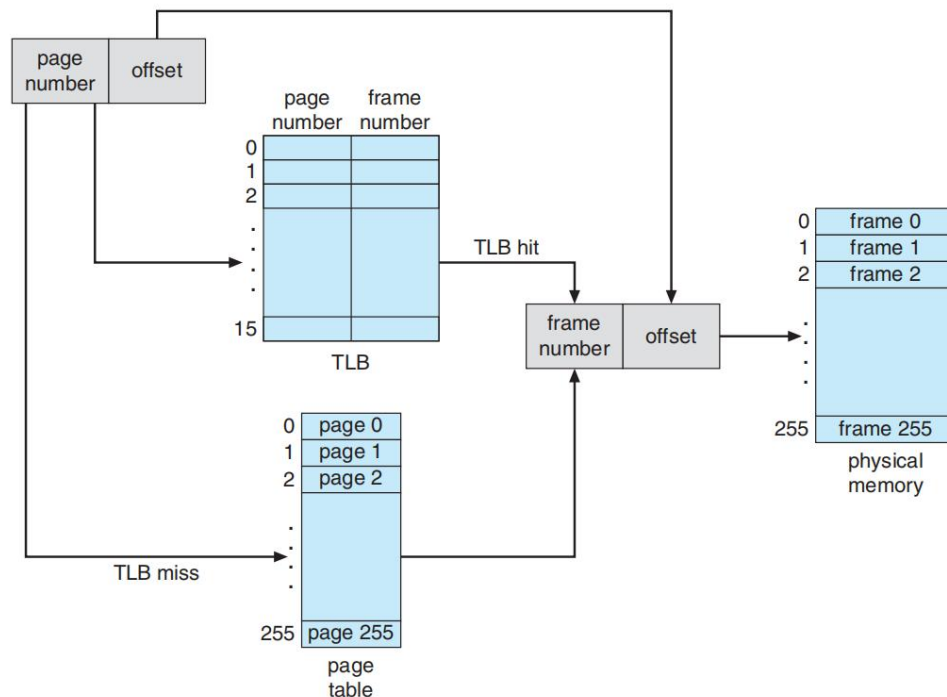


图 1 地址转换过程的表示

(1) 从 addresses.txt 读取逻辑地址

```

int addresses[1000]; // 声明存储逻辑地址的数组
FILE *fid; // 输入文件指针
FILE *fptr; // 输出文件指针
fid=fopen("addresses.txt","r"); // 打开存储逻辑地址的文件
fptr=fopen("out.txt","w+"); // 打开输出文件
int i=0;
while(fscanf(fid,"%d",&addresses[i])!=EOF){
    i++;
} // 读取逻辑地址
fclose(fid); // 关闭文件
  
```

(2) 建立页表、TLB 以及物理内存

```
int table[256]; // 页表
char memory[256][256]; // 物理内存
int frame_index=0; // 帧索引，方便物理内存的访问
int TLB[16][2]; // TLB[i][0] 存储页码，TLB[i][1] 存储对应帧码
int TLB_index=0; // TLB 索引
for(i=0; i<16; i++){
    for(int j=0; j<2; j++){
        TLB[i][j]=-1; // 将 TLB 设定为未存取状态
    }
}
for(int j=0; j<256; j++){
    table[j]=-1; // 将页表设定为未存取状态
}
```

(3) 将逻辑地址转换为二进制，方便读取页码和偏移

```
int temp;
int binary[1000][16]; // 将逻辑地址转换为二进制储存在二维数组 binary 中
for(i=0; i<1000; i++){
    temp=addresses[i];
    int j=0;
    while(j<16){
        if(temp>=1){
            binary[i][j]=temp%2;
            temp=temp/2;
        }
        else
            binary[i][j]=0;
        j++;
    }
}
```

(4) 由二进制逻辑地址计算获取页码和偏移，由下图可知 0~7 位为偏移，8~15 位为页码

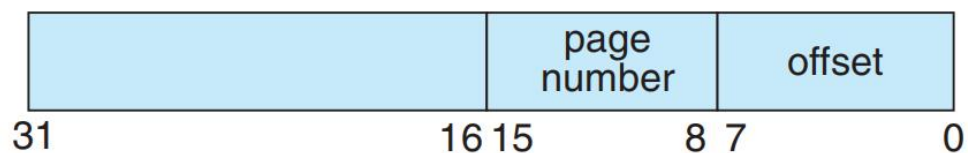


图 2 逻辑地址结构

```
int offset=0;
for(int j=0; j<8; j++){
    offset=offset+binary[i][j]*pow(2,j); // 由逻辑地址计算获取偏移
}
int page=0;
```

```
for(int j=8;j<16;j++)
```

```
page=page+binary[i][j]*pow(2,j-8); //由逻辑地址计算获取页码
```

