Experiments of Operating System 2

操作系统实验报告 2

2020.12.6

姓名：庞雨贤

学号： 18364075

邮箱：1909642325@qq.com

Experiment 1：虚存管理模拟程序

1.实验内容及要求

1.1 Chapter 10. Programming Projects: Designing a Virtual Memory Manager（OSC 10th ed.）

(1) 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 Page Replacement，TLB 分别实现 FIFO 和 LRU 两种策略。

(2) 实现基于 LRU 的 Page Replacement；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。提示：通过 getopt 函数，程序运行时通过命令行指定参数。

1.2 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 10000 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（数据点太密的话可以采样后绘图），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。

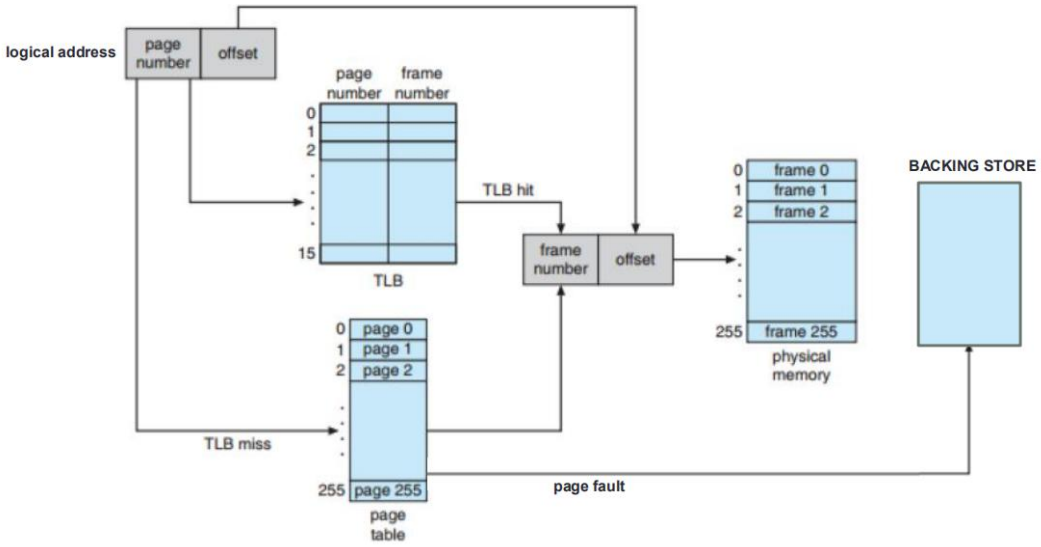
2.任务描述与解决步骤（参考教材）

该项目的目标为将大小为 2^{16}=65536 字节的虚拟地址空间将逻辑地址转换到物理地址。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。假设 TLB、页表和物理内存在开始时为空。其他要求包括：（1）页表有 2^8 个条目；（2）页面大小为 2^8 字节；（3）TLB 有 2^4 个条目（使用 LRU 页面置换策略）；（4）帧大小为 2^8 字节；（5）帧数为 2^8；（6）物理内存为 65536

字节（256 帧×256 字节/帧）。

程序将读取包含表示逻辑地址的整数数字的文件。这 16 位逻辑地址分为：由 8 位组成的页码、由 8 位组成的页面偏移量。

地址转换过程：首先，从逻辑地址提取页码，并且查阅 TLB。在 TLB 命中的情况下，从 TLB 中获取帧码。在 TLB 未命中的情况下，查阅页表。在后一种情况下，从页表获得帧码或发生缺页错误。地址转换过程如图所示：



处理缺页错误：后备存储由文件 BACKING_STORE.bin 表示，这是一个大小为 65536 字节的二进制文件。当发生缺页错误时，将从文件 BACKING_STORE 中读取一个 256 字节的页面，并将其存储在物理内存的可用页帧中。例如，如果页码为 15 的逻辑地址导致缺页错误，则程序将从 BACKING_STORE 中读取第 15 页，并且将其存储在物理内存的页帧中。一旦该帧被存储（并且页表和 TLB 也被更新），对 15 页的后续访问将由 TLB 或页表来解决。为了可以随机寻到文件的特定位置来读取，需要将 BACKING_STORE.bin 作为随机访问文件来处理。

addresses.txt 文件提供包含 0~65535 的整数值的逻辑地址，correct.txt 包含 addresses.txt 的正确输出值。最终程序应输出：（1）要翻译的逻辑地址（从 addresses.txt 中读取的整数值）；（2）相应的物理地址；（3）存储在转换的物理地址上的带符号字节值。

在实际中，物理内存通常比虚拟地址空间小得多，**所以之后我们修改页帧数为 128**。这种更改需要修改程序跟踪空闲页帧，采用 FIFO 或 LRU 来实现页面置换策略。

FIFO 页面置换算法为每个页面记录了调到内存的时间。当必须置换页面时，将选择最

旧的页面。创建一个 FIFO 队列来管理所有的内存页面。置换的是队列的首个页面。当需要调入页面到内存时，就将它加到队列的尾部。LRU 页面置换算法将每个页面与它的上次使用的时间关联起来。当需要置换页面时，LRU 选择最长时间没有使用的页面。LRU 最重要的是确定由上次使用时间定义的帧的顺序，通常有两种实现方法：计数器和堆栈。

最后，程序还应报告以下统计信息：（1）缺页错误率：导致缺页错误的地址引用的百分比；（2）TLB 命中率：在 TLB 中解析的地址引用的百分比。

3.实验环境

操作系统环境：WSL+Ubuntu20.04+vscode

编程语言：C&python

4.代码实现

4.1 Designing a Virtual Memory Manager

（1）vm.c

vm.c 分别用 FIFO 和 LRU 实现了 TLB 更新， 当使用 FIFO 时，须注释掉 LRU 部分代码,代码包含所需的头文件以及对自定义的函数进行声明：

```c
/*
vm.c
注释:
    本文件解决1.1(1)的问题
    分别用FIFO 和LRU 实现了TLB 更新
    当使用FIFO 时，须注释掉LRU 部分代码
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include<time.h>
#define TLB_SIZE 16

int bin2int(int*,int, int);//将二进制码start-end 位转换为int
int search_TLB(int, int*, int*);
void update_TLB(int*, int* , int, int);
int load_memory(char*, char [256][256], int);
```

bin2int(binary, start, end)函数将二进制码 binary 的第 start 到第 end 位转换成整数：

```
int bin2int(int binary[],int start,int end){
    int i=0;
    int ans = 0;
    for ( i = start; i < end; i++)
    {
        ans = ans + pow(2,i-start) * binary[i];
    }
    return ans;
}
```

使用 FIFO 方法实现 TLB：

```
int search_TLB(int page_number, int* TLB_page_number, int* TLB_frame_number)
{
    // 因为采用 int 数组 page_table[page] 表示页表，可以线性查找
    for(int i=0;i<TLB_SIZE;i++)
        if(TLB_page_number[i]==page_number)
            return TLB_frame_number[i];// 页表存在，返回 frame，TLB hit
    return -1;//不存在， TLB miss
}
void update_TLB(int* TLB_page_number, int* TLB_frame_number, int page_number, int f
rame_number)
{
    /*使用 FIFO 更新 TLB
    */
    static int TLB_FI = 0;
    // 搜索 TLB 以找到空闲 space
    for(int i=0;i<TLB_SIZE;i++)
    {
        if (TLB_page_number[i] == -1)
        {
            // TLB 还没有满，直接向空闲 sapce 添加元素
            TLB_page_number[i] = page_number;
            TLB_frame_number[i] = frame_number;
            return;
        }
    }
    // TLB 满时，元素应按 FIFO 方式更新
    TLB_page_number[TLB_FI] = page_number;
    TLB_frame_number[TLB_FI] = frame_number;
    TLB_FI = (TLB_FI + 1)%TLB_SIZE;
    return;
}
```

使用 LRU 方法实现 TLB：

```
time_t TLB_LRU_time[TLB_SIZE];
int search_TLB(int page_number, int* TLB_page_number, int* TLB_frame_number)
{
    // 线性搜索 TLB 以匹配页码
    for(int i=0;i<TLB_SIZE;i++)
        if(TLB_page_number[i]==page_number)
        {// 匹配成功, TLB hit
            time(TLB_LRU_time + i);// 更新最近使用时间
            return TLB_frame_number[i];
        }
    return -1;//匹配失败, TLB miss
}
void update_TLB(int* TLB_page_number, int* TLB_frame_number, int page_number, int f
rame_number)
{
    for(int i=0;i<TLB_SIZE;i++)
    {
        // 搜索 TLB 以找到空闲 space
        if (TLB_page_number[i] == -1)
        {
            time(TLB_LRU_time + i);//记录最近使用时间
            TLB_page_number[i] = page_number;
            TLB_frame_number[i] = frame_number;
            return;
        }
    }
    // TLB 满时，元素应按 LRU 方式更新
    time_t least_recently_used_time = TLB_LRU_time[0];
    int least_recently_used_index = 0;
    for(int i = 1;i < TLB_SIZE; i++)
    {
        if(TLB_LRU_time[i]<least_recently_used_time)
        {
            least_recently_used_time = TLB_LRU_time[i];
            least_recently_used_index = i;
        }
    }
    //新添加的元素取代了第一个元素
    TLB_page_number[least_recently_used_index] = page_number;
    TLB_frame_number[least_recently_used_index] = frame_number;
    time(TLB_LRU_time+least_recently_used_index);//记录新添加的元素最近使用的时间
    return;
}
```

因为内存的大小等于页表的大小，只需线性地添加所需的帧，不需要担心内存溢出，因为内存足够大，可以同时包含所有的页:

```
int load_memory(char* back_store_name, char physical_memory[256][256], int page_number)
{
    static int first_empty_frame = 0;
    //打开bin 文件
    FILE *bin_file;
    if((bin_file = fopen(back_store_name, "rb"))== NULL){
        printf("file read fail: %s", back_store_name);
        return -1;
    }
    fseek(bin_file,256 * page_number,0);//读指针设置为所需的页面
    fread(physical_memory[first_empty_frame],1,256,bin_file);// 将页面加载到空闲的物理内存帧中
    fclose(bin_file);
    return first_empty_frame++;//线性保留下一个自由帧位置
}
```

