操作系统大作业 2

提交截止日：12 月 18 日零时

1. 总体要求

在 github 上创建 os-assignment2 项目，项目包括 2 个题目的结果:

1. 虚存管理模拟程序
2. Linux 内存管理实验程序

该目录下，同时存放 1 个 pdf/word 文件，作为实验报告。

注：附加题的分数单独计算，累加到正常分数上面，最后总分不超过 100 分。

2. 虚存管理模拟程序，50 分+10 分（附加题）

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

(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：本小题不要求实现页置换（Page Replacement），TLB 用简单的 FIFO 策略。30 分。

编程思路：

include 头文件
定义全局变量
声明函数

main 函数
声明各种要用到的变量

读取 BACKING_STORE.bin
读取 addresses.txt

初始化页表（使用-1 初始化页表，-1 代表空）
初始化 TLB（使用-1 初始化 TLB， -1 代表空）

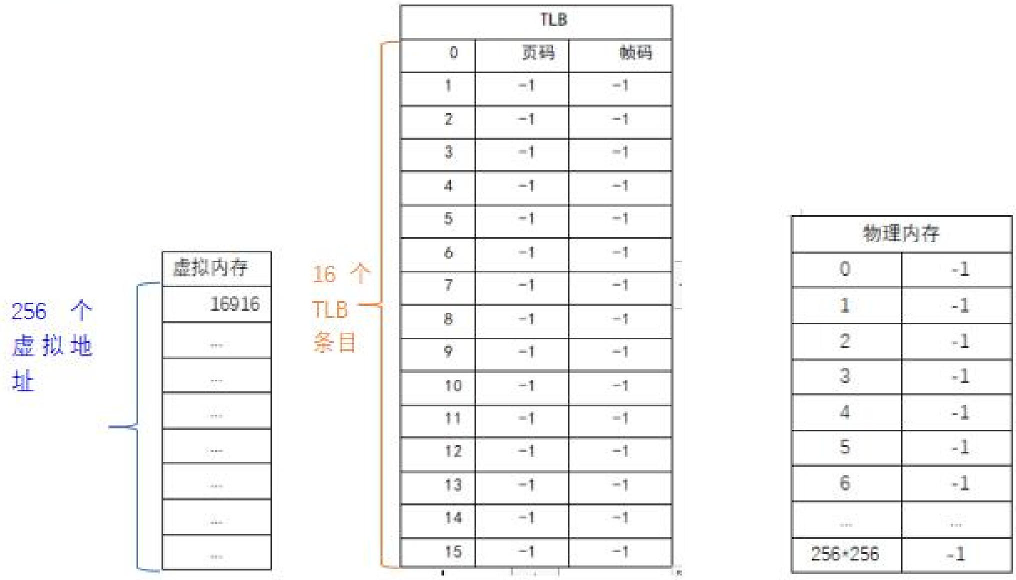
```
while(读取 addresses.txt) 循环一千次:
读取虚拟地址，计算页码和偏移

检查 tlb（FIFO）

若 tlb 未命中
{
          在页表项查找，若页表项无效，即读出的帧码为-1
    {
        调页，读取 BACKING_STORE.bin 的一页到物理内存的一帧
    }
          若页表项有效
    {
        读取帧码
        计算物理地址
        读取物理内存
                 }
    更新 tlb
}
若 tlb 命中
{
         读取帧码
         计算物理地址
         读取物理内存
}
   输出存储的数值
   格式为 printf("%d\n", Value);
   输出页错误数量，tlb 命中数量，关闭文件指针（一开始打开的那个地址文件的）
   printf("Page faults = %d\n",page_fault_counter);
   printf("TLB hits = %d\n",tlb_hit);

   关闭函数指针
   return 0;
}
```

TLB[16][2]被初始化为-1，-1 代表该 TLB 条目为空或无效

页表 page_table[256] 被初始化为-1，表示该缺页

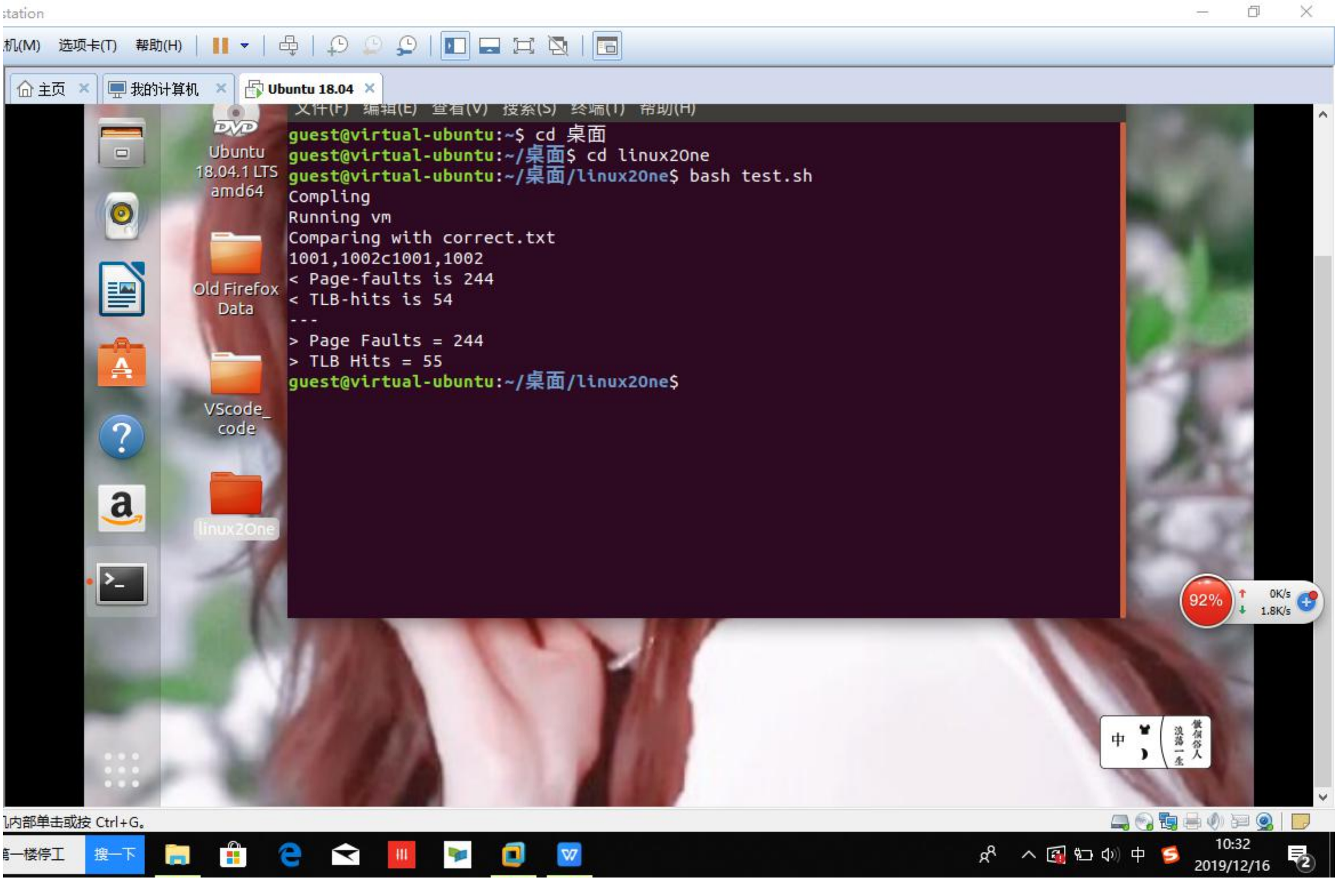
物理内存 memory[256*256]

采用 FIFO 策略替换 TLB 条目



页表	
0	-1
1	-1
2	-1
3	-1
4	-1
...	...
255	-1

程序运行结果截图：



代码部分：vm1.c

include 头文件:

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

定义常量:

```
#define PAGE_SIZE 256
#define PAGE_NUM 256
#define PAGE_TABLE_SIZE 256
#define FRAME_SIZE 256
#define FRAME_NUM 256
#define MEMORY_SIZE 256*256
#define TLB_SIZE 16
```

定义全局变量:

```
/* Virtual Addresses */

char buffer[8]; // read 8 characters once and put them in buffer[8]
int Page_Num; // Virtual Memory Page Number
int Page_Offset; // Virtual Memory Page Offset
int Virtual_Address; // Virtual Address
int mask = 255; // 0000 0000 0000 0000 0000 0000 1111 1111
int V_Address_Count = 0; // counting the numer of Virtual Addresses

/* page_table[256] */
int page_table[PAGE_TABLE_SIZE];
int Frame_Num; // the frame number stored in page table

/* TLB[16] */
int TLB[TLB_SIZE][2];
int tlb_hit = 0; // record the number of "tlb hit"
int tlb_count = 0; // record the number of the next valiable tlb

/* memory[256*256] */
char memory[MEMORY_SIZE];
int M_index = 0; // the frame number in memory
int Value; // the final output is stored in memory
int page_fault_counter = 0;
int Physical_Address;
```