(5) 从页表、TLB 中查找帧码，在 TLB 未命中、访问页表出现缺页的情况下，从 BACKING_STORE.bin 中读取页面并存入物理内存

```
int frame;
```

```
//在 TLB 中查找
```

```
int flag=0; //TLB 是否查找成功的标记值
```

```
for(int j=0;j<16;j++){
```

```
if(TLB[j][0]==page){
```

```
frame=TLB[j][1];
```

```
flag=1; //查找成功
```

```
}
```

```
}
```

```
if(flag!=1){ //TLB 查找失败，开始查找页表
```

```
if(table[page]==-1){
```

```
FILE *file; //文件指针
```

```
char *buffer; //缓冲区
```

```
file=fopen("BACKING_STORE.bin","rb"); //打开后备存储文件
```

```
fseek(file,256*page,0); //定位到页码对应位置
```

```
buffer=(char*)malloc(257); //分配内存
```

```
fread(buffer,1,256,file); //读入数据
```

```
for(int j=0;j<256;j++){
```

```
memory[frame_index][j]=buffer[j]; //存入物理内存中帧码对应位置
```

```
free(buffer); //释放内存
```

```
fclose(file); //关闭文件
```

```
table[page]=frame_index; //给页表中页码分配帧码
```

```
//将页码和帧码存入 TLB
```

```
if(TLB_index<16){
```

```
TLB[TLB_index][0]=page; //存入页码
```

```
TLB[TLB_index][1]=frame_index; //存入帧码
```

```
TLB_index++;
```

```
}
```

```
frame=frame_index;
```

```
frame_index++;
```

```
}
```

```
else //页表查找成功
```

```
frame=table[page];
```

```
}
```

(6) 由帧码和偏移计算得到物理地址，并输出至 out.txt

```
int phyaddr;
```

```
phyaddr=frame*256+offset; //由帧码和偏移计算物理地址
```

```
//将逻辑地址、物理地址以及物理地址处带符号字节的值
```

```
char output[100]="Virtual address: ";
```

```
char VirtualAddress[10];
```

```

sprintf(VirtualAddress,"%d",addresses[i]);
strcat(output,VirtualAddress);
strcat(output," Physical address: ");
char PhysicalAddress[10];
sprintf(PhysicalAddress,"%d",phyaddr);
strcat(output,PhysicalAddress);
strcat(output," Value: ");
char Value[10];
sprintf(Value,"%d",memory[frame][offset]);
strcat(output,Value);
strcat(output,"\n");
fputs(output,fptr);

```

地址转换测试:

```

#!/bin/bash -e

echo "Compiling"

gcc vm.c -o vm -lm

echo "Running vm"

./vm BACKING_STORE.bin addresses.txt > out.txt

echo "Comparing with correct.txt"

diff out.txt correct.txt

```

测试结果:

```

osc@ubuntu:~/final-src-osc10e/ch10$ bash test.sh
Compiling
Running vm
Comparing with correct.txt

```

out.txt 内容截图:

```

Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
Virtual address: 64815 Physical address: 2095 Value: 75
Virtual address: 18295 Physical address: 2423 Value: -35
Virtual address: 12218 Physical address: 2746 Value: 11
Virtual address: 22760 Physical address: 3048 Value: 0
Virtual address: 57982 Physical address: 3198 Value: 56
Virtual address: 27966 Physical address: 3390 Value: 27
Virtual address: 54894 Physical address: 3694 Value: 53

```

由测试结果可知, **correct.txt** 与 **out.txt** 内容完全相同, 证明地址转换成功。

1.1 (2) 页面置换

本题中分别实现基于 LRU 和 FIFO 的 Page Replacement，打印比较 Page-fault rate 和 TLB hit rate（TLB 和页置换统一策略）

FIFO 策略：

首先我们要搞懂什么是 FIFO。FIFO(First Input First Output)，即先进先出队列，在本题目中指的是当 TLB 和页表达到最大容量后，再从逻辑地址中读取到新的页码，需要进行 TLB 和页表的页置换，此时，最早存入 TLB 和页表的页码及其对应的帧码需要被移出，使得新的页码及其对应的帧码可以移入 TLB 和页表。程序可以分为以下几个步骤实现：

(1) 将第 1 题中页表的数据结构也改成和 TLB 一样，将页表视作一个大得多的 TLB，即一个 $length*2$ 的二维数组，用 `table[index][0]` 存储页码，用 `table[index][1]` 存储帧码

```
int table[length][2]; // 页表
int table_index=0; // 页表索引
for(i=0; i<length; i++){
    for(int j=0; j<2; j++){
        table[i][j]=-1; // 将页表设定为未存取状态
    }
}
```

