操作系统第二次大作业

张灿晖 18364114 智科 2 班

1 虚拟内存管理程序

1.1 vm.c 代码分析

宏定义和 TLB 结构体定义。

```
#define TLB_SIZE 16 // TLB大小
#define FRAME_SIZE 256 // 帧大小
#define ADDR_SIZE 1000 // 文件addresses.txt中的虚拟地址条目数

typedef struct
{
    int frameNum;
    int pageNum;
    int access;
} TLB; // TLB结构定义
```

一些全局变量定义，各变量的功能如注释所示。

```
char phyMem[FRAME_SIZE][FRAME_SIZE]; // 物理内存
int page[FRAME_SIZE];                // 页表
int exist[FRAME_SIZE];               // 用于说明某页是否存在数据
int virAddr[ADDR_SIZE];                 // 存储读入的虚拟地址，由于
        addresses.txt中有1000个虚拟地址，故设置数组大小为1000

int tlbHit = 0; // tlb命中次数
int miss = 0;   // page失效次数
int count = 0;  // 记录读入的虚拟地址数目

FILE *addrFilePtr;      // 虚拟地址文件
FILE *backStoreFilePtr; // 存储数据的文件
FILE *outFilePtr; // 储存输出的文件

TLB tlb[TLB_SIZE]; // 实例化TLB
int ind = 0; // 全局变量ind，用于模拟FIFO中的队列功能
```

下面是 main 函数，用于完成页面置换的相关功能。下图代码块先调用函数 getopt 获取传入的页面置换策略参数，也即-f 和-l，分别表示指定虚拟内存管理器后续的部分中是使用 FIFO 置换算法还是 LRU 置换算法。随后将命令行参数 argv[2]和 argv[3]，即两个存储内存数据的文件，读入两个字符串变量中并调用函数 fopen 将它们打开，并额外打开一个输出文件 out.txt 做好后续写入结果的准备。

```
// 实现虚拟内存管理器的功能模块
int main(int argc, char *argv[])
{
    int mode = 0; // 页面置换策略的标志位，f表示FIFO策略，l表示LRU策略
    mode = getopt(argc, argv, "fl"); // 获取传入的页面置换策略参数

    char *backStoreFilename = argv[2]; // 储存文件BACKING_STORE.bin文件
            名
    char *addrFilename = argv[3]; // 地址文件correct.txt文件名

    // 打开两个输入文件，一个输出文件
    addrFilePtr = fopen(addrFilename, "r");
    backStoreFilePtr = fopen(backStoreFilename, "r");
    outFilePtr = fopen("out.txt", "w");
```

初始化一系列的变量、数组等。

```
int pageNum = 0;
int frameNum = 0;
int offset = 0;
char res;

for (int ii = 0; ii < FRAME_SIZE; ii++)
    exist[ii] = 0;
for (int ii = 0; ii < TLB_SIZE; ii++)
{
    tlb[ii].frameNum = -1;
    tlb[ii].pageNum = -1;
    tlb[ii].access = 10000; // access设为一个较大的值，
}
```