接下来是 int main(int argc, char *argv[])部分，首先读取 addresses.txt：

```
    char* back_store_name = argv[1];
    char* input_address_name = argv[2];

    //读取addresses.txt 文件
    FILE *fp;
    if((fp = fopen(input_address_name,"r"))== NULL){
        printf("File read failed: %s", input_address_name);
        return -1;
    }
    int addresses[1500];
    int i = 0;
    while(fscanf(fp, "%d", &addresses[i]) != EOF) {
        i++;
    }
    fclose(fp);
```

获取 page 和 offSet：

```
        int temp = addresses[i];
        int count = 0;
        int binary[100];
        for(int l = 0; l < 100; l++){
            binary[l] = 0;
        }
        while(temp != 0){
```

```
            binary[count] = temp & 1;
            temp = (temp>>1);
            count++;
        }

        int page_number = 0;
        int off_set = 0;
        page_number = bin2int(binary,8,16);
        for (int m = 0; m < 8; m++)
        {
            off_set = off_set + pow(2,m) * binary[m];
        }
```

获取 page 和 offSet 后，我们要得到 frame 和 offSet。在该题中 offSet 不变，我们要获取 page 和 frame 的关系。在第一问中，假设虚拟地址和物理地址相同，不需要进行页面置换。我们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。我采用 int 数组 page_table[page] = frame 表示页表。

```
    int page_table[256];//通过 int 数组模拟页表映射，存储效率低，操作方便，速度快
    char memory[256][256];//通过二维数组模拟物理内存，memory[i][j]表示第 i 帧中的第 j 个字节
    int firstFrame = 0;
    for(int j = 0; j < 256 ; j++){
        page_table[j]  = -1;//设定这个-1 为我们未存取的情况
    }
   //TLB_page_number[i] -->> TLB_frame_number[i]
    int TLB_page_number[TLB_SIZE] = {
        -1,-1,-1,-1,-1, -1,-1,-1,-1,-1, -1,-1,-1,-1,-1,-1};//初始化为-1 意味着所有位置都是空闲的
    int TLB_frame_number[TLB_SIZE];
```

在对一个 page 进行处理时，先搜索页表：如果页表匹配成功，我们直接得到了 frame；如果匹配失败，我们就从上到下，把还没有分配的 frame 分配给 page。

```
        int frame_number;
        if ((frame_number = search_TLB(page_number, TLB_page_number, TLB_frame_number)) >= 0)
        {
            //TLB hit
            int physical_address = (frame_number << 8) | off_set;
            printf("Virtual address: %u Physical address: %u Value: %d\n",
            addresses[i], physical_address, (int)memory[frame_number][off_set]);
        }
        else
        {
```

```
            //TLB miss
            if ((frame_number = page_table[page_number]) == -1)
            {
                // 缺页错误，应该从 bin 文件加载页面到内存帧

                //将文件加载到内存
                frame_number = load_memory(back_store_name, memory, page_number);

                //更新页表
                page_table[page_number] = frame_number;
            }
            //更新 TLB
            update_TLB(TLB_page_number, TLB_frame_number, page_number, frame_number
);

            unsigned int physical_address = (frame_number << 8) | off_set;
            printf("Virtual address: %u Physical address: %u Value: %d\n",
             addresses[i], physical_address, (int)memory[frame_number][off_set]);
        }
```

（2）第二问修改页帧数为 128，不需要得到 value，而是关注于输入 page（虚拟地址的页表）和输出 frame（物理地址的帧表）转化的过程。并且要求输出缺页错误率（导致缺页错误的地址引用的百分比）和 TLB 命中率（在 TLB 中解析的地址引用的百分比），并实现分别基于 LRU 和 FIFO 的页置换。

a. vm_FIFO.c

代码包含所需的头文件以及对自定义的函数进行声明：

```
/*
vm_LRU.c
注释:
    本文件解决 1.1(2)的问题
    TLB 使用 FIFO 更新
    Page Replacement 使用 FIFO

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
```

```
int bin2int(int*,int, int);//将二进制码 start-end 位转换为 int
void TLB_update_FIFO(int [][2], int, int, int);// 用 FIFO 更新 TLB 数组
int page_replacement_FIFO(const char*, int, char [128][256], int [128][2]); //该函数用 FIFO 策略更新 page_table 数组和物理内存中存放的内容
```

bin2int 函数在第一问中有分析过，这里不给出详细代码。TLB_update_FIFO 函数用于更新 TLB，如果 TLB 数组未满则按顺序储存，如果满了则使用 FIFO 替换策略进行替换：

```
void TLB_update_FIFO(int TLB[][2], int len, int page_number, int frame_number){
    static int record = 0; //静态变量 record 用于记录当前 TLB 数组应该更新的行，TLB 初始为空，因此 record 初始为 0

    //更新 TLB 中索引为 record 的行
    TLB[record][0] = page_number;
    TLB[record][1] = frame_number;
    record++;
    //如果 TLB 的最后一行已被更新，则下一次调用该函数应该更新第一行，record 为 0
    if(record >= len)
        record = record % len;

    return;
}
```

page_replacement_FIFO 函数用于更新 page_table 数组和物理内存中存放的内容，如果物理内存、page_table 数组未满则按顺序储存，如果满了则使用 FIFO 替换策略进行替换：

```
int page_replacement_FIFO(const char* back_store_name, int page_number, char physical_memory[128][256], int page_table[128][2]){

    static int frame_number = -1; //静态变量 frame_number 用于记录物理内存当前应该更新的 frame 对应的 frame number
    //index 指示应该将文件指针定位到何处，这里是 page number 对应的内容的首地址
    int index = 256 * page_number;
    frame_number++;
    //如果 frame_number 为 128，则将其置为 0
    if(frame_number >= 128)
        frame_number = frame_number % 128;
    //定义文件指针，初始化为空
    FILE *fp = NULL;
    //打开 BACKING_STORE.bin 文件
    if((fp = fopen(back_store_name, "rb")) == NULL){
        printf("Cannot open file %s!\n", back_store_name);
        return -1;
    }
    //将文件指针定位到 page number 对应的内容的首地址
    fseek(fp, index, 0);
```

```
    //将 page number 对应的内容写入物理地址
    fread(physical_memory[frame_number],1,256,fp);
    //关闭 BACKING_STORE.bin 文件
    fclose(fp);
    fp = NULL;
    //更改 page_table 中相应位置的映射关系
    page_table[frame_number][0] = page_number;
    page_table[frame_number][1] = frame_number;
    //返回 page number 对应的 frame number
    return frame_number;
}
```

main 函数部分：

```
int main(int argc, char *argv[]){

    char* back_store_name = argv[1];
    char* input_address_name = argv[2];

    //TLB[i][0]为 page number，TLB[i][1]为该 page number 对应的 frame number
    int TLB[16][2];
    //page_table[i][0]为 page number，page_table[i][1]为该 page number 对应的
frame number
    int page_table[128][2];
    //将物理内存定义为 128*256 的二维数组，即有 128frames，每个 fram 有 256 个字节
    char physical_memory[128][256];
    //初始化 TLB，-1 表示为空
    for(int i=0; i<16; ++i)
        *TLB[i] = -1;
    //初始化 page table，-1 表示为空
    for(int i=0; i<128; ++i)
        *page_table[i] = -1;

    //TLB_hit 变量用于记录 TLB hit 的次数
    int TLB_hit = 0;
    //Page_fault 变量用于记录 Page-fault 的次数
    int Page_fault = 0;
    //total 变量用于记录虚拟地址的个数
    int total = 0;

    //读取 addresses.txt 文件
    FILE *fp;
    if((fp = fopen(input_address_name,"r"))== NULL){
        printf("File read failed: %s", input_address_name);
        return -1;
    }
```

```
    int addresses[1500];
    int i = 0;
    while(fscanf(fp, "%d", &addresses[i]) != EOF) {
        i++;
    }
    fclose(fp);

    //循环遍历addresses.txt文件中记录的每个虚拟地址，将其转换成物理地址，并获取该物理地址中储存的值
    for(int i = 0; i < 1000; i++){
        //获取page和offset
        int temp = addresses[i];
        int count = 0;
        int binary[100];
        for(int l = 0; l < 100; l++){
            binary[l] = 0;
        }
        while(temp != 0){
            binary[count] = temp & 1;
            temp = (temp>>1);
            count++;
        }
        int page_number = 0;
        int off_set = 0;
        page_number = bin2int(binary,8,16);
        for (int m = 0; m < 8; m++)
        {
            off_set = off_set + pow(2,m) * binary[m];
        }

        //定义变量frame_number，用于储存当前page number对应的frame number
        int frame_number;
        //设置标志变量flag1，如果flag1=0代表TLB miss，如果flag1=1代表TLB hit
        int flag1 = 0;
        //设置标志变量flag2，如果flag2=0代表page fault，如果flag2=1代表page success
        int flag2 = 0;
        total++;
        //查询TLB
        for(int i=0; i<16; i++){
            if(TLB[i][0] == -1)
                break;
            //如果TLB hit
            if(TLB[i][0] == page_number){
                //获取当前page number对应的frame number
```

```
                frame_number = TLB[i][1];
                flag1 = 1;
                TLB_hit++;
                break;
            }
        }
        //如果TLB miss，查询page table
        if(flag1 == 0){
            for(int i=0; i<128; ++i){
                if(page_table[i][0] == -1)
                    break;
                //如果page success
                if(page_table[i][0] == page_number){
                    //获取当前page number对应的frame number
                    frame_number = page_table[i][1];
                    flag2 = 1;
                    break;
                }
            }
            //如果标志变量flag2=1，说明page success
            if(flag2 == 1){
                TLB_update_FIFO(TLB, 16, page_number, frame_number);
            }
            //如果标志变量flag2=0，说明page fault
            else{
                //Page-fault的次数加一
                Page_fault++;
                frame_number = page_replacement_FIFO(back_store_name, page_number,
physical_memory, page_table);
                //frame_number为-1表示打开文件失败，此时返回-1
                if(frame_number == -1)
                    return -1;
                TLB_update_FIFO(TLB, 16, page_number, frame_number);
            }
        }
        //根据frame number和offset来计算当前虚拟地址对应的物理地址
        int physical_address = (frame_number << 8) + off_set;
        //打印当前的虚拟地址、当前的虚拟地址对应的物理地址、该物理地址中储存的值
        printf("Virtual address: %d Physical address: %d Value: %d\n", addresses[i]
, physical_address, physical_memory[frame_number][off_set]);

    }
    float TLB_hit_rate = 100 * (float)TLB_hit / total;
    float Page_fault_rate = 100 * (float)Page_fault / total;
```

```
    printf("TLB hit rate is %.4f%%, Page-fault rate is %.4f%%.\n", TLB_hit_rate, Page_fault_rate);
    return 0;
}
```