(2) 考虑到这是一个顺序数组，我们很容易知道每个页码存入地先后顺序。当 `table` 未滿时，依次存入新的页码和帧码；当需要进行页面置换时，`table[0][0]` 和 `table[0][1]` 被移除数组，数组中存储的其他页码及其帧码则依次地向前移一位，最后将新的页码及分配的帧码分别存入 `table[length-1][0]` 和 `table[length-1][1]`。TLB 的 FIFO 页面置换策略同理。

```
if(frame_index>=256)
    frame_index=0; // 所有帧码分配完后，将 frame_index 置 0，重新分配帧码
// 将页码和帧码存入 TLB
if(TLB_index<16){ // TLB 未滿
    TLB[TLB_index][0]=page; // 存入页码
    TLB[TLB_index][1]=frame_index; // 存入帧码
    TLB_index++;
}
else{ // TLB 已滿，FIFO 策略更新 TLB
    for(int j=0; j<15; j++){ // 将第一个进入 TLB 的页码剔除
        TLB[j][0]=TLB[j+1][0];
        TLB[j][1]=TLB[j+1][1];
    }
    TLB[15][0]=page; // 存入新页码
    TLB[15][1]=frame_index; // 存入新帧码
}
// 将页码和帧码存入页表
if(table_index<length){
```

```

        table[table_index][0]=page;
        table[table_index][1]=frame_index;
        table_index++;
    }
    else{//页表已满，FIFO 策略更新页表
        for(int j=0;j<length-1;j++){//将第一个进入页表的页码剔除
            table[j][0]=table[j+1][0];
            table[j][1]=table[j+1][1];
        }
        table[length-1][0]=page;//存入新页码
        table[length-1][1]=frame_index;//存入新帧码
    }
    frame=frame_index;
    frame_index++;
(3) 计算缺页率以及 TLB 命中率
    double page_fault_rate;//缺页率
    page_fault_rate=(double)page_fault/1000;
    double TLB_hit_rate;//TLB 命中率
    TLB_hit_rate=(double)TLB_hit/1000;
    printf("缺页率:%lf\nTLB 命中率:%lf\n",page_fault_rate,TLB_hit_rate);

```

运行结果：

```

缺页率:0.538000
TLB命中率:0.053000

```

由运行结果可知缺页率达到 53.8%，而 TLB 命中率仅有 5.3%。

LRU 策略：

LRU 是 Least Recently Used 的缩写，即最近最少使用，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t ，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。在本题中使用另一种思路获得最近最少使用的页面，程序可由以下几个步骤实现：

(1) 页表和 TLB 的数据结构沿用 FIFO 中的顺序二维数组，页面置换的策略仍旧是从队列中删除队头的页码及帧码，将新页码及帧码加入队尾

```

        if(TLB_index<16){//TLB 未满
            TLB[TLB_index][0]=page;//存入页码
            TLB[TLB_index][1]=frame_index;//存入帧码
            TLB_index++;
        }
        else{//TLB 已满，LRU 策略更新 TLB
            for(int j=0;j<15;j++){//将最长时间未使用的页码剔除
                TLB[j][0]=TLB[j+1][0];

```

```

        TLB[j][1]=TLB[j+1][1];
    }
    TLB[15][0]=page;//存入新页码
    TLB[15][1]=frame_index;//存入新帧码
}
//将页码和帧码存入页表
if(table_index<length){
    table[table_index][0]=page;
    table[table_index][1]=frame_index;
    table_index++;
}
else{//页表已满，LRU 策略更新页表
    for(int j=0;j<length-1;j++){//将最长时间未使用的页码剔除
        table[j][0]=table[j+1][0];
        table[j][1]=table[j+1][1];
    }
    table[length-1][0]=page;//存入新页码
    table[length-1][1]=frame_index;//存入新帧码
}
}

```

(2) 唯一的不同是，每一次访问一个已在页表和 TLB 中的页码时，需要将该页面移至队尾并更新队列，从而保证队列中越靠前的页面，自上次访问以来所经历的时间最长，这样一来当需要进行页面置换时，最近最少使用的页面就在队头，直接将队头删除，在队尾补入新页面，即可实现 LRU 页面置换

```

for(int j=0;j<16;j++){
    if(TLB[j][0]==page){
        frame=TLB[j][1];
        flag=1;//TLB 查找成功
        TLB_hit++;//TLB 命中次数+1
        if(j!=TLB_index-1){
            //如果访问的页不在队尾，更新队列，将访问的页移至队尾
            int temp1,temp2;
            temp1=TLB[j][0];
            temp2=TLB[j][1];
            for(int k=j;k<TLB_index-1;k++){
                TLB[k][0]=TLB[k+1][0];
                TLB[k][1]=TLB[k+1][1];
            }
            TLB[TLB_index-1][0]=temp1;
            TLB[TLB_index-1][1]=temp2;
        }
        break;
    }
}
}