定义函数:

```
// initialize the page_table[256] with -1
void init_page_table(void){
    for(int i = 0; i < PAGE_TABLE_SIZE; i++){
        page_table[i] = -1;
    }
}

//initialize the TLB with -1
void init_TLB(void){
    for(int i = 0; i <16; i++){
        TLB[i][0] = -1;
        TLB[i][1] = -1;
    }
}
```

```c
//check TLB[16]
int check_TLB(int page){
    for(int i = 0; i < 16; i++){
        if(TLB[i][0] == page){
            //TLB hit
            tlb_hit++;
            return TLB[i][1];
        }
    }
    //if TLB doesn't hit
    return -1;
}

// check page_table[256]
int check_page_table(int page){
    return page_table[page];
}
```

```c
// update the TLB
void update_TLB(int page, int frame){
    //update the TLB
    TLB[tlb_count][0] = page;
    TLB[tlb_count][1] = frame;

    //tlb_count point to the next available position
    tlb_count++;
    //FIFO:first in first out
    if(tlb_count % 16 == 0)
        tlb_count = 0;
}
```

main 函数:

```c
int main(int argc, char *argv[])
{
    char *V_Address_Dir; // addresses.txt
    char *STORE_Dir; // BACKING_STORE.bin
    int n = 0;

    if(argc != 3){
        printf("Enter input, store file names!");
        exit(EXIT_FAILURE);
    }
    // receive parameters from main
    STORE_Dir = argv[1];
    V_Address_Dir = argv[2];

    // define a file pointer
    FILE *file_ptr = NULL;

    //open the  BACKING_STORE.bin
    file_ptr = fopen(STORE_Dir, "rb");


    // Virtual Address file pointer
    FILE *filp = NULL;

    // open the addresses.txt
    filp = fopen(V_Address_Dir,"r");
```

```c

    // initializing the page table with -1;
    init_page_table();

    // initializing the TLB with -1
    init_TLB();
```

```c
// read a Virtual address once and store it in buffer[8]
while(fgets(buffer,sizeof(buffer),filp)){
    /* get virtual address*/
    // char buffer[8] -> int Virtual
    Virtual_Address = atoi(buffer);

    // Calculate the Virtual Page Offset and Virtual Page Number
    Page_Offset = Virtual_Address&mask;
    Page_Num = (Virtual_Address>>8)&mask;

    // Use V_Address_Count to count the number of Virtual_Address Addresses
    V_Address_Count++;

    /* get physical address*/
    //check TLB
    Frame_Num = check_TLB(Page_Num);

    //if TLB fail
    if (Frame_Num == -1){

        //check page table
        Frame_Num = check_page_table(Page_Num);

        if (Frame_Num == -1){
            // Page Fault
            page_fault_counter++;

            // read 256 bytes(a page) from store file to memory
            fseek(file_ptr, Page_Num*256, SEEK_SET);
            n = fread(memory+M_index*256, 1, 256, file_ptr);
            // read fail
            if(n == 0){
                printf("BACKING_STORE.bin could not be read");
                exit(EXIT_FAILURE);
            }

            // update the page table
            Frame_Num = M_index;
            page_table[Page_Num] = Frame_Num;

            //get final output from memory
            Physical_Address = Frame_Num * FRAME_SIZE + Page_Offset;
            Value = memory[Physical_Address];

            //指向物理内存下一帧
            M_index++;
        }
```

```c
            else{
                // No Frame Fault
                // figure out the physical address
                Physical_Address = Frame_Num * FRAME_SIZE + Page_Offset;

                // get the final output from the memory
                Value = memory[Physical_Address];
            }// memory

            //update the TLB with Page_Num, Frame_Num
            update_TLB(Page_Num, Frame_Num);

        }
        else{
            // TLB hit
            // figure out the physical address
            Physical_Address = Frame_Num * FRAME_SIZE + Page_Offset;

            // get the final output from the memory
            Value = memory[Physical_Address];
        }
        // print Virtual Address//
        printf("%d\n", Value);
        printf("Page faults = %d\n", page_fault_counter);
        printf("TLB hits = %d\n",tlb_hit);

    }

    fclose(file_ptr);
    fclose(filp);
    exit(0);
}
```

(2) 实现 LRU 的 TLB，8 分。

(3) 实现基于 LRU 的 Page Replacement，8 分。

(4) 代码可读性，4 分。

(5) 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。

编程思路：

定义头文件
定义全局变量
声明函数

main 函数
声明各种要用到的变量

接收 BACKING_STORE.bin
接收 addresses.txt
接收内存大小
接收页置换策略

读取 BACKING_STORE.bin
读取 addresses.txt

初始化页表（使用-1 初始化页表，-1 代表空）
初始化 TLB（使用-1 初始化 TLB， -1 代表空）
初始化 time_counter[256](这个数组每个元素是 memory 中每一页使用的时间计数器，采用 LRU 策略)

while(读取 addresses.txt) 循环一千次：
读取虚拟地址，计算页码 page 和偏移 offset

检查 tlb
调用 update_time_counter(page)，使 time_counter[page]=0,而其他 time_counter[i]+1 来标记 page 为最近使用的页

若 tlb 未命中
{
　　在页表项查找 page，若页表项无效，即读出的帧码为-1
{
　　调页，读取 BACKING_STORE.bin 的一页到物理内存的一帧
　　更新页表
　　计算物理地址
　　读取物理内存

```
        若物理内存还有空闲帧，M_index 指向下一个空闲帧
                       若是无空闲帧{
            若替换算法为 LRU
                调用 lru_replace_page( )函数，  得到下一个的用于页置换的帧号 M_index

            若替换算法为 FIFO
                M_index = (M_index + 1) % MEMORY_SIZE;

            如果是页置换，释放 free 对应页表项，使该被置换页表项的帧码为-1（使无效）
                并使对应的 time_counter[page] = -1（使无效）
            如果是页置换，释放 free 对应 TLB 项（使无效）
               }
    }
              若页表项有效，  读取帧码
    {
        计算物理地址
        读取物理内存
                    }

    更新 TLB（分为 LRU 和 FIFO 两种更新方法）
   }
   若 TLB 命中，读取帧码
   {
            计算物理地址
            读取物理内存
   }
      输出存储的数值 printf("%d\n", Value);
      输出页错误数量，tlb 命中数量，关闭文件指针（一开始打开的那个地址文件的）
      printf("Page faults = %d\n",page_fault_counter);
      printf("TLB hits = %d\n",tlb_hit);
      关闭函数指针
      return 0;
   }
```

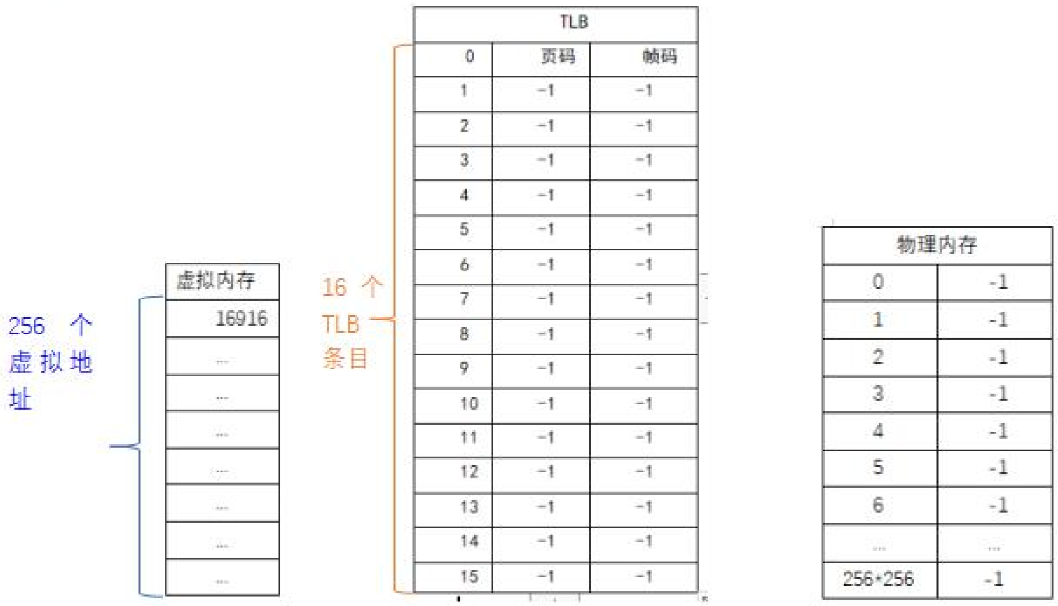
TLB[16][2]被初始化为-1，-1 代表该 TLB 条目为空或无效。

页表 page_table[256] 被初始化为-1，表示该缺页。

time_counter[256]被初始化为-1，表示没有页面被读取。

当替换算法为 LRU 时，使用 time_counter[256]来记录每一页的使用情况。

物理内存定义为 memory[256*256]，当用户定义物理内存大小 MEMORY_SIZE 为 256 时，使用全部的 memory[256*256]，当用户定义物理内存大小为 128 时，仅使用 memory[256*256]的前 128*256 的空间，即模拟物理内存为 memory[128*256]。



页表	
0	-1
1	-1
2	-1
3	-1
4	-1
...	...
255	-1

time_counter	
页码	时间
0	-1
1	-1
2	-1
3	-1
4	-1
...	...
255	-1

替换算法为 LRU，内存大小为 128 时，time_counter[256]最终结果输出部分截图和 page_table[256]的最终结果输出部分截图如下图所示：

time_counter 数组中，time_counter[i] = -1 表示：第 i 页不在物理内存中；time_counter 数组中的最大值为最近最少读取的页； time_counter 数组中的 time_counter[i] = 0 表示最近读取的页。