b. vm_LRU.c

代码包含所需的头文件以及对自定义的函数进行声明：

```
/*
vm_LRU.c
注释:
    本文件解决1.1(2)的问题
    TLB 使用LRU 更新
    Page Replacement 使用LRU

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

int bin2int(int*,int, int);//将二进制码start-end 位转换为int
void TLB_update_LRU(int [][2], int, int, int, clock_t*);// 用LRU 更新TLB 数组
int page_replacement_LRU(const char*, int, char [128][256], int [128][2], clock_t*);//该函数用LRU 策略更新page_table 数组和物理内存中存放的内容
```

TLB_update_LRU 函数用于更新 TLB 数组，如果 TLB 数组未满则按顺序储存，如果满了则使用 LRU 替换策略进行替换：

```
void TLB_update_LRU(int TLB[][2], int len, int page_number, int frame_number, clock_t* record_time_TLB){
    static int record = 0; //静态变量record 用于指示TLB 是否已满
    //如果TLB 已经存放满了
    if(record >= len){
        //min_time 用于记录TLB 所有行中最近一次被访问的最小时间
        clock_t min_time = record_time_TLB[0];
        //index 用于记录该最小时间对应的行的索引
        int index = 0;
        for(int i=1; i<len; i++)
            if(record_time_TLB[i] < min_time){
                min_time = record_time_TLB[i];
                index = i;
            }
        //将最近一次被访问的时间最小的一行替换掉
```

```
        TLB[index][0] = page_number;
        TLB[index][1] = frame_number;
        //更新该行最近一次的访问时间
        record_time_TLB[index] = clock();
    }
    //如果 TLB 未存放满，则按顺序继续存放
    else{
        TLB[record][0] = page_number;
        TLB[record][1] = frame_number;
        //更新该行最近一次的访问时间
        record_time_TLB[record] = clock();
        //在 TLB 存放满之前，每次调用完该函数，record 加 1
        record++;
    }

    return;
}
```

page_replacement_LRU 函数用于更新 page_table 数组和物理内存中存放的内容，如果物理内存、page_table 数组未满则按顺序储存，如果满了则使用 LRU 替换策略进行替换：

```
int page_replacement_LRU(const char* BACKING_STORE, int page_number, char physical_
memory[128][256], int page_table[128][2], clock_t* record_time_page){
    static int index = 0; //静态变量 index 用于指示物理内存、page_table 数组是否已满
    int frame_number;//frame_number 用于记录物理内存当前应该更新的 frame 对应的
frame number
    //如果 index 为 128，说明物理内存、page_table 数组存放满了，此时使用 LRU 替换策略进行替换
    if(index >= 128){
        //min_time 用于记录 page_table 所有行中最近一次被访问的最小时间，也即物理内存所有
frame 最近一次被访问的最小时间
        clock_t min_time = record_time_page[0];
        //record 用于记录该最小时间对应的行的索引
        int record = 0;
        for(int i=1; i<128; i++){
            if(record_time_page[i] < min_time){
                min_time = record_time_page[i];
                record = i;
            }
        }
        //获取该最小时间对应的 frame number，该 frame number 对应的物理内存及其映射关系需要
更新
        frame_number = record;
    }
    //如果 index 小于 128，说明物理内存、page_table 数组未满，此时按顺序存放
    //此时 index 记录的是需要更新的物理内存的位置，则 frame_number=index，该 frame number
```

```
对应的物理内存及其映射关系需要更新
    else{
        frame_number = index;
        index++;
    }

    ////定义文件指针，初始化为空
    FILE *fp = NULL;
    //打开BACKING_STORE.bin 文件
    //如果打开失败，则打印提示信息，返回-1
    if((fp = fopen(BACKING_STORE, "rb")) == NULL){
        printf("Cannot open file %s!\n", BACKING_STORE);
        return -1;
    }
    //将文件指针定位到page number 对应的内容的首地址
    fseek(fp, 256 * page_number, 0);
    //将page number 对应的内容写入物理内存
    fread(physical_memory[frame_number],1,256,fp);
    //关闭BACKING_STORE.bin 文件
    fclose(fp);
    fp = NULL;
    //更改page_table 中相应位置的映射关系
    page_table[frame_number][0] = page_number;
    page_table[frame_number][1] = frame_number;
    //更新该行最近一次的访问时间
    record_time_page[frame_number] = clock();
    //返回page number 对应的frame number

    return frame_number;
}
```

main 函数部分：

```
int main(int argc, char *argv[]){

    char* back_store_name = argv[1];
    char* input_address_name = argv[2];

    //TLB[i][0]为page number，TLB[i][1]为该page number 对应的frame number
    int TLB[16][2];
    //page_table[i][0]为page number，page_table[i][1]为该page number 对应的
frame number
    int page_table[128][2];
    //将物理内存定义为128*256 的二维数组，即有128frames，每个fram 有256 个字节
    char physical_memory[128][256];
    //初始化TLB，-1 表示为空
```

```
    for(int i=0; i<16; ++i)
        *TLB[i] = -1;
    //初始化 page table，-1 表示为空
    for(int i=0; i<128; ++i)
        *page_table[i] = -1;

    //定义 clock_t 类型的数组 record_time_TLB 与 record_time_page，其长度分别于与 TLB 数组、page_table 数组行数相同，
    //record_time_TLB 与 record_time_page 的每一个元素分别用于存放 TLB、page_table 中第一列存放的每个 page number 最近一次被访问的时间，
    //也即 record_time_TLB[i] 中存放的是 TLB[i][0] 最近一次被访问的时间，record_time_page[i] 中存放的是 page_table[i][0] 最近一次被访问的时间
    clock_t record_time_TLB[16];
    clock_t record_time_page[128];
    //TLB_hit 变量用于记录 TLB hit 的次数
    int TLB_hit = 0;
    //Page_fault 变量用于记录 Page-fault 的次数
    int Page_fault = 0;
    //total 变量用于记录虚拟地址的个数
   int total = 0;

    //读取 addresses.txt 文件
    FILE *fp;
    if((fp = fopen(input_address_name,"r"))== NULL){
        printf("File read failed: %s", input_address_name);
        return -1;
    }
    int addresses[1500];
    int i = 0;
    while(fscanf(fp, "%d", &addresses[i]) != EOF) {
        i++;
    }
   fclose(fp);

    //循环遍历 addresses.txt 文件中记录的每个虚拟地址，将其转换成物理地址，并获取该物理地址中储存的值
    for(int i = 0; i < 1000; i++){
        //获取 page 和 offset
        int temp = addresses[i];
        int count = 0;
        int binary[100];
        for(int l = 0; l < 100; l++){
            binary[l] = 0;
        }
```

```
        while(temp != 0){
            binary[count] = temp & 1;
            temp = (temp>>1);
            count++;
        }
        int page_number = 0;
        int off_set = 0;
        page_number = bin2int(binary,8,16);
        for (int m = 0; m < 8; m++)
        {
            off_set = off_set + pow(2,m) * binary[m];
        }

        //定义变量frame_number，用于储存当前page number对应的frame number
        int frame_number;
        //设置标志变量flag1，如果flag1=0代表TLB miss，如果flag1=1代表TLB hit
        int flag1 = 0;
        //设置标志变量flag2，如果flag2=0代表page fault，如果flag2=1代表page success
        int flag2 = 0;
        total++;
        //查询TLB
        for(int i=0; i<16; i++){
            if(TLB[i][0] == -1)
                break;
            //如果TLB hit
            if(TLB[i][0] == page_number){
                //获取当前page number对应的frame number
                frame_number = TLB[i][1];
                flag1 = 1;
                //更新该page number在TLB中最近一次访问的时间
                record_time_TLB[i] = clock();
                TLB_hit++;
                break;
            }
        }
        //如果TLB miss，查询page table
        if(flag1 == 0){
            for(int i=0; i<128; ++i){
                if(page_table[i][0] == -1)
                    break;
                //如果page success
                if(page_table[i][0] == page_number){
                    //获取当前page number对应的frame number
                    frame_number = page_table[i][1];
```

```
                    flag2 = 1;
                    //更新该page number在page table中最近一次访问的时间
                    record_time_page[i] = clock();
                    break;
                }
            }
            //如果标志变量flag2=1，说明page success
            if(flag2 == 1){
                TLB_update_LRU(TLB, 16, page_number, frame_number, record_time_TLB)
;

            }
            //如果标志变量flag2=0，说明page fault
            else{
                //Page-fault的次数加一
                Page_fault++;
                frame_number = page_replacement_LRU(back_store_name, page_number, p
hysical_memory, page_table, record_time_page);
                //frame_number为-1表示打开文件失败，此时返回-1
                if(frame_number == -1)
                    return -1;
                TLB_update_LRU(TLB, 16, page_number, frame_number, record_time_TLB)
;

            }
        }
        //根据frame number和offset来计算当前虚拟地址对应的物理地址
        int physical_address = (frame_number << 8) + off_set;
        //打印当前的虚拟地址、当前的虚拟地址对应的物理地址、该物理地址中储存的值
        printf("Virtual address: %d Physical address: %d Value: %d\n", addresses[i]
, physical_address, physical_memory[frame_number][off_set]);
    }
    float TLB_hit_rate = 100 * (float)TLB_hit / total;
    float Page_fault_rate = 100 * (float)Page_fault / total;
    printf("TLB hit rate is %.4f%%, Page-fault rate is %.4f%%.\n", TLB_hit_rate, Pa
ge_fault_rate);
    return 0;
}
```