```

```

for(int j=0;j<length;j++){
    if(table[j][0]==page){
        frame=table[j][1];
        sign=1;//页表查找成功
        if(j!=table_index-1){
            //如果访问的页不在队尾，更新队列，将访问的页移至队尾
            int temp1,temp2;
            temp1=table[j][0];
            temp2=table[j][1];
            for(int k=j;k<table_index-1;k++){
                table[k][0]=table[k+1][0];
                table[k][1]=table[k+1][1];
            }
            table[table_index-1][0]=temp1;
            table[table_index-1][1]=temp2;
        }
        break;
    }
}
}

```

(3) 计算缺页率以及 TLB 命中率

```

double page_fault_rate;//缺页率
page_fault_rate=(double)page_fault/1000;
double TLB_hit_rate;//TLB 命中率
TLB_hit_rate=(double)TLB_hit/1000;
printf("缺页率:%lf\nTLB 命中率:%lf\n",page_fault_rate,TLB_hit_rate);

```

运行结果：

```

缺页率:0.554000
TLB命中率:0.056000

```

由运行结果可知缺页率达到 55.4%，TLB 命中率为 5.6%。

1.2 内存访问的局部性

编写一个简单 trace 生成器程序，运行生成自己的 addresses-locality.txt，包含 10000 条访问记录，体现内存访问的局部性，并比较 FIFO 和 LRU 策略下的性能指标。

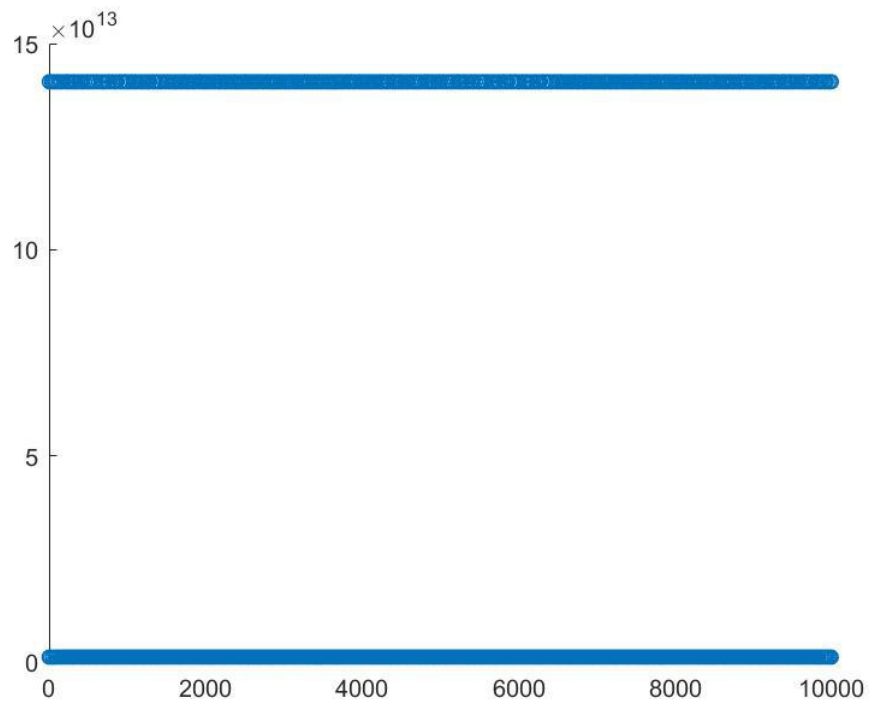
绘制地址分布图：

trace 程序将一个神经网络的 python 程序中用到的地址记录下来，由于不停重复调用子函数，因此形成了如下图所示的地址分布。调用不同子函数时访问的内存有很大间隔，而再次调用同一子函数时访问的内存间隔很小，体现了内存访问的局部性。


```

y=table2array(addresseslocality);
for i=1:10000
    x(i)=i;
end
x=reshape(x, 10000, 1);
scatter(x, y);

```



地址分布图

比较不同策略下的性能指标:

FIFO:

缺页率:0.058000
TLB命中率:0.500000

LRU:

缺页率:0.058000
TLB命中率:0.612000

通过比较我们可以发现，两种策略的缺页率相同，但是 LRU 的 TLB 有着更高的 TLB 命中率，因此可以得出 LRU 策略的性能优于 FIFO 的结论。

2 xv6-lab-2020 页表实验

2.1 打印页表内容

具体实现：

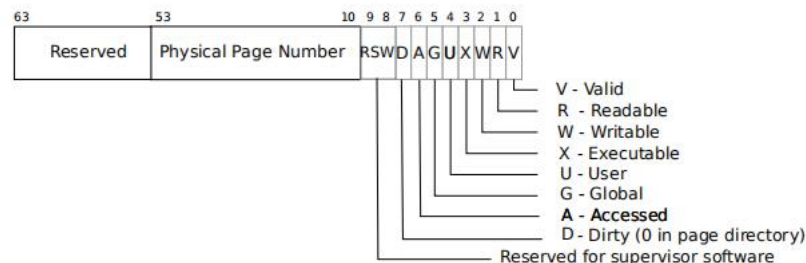


Figure 3.2: RISC-V address translation details.