页表 page_table 数组中，page_table[i] = -1 表示：第 i 页不在物理内存中。

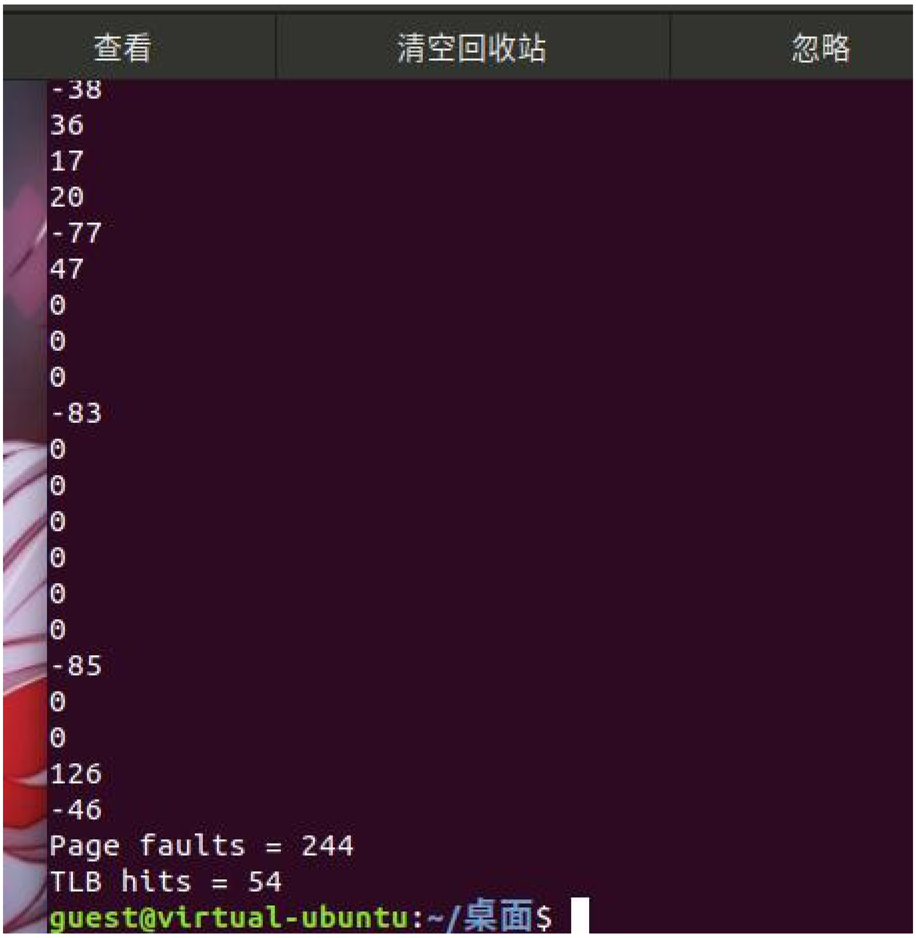
```
time_counter[0] = -1
time_counter[1] = 116
time_counter[2] = 65
time_counter[3] = -1
time_counter[4] = 118
time_counter[5] = -1
time_counter[6] = -1
time_counter[7] = 133
time_counter[8] = 7
time_counter[9] = -1
time_counter[10] = 158
time_counter[11] = -1
time_counter[12] = -1
time_counter[13] = -1
time_counter[14] = 78
time_counter[15] = -1
time_counter[16] = -1
time_counter[17] = -1
time_counter[18] = -1
time_counter[19] = -1
time_counter[20] = -1
```

```
page_table[0] = -1
page_table[1] = 89
page_table[2] = 83
page_table[3] = -1
page_table[4] = 58
page_table[5] = -1
page_table[6] = -1
page_table[7] = 42
page_table[8] = 51
page_table[9] = -1
page_table[10] = 85
page_table[11] = -1
page_table[12] = -1
page_table[13] = -1
page_table[14] = 101
page_table[15] = -1
page_table[16] = -1
page_table[17] = -1
page_table[18] = -1
page_table[19] = -1
page_table[20] = -1
```

代码运行结果截图：

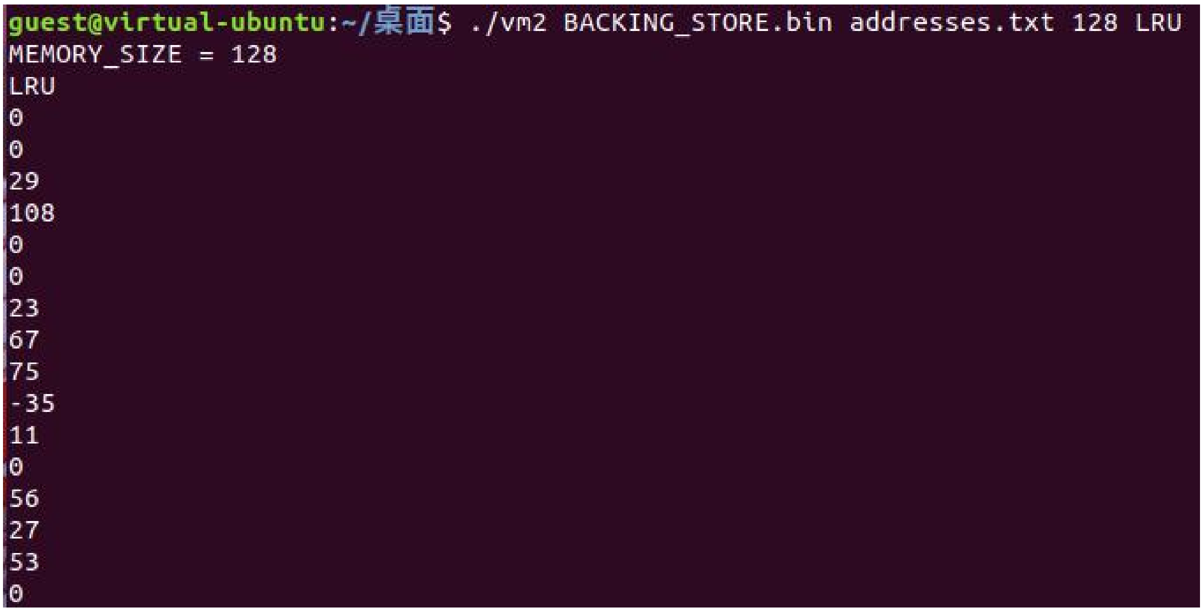
1、

```
guest@virtual-ubuntu:~/桌面$ ./vm2 BACKING_STORE.bin addresses.txt 256 LRU
MEMORY_SIZE = 256
LRU
0
0
29
108
0
```



```
查看        清空回收站        忽略
-38
36
17
20
-77
47
0
0
0
-83
0
0
0
0
0
0
-85
0
0
126
-46
Page faults = 244
TLB hits = 54
guest@virtual-ubuntu:~/桌面$
```

2、

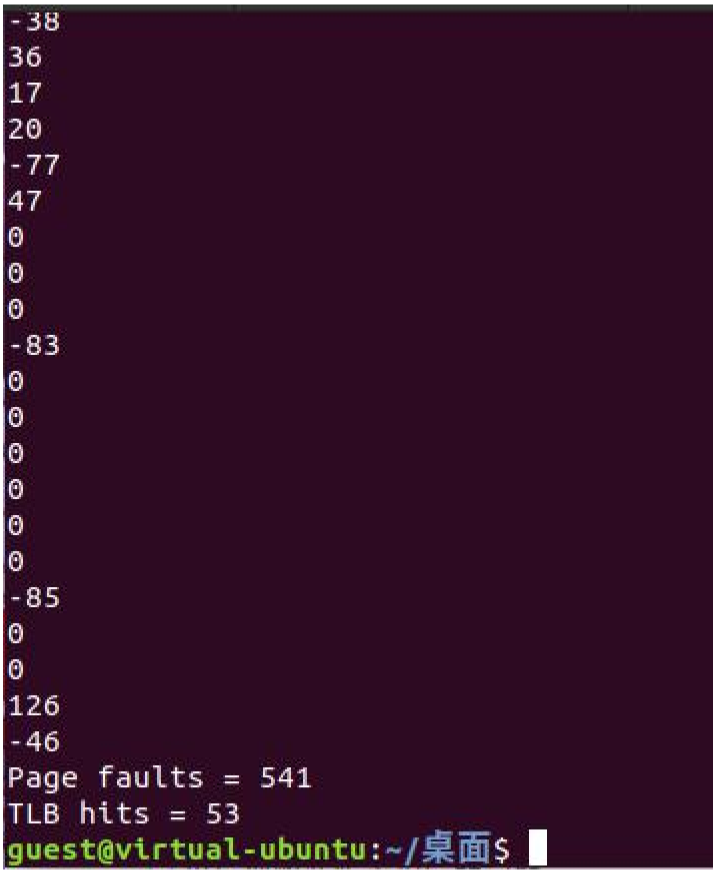


```
guest@virtual-ubuntu:~/桌面$ ./vm2 BACKING_STORE.bin addresses.txt 128 LRU
MEMORY_SIZE = 128
LRU
0
0
29
108
0
0
23
67
75
-35
11
0
56
27
53
0
```

```
-38
36
17
20
-77
47
0
0
0
-83
0
0
0
0
0
0
-85
0
0
126
-46
Page faults = 541
TLB hits = 53
```

3、

```
guest@virtual-ubuntu:~/桌面$ ./vm2 BACKING_STORE.bin addresses.txt 128 FIFO
MEMORY_SIZE = 128
FIFO
0
0
29
108
0
0
23
67
75
-35
11
0
56
27
53
0
```


-38
36
17
20
-77
47
0
0
0
-83
0
0
0
0
0
0
-85
0
0
126
-46
Page faults = 541
TLB hits = 53
guest@virtual-ubuntu:~/桌面$