4.2 编写 trace 生成器

本题采用 python 语言编写，具体代码见附件 trace.py，运行结果打印见 5.2。

5.实验结果及分析

5.1 Designing a Virtual Memory Manager

（1）运行 test.sh 结果如图（上图为 FIFO，下图为 LRU）：

```
pyx@DESKTOP-ETHO4RO:~/os-assignment2$ sh test.sh
Compiling
Running vm
Comparing with correct.txt
```

```
pyx@DESKTOP-ETHO4RO:~/os-assignment2$ sh test.sh
Compiling
Running vm
Comparing with correct.txt
```

（2）编译运行 vm_FIFO.c（TLB 和 page replacement 策略为 FIFO）和 vm_LRU.c（TLB 和 page replacement 策略为 LRU）

a.TLB 和 page replacement 策略为 FIFO

```
pyx@DESKTOP-ETHO4RO:~/os-assignment2$ g++ vm_FIFO.c -o vm_FIFO -fno-stack-protector
pyx@DESKTOP-ETHO4RO:~/os-assignment2$ ./vm_FIFO BACKING_STORE.bin addresses.txt
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
Virtual address: 38929 Physical address: 3857 Value: 0
Virtual address: 32865 Physical address: 4193 Value: 0
Virtual address: 64243 Physical address: 4595 Value: -68
Virtual address: 2315 Physical address: 4619 Value: 66
Virtual address: 64454 Physical address: 5062 Value: 62
Virtual address: 55041 Physical address: 5121 Value: 0
Virtual address: 18633 Physical address: 5577 Value: 0
Virtual address: 14557 Physical address: 5853 Value: 0
Virtual address: 61006 Physical address: 5966 Value: 59
Virtual address: 62615 Physical address: 407 Value: 37
Virtual address: 7591 Physical address: 6311 Value: 105
Virtual address: 64747 Physical address: 6635 Value: 58
Virtual address: 6727 Physical address: 6727 Value: -111
Virtual address: 32315 Physical address: 6971 Value: -114
Virtual address: 60645 Physical address: 7397 Value: 0
Virtual address: 6308 Physical address: 7588 Value: 0
Virtual address: 45688 Physical address: 7800 Value: 0
Virtual address: 969 Physical address: 8137 Value: 0
Virtual address: 40891 Physical address: 8379 Value: -18
```

使用 FIFO 替换策略得到的 TLB hit 以及 page fault 如下：

```
Virtual address: 7736 Physical address: 13112 Value: 0
Virtual address: 31260 Physical address: 5148 Value: 0
Virtual address: 17071 Physical address: 5551 Value: -85
Virtual address: 8940 Physical address: 5868 Value: 0
Virtual address: 9929 Physical address: 6089 Value: 0
Virtual address: 45563 Physical address: 6395 Value: 126
Virtual address: 12107 Physical address: 6475 Value: -46
TLB hit rate is 5.4000%, Page-fault rate is 53.8000%.
```

b.TLB 和 page replacement 策略为 LRU

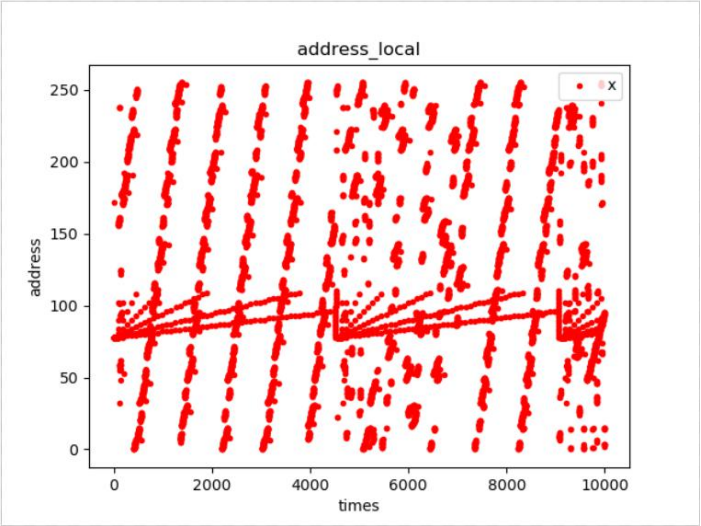
```
pyx@DESKTOP-ETHO4RO:~/os-assignment2$ g++ vm_LRU.c -o vm_LRU -fno-stack-protector
pyx@DESKTOP-ETHO4RO:~/os-assignment2$ ./vm_LRU BACKING_STORE.bin addresses.txt
Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical 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
Virtual address: 38929 Physical address: 3857 Value: 0
Virtual address: 32865 Physical address: 4193 Value: 0
Virtual address: 64243 Physical address: 4595 Value: -68
Virtual address: 2315 Physical address: 4619 Value: 66
Virtual address: 64454 Physical address: 5062 Value: 62
Virtual address: 55041 Physical address: 5121 Value: 0
Virtual address: 18633 Physical address: 5577 Value: 0
Virtual address: 14557 Physical address: 5853 Value: 0
Virtual address: 61006 Physical address: 5966 Value: 59
Virtual address: 62615 Physical address: 407 Value: 37
Virtual address: 7591 Physical address: 6311 Value: 105
Virtual address: 64747 Physical address: 6635 Value: 58
Virtual address: 6727 Physical address: 6727 Value: -111
Virtual address: 32315 Physical address: 6971 Value: -114
Virtual address: 60645 Physical address: 7397 Value: 0
Virtual address: 6308 Physical address: 7588 Value: 0
Virtual address: 45688 Physical address: 7800 Value: 0
Virtual address: 969 Physical address: 8137 Value: 0
```

使用 LRU 替换策略得到的 TLB hit 以及 page fault 如下：

```
Virtual address: 7736 Physical address: 31032 Value: 0
Virtual address: 31260 Physical address: 284 Value: 0
Virtual address: 17071 Physical address: 21423 Value: -85
Virtual address: 8940 Physical address: 9708 Value: 0
Virtual address: 9929 Physical address: 6857 Value: 0
Virtual address: 45563 Physical address: 251 Value: 126
Virtual address: 12107 Physical address: 24395 Value: -46
TLB hit rate is 5.5000%, Page-fault rate is 53.7000%.
```

5.2 编写 trace 生成器

运行 trace.py 绘制【访问地址-时间】图表如图：



生成 addresses-locality.txt 如图：

addresses-locality.txt

```
9981    24224
9982    24272
9983    24176
9984    24320
9985    24416
9986    24464
9987    24368
9988    24512
9989    720
9990    816
9991    3696
9992    3792
9993    3888
9994    864
9995    912
9996    3840
9997    3984
9998    960
9999    1008
10000   3936
10001
```

以该文件为参数运行 vm_FIFO.c，结果如图：

```
TLB hit rate is 82.2133%, Page-fault rate is 12.6533%.
```

以该文件为参数运行 vm_LRU.c，结果如图：

```
TLB hit rate is 82.2333%, Page-fault rate is 12.3600%.
```

Experiment 2：xv6-lab-2020 页表实验

1.实验内容及要求

完成 Print a page table 任务。要求按图 1 格式打印页表内容；其中括号内表示页表项权限，R 表示可读，W 表示可写，X 表示可执行，U 表示用户可访问。物理页后的数字（pa 32618）表示第几个物理页帧。要求在报告中提供实现所需的源代码和运行截屏，代码要求有充分注释。然后，回答接下来的 6 个问题（分别对应代码注释行中的标签）。

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 () pa 32618(th pages) //问题1
.. ..0: pte 0x0000000021fda401 () pa 32617(th pages)
.. .. ..0: pte 0x0000000021fdac1f (RWXU) pa 32619(th pages) //问题2
.. .. ..1: pte 0x0000000021fda00f (RWX) pa 32616(th pages) //问题3
.. .. ..2: pte 0x0000000021fd9c1f (RWXU) pa 32615(th pages) //问题4
..255: pte 0x0000000021fdb401 () pa 32621(th pages)
.. ..511: pte 0x0000000021fdb001 () pa 32620(th pages)
.. .. ..510: pte 0x0000000021fdd807 (RW) pa 32630(th pages) //问题5
.. .. ..511: pte 0x0000000020001c0b (RX) pa 7(th pages) //问题6
```

图 1. init 进程的页表内容

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。

2.任务描述

任务是编写一个打印页表内容的函数。定义一个名为 vmprint（）的函数。它应该带有 pagetable_t 参数，并以图 1 中的格式打印该 pagetable。在返回 argc 之前，在 exec.c 中插入 if（p-> pid == 1）vmprint（p-> pagetable）以打印第一个进程的页表。

如图一所示，第一行显示 vmprint 的参数。之后，每个 PTE 都有一行，包括引用树中更深的页表页面的 PTE。每条 PTE 线都以“ ..”缩进，以表示其在树中的深度。每条 PTE 行在其页表页中显示 PTE 索引，pte 位以及从 PTE 中提取的物理地址。不要打印无效的 PTE。在上面的示例中，顶层页表页面具有条目 0 和 255 的映射。条目 0 的下一层仅映射了索引 0，

而索引 0 的最低层具有条目 0、1 和 2 映射。

一些提示：

（1）可以将 vmprint()放在 kernel / vm.c 中；

（2）在文件 kernel / riscv.h 的末尾使用宏；

（3）在 kernel / defs.h 中定义 vmprint 的原型，以便从 exec.c 中调用它；

（4）在 printf 调用中使用%p 打印出完整的 64 位十六进制 PTE 和地址。

3.实验环境

操作系统环境：WSL+Ubuntu20.04

编程语言：C

4.代码实现

在 kernel/vm.c 中加入 vmprint 函数代码如下：