1) 在 kernel/vm.c 中定义

```
static void recursion(paetable_t paetable, int level){
    for(int i=0; i<512; i++){
        pte_t pte=paetable[i];
        if(pte && PTE_V){ //PTE is valid
            // shift a physical address to the right place for a PTE.
            uint64 child=PTE2PA(pte);
            //题目中要求物理页后的数字表示第几个物理页帧，求解的方法即是用当前的
            //物理地址减去物理地址下界 KERNBASE，再除以每一页的字节大小
            uint64 result=(child-KERNBASE)/PGSIZE;
            //获取地址的 flag 位，根据上面的 Figure3.2，分别查询 RISC-V 地址每一位的信息
            //来获取页表权限
            uint64 flag=PTE_FLAGS(pte);
            uint64 judge[8];
            for(int i=0; i<8; i++){
                judge[i]=flag%2;
                flag=flag/2;
            }
            char *output="";
            if(judge[1]==1 && judge[2]==1 && judge[3]==1 && judge[4]==1){
                output="RWXU";
            }
            else if(judge[1]==1 && judge[2]==1 && judge[3]==1 && judge[4]!=1){
                output="RWX";
            }
            else if(judge[1]==1 && judge[2]==1 && judge[3]!=1 && judge[4]==1){
                output="RW";
            }
            else if(judge[1]==1 && judge[2]!=1 && judge[3]==1 && judge[4]==1){
```

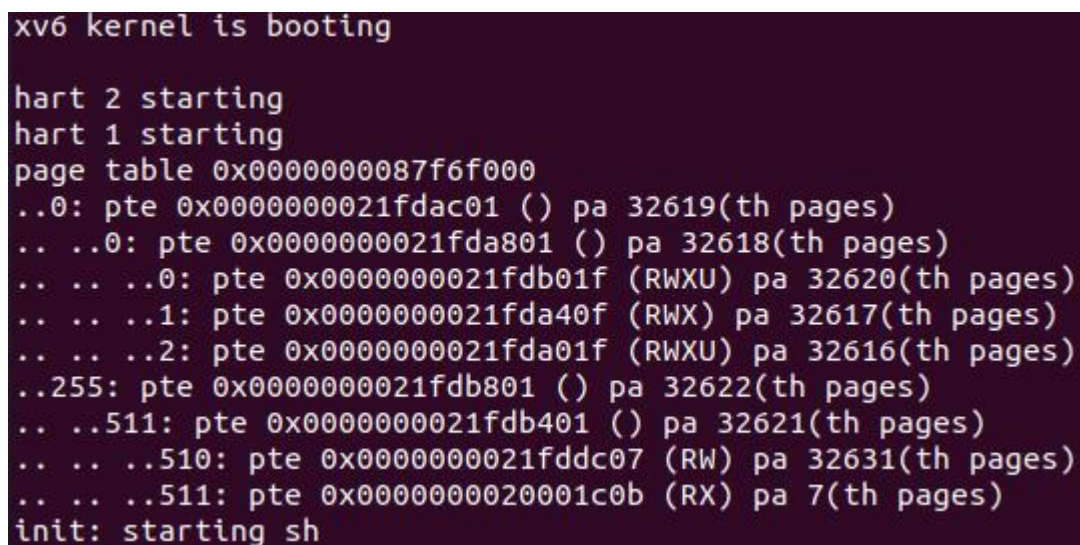

3) 在 kernel/exec.c 中的 return argc 之前增加对 vmprint 的调用

```
// Commit to the user image.
oldpagetable = p->pagetable;
p->pagetable = pagetable;
p->SZ = SZ;
p->trapframe->epc = elf.entry; // initial program counter = main
p->trapframe->sp = sp; // initial stack pointer
proc_freepagetable(oldpagetable, oldsz);

if(p->pid==1)
    vmprint(p->pagetable);

return argc; // this ends up in a0, the first argument to main(argc,
argv)
```

运行结果:



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f6f000
..0: pte 0x0000000021fdac01 () pa 32619(th pages)
.. ..0: pte 0x0000000021fda801 () pa 32618(th pages)
.. .. ..0: pte 0x0000000021fdb01f (RWXU) pa 32620(th pages)
.. .. ..1: pte 0x0000000021fda40f (RWX) pa 32617(th pages)
.. .. ..2: pte 0x0000000021fda01f (RWXU) pa 32616(th pages)
..255: pte 0x0000000021fdb801 () pa 32622(th pages)
.. ..511: pte 0x0000000021fdb401 () pa 32621(th pages)
.. .. ..510: pte 0x0000000021fddc07 (RW) pa 32631(th pages)
.. .. ..511: pte 0x0000000020001c0b (RX) pa 7(th pages)
init: starting sh
```

2.2 回答问题

问题 1:

第一对括号为空的原因: vm.c 的 walk 函数中, 在 for 循环中从 level=2 开始, 页表权限项就只有 PTE_V 会被取值, 其他关键位会被忽略; 当 level=1 时, 页表权限项没有被取值, 所以第一对括号为空。

不从低地址开始的原因: 内核加载到内存后, main() 函数中首先调用 kinit 函数, 由于内核实际能用的虚拟地址空间是不足已完成初始化工作的, 所以初始化过程中需要重新设置页表。需要调用 kalloc.c 中的 kinit、freerange 和 kfree 函数, 函数运行步骤如下:

1. Kinit() 被 initlock() 锁死
2. 通过 freerange() 进行从 end 到 PHYSTOP 的内存删除
3. kfree() 中将 freerange() 删除的内存空间指向了 freelist, 因为删除的为高地址空间, 而进程又按照 freelist 的顺序运行, 所以从高地址开始。

问题 2:

这一页装载的是可执行 ELF 目标文件中的代码段和数据段。这个页的物理内存分配由下图中的 `uvmalloc` 函数完成，在下图所示的代码中，`exec` 函数对 ELF 中的每个代码段和数据段分配页表(`uvmalloc`)，并将其加载进内存中(`loadseg`)。

```
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64*)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
        goto bad;
    uint64 sz1;
    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
        goto bad;
    sz = sz1;
    if(ph.vaddr % PGSIZE != 0)
        goto bad;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
        goto bad;
}
```

问题 3:

这一页是用户栈的 **guard page**。阅读代码可以发现，这一页与问题 4 所指的页一起分配，但只有后者被作为用户的栈使用，可知前者并不具有用户权限，因此没有 **U** 标志位。

问题 4:

这一页负责存储用户的栈中的内容，比如 `exec` 的参数 `argc`、`argv` 等。源代码中初始化该页的位置如下。

```
// Allocate two pages at the next page boundary.
// Use the second as the user stack.
sz = PGROUNDUP(sz);
uint64 sz1;
if((sz1 = uvmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
sz = sz1;
uvmclear(pagetable, sz-2*PGSIZE);
sp = sz;
stackbase = sp - PGSIZE;
```


问题 5:

这一页存储的是 **TRAMPOLINE** 下面的 **trapframe**，没有 **X** 标志位的原因是 **trapframe** 是中断、自陷、异常进入内核后，在堆栈上形成的一种数据结构，不是可执行的代码段或数据段，源代码中初始化该页的位置如下。

```
// map the trapframe just below TRAMPOLINE, for trampoline.S.  
if(mappages(pagetable, TRAPFRAME, PGSIZE,  
            (uint64)(p->trapframe), PTE_R | PTE_W) < 0){  
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);  
    uvmfree(pagetable, 0);  
    return 0;  
}
```

问题 6:

这一页存储的是 **TRAMPOLINE**，没有 **W** 标志位的原因是 **TRAMPOLINE** 是没有写入权限的。这里的物理页号处于低地址区域的原因是 **TRAMPOLINE** 在内核和用户地址空间中的地址相同，其地址映射方式如下图所示。

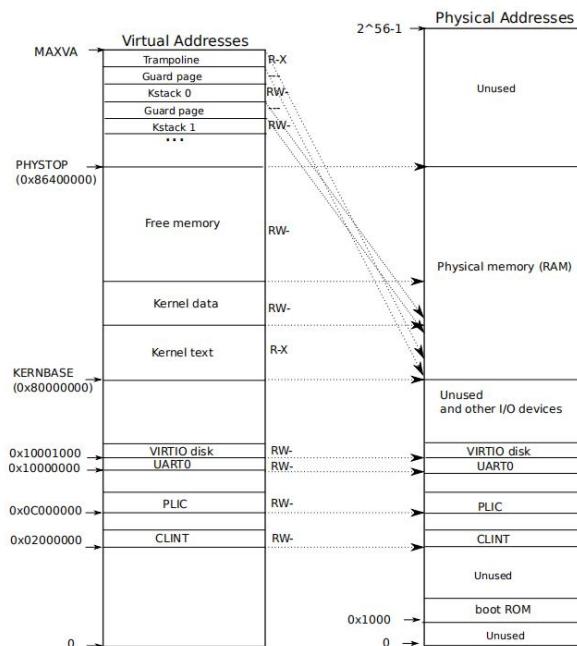


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

源代码中初始化该页的位置如下