程序代码截图：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>


#define PAGE_SIZE 256
#define PAGE_NUM 256
#define PAGE_TABLE_SIZE 256
#define FRAME_SIZE 256
#define FRAME_NUM 256
#define TLB_SIZE 16


/* Virtual Addresses */

char buffer[8]; // read 8 characters once and put them in buffer[8]
int Page_Num; // Virtual Memory Page Number
int Page_Offset; // Virtual Memory Page Offset
int Virtual_Address; // Virtual Address
int mask = 255; // 0000 0000 0000 0000 0000 0000 1111 1111
int V_Address_Count = 0; // counting the numer of Virtual Addresses

/* page_table[256] */
int page_table[PAGE_TABLE_SIZE];
int Frame_Num; // the frame number stored in page table
```

```c
/* TLB[16] */
int TLB[TLB_SIZE][2];
int tlb_hit = 0; // record the number of "tlb hit"
int tlb_count = 0; // record the number of the next valiable tlb

/* memory[256*256] */
char memory[256*256];
int M_index = 0; // the frame number in memory
int Value; // the final output is stored in memory
int page_fault_counter = 0;
int Physical_Address;

/* LRU frame time counter*/
int time_counter[256];
```

```c
/* Defining function */
// the function to initialize the page table
void init_page_table(void);

// the function to intialize the TLB
void init_TLB(void);

// the function to initialize the time_counter[256]
void init_time_counter(void);

// the function to check the TLB
int check_TLB(int page);

// the function to check the page table
int check_page_table(int page);

// free page_table
void free_page_table(int frame);

// free TLB
void free_TLB(int frame);

// the function to update the TLB
void fifo_update_TLB(int page, int frame);

// the function to update the TLB
void lru_update_TLB(int page, int frame);

// the function to record the used time of pages
void  update_time_counter(int page);
```

```c
// the function to get nex available page to replace
int lru_replace_page(void);
```

```c
int main(int argc, char *argv[])
{
    char *V_Address_Dir;// = "addresses.txt";
    char *STORE_Dir;// = "BACKING_STORE.bin";
    char *Memory_Size;// = "128";
    char *Replace_Algorithm;// = "LRU";

    int MEMORY_SIZE;
    int mem_empty = 1;

    if(argc != 5 & argc!=2){
        printf("Enter BACKING_STORE.bin, addresses.txt, Memory Size, FIFO or LRU!");
        exit(EXIT_FAILURE);
    }

    if(argc == 5){
        STORE_Dir = argv[1];
        V_Address_Dir = argv[2];
        Memory_Size = argv[3];
        Replace_Algorithm = argv[4];
    }

    if(argc == 2){
        STORE_Dir = argv[1];
        V_Address_Dir = argv[2];
        Memory_Size = "256";
        Replace_Algorithm = "FIFO";
    }
```

```c
    // get the real memory size
    MEMORY_SIZE = atoi(Memory_Size);
    printf("MEMORY_SIZE = %d\n", MEMORY_SIZE);
    printf("%s\n",Replace_Algorithm);

    // define a file pointer
    FILE *file_ptr = NULL;
    int n;
    //open the  BACKING_STORE.bin
    file_ptr = fopen(STORE_Dir, "rb");


    // Virtual Address file pointer
    FILE *filp = NULL;

    // open the addresses.txt
    filp = fopen(V_Address_Dir,"r");

    // initializing the page table with -1;
    init_page_table();

    // initializing the TLB with -1
    init_TLB();

    //initializing the time_counter with -1
    init_time_counter();
```

```c
// read a Virtual address once and store it in buffer[8]
while(fgets(buffer,sizeof(buffer),filp)){
    /* get virtual address*/
    // char buffer[8] -> int Virtual
    Virtual_Address = atoi(buffer);

    // Calculate the Virtual Page Offset and Virtual Page Number
    Page_Offset = Virtual_Address&mask;
    Page_Num = (Virtual_Address>>8)&mask;

    // Use V_Address_Count to count the number of Virtual_Address Addresses
    V_Address_Count++;

    /* get physical address*/
    //check TLB
    Frame_Num = check_TLB(Page_Num);

    // record the least used page
    update_time_counter(Page_Num);
```

```c
    //if TLB fail
    if (Frame_Num == -1){

        //check page table
        Frame_Num = check_page_table(Page_Num);

        if (Frame_Num == -1){
            // Page Fault
            page_fault_counter++;

            // read 256 bytes(a page) from store file to memory
            fseek(file_ptr, Page_Num*256, SEEK_SET);
            n = fread(memory+M_index*256, 1, 256, file_ptr);

            // read fail
            if(n == 0){
                printf("BACKING_STORE.bin could not be read");
                exit(EXIT_FAILURE);
            }

            // update the page table
            Frame_Num = M_index;
            page_table[Page_Num] = Frame_Num;

            //get final output from memory
            Physical_Address = Frame_Num * FRAME_SIZE + Page_Offset;
            Value = memory[Physical_Address];
```

```
            // if memory[i] is empty
            if(mem_empty){

                // M_index point to the next available memory
                M_index++;

                // if memory is full
                if(M_index == (MEMORY_SIZE - 1)){
                    // set men_empty = 0
                    mem_empty = 0;
                }
            }
            else{
                // if memory is full
                // replace page
                if(Replace_Algorithm == "LRU"){
                    // LRU: find the next availabe frame
                    // memory is full, replace page
                    M_index = lru_replace_page();

                    // if replace page, free page_table[i] and time_counter[i]
                    free_page_table(M_index);

                    // if replace page, free TLB[i]
                    free_TLB(M_index);
                }
                else{
                    // FIFO: find the next availabe frame
                    // if memory is full, replace page
                    M_index = (M_index + 1) % MEMORY_SIZE;

                    // if replace page, free page_table[i] and time_counter[i]
                    free_page_table(M_index);

                    // if replace page, free TLB[i]
                    free_TLB(M_index);
                }

            }

        }
        else{
            // No Frame Fault
            // figure out the physical address
            Physical_Address = Frame_Num * FRAME_SIZE + Page_Offset;

            // get the final output from the memory
            Value = memory[Physical_Address];

        }// memory
```

```
                //update the TLB with Page_Num, Frame_Num
                if(Replace_Algorithm == "LRU"){

                    //if Replace Algorithm is LRU
                    lru_update_TLB(Page_Num, Frame_Num);
                }
                else{

                    //if Replace Algorithm is FIFO
                    fifo_update_TLB(Page_Num, Frame_Num);
                }

            }
            else{
                // TLB hit
                // figure out the physical address
                Physical_Address = Frame_Num * FRAME_SIZE + Page_Offset;

                // get the final output from the memory
                Value = memory[Physical_Address];
            }


            // print Virtual Address//
            printf("%d\n", Value);

        }

    printf("Page faults = %d\n", page_fault_counter);
        printf("TLB hits = %d\n",tlb_hit);
    fclose(file_ptr);
    fclose(filp);
    exit(0);
}
```

```c
// initialize the page_table[256] with -1
void init_page_table(void){
    for(int i = 0; i < PAGE_TABLE_SIZE; i++){
        page_table[i] = -1;
    }
}


// initialize the TLB with -1
void init_TLB(void){
    for(int i = 0; i <16; i++){
        TLB[i][0] = -1;
        TLB[i][1] = -1;
    }
}


// initialize lru time counter[256] with -1
void init_time_counter(void){
    for(int i = 0; i < 256; i++){
        time_counter[i] = -1;
    }
}
```

```c
// check TLB[16]
int check_TLB(int page){
    for(int i = 0; i < 16; i++){
        if(TLB[i][0] == page){
            //TLB hit
            tlb_hit++;
            return TLB[i][1];
        }
    }
    //if TLB doesn't hit
    return -1;
}


// check page_table[256]
int check_page_table(int page){
    return page_table[page];
}
```

```c
 // free page table[i] and time_counter[i]
void free_page_table(int frame){

    // if replace page in memory
    // free page_table[i]
    for(int i = 0; i < PAGE_TABLE_SIZE; i++){

        // find out the replaced frame and free it from page_table[256]
        if(page_table[i] == frame){
            page_table[i] = -1;
            time_counter[i] = -1;
        }
    }
}


// free TLB
void free_TLB(int frame){
    // if frame is always in TLB[i], then free TLB[i]
    // set page=-1 and frame = -1
    for(int i = 0; i < 16; i++){
        if(TLB[i][1] == frame){
            TLB[i][0] = -1; // get page = -1
            TLB[i][1] = -1; // set frame = -1
        }
    }

}
```

```c
// update time counter
void  update_time_counter(int page){

    // time_counter of the recently used page is 0
    // time_counter of the least recently used page is the maximum.
    for(int i =0; i < 256; i++){
        if(i == page){
            time_counter[i] = 0;
        }
        else{
            if(time_counter[i] != -1){
                time_counter[i]++;
            }
        }
    }
}
```

```c
//FIFO: update TLB
 void fifo_update_TLB(int page, int frame){
        //update the TLB
        TLB[tlb_count][0] = page;
        TLB[tlb_count][1] = frame;

        //tlb_count point to the next available position
        tlb_count++;

        //FIFO:first in first out
        if(tlb_count % 16 == 0)
            tlb_count = 0;
}
```

```c
//LRU: update TLB
void lru_update_TLB(int page, int frame){
    int tlb_p;
    int max_p; // the max time page number
    int max_time = 0;
    int lru_n; // the relative TLB number
    int empty = 0; // TLB is full

    for(int i = 0; i < TLB_SIZE; i++){

        // if TLB[i] is empty
        if (TLB[i][0] == -1){

            // get the empty TLB number
            empty = 1;