```
void vmprint(pagetable_t pagetable){
  //打印第一级页表的地址
  printf("page table %p\n", pagetable);
  //遍历第一级页表，第一级页表有512个pte
  for(int i = 0; i < 512; i++){
    //获取第一级页表的第i个pte
    pte_t pte0 = pagetable[i];
    //如果该pte有建立有效映射关系：第一级页表—>第二级页表
    if(pte0 & PTE_V){
      //打印该pte所在的行号以及该pte的内容
      printf("..%d: pte %p (", i, pte0);
      //如果可读标志位为1，打印“R”
      if(pte0 & PTE_R)
        printf("R");
      //如果可写标志位为1，打印“W”
      if(pte0 & PTE_W)
        printf("W");
      //如果可执行标志位为1，打印“X”
      if(pte0 & PTE_X)
        printf("X");
      //如果用户标志位为1，打印“U”
      if(pte0 & PTE_U)
        printf("U");
      printf(") ");
      //获取对应的第二级页表的地址
      pagetable_t physical_address0 = (pagetable_t)PTE2PA(pte0);
      //计算出该地址在物理内存的第几帧并打印
```

```
    uint64 pa0 =  ((uint64)physical_address0 >> 12) - (KERNBASE >> 12);
    printf("pa %d(th pages)\n", pa0);
    //遍历该第二级页表，该第二级页表有512 个pte
    for(int j = 0; j < 512; j++){
      //获取该第二级页表的第j 个pte
      pte_t pte1 = physical_address0[j];
      //如果该pte 有有效建立映射关系：第二级页表—>第三级页表
      if(pte1 & PTE_V){
        printf(".. ..%d: pte %p (", j, pte1);
        if(pte1 & PTE_R)
          printf("R");
        if(pte1 & PTE_W)
          printf("W");
        if(pte1 & PTE_X)
          printf("X");
        if(pte1 & PTE_U)
          printf("U");
        printf(") ");
        //获取对应的第三级页表的地址
        pagetable_t physical_address1 = (pagetable_t)PTE2PA(pte1);
        //计算出该地址在物理内存的第几帧并打印
        uint64 pa1 =  ((uint64)physical_address1 >> 12) - (KERNBASE >> 12);
        printf("pa %d(th pages)\n", pa1);
        //遍历该第三级页表，该第三级页表有512 个pte
        for(int k = 0; k < 512; k++){
          //获取该第三级页表的第k 个pte
          pte_t pte2 = physical_address1[k];
          //如果该pte 有建立有效映射关系：第三级页表—>物理地址
          if(pte2 & PTE_V){
            printf(".. .. ..%d: pte %p (", k, pte2);
            if(pte2 & PTE_R)
              printf("R");
            if(pte2 & PTE_W)
              printf("W");
            if(pte2 & PTE_X)
              printf("X");
            if(pte2 & PTE_U)
              printf("U");
            printf(") ");
            //获取对应的物理地址
            pagetable_t physical_address2 = (pagetable_t)PTE2PA(pte2);
            //计算出该地址在物理内存的第几帧并打印
            uint64 pa2 =  ((uint64)physical_address2 >> 12) - (KERNBASE >> 12);
            printf("pa %d(th pages)\n", pa2);
```

```
            }
          }
        }
      }
    }
  }
}
```

修改 kernel/defs.h 文件如下，在 174 行定义 vmprint 的原型，以便从 exec.c 中调用它：

```
156  //·vm.c
157  void·····kvminit(void);
158  void·····kvminithart(void);
159  uint64····kvmpa(uint64);
160  void·····kvmmap(uint64,·uint64,·uint64,·int);
161  int·····mappages(pagetable_t,·uint64,·uint64,·uint64,·int);
162  pagetable_t···uvmcreate(void);
163  void·····uvminit(pagetable_t,·uchar·*,·uint);
164  uint64····uvmalloc(pagetable_t,·uint64,·uint64);
165  uint64····uvmdealloc(pagetable_t,·uint64,·uint64);
166  int·····uvmcopy(pagetable_t,·pagetable_t,·uint64);
167  void·····uvmfree(pagetable_t,·uint64);
168  void·····uvmunmap(pagetable_t,·uint64,·uint64,·int);
169  void·····uvmclear(pagetable_t,·uint64);
170  uint64····walkaddr(pagetable_t,·uint64);
171  int·····copyout(pagetable_t,·uint64,·char·*,·uint64);
172  int·····copyin(pagetable_t,·char·*,·uint64,·uint64);
173  int·····copyinstr(pagetable_t,·char·*,·uint64,·uint64);
174  void·····vmprint(pagetable_t);
```

修改 exec.c 如下，在 119-120 行插入以下代码：

```
119  ··if(p->·pid·==·1)
120  ····vmprint(p->·pagetable);
121  ··return·argc;·//·this·ends·up·in·a0,·the·first·argument·to·main(argc,·argv)
```

5.实验结果打印

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 () pa 32618(th pages)
.. ..0: pte 0x0000000021fda401 () pa 32617(th pages)
.. .. ..0: pte 0x0000000021fdac1f (RWXU) pa 32619(th pages)
.. .. ..1: pte 0x0000000021fda00f (RWX) pa 32616(th pages)
.. .. ..2: pte 0x0000000021fd9c1f (RWXU) pa 32615(th pages)
..255: pte 0x0000000021fdb401 () pa 32621(th pages)
.. ..511: pte 0x0000000021fdb001 () pa 32620(th pages)
.. .. ..510: pte 0x0000000021fdd807 (RW) pa 32630(th pages)
.. .. ..511: pte 0x0000000020001c0b (RX) pa 7(th pages)
init: starting sh
$
```

6.问题回答与分析

6.1 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。

答：第一对括号中的内容表示页表项权限。因为该 pte 建立的是一级页表到二级页表的映射关系，而二级页表在进程执行期间是不可读、不可写、不可执行且用户模式不可访问的，所以第一对括号为空。通过阅读代码可知，建立页表映射的函数是 mappages，该函数用于建立虚拟页到物理页的映射，虚拟地址到物理地址的映射为三级页表映射，这三级页表映射的创建或获取通过调用 walk 函数来实现。

walk 代码如下：

```
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
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)];
}
```

mappages 代码如下：

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned. Returns 0 on success, -1 if walk() couldn't
// allocate a needed page-table page.
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if(*pte & PTE_V)
      panic("remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

阅读 walk 函数代码注释我们知道，虚拟地址到物理地址的映射为三级页表映射，三级页表中每个页表都有 512 个 pte，每个 pte 对应下一级别的一张页表，假设某个虚拟地址到物理地址的三级映射分别由 $pte2$、$pte1$ 以及 $pte0$ 记录，其中 $pte2$ 位于第一级页表中，记录了对应的第二级页表的首地址；$pte1$ 位于该第二级页表中，记录了对应的第三级页表的首地址；$pte0$ 位于该第三级页表中，记录了要映射到的物理地址。

walk 函数会从虚拟地址中获取 $pte2$ 在第一级页表中的位置，然后通过第一级页表获取 $pte2$，如果 $pte2$ 中的有效标志位为 1，则直接获取 $pte2$ 中记录的第二级页表的首地址，当第二级页表或第三级页表不存在时 walk 函数会先创建不存在的页表再获取 $pte0$ 的地址并返回，当第二级页表与第三级页表已经存在 walk 函数会直接获取 $pte0$ 的地址并返回。

mappages 函数在循环中每次将第一级页表的首地址、当前需要建立映射关系的虚拟地址以及 1 传给 walk 函数，walk 函数返回 $pte0$ 的地址。mappages 函数获取到 $pte0$ 的地址后，将当前虚拟地址要映射到的物理地址记录在 $pte0$ 中，然后根据输入参数 perm 设置其页表项权限，并将其有效标志位置为 1。通过代码以及上述分析可以知道，walk 函数只将 $pte2$、$pte1$ 的有效标志位置为 1，而没有设置它们的页表项权限位，而 mappages 函数则在 walk 函数的基础上，将 $pte0$ 的有效标志位置为 1 并且根据输入参数 perm 设置其页表项权限位。问题 1 处第一对括号中的内容表示页表项权限，对应的 pte 是 $pte2$，$pte2$ 只设置了有效标志位而没有设置页表项权限位，因此该括号的内容为空。

32618 在物理内存的高地址位置，不从低地址开始的原因通过阅读 kalloc.c 中 kinit 函数和 freerange 函数可知：空闲内存是通过链表连接在一起的，kinit 函数调用 freerange 函数初始化的时候从低地址的空闲内存开始，通过头插法将空闲内存一页一页的插入链表中。由于初始化的时候内存是按地址由低到高的顺序插入链表的，因此初始化之后低地址内存在链表尾部，高地址内存在链表头部。通过观察 kalloc 函数可知，kalloc 函数分配内存的时候是从链表头处开始分配的，也即每次都先将高地址内存分配出去。

kinit 和 freerange 函数代码：

```
void
kinit()
{
  initlock(&kmem.lock, "kmem");
  freerange(end, (void*)PHYSTOP);
}

void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
    kfree(p);
}
```

6.2 这是什么页？装载的什么内容？结合源代码内容进行解释。

答：这是用户空间页，装载用户空间的代码，该页是由 exec.c 文件中的 exec 函数第 52 行通过调用 uvmalloc 函数进行分配的：

```
41    //Load program into memory.
42    for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
43      if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
44        goto bad;
45      if(ph.type != ELF_PROG_LOAD)
46        continue;
47      if(ph.memsz < ph.filesz)
48        goto bad;
49      if(ph.vaddr + ph.memsz < ph.vaddr)
50        goto bad;
51      uint64 sz1;
52      if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
53        goto bad;
```