```
// map the trampoline code (for system call return)  
// at the highest user virtual address.  
// only the supervisor uses it, on the way  
// to/from user space, so not PTE_U.  
if(mappages(pagetable, TRAMPOLINE, PGSIZE,  
            (uint64)trampoline, PTE_R | PTE_X) < 0){  
    uvmfree(pagetable, 0);  
    return 0;  
}
```

3 xv6-lab-2020 内存分配实验

3.1 Lazy allocation

具体实现:

1) 在 trap.c 中修改 usertrap()函数

```
else if(r_scause()==13||r_scause()==15){
    //13:Page load fault,15:Page store fault
    //get the virtual address that causes the page fault
    uint64 fault_vaddr=(uint64)r_stval();
    //round the faulting virtual address down to a page boundry
    uint64 vpagebase;
    vpagebase=PGROUNDDOWN(fault_vaddr);
    //allocate a new page
    char *mem;
    mem=kalloc();
    if(mem==0){//fail to kalloc
        printf("kalloc out of memory!\n");
        p->killed=1;//kill the process
    }
    else{
        //map the page
        memset(mem,0,PGSIZE);
        if(mappages(p->pagetable,vpagebase,PGSIZE,(uint64)mem,PTE_W|PTE_U)<0)
        {
            //fail to map
            kfree(mem);
            p->killed=1;
        }
    }
}
```

2) 修改 uvmunmap()函数防止报错

```
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    uint64 a;
    pte_t *pte;

    if((va % PGSIZE) != 0)
        panic("uvmunmap: not aligned");

    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            continue;
        //panic("uvmunmap: walk");
        if((*pte & PTE_V) == 0)
            continue;
        //panic("uvmunmap: not mapped");
        if(PTE_FLAGS(*pte) == PTE_V)
            panic("uvmunmap: not a leaf");
        if(do_free){
            uint64 pa = PTE2PA(*pte);
            kfree((void*)pa);
        }
        *pte = 0;
    }
}
```

echo hi 运行结果:

```
init: starting sh
$ echo hi
hi
```

3.2 Lazytests and Usertests

具体实现:

1) 处理 sbrk()参数为负数的情况

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0)
        return -1;

    struct proc*p=myproc();
    addr=p->sz;
    if(n<0){//when the argument is negative,allocate memory n
        if(p->sz+n<0)
            return -1;
        else
            uvmdealloc(p->pagetable,p->sz,p->sz+n);
    }
    p->sz+=n;
    //if(growproc(n) < 0)
    //return -1;
    return addr;
}
```

2) 当发现缺页异常时,读入虚拟地址比 p->sz 大,或者当虚拟地址比进程的用户栈还小,或者申请空间不够的时候终止进程

```
else if(r_scause()==13||r_scause()==15){
    //13:Page load fault,15:Page store fault
    //get the virtual address that causes the page fault
    uint64 fault_vaddr=(uint64)r_stval();
    if(fault_vaddr<p->sz&&fault_vaddr>PGROUNDDOWN(p->trapframe->sp)){
        //round the faulting virtual address down to a page boundry
        uint64 vpagebase;
        vpagebase=PGROUNDDOWN(fault_vaddr);
        //allocate a new page
        char *mem;mem=kalloc();
        if(mem==0){//fail to kalloc
            printf("kalloc out of memory!\n");p->killed=1;//kill the process
        }
        else{
            //map the page
            memset(mem,0,PGSIZE);
            if(mappages(p->pagetable,vpagebase,PGSIZE,(uint64)mem,PTE_U|PTE_W|PTE_R)!=0){
                //fail to map
                kfree(mem);p->killed=1;
            }
        }
    }
    else p->killed=1;
```


3) 修改 fork 函数中子进程复制父进程地址空间的过程，发现地址空间不存在时直接忽略

```
for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
        continue;
    //panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
        continue;
    //panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
        goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
        kfree(mem);
        goto err;
    }
}
```

4) 处理系统从进程调用一个有效地址时，该地址尚未分配内存的情况。遇到这种情况时，需要添加相应的物理地址映射到 page table 里。vm.c 中的 walkaddr 函数负责将虚拟地址转换为物理地址，修改 walkaddr 函数

```
uint64 walkaddr(pagetable_t pagetable, uint64 va)
{
    pte_t *pte;
    uint64 pa;
    if(va >= MAXVA)
        return 0; //Invalid virtual address
    pte = walk(pagetable, va, 0);
    if(pte == 0 || (*pte & PTE_V) == 0){
        if(va >= p->sz || va < PGROUNDUP(p->trapframe->sp))
            return 0;
        uint64 ka = (uint64)kalloc();
        if(ka == 0) //fail to kalloc
            return 0;
        if(mappages(p->pagetable, PGROUNDUP(va), PGSIZE, ka, PTE_W | PTE_X | PTE_R | PTE_U) != 0) { //fail to map
            kfree((void*)ka);
            return 0;
        }
        return ka;
    }
    if((*pte & PTE_U) == 0)
        return 0;
    pa = PTE2PA(*pte);
    return pa;
}
```