            // get the relative TLB number
            lru_n = i;
        }
```

```
        //if TLB[i] is full
        if (!empty){
            // get the pages number stored in TLB
            tlb_p = TLB[i][0];

            // find out the max time of those pages
            // the max-time-page is the least used page
            if(time_counter[tlb_p] > max_time){
                max_time = time_counter[tlb_p];

                // get the max time page number
                max_p = tlb_p;

                // get the relative TLB number
                lru_n = i;
            }
        }
    }

    // update TLB in LRU
    TLB[lru_n][0] = page;
    TLB[lru_n][1] = frame;
}
```

```
// LRU: replace page
int lru_replace_page(void){
    int max_p;
    int max_time = 0;
    int page_table_p;

    // go through the page table and time counter to find the max used page
    for(int i = 0; i < 256; i++){
        page_table_p = i;
        if(time_counter[page_table_p] > max_time){
            max_time = time_counter[page_table_p];
            max_p = page_table_p;
        }
    }

    // return frame number which store the max used page
    return page_table[max_p];
}
```

2. （**附加题 10 分**）编写一个简单 trace 生成器程序（可以用任意语言，报告里面作为附件提供），运行生成自己的 addresses-locality.txt，包含 1 万条访问记录，体现内存访问的局部性（参考 Figure 10.21, OSC 10th ed.），绘制类似图表，表现内存页的局部性访问轨迹。然后以该文件为

参数运行 vm，比较 FIFO 和 LRU 策略下的性能指标，最好用图对比。给出结果及分析。

3. Linux 内存管理实验，50 分+10 分（附加题）

阅读 Linux 内存管理相关代码片段，提供程序和阅读报告，描述关键数据结构中和内存相关的成员的意义，以及指针指向关系。涉及的数据结构包括（但不限于）task_struct，mm_struct, vm_area_struct, vm_operations_struct, page 等

1. 分析图 1（注：图 1 是 2 级页表，对应于 IA-32 位系统），解释图中**每一类方框**和箭头的含义，在代码树中寻找相关数据结构片段，做简单解释。30 分。

2. 参考图 2 解释内核层不同内存分配接口的区别，包括__get_free_pages，kmalloc，vmalloc 等，3 分。

3. 参考 Anatomy of a Program in Memory 和 User-Level Memory Management 中例程，写一个实验程序 mtest.c，生成可执行程序 mtest；打印代码段、数据段、BSS，栈、堆等的相关地址；需要创建自己的例子，不允许简单照搬，8 分。

4. 参考 How The Kernel Manages Your Memory，通过/proc/pid_number/maps，分析 mtest 各个内存段（参考链接）。绘制图表，解释输出的每一段的各种属性，包括每一列的内容。为了让 mtest 程序驻留内存，可以在程序末尾加上长时睡眠，并将 mtest 在后台运行，即./mtest & 6 分。

5. 参考 A Malloc Tutorial 以及相关资料（如链接）回答以下问题：3 分
   （1）用户程序的内存分配涉及 brk/sbrk 和 mmap 两个系统调用，这两种方式的区别是什么，什么时候用 brk/sbrk，什么时候用 mmap？
   （2）应用程序开发时，为什么需要用标准库里的 malloc 而不是直接用这些系统调用接口？malloc 额外做了哪些工作？
   （3）malloc 的内存分配，是分配的虚拟内存还是物理内存？两者之间如何转换？

6. （**附加题，10 分**）模仿 malloc 接口，实现一对简单的函数，命名为 myalloc/myfree，实现堆上的动态内存分配和释放，并提供测试函数。相关代码以 myalloc.c 文件提供在项目目录下面。在自己的机器上进行实验，观察随着 malloc/free 的行为，/proc/pid_number/maps 中如何反映堆内存的变化情况，给出截屏和解释。实现基本功能 5 分，在内存块管理方面进行专门优化 5 分。

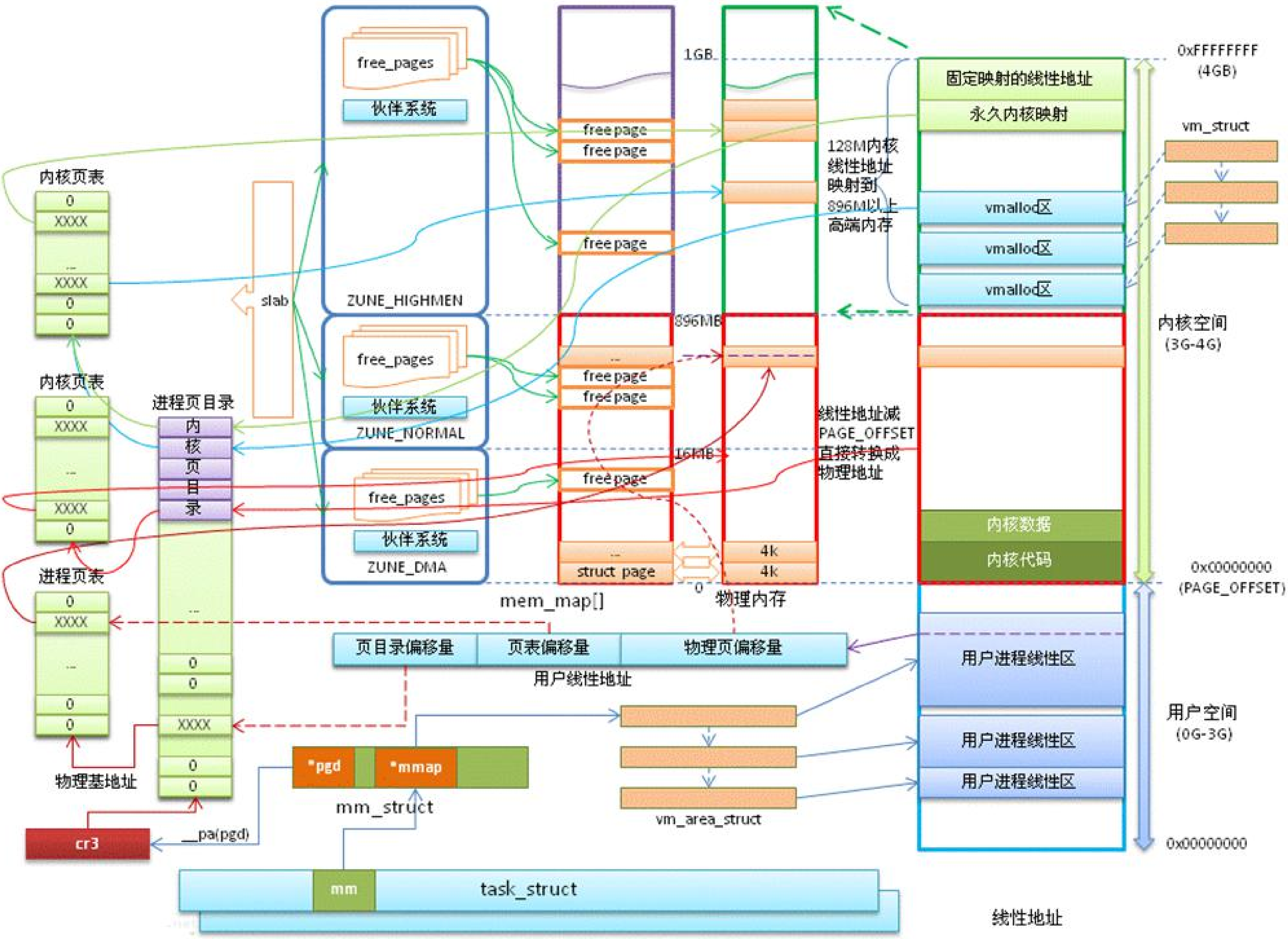


图 1. Linux 内核内存管理示意图（IA_32）

、

1、分析图 1（注：图 1 是 2 级页表，对应于 IA-32 位系统），解释图中**每一类方框**和箭头的含义，在代码树中寻找相关数据结构片段，做简单解释。30 分。

task_struct：Linux 内核通过一个被称为进程描述符的 task_sruct 结构体来管理进程，这个 task_sruct 结构体包含了一个进程所需的所有信息。

```c
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;

#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
    unsigned int wakee_flips;
    unsigned long wakee_flip_decay_ts;
    struct task_struct *last_wakee;

    int wake_cpu;
#endif
    int on_rq;

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
    struct task_group *sched_task_group;
#endif
    struct sched_dl_entity dl;

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers;
#endif
```

```c
#ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
#endif

    unsigned int policy;
    int nr_cpus_allowed;
    cpumask_t cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU
    int rcu_read_lock_nesting;
    union rcu_special rcu_read_unlock_special;
    struct list_head rcu_node_entry;
    struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TASKS_RCU
    unsigned long rcu_tasks_nvcsw;
    bool rcu_tasks_holdout;
    struct list_head rcu_tasks_holdout_list;
    int rcu_tasks_idle_cpu;
#endif /* #ifdef CONFIG_TASKS_RCU */

#ifdef CONFIG_SCHED_INFO
    struct sched_info sched_info;
#endif

    struct list_head tasks;
#ifdef CONFIG_SMP
    struct plist_node pushable_tasks;
    struct rb_node pushable_dl_tasks;
#endif

    struct mm_struct *mm, *active_mm;
    /* per-thread vma caching */
```

```
    u64 vmacache_seqnum;
    struct vm_area_struct *vmacache[VMACACHE_SIZE];
#if defined(SPLIT_RSS_COUNTING)
    struct task_rss_stat    rss_stat;
#endif
/* task state */
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */
    unsigned long jobctl;   /* JOBCTL_*, siglock protected */