随后，下图代码块中展示了一个 for 循环，ADDR_SIZE 表示待读取的行数，根据 addresses.txt 中的数据，在本实验中有 1000 行需要读取，读取的方式是在 for 循环中利用 fscanf 读取虚拟地址字符串并将其转化为整数。随后，我们利用位运算，取出虚拟地址的高 8 位，即页号，并将其写入 pageNum 变量，同时将虚拟地址与帧的规模取余，以计算需要处理的偏移量 offset。随后，我们判定下标 pageNum 中对应的 exist 数组元素是否非空，如果非空，则说明该页存在于 page 中，不需要从文件 BACKING_STORE.bin 中调入，因此，我们下一步需要访问 TLB 来判断该页是否位于 TLB 中。访问 TLB 需要利用 for 循环遍历其中的每个元素，如果哪个元素中的 pageNum 数据域与当前待查找的 pageNum 相等，就说明 TLB 命中，可以从“物理内存”中提取结果存储到变量 res 中，再调用函数 printf 打印访问结果。最后，我们让 tlbHit 变量自增 1，表示又击中了一次 TLB；将 TLB 数组中被击中元素的 access 域更新为 0，表明该页最近被访问过；将标志变量 inTlb 设为 1，表示本轮中 TLB 被击中。反之，如果 TLB 未被击中，我们就在遍历 TLB 的过程中让其他元素的 access 数据域自增 1，以表示未被使用。

```
for (int count = 0; count < ADDR_SIZE; count++)
{
    fscanf(addrFilePtr, "%d", &virAddr[count]); // 从文件中读入虚拟
            地址
    pageNum = (virAddr[count] >> 8) & 0xff; // 取得虚拟地址的高8位
            写入pageNum
    offset = virAddr[count] % FRAME_SIZE;
    int inTlb = 0;
    if (exist[pageNum]) // 页存在于page中
    {
        for (int ii = 0; ii < TLB_SIZE; ii++) // 访问TLB
        {
            if (tlb[ii].pageNum == pageNum) // TLB命中
            {
                res = phyMem[tlb[ii].frameNum][offset];
                printf("Virtual address: %d Physical address: %d
                        Value: %d\n", virAddr[count], FRAME_SIZE
                        * tlb[ii].frameNum + offset, res);
                tlbHit++;
                tlb[ii].access = 0; // 更新为0，说明该页最近被访问过
                inTlb = 1;
            }
            else
                tlb[ii].access++;
        }
```

紧接上文，我们要对标志变量 inTlb 进行判定，如果该值为 0，即 TLB 未击中，我们就要去页表中查找页。如下图所示，我们从“物理内存”中提取出对应值，再输出查找结果。

```
if (!inTlb) // 在page中，但是不在tlb中，利用LRU放入tlb中
{
    res = phyMem[page[pageNum]][offset];
    printf("Virtual address: %d Physical address: %d
            Value: %d\n", virAddr[count], FRAME_SIZE *
            page[pageNum] + offset, res);
```

接下来，当我们在页表中找到页以后，要利用置换算法将其换入 TLB 中，下面展示了一个 switch 块，以判定置换模式。下图所示的 case ‘f’ 表示用 FIFO 置换，其核心思路为利用一个全局变量 ind 模拟循环队列，再通过 ind 对 TLB 进行 FIFO 访问。

```
// 根据传入的命令行参数选择页面置换算法
switch (mode)
{
case 'f': // FIFO
{
    ind = (ind++) % TLB_SIZE; // 利用全局变量ind实现
            FIFO策略
    // 换入页面，即直接向TLB中写入数据
    tlb[ind].frameNum = page[pageNum];
    tlb[ind].pageNum = pageNum;
    // 注意，FIFO策略不用更新access值
    break;
}
```

接下来，case ‘1’表示利用 LRU 算法进行置换，其核心思路在于通过遍历 TLB，找到最近最久未被使用的页面下标，选出换出页，再将换入页写入 TLB 的该位置，并设置数据域 access 为 0，表示刚刚对该页进行了访问。

```
case '1': // LRU
{
    int max = 0;
    int maxInd = 0;
    // 找到最近最久未被使用的页面下标
    for (int jj = 0; jj < TLB_SIZE; jj++)
    {
        if (tlb[jj].access > max)
        {
            maxInd = jj;
            max = tlb[jj].access;
        }
    }
    // LRU策略写入数据到TLB中
    tlb[maxInd].frameNum = page[pageNum];
    tlb[maxInd].pageNum = pageNum;
    tlb[maxInd].access = 0;
    break;
}
default:
    printf("Mode Error\n");
    return 0;
}
```