为了获得如题所示打印结果，在 trap.c 中增加一条打印语句

```
else if(r_scause() == 13 || r_scause() == 15){
    //13:Page load fault, 15:Page store fault
    //get the virtual address that causes the page fault
    uint64 fault_vaddr = (uint64)r_stval();
    printf("usertrap(): fault address %p\n", fault_vaddr);
}
```

lazystests 运行结果:

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ lazytests
usertrap(): fault address 0x0000000000004008
usertrap(): fault address 0x00000000000013f48
lazystests starting
running test lazy alloc
usertrap(): fault address 0x0000000000004000
usertrap(): fault address 0x00000000000044000
usertrap(): fault address 0x00000000000084000
usertrap(): fault address 0x000000000000c4000
usertrap(): fault address 0x00000000000104000
usertrap(): fault address 0x00000000000144000
usertrap(): fault address 0x00000000000184000
usertrap(): fault address 0x000000000001c4000
usertrap(): fault address 0x00000000000204000
usertrap(): fault address 0x00000000000244000
usertrap(): fault address 0x00000000000284000
usertrap(): fault address 0x000000000002c4000
usertrap(): fault address 0x00000000000304000
usertrap(): fault address 0x00000000000344000
usertrap(): fault address 0x00000000000384000
usertrap(): fault address 0x000000000003c4000
usertrap(): fault address 0x00000000000404000
usertrap(): fault address 0x00000000000444000
usertrap(): fault address 0x00000000000484000
usertrap(): fault address 0x000000000004c4000
usertrap(): fault address 0x00000000000504000
usertrap(): fault address 0x00000000000544000
usertrap(): fault address 0x00000000000584000
usertrap(): fault address 0x000000000005c4000
usertrap(): fault address 0x00000000000604000
usertrap(): fault address 0x00000000000644000
usertrap(): fault address 0x00000000000684000
usertrap(): fault address 0x000000000006c4000
usertrap(): fault address 0x00000000000704000
```

```
usertrap(): fault address 0x000000003d004000
usertrap(): fault address 0x000000003e004000
usertrap(): fault address 0x000000003f004000
test lazy unmap: OK
running test out of memory
usertrap(): fault address 0x0000000000003008
usertrap(): fault address 0x0000000001003018
usertrap(): fault address 0x0000000002003028
usertrap(): fault address 0x0000000003003038
usertrap(): fault address 0x0000000004003048
usertrap(): fault address 0x0000000005003058
usertrap(): fault address 0x0000000006003068
usertrap(): fault address 0x0000000007003078
usertrap(): fault address 0x0000000008003088
usertrap(): fault address 0x0000000009003098
usertrap(): fault address 0x000000000a0030a8
usertrap(): fault address 0x000000000b0030b8
usertrap(): fault address 0x000000000c0030c8
```

```
usertrap(): fault address 0x000000006a0036a8
usertrap(): fault address 0x000000006b0036b8
usertrap(): fault address 0x000000006c0036c8
usertrap(): fault address 0x000000006d0036d8
usertrap(): fault address 0x000000006e0036e8
usertrap(): fault address 0x000000006f0036f8
usertrap(): fault address 0x0000000070003708
usertrap(): fault address 0x0000000071003718
usertrap(): fault address 0x0000000072003728
usertrap(): fault address 0x0000000073003738
usertrap(): fault address 0x0000000074003748
usertrap(): fault address 0x0000000075003758
usertrap(): fault address 0x0000000076003768
usertrap(): fault address 0x0000000077003778
usertrap(): fault address 0x0000000078003788
usertrap(): fault address 0x0000000079003798
usertrap(): fault address 0x000000007a0037a8
usertrap(): fault address 0x000000007b0037b8
usertrap(): fault address 0x000000007c0037c8
usertrap(): fault address 0x000000007d0037d8
usertrap(): fault address 0x000000007e0037e8
usertrap(): fault address 0x000000007f0037f8
usertrap(): fault address 0xffffffff80003808
test out of memory: OK
ALL TESTS PASSED
```


usertests 运行结果:

```
$ usertests
usertests starting
test manywrites: OK
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test rwsbrk: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
```

```
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: OK
test sbrkfail: OK
test sbrkarg: OK
test validatetest: OK
test stacktest: OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```