6.3 这是什么页，有何功能？为什么没有 U 标志位？

答：这是用户空间页，功能是避免越界访问和允许 exec 处理参数太多的情况，当用户进程越界访问内存的时候，会访问到这一页，但是由于该页没有 U 标志位，用户进程没有权限，访问时会引发 page fault，从而终止该错误操作。exec.c 文件中的 exec 函数的第 71 行调用 uvmalloc 请求了两个用户页，但是在 74 行调用 uvmclear 函数将该虚拟地址的 PTE 的 U 标志位设为了 0：

```
67    //·Allocate·two·pages·at·the·next·page·boundary.
68    //·Use·the·second·as·the·user·stack.
69    sz·=·PGROUNDUP(sz);
70    uint64·sz1;
71    if((sz1·=·uvmalloc(pagetable,·sz,·sz·+·2*PGSIZE))·==·0)
72      goto·bad;
73    sz·=·sz1;
74    uvmclear(pagetable,·sz-2*PGSIZE);
75    sp·=·sz;
76    stackbase·=·sp·-·PGSIZE;
```

uvmclear 函数将该页的 U 标志位取反，从而就实现了将该页的 U 标志位清零。

```
338   //·mark·a·PTE·invalid·for·user·access.
339   //·used·by·exec·for·the·user·stack·guard·page.
340   void
341   uvmclear(pagetable_t·pagetable,·uint64·va)
342   {
343     pte_t·*pte;
344     
345     pte·=·walk(pagetable,·va,·0);
346     if(pte·==·0)
347       panic("uvmclear");
348     *pte·&=·~PTE_U;
349   }
```

6.4 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。

答：这是用户空间页，装载用户程序栈，如图，源代码初始化该页的位置在 exec.c 文件中的第 71 行：

```
67    //·Allocate·two·pages·at·the·next·page·boundary.
68    //·Use·the·second·as·the·user·stack.
69    sz·=·PGROUNDUP(sz);
70    uint64·sz1;
71    if((sz1·=·uvmalloc(pagetable,·sz,·sz·+·2*PGSIZE))·==·0)
72      goto·bad;
73    sz·=·sz1;
74    uvmclear(pagetable,·sz-2*PGSIZE);
75    sp·=·sz;
76    stackbase·=·sp·-·PGSIZE;
```

通过调用 uvmalloc 进行初始化，uvmalloc 函数位于 vm.c 文件中的第 228 行到第 252 行：

```
226  //·Allocate·PTEs·and·physical·memory·to·grow·process·from·oldsz·to
227  //·newsz,·which·need·not·be·page·aligned.··Returns·new·size·or·0·on·error.
228  uint64
229  uvmalloc(pagetable_t·pagetable,·uint64·oldsz,·uint64·newsz)
230  {
231  ··char·*mem;
232  ··uint64·a;
233  
234  ··if(newsz·<·oldsz)
235  ····return·oldsz;
236  
237  ··oldsz·=·PGROUNDUP(oldsz);
238  ··for(a·=·oldsz;·a·<·newsz;·a·+=·PGSIZE){
239  ····mem·=·kalloc();
240  ····if(mem·==·0){
241  ····uvmdealloc(pagetable,·a,·oldsz);
242  ····return·0;
243  ····}
244  ····memset(mem,·0,·PGSIZE);
245  ····if(mappages(pagetable,·a,·PGSIZE,·(uint64)mem,·PTE_W|PTE_X|PTE_R|PTE_U)·!=·0){
246  ····kfree(mem);
247  ····uvmdealloc(pagetable,·a,·oldsz);
248  ····return·0;
249  ····}
250  ··}
251  ··return·newsz;
252  }
```

6.5 这是什么页，为何没有 X 标志位？

答：这是内核空间页，装载进程的内核栈，没有 X 标志位是因为该页装载的内容是指向当前进程的内核堆栈、当前 CPU 的 hartid、usertrap 的地址和内核页表的地址的指针等，而不是可执行的指令，因此没有 X 标志位。

```
24  //·initialize·the·proc·table·at·boot·time.
25  void
26  procinit(void)
27  {
28  ··struct·proc·*p;
29  ··
30  ··initlock(&pid_lock,·"nextpid");
31  ··for(p·=·proc;·p·<·&proc[NPROC];·p++)·{
32  ····initlock(&p->lock,·"proc");
33  
34  ····//·Allocate·a·page·for·the·process's·kernel·stack.
35  ····//·Map·it·high·in·memory,·followed·by·an·invalid
36  ····//·guard·page.
37  ····char·*pa·=·kalloc();
38  ····if(pa·==·0)
39  ····panic("kalloc");
40  ····uint64·va·=·KSTACK((int)·(p·-·proc));
41  ····kvmmap(va,·(uint64)pa,·PGSIZE,·PTE_R·|·PTE_W);
42  ····p->kstack·=·va;
43  ··}
44  ··kvminithart();
45  }
```

6.6 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。

答：这是内核空间页，没有 W 标志是因为该页内容不允许更改，装载的是 trampoline。kernel 从这里读取指令来执行系统调用的返回。物理页号处于低地址区域是因为虚拟地址映射中，trampoline 映射到物理空间的低地址位置，在 proc.c 文件中的 proc_pagetable 函数将虚拟地址的最高地址映射到 tranpoline 的物理地址，也因此该 pte 是在三级页表树状结构的最后一页：

```
155  //·Create·a·user·page·table·for·a·given·process,
156  //·with·no·user·memory,·but·with·trampoline·pages.
157  pagetable_t
158  proc_pagetable(struct·proc·*p)
159  {
160  ··pagetable_t·pagetable;
161
162  ··//·An·empty·page·table.
163  ··pagetable·=·uvmcreate();
164  ··if(pagetable·==·0)
165  ····return·0;
166
167  ··//·map·the·trampoline·code·(for·system·call·return)
168  ··//·at·the·highest·user·virtual·address.
169  ··//·only·the·supervisor·uses·it,·on·the·way
170  ··//·to/from·user·space,·so·not·PTE_U.
171  ··if(mappages(pagetable,·TRAMPOLINE,·PGSIZE,
172  ··········(uint64)trampoline,·PTE_R·|·PTE_X)·<·0){
173  ····uvmfree(pagetable,·0);
174  ····return·0;
175  ··}
176
177  ··//·map·the·trapframe·just·below·TRAMPOLINE,·for·trampoline.S.
178  ··if(mappages(pagetable,·TRAPFRAME,·PGSIZE,
179  ··········(uint64)(p->trapframe),·PTE_R·|·PTE_W)·<·0){
180  ····uvmunmap(pagetable,·TRAMPOLINE,·1,·0);
181  ····uvmfree(pagetable,·0);
182  ····return·0;
183  ··}
```

Experiment 3：xv6-lab-2020 内存分配实验

1.实验内容及要求

1.1 完成 Lazy allocation 子任务，要求 echo hi 正常运行，报告中可以描述自己的尝试过程，以及一些中间变量。

1.2 完成 Lazytests and Usertests 子任务。对于 Lazytests，要求屏幕输出如下图所示；对于 usertests 任务，要求通过所有除 sbrkarg 之外的测试。给出运行截屏。在阅读报告中提供代码修改片段，说明针对哪些文件，哪些函数进行了修改，新代码加上充分注释；可以写一些体会。

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
usertrap(): fault address 0x0000000000004000 beyond heap range
usertrap(): fault address 0x0000000001004000 beyond heap range
usertrap(): fault address 0x0000000002004000 beyond heap range
usertrap(): fault address 0x0000000003004000 beyond heap range
...
test lazy unmap: OK
running test out of memory
usertrap(): mappages failed, va=0x0000003ef5041000
test out of memory: OK
ALL TESTS PASSED
```

图 2. Lazytests 运行输出示例

2.实验环境

操作系统环境：WSL+Ubuntu20.04

编程语言：C

3.实验过程

3.1 Eliminate allocation from sbrk()

第一个任务是从 sbrk(n)系统调用实现中删除页面分配，该实现是 sysproc.c 中的 sys_sbr()函数。sbrk(n)系统调用将进程的内存大小增加 n 个字节然后返回新分配区域开始位置（即原来的内存大小），新的 sbrk(n)只需要将进程的大小(myproc()->sz)增加 n，然后返回原来的大小。它不应该分配内存，因此应该删除对 growproc()的调用，但是仍然需要增加进程的大小。修改后的代码如图：

```
41  uint64
42  sys_sbrk(void)
43  {
44    int addr;
45    int n;
46
47    if(argint(0, &n) < 0)
48      return -1;
49    addr = myproc()->sz;
50    //if(growproc(n) < 0)
51    //  return -1;
52    myproc()->sz += n;
53    if(n < 0)
54      uvmdealloc(myproc()->pagetable, addr, myproc()->sz);
55
56    return addr;
57  }
```

运行 echo hi 输出如图：

```
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000012ac stval=0x0000000000004008
panic: uvmunmap: not mapped
```

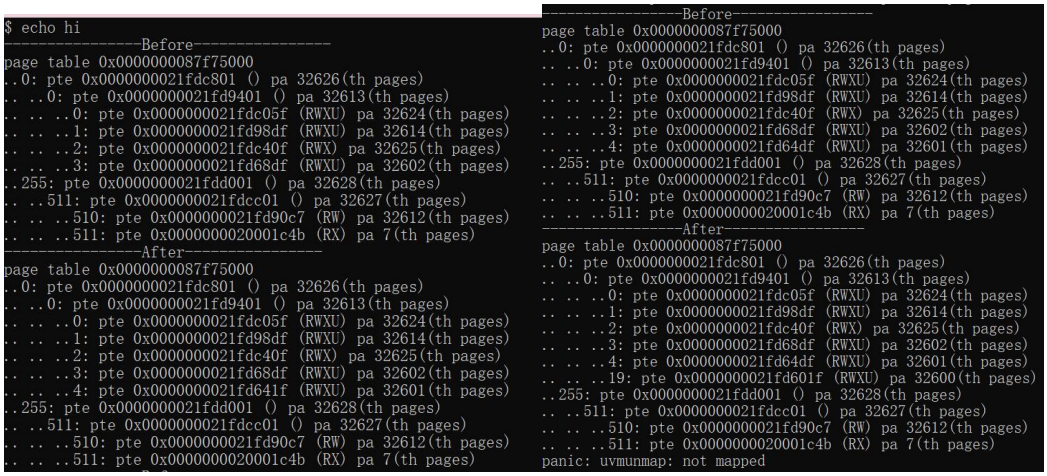
“ usertrap（）：...”消息来自 trap.c 中的用 usertrap 函数，它捕获了一个不知道如何处理的异常。“ stval = 0x0..04008”指示导致页面错误的虚拟地址为 0x4008。

3.2 Lazy allocation

为了让 echo hi 正常运行，我们应该修改 usertrap 函数，在其打印报错信息之前为发生错误的虚拟地址分配新的物理内存页并建立它们之间的映射。修改代码如图：