随后，我们要处理待查找页不在页表中的情况，也就是发生缺页错误时的情况，先将 miss 变量自增，说明又发生了一次缺页错误，再调用函数 fseek 和 fread 来查找对应的页并将其读取到“物理内存”和页表中，并将 exist 中的对应位置置为 1，随后求算结果 res 并打印，最后将帧数自增 1。

```
else // 不在page中，发生缺页错误
{
    miss++;
    // 寻找对应的页并且进行读取
    fseek(backStoreFilePtr, pageNum * FRAME_SIZE * sizeof
            (char), 0);
    fread(phyMem[frameNum], sizeof(char), FRAME_SIZE,
            backStoreFilePtr);
    page[pageNum] = frameNum;
    exist[pageNum] = 1;

    res = phyMem[frameNum][offset];
    printf("Virtual address: %d Physical address: %d Value:
            %d\n", virAddr[count], FRAME_SIZE * frameNum +
            offset, res);
    frameNum++;
```

随后即再利用一次 switch 语句确定换入换出算法，再采用对应的算法将页从后备存储中换入页表，其大体思路是与将页面从 TLB 中换出，从页表中换入一致的，故此处不再赘述算法的设计，有兴趣的读者可自行查阅源代码。

```
// 逻辑同上一个switch，不再赘述
switch (mode)
{
case 'f': // FIFO
case 'l': // LRU
```

程序的最后，我们计算出缺页错误的发生率和 TLB 命中率并打印有关信息，最后关闭已打开的文件。

```
double tlbRate = tlbHit / (double)ADDR_SIZE; // TLB命中率
double pageRate = miss / (double)ADDR_SIZE;  // 缺页率
printf("Page miss rate %f , TLB hit rate %f\n", pageRate, tlbRate)
        ; // 输出统计信息

// 关闭文件
fclose(addrFilePtr);
fclose(backStoreFilePtr);
fclose(outFilePtr);
```

1.2 地址转换测试 1.1(1)

test.sh 测试脚本。先对 vm.c 文件进行编译得到可执行文件 vm，再传入三个命令行参数并将标准输出和标准错误重定向到文件 out.txt 中，最后利用命令 diff 将输出文件 out.txt 与标准文件 correct.txt 比较。

```
#!/bin/bash

echo "Compiling..."
gcc vm.c -o vm -std=c99 # 以C99标准编译文件

echo "Running vm: mode LRU"
./vm -l BACKING_STORE.bin addresses.txt 1>out.txt 2>>out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt # 将输出与标准文件correct.txt比较
```

运行结果如下图，可以看出，编译 vm.c 文件和运行可执行文件 vm 成功，说明程序 vm.c 无结构性问题，随后与 correct.txt 文件比较的结果表明 out.txt 仅多了第 1001 行的输出最终比较结果行，这说明我们程序的输出，即 out.txt，其结果符合标准文件，即地址转换成功。

```
(base) [root@izwz9b4fdjken8sstzwx90z Q1]# ./test.sh
Compiling...
Running vm: mode LRU
Comparing with correct.txt
1001d1000
< Page miss rate 0.244000 , TLB hit rate 0.055000
```

1.3 基于不同策略运行虚拟内存管理系统 1.1(2)

测试脚本 test_FIFO_LRU.sh，先对 mv.c 文件进行编译，得到可执行文件 vm.out，再传入命令行参数分别按照 FIFO 策略和 LRU 策略运行，并将输出与 tail 串接，显示最后一行输出，即统计的缺页率和 TLB 命中率。

```
#!/bin/bash

gcc -std=c99 vm.c -o vm.out

# 使用FIFO策略
echo "FIFO Strategy Result:"
./vm.out -f BACKING_STORE.bin addresses.txt | tail -n 1


echo "LRU Strategy Result:"
# 使用LRU策略
./vm.out -l BACKING_STORE.bin addresses.txt | tail -n 1
```