    /* Used for emulating ABI behavior of previous Linux versions */
    unsigned int personality;

    /* scheduler bits, serialized by scheduler locks */
    unsigned sched_reset_on_fork:1;
    unsigned sched_contributes_to_load:1;
    unsigned sched_migrated:1;
    unsigned :0; /* force alignment to the next boundary */

    /* unserialized, strictly 'current' */
    unsigned in_execve:1; /* bit to tell LSMs we're in execve */
    unsigned in_iowait:1;
#ifdef CONFIG_MEMCG
    unsigned memcg_may_oom:1;
#endif
#ifdef CONFIG_MEMCG_KMEM
    unsigned memcg_kmem_skip_account:1;
#endif
#ifdef CONFIG_COMPAT_BRK
    unsigned brk_randomized:1;
#endif
#ifdef CONFIG_CGROUPS
    /* disallow userland-initiated cgroup migration */

    /* disallow userland-initiated cgroup migration */
    unsigned no_cgroup_migration:1;
#endif

    unsigned long atomic_flags; /* Flags needing atomic access. */

    struct restart_block restart_block;

    pid_t pid;
    pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
#endif
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->real_parent->pid)
     */
    struct task_struct __rcu *real_parent; /* real parent process */
    struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
    /*
     * children/sibling forms the list of my natural children
     */
    struct list_head children;  /* list of my children */
    struct list_head sibling;   /* linkage in my parent's children list */
    struct task_struct *group_leader;   /* threadgroup leader */

    /*
     * ptraced is the list of tasks this task is using ptrace on.
     * This includes both natural children and PTRACE_ATTACH targets.
     * p->ptrace_entry is p's link on the p->parent->ptraced list.
```

```
    struct list_head ptraced;
    struct list_head ptrace_entry;

    /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;
    struct list_head thread_node;

    struct completion *vfork_done;      /* for vfork() */
    int __user *set_child_tid;      /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */

    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
    struct prev_cputime prev_cputime;
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
    seqlock_t vtime_seqlock;
    unsigned long long vtime_snap;
    enum {
        VTIME_SLEEPING = 0,
        VTIME_USER,
        VTIME_SYS,
    } vtime_snap_whence;
#endif
    unsigned long nvcsw, nivcsw; /* context switch counts */
    u64 start_time;     /* monotonic time in nsec */
    u64 real_start_time;    /* boot based time in nsec */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
    unsigned long min_flt, maj_flt;

    struct task_cputime cputime_expires;
    struct list_head cpu_timers[3];

/* process credentials */
    const struct cred __rcu *ptracer_cred; /* Tracer's credentials at attach */
```

```
                     * credentials (COW) */
    const struct cred __rcu *cred;  /* effective (overridable) subjective task
                     * credentials (COW) */
    char comm[TASK_COMM_LEN]; /* executable name excluding path
                     - access with [gs]et_task_comm (which lock
                       it with task_lock())
                     - initialized normally by setup_new_exec */
/* file system info */
    struct nameidata *nameidata;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
    struct sysv_sem sysvsem;
    struct sysv_shm sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
    unsigned long last_switch_count;
#endif
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespaces */
    struct nsproxy *nsproxy;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
    struct sigpending pending;

    unsigned long sas_ss_sp;
```

```
    size_t sas_ss_size;

    struct callback_head *task_works;

    struct audit_context *audit_context;
#ifdef CONFIG_AUDITSYSCALL
    kuid_t loginuid;
    unsigned int sessionid;
#endif
    struct seccomp seccomp;

/* Thread group tracking */
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings, mems_allowed,
 * mempolicy */
    spinlock_t alloc_lock;

    /* Protection of the PI data structures: */
    raw_spinlock_t pi_lock;

    struct wake_q_node wake_q;

#ifdef CONFIG_RT_MUTEXES
    /* PI waiters blocked on a rt_mutex held by this task */
    struct rb_root pi_waiters;
    struct rb_node *pi_waiters_leftmost;
    /* Deadlock detection and priority inheritance handling */
    struct rt_mutex_waiter *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
    /* mutex deadlock detection */
    struct mutex_waiter *blocked_on;
#endif
```

```
#ifdef CONFIG_TRACE_IRQFLAGS
    unsigned int irq_events;
    unsigned long hardirq_enable_ip;
    unsigned long hardirq_disable_ip;
    unsigned int hardirq_enable_event;
    unsigned int hardirq_disable_event;
    int hardirqs_enabled;
    int hardirq_context;
    unsigned long softirq_disable_ip;
    unsigned long softirq_enable_ip;
    unsigned int softirq_disable_event;
    unsigned int softirq_enable_event;
    int softirqs_enabled;
    int softirq_context;
#endif
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 48UL
    u64 curr_chain_key;
    int lockdep_depth;
    unsigned int lockdep_recursion;
    struct held_lock held_locks[MAX_LOCK_DEPTH];
    gfp_t lockdep_reclaim_gfp;
#endif

/* journalling filesystem info */
    void *journal_info;

/* stacked block device info */
    struct bio_list *bio_list;

#ifdef CONFIG_BLOCK
/* stack plugging */
    struct blk_plug *plug;
#endif
```

```
/* VM state */
	struct reclaim_state *reclaim_state;

	struct backing_dev_info *backing_dev_info;

	struct io_context *io_context;

	unsigned long ptrace_message;
	siginfo_t *last_siginfo; /* For ptrace use.  */
	struct task_io_accounting ioac;
#if defined(CONFIG_TASK_XACCT)
	u64 acct_rss_mem1;	/* accumulated rss usage */
	u64 acct_vm_mem1;	/* accumulated virtual memory usage */
	cputime_t acct_timexpd;	/* stime + utime since last update */
#endif
#ifdef CONFIG_CPUSETS
	nodemask_t mems_allowed;	/* Protected by alloc_lock */
	seqcount_t mems_allowed_seq;	/* Seqence no to catch updates */
	int cpuset_mem_spread_rotor;
	int cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
	/* Control Group info protected by css_set_lock */
	struct css_set __rcu *cgroups;
	/* cg_list protected by css_set_lock and tsk->alloc_lock */
	struct list_head cg_list;
#endif
#ifdef CONFIG_FUTEX
	struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
	struct compat_robust_list_head __user *compat_robust_list;
#endif
	struct list_head pi_state_list;
	struct futex_pi_state *pi_state_cache;
#endif
```

```
#ifdef CONFIG_PERF_EVENTS
	struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
	struct mutex perf_event_mutex;
	struct list_head perf_event_list;
#endif
#ifdef CONFIG_DEBUG_PREEMPT
	unsigned long preempt_disable_ip;
#endif
#ifdef CONFIG_NUMA
	struct mempolicy *mempolicy;	/* Protected by alloc_lock */
	short il_next;
	short pref_node_fork;
#endif
#ifdef CONFIG_NUMA_BALANCING
	int numa_scan_seq;
	unsigned int numa_scan_period;
	unsigned int numa_scan_period_max;
	int numa_preferred_nid;
	unsigned long numa_migrate_retry;
	u64 node_stamp;		/* migration stamp  */
	u64 last_task_numa_placement;
	u64 last_sum_exec_runtime;
	struct callback_head numa_work;

	struct list_head numa_entry;
	struct numa_group *numa_group;

	/*
	 * numa_faults is an array split into four regions:
	 * faults_memory, faults_cpu, faults_memory_buffer, faults_cpu_buffer
	 * in this precise order.
```

```
	 * faults_memory: Exponential decaying average of faults on a per-node
	 * basis. Scheduling placement decisions are made based on these
	 * counts. The values remain static for the duration of a PTE scan.
	 * faults_cpu: Track the nodes the process was running on when a NUMA
	 * hinting fault was incurred.
	 * faults_memory_buffer and faults_cpu_buffer: Record faults per node
	 * during the current scan window. When the scan completes, the counts
	 * in faults_memory and faults_cpu decay and these values are copied.
	 */
	unsigned long *numa_faults;
	unsigned long total_numa_faults;

	/*
	 * numa_faults_locality tracks if faults recorded during the last
	 * scan window were remote/local or failed to migrate. The task scan
	 * period is adapted based on the locality of the faults with different
	 * weights depending on whether they were shared or private faults
	 */
	unsigned long numa_faults_locality[3];

	unsigned long numa_pages_migrated;
#endif /* CONFIG_NUMA_BALANCING */

#ifdef CONFIG_ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH
	struct tlbflush_unmap_batch tlb_ubc;
#endif

	struct rcu_head rcu;

	/*
	 * cache last used pipe for splice
	 */
	struct pipe_inode_info *splice_pipe;
```

```
	struct page_frag task_frag;

#ifdef	CONFIG_TASK_DELAY_ACCT
	struct task_delay_info *delays;
#endif
#ifdef CONFIG_FAULT_INJECTION
	int make_it_fail;
#endif
	/*
	 * when (nr_dirtied >= nr_dirtied_pause), it's time to call
	 * balance_dirty_pages() for some dirty throttling pause
	 */
	int nr_dirtied;
	int nr_dirtied_pause;
	unsigned long dirty_paused_when; /* start of a write-and-pause period */