```
72  //////////////////////////////////////////////////////
73  else if (r_scause() == 15 || r_scause() == 13)
74  {
75    uint64 va = r_stval();
76    if (va < p->sz && va > PGROUNDDOWN(p->trapframe->sp))
77    {
78      uint64 ka = (uint64) kalloc();
79      if (ka == 0) p->killed = -1;
80      else
81      {
82        memset((void*)ka, 0, PGSIZE);
83        va = PGROUNDDOWN(va);
84        if (mappages(p->pagetable, va, PGSIZE, ka, PTE_U | PTE_W| PTE_R) != 0)
85        {
86          kfree((void*)ka);
87          p->killed = -1;
88        }
89      }
90    }
91  else p->killed = -1;
92  }
93  //////////////////////////////////////////////////////
```

在 else 之前再添加一个 else if 语句，如果 r_scause()返回的值为 13 或 15，说明发生的错误为 page fault，此时先调用 vmprint 函数打印此时页表的内容，然后调用 r_stval 函数获取发生错误的虚拟地址，然后正确处理缺页异常，完成必要的检查（访问非法的线性地址和栈溢出），再调用 kalloc 函数为其分配新的物理内存页并使用 memset 函数进行初始化，再调用 mappages 函数在发生错误的虚拟地址与新分配的物理内存页之间建立映射。如果映射建立失败，则调用 kfree 函数回收分配的物理内存，如果映射建立成功则调用 vmprint 函数打印此时页表的内容。修改之后得到的运行结果如图：



可知已经成功为发生错误的两个虚拟地址分配了物理内存并建立了映射，但是调用 uvmunmap 函数时发生了 panic。找到 uvmunmap 函数中打印报错信息的地方，将其注释掉：

```
173  void
174  uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
175  {
176    uint64 a;
177    pte_t *pte;
178
179    if((va % PGSIZE) != 0)
180      panic("uvmunmap: not aligned");
181
182    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
183      if((pte = walk(pagetable, a, 0)) == 0)
184        panic("uvmunmap: walk");
185      //if((*pte & PTE_V) == 0)
186      //  panic("uvmunmap: not mapped");
187      if(PTE_FLAGS(*pte) == PTE_V)
188        panic("uvmunmap: not a leaf");
189      if(do_free){
190        uint64 pa = PTE2PA(*pte);
191        kfree((void*)pa);
192      }
193      *pte = 0;
194    }
195  }
```

再次运行 echo hi 结果如图：

```
----------------After----------------
page table 0x0000000087f75000
..0: pte 0x0000000021fdc801 () pa 32626(th pages)
.. ..0: pte 0x0000000021fd9401 () pa 32613(th pages)
.. .. ..0: pte 0x0000000021fdc05f (RWXU) pa 32624(th pages)
.. .. ..1: pte 0x0000000021fd98df (RWXU) pa 32614(th pages)
.. .. ..2: pte 0x0000000021fdc40f (RWX) pa 32625(th pages)
.. .. ..3: pte 0x0000000021fd68df (RWXU) pa 32602(th pages)
.. .. ..4: pte 0x0000000021fd64df (RWXU) pa 32601(th pages)
.. .. ..19: pte 0x0000000021fd601f (RWXU) pa 32600(th pages)
..255: pte 0x0000000021fdd001 () pa 32628(th pages)
.. ..511: pte 0x0000000021fdcc01 () pa 32627(th pages)
.. .. ..510: pte 0x0000000021fd90c7 (RW) pa 32612(th pages)
.. .. ..511: pte 0x0000000020001c4b (RX) pa 7(th pages)
panic: kfree
```

调用 kfree 函数的时候发生了 panic，将 uvmunmap 函数中调用 kfree 函数的语句注释掉：

```
173   void
174 uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
175   {
176     uint64 a;
177     pte_t *pte;
178
179     if((va % PGSIZE) != 0)
180       panic("uvmunmap: not aligned");
181
182     for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
183       if((pte = walk(pagetable, a, 0)) == 0)
184         panic("uvmunmap: walk");
185       //if((*pte & PTE_V) == 0)
186       //  panic("uvmunmap: not mapped");
187       if(PTE_FLAGS(*pte) == PTE_V)
188         panic("uvmunmap: not a leaf");
189       if(do_free){
190         //uint64 pa = PTE2PA(*pte);
191         //kfree((void*)pa);
192       }
193       *pte = 0;
194     }
195   }
```

再次运行，echo hi 正常运行：

```
$ echo hi
----------------Before----------------
page table 0x0000000087f75000
..0: pte 0x0000000021fd9801 () pa 32614(th pages)
.. ..0: pte 0x0000000021fd6401 () pa 32601(th pages)
.. .. ..0: pte 0x0000000021fdc05f (RWXU) pa 32624(th pages)
.. .. ..1: pte 0x0000000021fd60df (RWXU) pa 32600(th pages)
.. .. ..2: pte 0x0000000021fd5c0f (RWX) pa 32599(th pages)
.. .. ..3: pte 0x0000000021fd58df (RWXU) pa 32598(th pages)
..255: pte 0x0000000021fdd001 () pa 32628(th pages)
.. ..511: pte 0x0000000021fdcc01 () pa 32627(th pages)
.. .. ..510: pte 0x0000000021fd8cc7 (RW) pa 32611(th pages)
.. .. ..511: pte 0x0000000020001c4b (RX) pa 7(th pages)
----------------After----------------
page table 0x0000000087f75000
..0: pte 0x0000000021fd9801 () pa 32614(th pages)
.. ..0: pte 0x0000000021fd6401 () pa 32601(th pages)
.. .. ..0: pte 0x0000000021fdc05f (RWXU) pa 32624(th pages)
.. .. ..1: pte 0x0000000021fd60df (RWXU) pa 32600(th pages)
.. .. ..2: pte 0x0000000021fd5c0f (RWX) pa 32599(th pages)
.. .. ..3: pte 0x0000000021fd58df (RWXU) pa 32598(th pages)
.. .. ..4: pte 0x0000000021fd541f (RWXU) pa 32597(th pages)
..255: pte 0x0000000021fdd001 () pa 32628(th pages)
.. ..511: pte 0x0000000021fdcc01 () pa 32627(th pages)
.. .. ..510: pte 0x0000000021fd8cc7 (RW) pa 32611(th pages)
.. .. ..511: pte 0x0000000020001c4b (RX) pa 7(th pages)
```

```
----------------Before----------------
page table 0x0000000087f75000
..0: pte 0x0000000021fd9801 () pa 32614(th pages)
.. ..0: pte 0x0000000021fd6401 () pa 32601(th pages)
.. .. ..0: pte 0x0000000021fdc05f (RWXU) pa 32624(th pages)
.. .. ..1: pte 0x0000000021fd60df (RWXU) pa 32600(th pages)
.. .. ..2: pte 0x0000000021fd5c0f (RWX) pa 32599(th pages)
.. .. ..3: pte 0x0000000021fd58df (RWXU) pa 32598(th pages)
.. .. ..4: pte 0x0000000021fd54df (RWXU) pa 32597(th pages)
..255: pte 0x0000000021fdd001 () pa 32628(th pages)
.. ..511: pte 0x0000000021fdcc01 () pa 32627(th pages)
.. .. ..510: pte 0x0000000021fd8cc7 (RW) pa 32611(th pages)
.. .. ..511: pte 0x0000000020001c4b (RX) pa 7(th pages)
----------------After----------------
page table 0x0000000087f75000
..0: pte 0x0000000021fd9801 () pa 32614(th pages)
.. ..0: pte 0x0000000021fd6401 () pa 32601(th pages)
.. .. ..0: pte 0x0000000021fdc05f (RWXU) pa 32624(th pages)
.. .. ..1: pte 0x0000000021fd60df (RWXU) pa 32600(th pages)
.. .. ..2: pte 0x0000000021fd5c0f (RWX) pa 32599(th pages)
.. .. ..3: pte 0x0000000021fd58df (RWXU) pa 32598(th pages)
.. .. ..4: pte 0x0000000021fd54df (RWXU) pa 32597(th pages)
.. .. ..19: pte 0x0000000021fd501f (RWXU) pa 32596(th pages)
..255: pte 0x0000000021fdd001 () pa 32628(th pages)
.. ..511: pte 0x0000000021fdcc01 () pa 32627(th pages)
.. .. ..510: pte 0x0000000021fd8cc7 (RW) pa 32611(th pages)
.. .. ..511: pte 0x0000000020001c4b (RX) pa 7(th pages)
hi
$
```

3.3 Lazytests and Usertests

lazytests 和 usertests 是 xv6 用户程序，它们可以测试可能会使惰性内存分配器产生压力的某些特定情况。我们需要修改内核代码，以便所有 lazytests 和 usertests 都能通过。

（1）给 sbrk 添加处理参数为负数的情况，dealloc 相应的内存 n，注意 n 不能大于 p->sz：