运行结果如下所示，FIFO 策略的 TLB 击中率明显低于 LRU 策略的 TLB 击中率。

```
(base) [root@izwz9b4fdjken8sstzwx90z Q1]# ./test_FIFO_LRU.sh
FIFO Strategy Result:
Page miss rate 0.244000 , TLB hit rate 0.002000
LRU Strategy Result:
Page miss rate 0.244000 , TLB hit rate 0.055000
```

1.4 trace 生成器生成内存文件运行 1.2

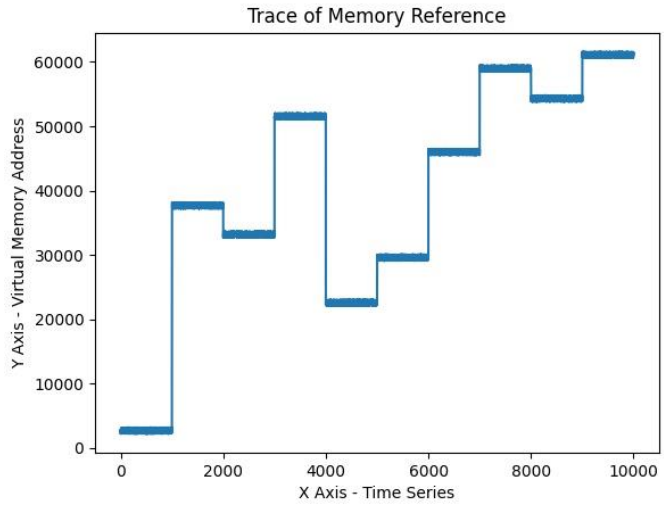
采用 python 语言编写 trace 生成器，利用两种策略生成地址文件，分别为模拟内存访问的局部性生成地址和利用随机函数随机生成地址，模拟内存访问的局部性生成地址算法如下图。其思路为先确定一个内存锚点，再取其左右 500 的范围为局部访问地址的生成空间，随机生成访问地址，这样就保证了访问地址的局部性。生成的地址文件保存为 addresses-locality.txt

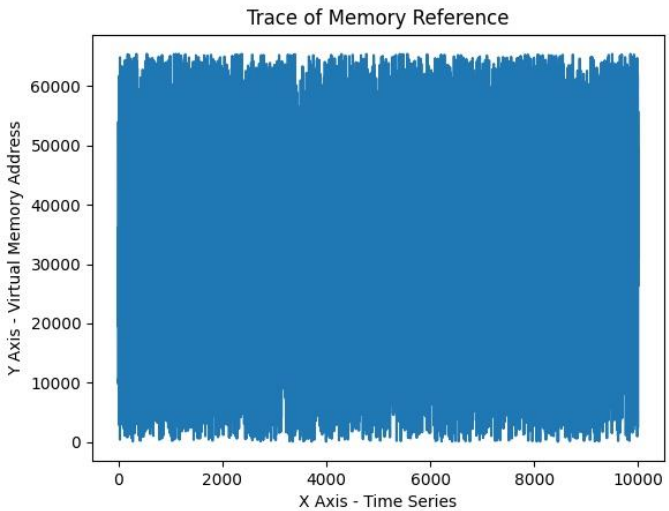
```
# 模拟内存访问的局部性生成访问地址
addr_list = []
print(min(addr_ref), max(addr_ref))
with open(''.join([FILE_ROOT, 'addresses-locality.txt']), 'w') as fp:
    for ii in range(int(YIELD_NUM/MEM_SIZE)):
        anchor = random.randint(0, max(addr_ref))
        for jj in range(MEM_SIZE):
            tmp_addr = random.randint(anchor-500, anchor+500) # 在锚点的正负500区间生成地址
            addr_list.append(tmp_addr)
            fp.write(''.join([str(tmp_addr), '\n']))
plot_trace(addr_list, 'locality_plot.jpg'); # 绘制内存访问分布图
```