#ifdef CONFIG_LATENCYTOP
	int latency_record_count;
	struct latency_record latency_record[LT_SAVECOUNT];
#endif
	/*
	 * time slack values; these are used to round up poll() and
	 * select() etc timeout values. These are in nanoseconds.
	 */
	unsigned long timer_slack_ns;
	unsigned long default_timer_slack_ns;

#ifdef CONFIG_KASAN
	unsigned int kasan_depth;
#endif
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
	/* Index of current stored address in ret_stack */
	int curr_ret_stack;
```

```
	/* Stack of return addresses for return function tracing */
	struct ftrace_ret_stack *ret_stack;
	/* time stamp for last schedule */
	unsigned long long ftrace_timestamp;
	/*
	 * Number of functions that haven't been traced
	 * because of depth overrun.
	 */
	atomic_t trace_overrun;
	/* Pause for the tracing */
	atomic_t tracing_graph_pause;
#endif
#ifdef CONFIG_TRACING
	/* state flags for use by tracers */
	unsigned long trace;
	/* bitmask and counter of trace recursion */
	unsigned long trace_recursion;
#endif /* CONFIG_TRACING */
#ifdef CONFIG_MEMCG
	struct mem_cgroup *memcg_in_oom;
	gfp_t memcg_oom_gfp_mask;
	int memcg_oom_order;

	/* number of pages to reclaim on returning to userland */
	unsigned int memcg_nr_pages_over_high;
#endif
#ifdef CONFIG_UPROBES
	struct uprobe_task *utask;
#endif
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
	unsigned int	sequential_io;
	unsigned int	sequential_io_avg;
#endif
#ifdef CONFIG_DEBUG_ATOMIC_SLEEP

	unsigned long	task_state_change;
#endif
	int pagefault_disabled;
/* CPU-specific state of this task */
	struct thread_struct thread;
/*
 * WARNING: on x86, 'thread_struct' contains a variable-sized
 * structure.  It *MUST* be at the end of 'task_struct'.
 *
 * Do not put anything below here!
 */
} « end task_struct » ;
```

mm： 进程所拥有的用户空间内存描述符。

```
	struct mm_struct *mm, *active_mm;
```

mm_struct： Linux 内核通过一个被称为内存描述符的 mm_sruct 结构体来管理进程，抽象描述了 linux 视角下一个进程整个虚拟地址空间的所有信息。每个进程都拥有自己一个 mm_sruct 结构体。

```c
struct mm_struct {
    struct vm_area_struct *mmap;        /* list of VMAs */
    struct rb_root mm_rb;
    u64 vmacache_seqnum;                    /* per-thread vmacache */
#ifdef CONFIG_MMU
    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
#endif
    unsigned long mmap_base;        /* base of mmap area */
    unsigned long mmap_legacy_base;         /* base of mmap area in bottom-up allocations */
    unsigned long task_size;        /* size of task vm space */
    unsigned long highest_vm_end;       /* highest vma end address */
    pgd_t * pgd;
    atomic_t mm_users;          /* How many users with user space? */
    atomic_t mm_count;          /* How many references to "struct mm_struct" (users count as 1) */
    atomic_long_t nr_ptes;          /* PTE page table pages */
#if CONFIG_PGTABLE_LEVELS > 2
    atomic_long_t nr_pmds;          /* PMD page table pages */
#endif
    int map_count;              /* number of VMAs */

    spinlock_t page_table_lock;     /* Protects page tables and some counters */
    struct rw_semaphore mmap_sem;

    struct list_head mmlist;        /* List of maybe swapped mm's.  These are globally strung
                         * together off init_mm.mmlist, and are protected
                         * by mmlist_lock
                         */


    unsigned long hiwater_rss;  /* High-watermark of RSS usage */
    unsigned long hiwater_vm;   /* High-water virtual memory usage */

    unsigned long total_vm;     /* Total pages mapped */
```

```c
    unsigned long locked_vm;    /* Pages that have PG_mlocked set */
    unsigned long pinned_vm;    /* Refcount permanently increased */
    unsigned long shared_vm;    /* Shared pages (files) */
    unsigned long exec_vm;      /* VM_EXEC & ~VM_WRITE */
    unsigned long stack_vm;     /* VM_GROWSUP/DOWN */
    unsigned long def_flags;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;

    unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */

    /*
     * Special counters, in some configurations protected by the
     * page_table_lock, in other configurations by being atomic.
     */
    struct mm_rss_stat rss_stat;

    struct linux_binfmt *binfmt;

    cpumask_var_t cpu_vm_mask_var;

    /* Architecture-specific MM context */
    mm_context_t context;

    unsigned long flags; /* Must use atomic bitops to access the bits

    struct core_state *core_state; /* coredumping support */
#ifdef CONFIG_AIO
    spinlock_t          ioctx_lock;
    struct kioctx_table __rcu   *ioctx_table;
#endif
#ifdef CONFIG_MEMCG
    /*
     * "owner" points to a task that is regarded as the canonical
```

```
	 * user/owner of this mm. All of the following must be true in
	 * order for it to be changed:
	 *
	 * current == mm->owner
	 * current->mm != mm
	 * new_owner->mm == mm
	 * new_owner->alloc_lock is held
	 */
	struct task_struct __rcu *owner;
#endif
	struct user_namespace *user_ns;

	/* store ref to file /proc/<pid>/exe symlink points to */
	struct file __rcu *exe_file;
#ifdef CONFIG_MMU_NOTIFIER
	struct mmu_notifier_mm *mmu_notifier_mm;
#endif
#if defined(CONFIG_TRANSPARENT_HUGEPAGE) && !USE_SPLIT_PMD_PTLOCKS
	pgtable_t pmd_huge_pte; /* protected by page_table_lock */
#endif
#ifdef CONFIG_CPUMASK_OFFSTACK
	struct cpumask cpumask_allocation;
#endif
#ifdef CONFIG_NUMA_BALANCING
	/*
	 * numa_next_scan is the next time that the PTEs will be marked
	 * pte_numa. NUMA hinting faults will gather statistics and migrate
	 * pages to new nodes if necessary.
	 */
	unsigned long numa_next_scan;

	/* Restart point for scanning and setting pte_numa */
	unsigned long numa_scan_offset;

	/* numa_scan_seq prevents two threads setting pte_numa */
```

```
	int numa_scan_seq;
#endif
#if defined(CONFIG_NUMA_BALANCING) || defined(CONFIG_COMPACTION)
	/*
	 * An operation with batched TLB flushing is going on. Anything that
	 * can move process memory needs to flush the TLB when moving a
	 * PROT_NONE or PROT_NUMA mapped page.
	 */
	bool tlb_flush_pending;
#endif
#ifdef CONFIG_ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH
	/* See flush_tlb_batched_pending() */
	bool tlb_flush_batched;
#endif
	struct uprobes_state uprobes_state;
#ifdef CONFIG_X86_INTEL_MPX
	/* address of the bounds directory */
	void __user *bd_addr;
#endif
#ifdef CONFIG_HUGETLB_PAGE
	atomic_long_t hugetlb_usage;
#endif
} « end mm_struct » ;
```

* **mmap** ：指向虚拟区间的链表，来查找线性区

```
	struct vm_area_struct *mmap;		/* list of VMAs */
```

vm_area_struct : linux 通过 vm_area_struct 结构的对象实现线性区，每个线性区描述符表示一个线性地址区间。

```c
struct vm_area_struct {
    /* The first cache line has the info for VMA tree walking. */

    unsigned long vm_start;     /* Our start address within vm_mm. */
    unsigned long vm_end;       /* The first byte after our end address
                       within vm_mm. */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;

    struct rb_node vm_rb;

    /*
     * Largest free memory gap in bytes to the left of this VMA.
     * Either between this VMA and vma->vm_prev, or between one of the
     * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
     * get_unmapped_area find a free area of the right size.
     */
    unsigned long rb_subtree_gap;

    /* Second cache line starts here. */

    struct mm_struct *vm_mm;    /* The address space we belong to. */
    pgprot_t vm_page_prot;      /* Access permissions of this VMA. */
    unsigned long vm_flags;     /* Flags, see mm.h. */

    /*
     * For areas with an address space and backing store,
     * linkage into the address_space->i_mmap interval tree.
     */
    struct {
        struct rb_node rb;
        unsigned long rb_subtree_last;
    } shared;

    /*
     * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
     * list, after a COW of one of the file pages.  A MAP_SHARED vma
     * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
     * or brk vma (with NULL file) can only be in an anon_vma list.
     */
    struct list_head anon_vma_chain; /* Serialized by mmap_sem &
                      * page_table_lock */
    struct anon_vma *anon_vma;  /* Serialized by page_table_lock */

    /* Function pointers to deal with this struct. */
    const struct vm_operations_struct *vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;     /* Offset (within vm_file) in PAGE_SIZE
                       units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;      /* File we map to (can be NULL). */
    void * vm_private_data;     /* was vm_pte (shared mem) */

#ifndef CONFIG_MMU
    struct vm_region *vm_region;    /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
    struct vm_userfaultfd_ctx vm_userfaultfd_ctx;
} « end vm_area_struct » ;
```

用户进程线性区：系统中进程的虚拟地址空间起始于地址 0，延伸至 TASK_SIZE -1， 总的地址空间按 3：1 划分，内核分配 1GB，各个用户空间进程可用的部分为 3GB。用户空间从 0G 到 3G，内核空间从 3G 到 4G。

mm -> *mmap： 进程描述符 task_struct 数据结构中进程所拥有的用户空间内存描述符 mm 指向*mmap，来查找线性区。

vm_area_struct -> 用户进程线性区 :linux 通过 vm_area_struct 结构的对象实现线性区，每个线性区描述符表示一个线性地址区间，vm_start 包含区域的第一个线性地址，vm_end 表示区域之外的第一个线性地址，vm_end-vm_start 和线性区的长度。所有线性区通过简单的链表链接在一起，链表按内存地址升序排序。

*** pgd：** 页表目录指针