```
41  uint64
42  sys_sbrk(void)
43  {
44    int addr;
45    int n;
46
47    if(argint(0, &n) < 0)
48      return -1;
49    struct proc* p = myproc();
50    addr = p->sz;
51    if (n < 0)
52    {
53      if (p->sz + n < 0) return -1;
54      else uvmdealloc(p->pagetable, p->sz, p->sz + n);
55    }
56    p->sz += n;
57    //if(growproc(n) < 0)
58    //  return -1;
59    
60    return addr;
61  }
```

（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：

```
72  ////////////////////////////////////////////////////////////
73  else if (r_scause() == 15 || r_scause() == 13)
74  {
75    uint64 va = r_stval();
76    if (va < p->sz && va > PGROUNDDOWN(p->trapframe->sp))
77    {
78      uint64 ka = (uint64) kalloc();
79      if (ka == 0) p->killed = -1;
80      else
81      {
82        memset((void*)ka, 0, PGSIZE);
83        va = PGROUNDDOWN(va);
84        if (mappages(p->pagetable, va, PGSIZE, ka, PTE_U | PTE_W| PTE_R) != 0)
85        {
86          kfree((void*)ka);
87          p->killed = -1;
88        }
89      }
90    }
91  else p->killed = -1;
92  }
93  ////////////////////////////////////////////////////////////
```

（3）对 fork 函数进行修改，子进程复制父进程地址空间的时候，发现地址空间不存在时需要忽略，修改 uvmcopy 和 uvmunmap 函数如下：

```
311  uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
312  {
313    pte_t *pte;
314    uint64 pa, i;
315    uint flags;
316    char *mem;
317
318    for(i = 0; i < sz; i += PGSIZE){
319      if((pte = walk(old, i, 1)) == 0)
320        //panic("uvmcopy: pte should exist");
321        continue;
322      if((*pte & PTE_V) == 0)
323        //panic("uvmcopy: page not present");
324        continue;
325      pa = PTE2PA(*pte);
326      flags = PTE_FLAGS(*pte);
327      if((mem = kalloc()) == 0)
328        goto err;
329      memmove(mem, (char*)pa, PGSIZE);
330      if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
331        kfree(mem);
332        goto err;
333      }
334    }
335    return 0;
336
337  err:
338    uvmunmap(new, 0, i / PGSIZE, 1);
339    return -1;
340  }
```

```
173  void
174  uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
175  {
176    uint64 a;
177    pte_t *pte;
178
179    if((va % PGSIZE) != 0)
180      panic("uvmunmap: not aligned");
181
182    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
183      if((pte = walk(pagetable, a, 0)) == 0)
184        //panic("uvmunmap: walk");
185        continue;
186      if((*pte & PTE_V) == 0)
187        //panic("uvmunmap: not mapped");
188        continue;
189      if(PTE_FLAGS(*pte) == PTE_V)
190        panic("uvmunmap: not a leaf");
191      if(do_free){
192        // uint64 pa = PTE2PA(*pte);
193        // kfree((void*)pa);
194      }
195      *pte = 0;
196    }
197  }
```

（4）进程将有效的地址从 sbrk()传递给系统调用，比如读或写，但是用于该地址的内存还没有分配，我们需要添加相应物理地址映射到 page table 里，找到 exec.c 里系统调用时虚拟地址转换为物理地址的代码，可以发现 exec 函数将虚拟地址传入 walkaddr 函数里得到相应物理地址：

```
144   for(i = 0; i < sz; i += PGSIZE){
145     pa = walkaddr(pagetable, va + i);
146     if(pa == 0)
147       panic("loadseg: address should exist");
148     if(sz - i < PGSIZE)
149       n = sz - i;
150     else
151       n = PGSIZE;
152     if(readi(ip, 0, (uint64)pa, offset+i, n) != n)
153       return -1;
154   }
```

修改 walkaddr 函数：

```
103   pte = walk(pagetable, va, 0);
104   ////////////////////////////////////
105   if(pte == 0 || (*pte & PTE_V) == 0)
106   {
107     uint64 ka = (uint64)kalloc();
108     if (ka == 0)
109     {
110       return 0;
111     }
112     // 因为考虑到系统调用，需要和之前额外添加PTE_X
113     if (mappages(pagetable, PGROUNDDOWN(va), PGSIZE, ka, PTE_W|PTE_X|PTE_R|PTE_U) != 0)
114     {
115       kfree((void*)ka);
116       return 0;
117     }
118     return ka;
119   }
120   ////////////////////////////////////
121   if((*pte & PTE_U) == 0)
122     return 0;
```

由于 sbrk 的懒惰分配，不仅在会产生进入 usertrap 的错误，在 write 和 read 时也会导致 panic 的错误而不会进入 usertrap，因此在这些地方也需要进行内存的分配：对 vm.c 的 copyin 函数的修改如下：

```
380  //·Copy·from·user·to·kernel.
381  //·Copy·len·bytes·to·dst·from·virtual·address·srcva·in·a·given·page·table.
382  //·Return·0·on·success,·-1·on·error.
383  int
384  copyin(pagetable_t·pagetable,·char·*dst,·uint64·srcva,·uint64·len)
385  {
386  ··uint64·n,·va0,·pa0;
387  ··////////////////////////////////
388  ··if(srcva·>·MAXVA)
389  ····return·-1;
390  ··while(len·>·0){
391  ····va0·=·PGROUNDDOWN(srcva);
392  ····pa0·=·walkaddr(pagetable,·va0);
393  ····if(pa0·==·0)
394  ····  return·-1;
395  ····n·=·PGSIZE·-·(srcva·-·va0);
396  ····if(n·>·len)
397  ····  n·=·len;
398  ····memmove(dst,·(void·*)(pa0·+·(srcva·-·va0)),·n);
399
400  ····len·-=·n;
401  ····dst·+=·n;
402  ····srcva·=·va0·+·PGSIZE;
403  ··}
404  ··return·0;
405  }
```

修改 copyout 函数：

```
357  //·Return·0·on·success,·-1·on·error.
358  int
359  copyout(pagetable_t·pagetable,·uint64·dstva,·char·*src,·uint64·len)
360  {
361  ··uint64·n,·va0,·pa0;
362  ··////////////////////////////////
363  ··if(dstva·>·MAXVA)
364  ····return·-1;
365
366  ··while(len·>·0){
367  ····va0·=·PGROUNDDOWN(dstva);
368  ····pa0·=·walkaddr(pagetable,·va0);
369  ····////////////////////////////////
370  ····if(pa0·==·0){
371  ····  ····//lazy·allocation·here
372  ····  ····char·*mem;
373  ····  ····if((mem·=·kalloc())·==·0)
374  ····  ····{
375  ····  ····  ··printf("kalloc·failed\n");
376  ····  ····  ··return·-1;
377  ····  ····}
378  ····  ····else
379  ····  ····{
380  ····  ····  ··pa0·=·(uint64)mem;
381  ····  ····  ··memset(mem,·0,·PGSIZE);
382  ····  ····  ··mappages(pagetable,·va0,·PGSIZE,·pa0,·PTE_W|PTE_X|PTE_R|PTE_U);
383  ····  ····}·
384  ····}
385  ····n·=·PGSIZE·-·(dstva·-·va0);
```

修改 file.c 中的 fileread 函数：

```
106  int
107  fileread(struct file *f, uint64 addr, int n)
108  {
109    int r = 0;
110
111    ////////////////////////////////
112    if(addr > (myproc()->sz))
113      return -1;
114    if(n > ((myproc()->sz) - addr))
115      return -1;
116    if(addr < 2*PGSIZE)
117      return -1;
118    if(f->readable == 0)
119      return -1;
120
121    if(f->type == FD_PIPE){
122      r = piperead(f->pipe, addr, n);
123    } else if(f->type == FD_DEVICE){
124      if(f->major < 0 || f->major >= NDEV || !devsw[f->major].read)
125        return -1;
126      r = devsw[f->major].read(1, addr, n);
127    } else if(f->type == FD_INODE){
128      ilock(f->ip);
```

修改 filewrite 函数：

```
139  // Write to file f.
140  // addr is a user virtual address.
141  int
142  filewrite(struct file *f, uint64 addr, int n)
143  {
144    int r, ret = 0;
145
146    ////////////////////////////////
147    if(addr > (myproc()->sz))
148      return -1;
149    if(n > ((myproc()->sz) - addr))
150      return -1;
151    if(addr < 2*PGSIZE)
152      return -1;
153    if(f->writable == 0)
154      return -1;
155
156    if(f->type == FD_PIPE){
157      ret = pipewrite(f->pipe, addr, n);
158    } else if(f->type == FD_DEVICE){
159      if(f->major < 0 || f->major >= NDEV || !devsw[f->major].write)
160        return -1;
161      ret = devsw[f->major].write(1, addr, n);
162    } else if(f->type == FD_INODE){
```

修改 syscall.c 中的 fetchstr 函数如下：

```
22  //·Fetch·the·nul-terminated·string·at·addr·from·the·current·process.
23  //·Returns·length·of·string,·not·including·nul,·or·-1·for·error.
24  int
25  fetchstr(uint64·addr,·char·*buf,·int·max)
26  {
27  ··struct·proc·*p·=·myproc();
28  ··//////////////////////////////
29  ··if(addr·>·(p->sz))
30  ····return·-1;
31  ··extern·struct·proc·*·initproc;
32  ··if((max·>·(p->sz)-addr)·&&·p·!=·initproc)
33  ····return·-1;
34  ··int·err·=·copyinstr(p->pagetable,·buf,·addr,·max);
35  ··if(err·<·0)
36  ····return·err;
37  ··return·strlen(buf);
38  }
```

4.实验结果

运行 lazytests 截图：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
test lazy unmap: OK
running test out of memory
test out of memory: OK
ALL TESTS PASSED
```

运行 usertests 截图：

```
$ usertests
usertests starting
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: usertrap(): unexpected scause 0x000000000000000c pid=3235
            sepc=0x0000000000005580 stval=0x0000000000005580
usertrap(): unexpected scause 0x000000000000000c pid=3236
            sepc=0x0000000000005580 stval=0x0000000000005580
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 sharedfd: OK
test exectest: 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
$
```