随后，我们还提供了随机生成内存地址的算法，以作为对照，算法如下图。即简单地利用随机数生成函数 randint 进行生成。生成的地址文件保存为 addresses-random.txt

```
# 随机生成访问地址
addr_list = []
with open(''.join([FILE_ROOT, 'addresses-random.txt']), 'w') as fp:
    for ii in range(YIELD_NUM):
        addr_tmp = random.randint(min(addr_ref), max(addr_ref))
        addr_list.append(addr_tmp)
        fp.write(''.join([str(addr_tmp),'\n']))
plot_trace(addr_list, 'random_plot.jpg'); # 绘制内存访问分布图
```

接下来展示利用 python 包 matplotlib 绘制的内存访问分布图像，第一幅图像为局部访存算法生成的地址分布，第二幅图像则表示随机访存算法生成的地址分布。可以看出，局部访存算法生成的内存访问地址更符合计算机访存的规律，而随机访存算法生成的地址分布则完全无规律。





下面，我们编写脚本 test_lcl_rand.sh，来对不同策略和不同内存记录分布的组合进行测试，脚本内容如下图。其中，文件 addresses-locality.txt 和 addresses-random.txt 分别表示局部内存分布和随机内存分布。运输该脚本，先编译文件 vm.c 生成可执行文件 vm.o，再分别运行用 FIFO 算法测试局部内存分布、用 FIFO 算法测试随机内存分布、用 LRU 算法测试局部内存分布、用 LRU 算法测试随机内存分布四种策略的测试，并将输出重定向到四个输出文件，随后在屏幕上利用 tail 命令输出比较结果。

```
#!/bin/bash

echo "Compling..."
gcc -xc -std=c99 vm.c -o vm.o

echo "Mode: FIFO Memory: Locality"
./vm.o -f BACKING_STORE.bin addresses-locality.txt 1>FIFO_out_addresses-locality.txt
tail -n 1 FIFO_out_addresses-locality.txt

echo "Mode: FIFO Memory: Random"
./vm.o -f BACKING_STORE.bin addresses-random.txt 1>FIFO_out_addresses-random.txt
tail -n 1 FIFO_out_addresses-random.txt

echo "Mode: LRU Memory: Locality"
./vm.o -l BACKING_STORE.bin addresses-locality.txt 1>LRU_out_addresses-locality.txt
tail -n 1 LRU_out_addresses-locality.txt

echo "Mode: LRU Memory: Random"
./vm.o -l BACKING_STORE.bin addresses-random.txt 1>LRU_out_addresses-random.txt
tail -n 1 LRU_out_addresses-random.txt
```

运行结果如下图，其中，LRU 算法加上局部内存分布的组合获得了最低的缺页率和最高的 TLB 击中率。这说明了 LRU 算法的优越性，以及计算机内存的局部性确实可以带来访存性能的提升。

```
(base) [root@izwz9b4fdjken8sstzwx90z Q1]# ./test_lcl_rand.sh
Compling...
Mode: FIFO Memory: Locality
Page miss rate 0.005000 , TLB hit rate 0.250000
Mode: FIFO Memory: Random
Page miss rate 0.254000 , TLB hit rate 0.005000
Mode: LRU Memory: Locality
Page miss rate 0.005000 , TLB hit rate 0.995000
Mode: LRU Memory: Random
Page miss rate 0.254000 , TLB hit rate 0.056000
```

2 xv6-lab-2020 页表实验

2.1 代码实现

为了实现题目要求，我们按题目提示定义函数 vmprint。它应该接收一个 pagetable_t 参数，并且按照规定格式打印页表。函数 vmprint 声明如下，它接收一个页表变量 pagetable，随后打印页表地址，并且调用实用函数 print_recursively 递归打印页表内容。

```c
void vmprint(pagetable_t pagetable)
{
  // 打印页表地址
  printf("page table %p\n", pagetable);
  // 递归打印页表内容
  print_recursively(pagetable, 0);
}
```

函数定义如下，接收传入的需要打印的页表 pagetable，和递归层数 level，level 用于控制函数在不同的递归层数输出不同的内容

```c
static void print_recursively(pagetable_t pagetable, int level)
{
  //! 递归打印页表，该函数为函数vmprint的效用函数，通过递归的方式打印页表相
     关内容，加上static表示该函数仅在本文件中有效
  //? 参数：
  //*   pagetable_t pagetable: 传入的需要打印的页表
  //*   int level: 打印的递归层数
  //? 返回值：
  //*   无返回值
```

接下来，函数开始打印页表内容。在 for 循环的开始处先利用位运算构造一个表征该页相关权限的字符串，利用下标变量 ind 将表征的权限字符变量附加到字符数组 state 中，并在其尾部加上空字符’\0’。

```c
  for(int i = 0; i < 512; i++)
  {
    pte_t pte = pagetable[i]; // 获取页表pagetable中的项

    char state[5]; // 用于输出该页权限的字符串
    int ind = 0; // 控制字符串state的写入
    // 表征权限的四个位：PTE_R, PTE_W, PTE_X, PTE_U
    // 将pte与四个位分别做与运算，取出对应位置的置位值
    if(pte & PTE_R)
      state[ind++] = 'R';
    if(pte & PTE_W)
      state[ind++] = 'W';
    if(pte & PTE_X)
      state[ind++] = 'X';
    if(pte & PTE_U)
      state[ind++] = 'U';
    state[ind] = '\0';
```

接下来，先利用位运算 pte & PTE_V 判定页面 valid 字段的情况，如果该位值为 1，则说明该页的状态是合法的，可以进行访问。随后，函数调用移位函数 PTE2PA，取出子页面的地址，再进入 if-else if-else 控制流根据传入的 level 值，打印不同递归层数的输出信息，主要信息包括变量 i 的值、pte 地址、页面状态（RWXU 情况）、地址等。

```
if(pte & PTE_V) // 该页合法，可访问
{
  uint64 child = PTE2PA(pte); // 利用移位函数PTE2PA取出子页面的地址
  // 根据递归层数打印信息
  if(level == 0)
  {
    // %p -> %d
    printf("..%d: pte %p (%s) pa %p\n", i, pte, state, child);
    print_recursively((pagetable_t)child, level + 1);
  }
  else if(level == 1)
  {
    printf(".. ..%d: pte %p (%s) pa %p\n", i, pte, state, child);
    print_recursively((pagetable_t)child, level + 1);
  }
  else // 最后一层递归，打印后返回
  {
    printf(".. .. ..%d: pte %p (%s) pa %p\n", i, pte, state, child);
  }
}
}
```

运行结果如下所示：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6f000
..0: pte 0x0000000021fdac01 () pa 0x0000000087f6b000
.. ..0: pte 0x0000000021fda801 () pa 0x0000000087f6a000
.. .. ..0: pte 0x0000000021fdb01f (RWXU) pa 0x0000000087f6c000
.. .. ..1: pte 0x0000000021fda40f (RWX) pa 0x0000000087f69000
.. .. ..2: pte 0x0000000021fda01f (RWXU) pa 0x0000000087f68000
..255: pte 0x0000000021fdb801 () pa 0x0000000087f6e000
.. ..511: pte 0x0000000021fdb401 () pa 0x0000000087f6d000
.. .. ..510: pte 0x0000000021fddc07 (RW) pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b (RX) pa 0x0000000080007000
init: starting sh
$
```

2.2 问题分析

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。↵

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。↵

问题 3：这是什么页，有何功能？为什么没有 U 标志位？↵

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。↵

问题 5：这是什么页，为何没有 X 标志位？↵

问题 6：这是什么页，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。↵

问题 1：

因为该页中保存的是页目录，即 page dirtionary，RWXU 为用户所具有的权限，而用户对页目录并不具有 RWXU 中的任意一种操作权限，故为空。32618 位于物理内存的内核末尾和 PHYSTOP 之间。由于用户栈地址是从高向低分配，故不从低地址开始，再者，阅读如下图所示的 walk 函数定义，该函数从 level=2 开始遍历，当 alloc 非零时就分配 pte，故等级高的 pte 对应高地址。

```
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

问题二：

该页是用于保存进程参数的页。装载了用于存储进程的二进制文件。

```
// Load program into memory.
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;
}
iunlockput(ip);
end_op();
ip = 0;
```

问题三：

该页用于设立用户栈标志，是防护页，其功能为防止用户栈溢出。没有 U 标志位的原因在于 PTE_U 被设为 0，由于用户使用的栈空间是线性增长的，PTE_U 被设为 0 可以使用户在使用栈空间将要达到临界值时，即申请该防护页时，让用户无法访问该页，从而触发错误，以实现防护的作用。

问题四：

这是用户栈，装载了一些地址参数，源代码初始化该页的位置如下图：

```
// 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;
```

```
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp -= strlen(argv[argc]) + 1;
  sp -= sp % 16; // riscv sp must be 16-byte aligned
  if(sp < stackbase)
    goto bad;
  if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]
  ) + 1) < 0)
    goto bad;
  ustack[argc] = sp;
}
ustack[argc] = 0;
```

```
// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
  goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof
(uint64)) < 0)
  goto bad;
```

问题五：

这是陷入框架（trapframe）的装载页。没有标志位 X 说明该页不可执行，其原因在于 trapframe 中存储的是寄存器的值，trapframe 其用途是保存它们，其本身不具备执行功能，自然没有标志位 X。

```
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's
  kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program
  counter
  /*  32 */ uint64 kernel_hartid; // saved kernel tp
  /*  40 */ uint64 ra;
  /*  48 */ uint64 sp;
  /*  56 */ uint64 gp;
  /*  64 */ uint64 tp;
  /*  72 */ uint64 t0;
  /*  80 */ uint64 t1;
  /*  88 */ uint64 t2;
  /*  96 */ uint64 s0;
  /* 104 */ uint64 s1;
  /* 112 */ uint64 a0;
  /* 120 */ uint64 a1;
  /* 128 */ uint64 a2;
  /* 136 */ uint64 a3;
  /* 144 */ uint64 a4;
```

问题六：

这是蹦床（trampoline）页，下面的两张图表明了其装载过程。没有标志位 W 是因为其内容不可被用户写入。装载的内容为汇编代码 trampoline.S 所生成的二进制文件，该文件的作用实现在进程发生中断时保护现场之一功能，不能被用户随意更改，故标志位 W 为 0。

```
if((pagetable = proc_pagetable(p)) == 0)
  goto bad;
```

```
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  // An empty page table.
  pagetable = uvmcreate();
  if(pagetable == 0)
    return 0;

  // 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;
  }

  // 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;
  }

  return pagetable;
}
```

3 xv6-lab-2020 内存分配实验

3.1 Lazy Allocation 子任务

在 kernel/trap.c 文件中的函数 usertrap 修改其中的代码，对函数 usertrap 中的 else 情况作更加细化的判断。在 else 块中先调用函数 r_scause，将结果存储在变量 scause 中，如果 scause 等于 13 或 15，根据题目描述，这代表发生了页错误。随后，我们需要找出是哪个虚拟地址引发了页错误，因此我们调用函数 r_stval，以获取目标页的虚拟地址。接下来，我们就需要对 va 的情况进行分类判断了。

先判断 va 值是否高于由函数 sbrk 分配的地址，即判定 va >= p->sz 是否成立。如果成立，则打印错误信息，并且设置 p 的 killed 域为 1，这是为了在之后的 done 块将该进程终止。

```
} else if((which_dev = devintr()) != 0){
  // ok
} else {
  uint64 scause = r_scause();
  // 发生页错误
  if(scause == 13 || scause == 15)
  {
    uint64 va = r_stval(); // 获取页的虚拟地址

    // 如果在虚拟内存中页错误的地址高于函数sbrk()所分配的地址，则终止进程
    if(va >= p->sz)
    {
      printf("usertrap(): virtual address is invalid\n");
      p->killed = 1;
      goto done;
    }
```

随后，先判断进程是否在用户堆栈下的无效页面上出现页面错误，即判定语句 va <= PGROUNDDOWN(p->trapframe->sp)，这一句的含义是先从进程 p 的寄存器状态中取出其栈顶指针 sp 的值，再调用宏 PGROUNDDOWN 对其取整，如果虚拟地址 va 小于该值，则说明其位于用户堆栈下的无效页面，进程需要被终止。因此，我们打印出错信息，并设置进程 p 的 killed 域值，并快进到 done 块。

接下来，我们先调用宏 PGROUNDDOWN 对 va 这一虚拟地址向下取整，再调用函数 kalloc 分配内存，如果指针 mem 为空，则说明内存分配失败，进程需要被终结，否则调用函数 memset 对内存进行置位。

```
// 如果进程在用户堆栈下的无效页面上出现页面错误，则终止该进程
if(va <= PGROUNDDOWN(p->trapframe->sp))
{
  printf("usertrap(): guard page\n");
  p->killed = 1;
  goto done;
}

va = PGROUNDDOWN(va); // 对出错的页面虚拟地址向下取整
char* mem = kalloc(); // 分配内存
// 内存溢出，终止进程
if(mem == 0)
{
  printf("usertrap(): memory out\n");
  p->killed = 1;
  goto done;
}
memset(mem, 0, PGSIZE);
```

接下来，我们在 if 条件语句中调用函数 mappages，并传入包含页表 p->pagetable、虚拟地址 va、页规模 PGSIZE、先前分配的内存 mem，四个状态位的值作或运算 PTE_W|PTE_X|PTE_R|PTE_U 这一系列参数，尝试将虚拟地址映射到物理地址，并对该函数的返回值进行判断，如果返回值非 0，说明映射失败，即这一虚拟内存无法映射到有效的物理地址，故我们在 if 块中打印错误信息，释放先前分配的内存 mem，设置进程 p 的 killed 位并快进到 done 块

```
  // 页面分配失败
  if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, PTE_W|PTE_X|
      PTE_R|PTE_U) != 0)
  {
    printf("usertrap(): mappages failed\n");
    kfree(mem); // 释放先前分配的内存
    p->killed = 1;
    goto done;
  }
}
```

else 块是针对其他情况进行处理的块，即处理当 scause 不为 13 或 15 的情况。在这一部分，我们打印有关信息后设置进程 p 的 killed 位让进程终止即可。

```
  else // 其他未预料的情况
  {
    // 打印有关信息
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(),
        p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }
}
```

done 部分要做的事是根据进程 p 的 killed 域的值确定进程 p 是否需要被终止，如果 p->killed 值非 0，则调用函数 exit 终止。随后，由于变量 which_dev 的值来自于函数 devintr 的返回值，故我们要对变量 which_dev 的值进行判断，如果为 2，则说明为计时器中断，故根据规则函数要放弃 CPU。在最后，我们调用函数 usertrapret，使调用被返还给用户空间。

```
  done:
  if(p->killed) // 该进程需要被终止
    exit(-1);

  // 如果此为计时器中断，则放弃CPU
  if(which_dev == 2)
    yield();

  usertrapret(); // 返回用户空间
}
```

echo hi 可以成功调用。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo hi
hi
```

3.2 Lazytests and Usertests 子任务

传入 lazytests 并运行，结果如下图，lazytests 通过

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
```

```
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
usertrap: invalid virtual address
test lazy unmap: OK
running test out of memory
usertrap: mappages falied
test out of memory: OK
ALL TESTS PASSED
$
```

传入 usertests 并运行，结果如下图，usertests 通过。