```
struct mm_struct {
    struct vm_area_struct *mmap;        /* list of VMAs */
    struct rb_root mm_rb;
    u64 vmacache_seqnum;                   /* per-thread vmacache */
#ifdef CONFIG_MMU
    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
#endif
    unsigned long mmap_base;        /* base of mmap area */
    unsigned long mmap_legacy_base;         /* base of mmap area in bottom-up allocations */
    unsigned long task_size;        /* size of task vm space */
    unsigned long highest_vm_end;       /* highest vma end address */
    pgd_t * pgd;
```

cr3： 是 X86 的一个寄存器用来存放进程页目录的物理地址

```
/* Condition Register Bit Fields */

#define cr0 0
#define cr1 1
#define cr2 2
#define cr3 3
#define cr4 4
#define cr5 5
#define cr6 6
#define cr7 7
```

进程页目录 ： 进程页目录中存储的是页表的物理地址。

***pgd -> cr3 - > 进程页目录 ：** mm_sturct 中的页表目录指针*pgd 指向 cr3 寄存器，cr3 寄存器中存放进程页目录的物理地址，进而指向进程页目录。

用户线性地址 ： 被分为三个部分：页目录表偏移量、页表偏移量、物理页内的字节偏移量。

通过页目录表偏移量可在进程页目录中找到对应进程的页表的物理地址，通过页表偏移量可在对应进程的页表地址中找到最终物理页的物理起始地址，物理页基地址加上线性地址中的偏移量，CPU 就找到了线性地址最终对应的物理内存单元。

内核空间： 内核空间是由内核负责映射，是固定的。内核空间地址有自己对应的内核页表。

内核空间线性地址 -> 进程页目录 -> 内核页表 -> 物理内存： 由内核空间线性地址得

到进程页目录存放的内核页表的物理地址，进而找到内核页表得到物理内存的地址。

struct_page： 内核用 struct_page 结构体表示系统每一个物理页。flags 存放页的状态，如改页是不是脏页，_count 域表示该页的使用计数。_mapcout 表示在页表中存在多少个指向该页的项。lru 是一个表头，用于在各种链表上维护该页，mapping 指定了页帧所在的地址空间，private 是一个指向私有数据的指针，virtual 用于高端内存区的页。

```
struct page {
	/* First double word block */
	unsigned long flags;		/* Atomic flags, some possibly
					 * updated asynchronously */
	union {
		struct address_space *mapping;	/* If low bit clear, points to
						 * inode address_space, or NULL.
						 * If page mapped as anonymous
						 * memory, low bit is set, and
						 * it points to anon_vma object:
						 * see PAGE_MAPPING_ANON below.
						 */
		void *s_mem;			/* slab first object */
	};

	/* Second double word */
	struct {
		union {
			pgoff_t index;		/* Our offset within mapping. */
			void *freelist;		/* sl[aou]b first free object */
		};

		union {
#if defined(CONFIG_HAVE_CMPXCHG_DOUBLE) && \
	defined(CONFIG_HAVE_ALIGNED_STRUCT_PAGE)
			/* Used for cmpxchg_double in slub */
			unsigned long counters;
#else
			/*
			 * Keep _count separate from slub cmpxchg_double data.
			 * As the rest of the double word is protected by
			 * slab_lock but _count is not.
			 */
			unsigned counters;
```

```
#endif
            struct {
                union {
                    /*
                     * Count of ptes mapped in
                     * mms, to show when page is
                     * mapped & limit reverse map
                     * searches.
                     *
                     * Used also for tail pages
                     * refcounting instead of
                     * _count. Tail pages cannot
                     * be mapped and keeping the
                     * tail page _count zero at
                     * all times guarantees
                     * get_page_unless_zero() will
                     * never succeed on tail
                     * pages.
                     */
                    atomic_t _mapcount;

                    struct { /* SLUB */
                        unsigned inuse:16;
                        unsigned objects:15;
                        unsigned frozen:1;
                    };
                    int units;  /* SLOB */
                } « end {anon_union} » ;
                atomic_t _count;        /* Usage count, see below. */
            } « end {anon_struct} » ;
            unsigned int active;    /* SLAB */
        } « end {anon_union} » ;

    } « end {anon_struct} » ;

    /*
     * Third double word block
     *
     * WARNING: bit 0 of the first word encode PageTail(). That means
     * the rest users of the storage space MUST NOT use the bit to
     * avoid collision and false-positive PageTail().
     */
    union {
        struct list_head lru;   /* Pageout list, eg. active_list
                                 * protected by zone->lru_lock !
                                 * Can be used as a generic list
                                 * by the page owner.
                                 */
        struct {        /* slub per cpu partial pages */
            struct page *next;  /* Next partial slab */
#ifdef CONFIG_64BIT
            int pages;  /* Nr of partial slabs left */
            int pobjects;   /* Approximate # of objects */
#else
            short int pages;
            short int pobjects;
#endif
        };

        struct rcu_head rcu_head;   /* Used by SLAB
                                     * when destroying via RCU
                                     */
        /* Tail pages of compound page */
        struct {
```

```
			unsigned long compound_head; /* If bit zero is set */

			/* First tail page only */
#ifdef CONFIG_64BIT
			/*
			 * On 64 bit system we have enough space in struct page
			 * to encode compound_dtor and compound_order with
			 * unsigned int. It can help compiler generate better or
			 * smaller code on some archtectures.
			 */
			unsigned int compound_dtor;
			unsigned int compound_order;
#else
			unsigned short int compound_dtor;
			unsigned short int compound_order;
#endif
		};

#if defined(CONFIG_TRANSPARENT_HUGEPAGE) && USE_SPLIT_PMD_PTLOCKS
		struct {
			unsigned long __pad;	/* do not overlay pmd_huge_pte
						 * with compound_head to avoid
						 * possible bit 0 collision.
						 */
			pgtable_t pmd_huge_pte; /* protected by page->ptl */
		};
#endif
	} « end {anon_union} » ;
```

```
	/* Remainder is not double word aligned */
	union {
		unsigned long private;		/* Mapping-private opaque data:
						 * usually used for buffer_heads
						 * if PagePrivate set; used for
						 * swp_entry_t if PageSwapCache;
						 * indicates order in the buddy
						 * system if PG_buddy is set.
						 */
#if USE_SPLIT_PTE_PTLOCKS
#if ALLOC_SPLIT_PTLOCKS
		spinlock_t *ptl;
#else
		spinlock_t ptl;
#endif
#endif
		struct kmem_cache *slab_cache;	/* SL[AU]B: Pointer to slab */
	};

#ifdef CONFIG_MEMCG
	struct mem_cgroup *mem_cgroup;
#endif

	/*
	 * On machines where all RAM is mapped into kernel address space,
	 * we can simply calculate the virtual address. On machines with
	 * highmem some memory is mapped into kernel virtual memory
	 * dynamically, so we need a place to store that address.
	 * Note that this field could be 16 bits on x86 ... ;)
	 *
	 * Architectures with slow multiplication can define
	 * WANT_PAGE_VIRTUAL in asm/page.h
	 */
```

```
#if defined(WANT_PAGE_VIRTUAL)
    void *virtual;          /* Kernel virtual address (NULL if
                       not kmapped, ie. highmem) */
#endif /* WANT_PAGE_VIRTUAL */

#ifdef CONFIG_KMEMCHECK
    /*
     * kmemcheck wants to track the status of each byte in a page; this
     * is a pointer to such a status block. NULL if not tracked.
     */
    void *shadow;
#endif

#ifdef LAST_CPUPID_NOT_IN_PAGE_FLAGS
    int _last_cpupid;
#endif
} « end page »
```

区：区是逻辑上分组的概念，在 X86 体系结构中主要分为 3 个区：ZONE_DMA, ZONE_NORMAL, ZONE_HIGHMEM。ZONE_DMA 区中的页用来进行 DMA 时使用，ZONE_HIGHMEM 是高端地址，其中的页没有虚拟地址。剩余的内存就属于 ZONE_NORMAL 区。

```
struct zone {
    /* Read-mostly fields */

    /* zone watermarks, access with *_wmark_pages(zone) macros */
    unsigned long watermark[NR_WMARK];

    unsigned long nr_reserved_highatomic;

    /*
     * We don't know if the memory that we're going to allocate will be
     * freeable or/and it will be released eventually, so to avoid totally
     * wasting several GB of ram we must reserve some of the lower zone
     * memory (otherwise we risk to run OOM on the lower zones despite
     * there being tons of freeable ram on the higher zones).  This array is
     * recalculated at runtime if the sysctl_lowmem_reserve_ratio sysctl
     * changes.
     */
    long lowmem_reserve[MAX_NR_ZONES];

#ifdef CONFIG_NUMA
    int node;
#endif

    /*
     * The target ratio of ACTIVE_ANON to INACTIVE_ANON pages on
     * this zone's LRU.  Maintained by the pageout code.
     */
    unsigned int inactive_ratio;

    struct pglist_data  *zone_pgdat;
    struct per_cpu_pageset __percpu *pageset;

    /*
     * This is a per-zone reserve of pages that should not be
     * considered dirtyable memory.
```

```
	unsigned long		dirty_balance_reserve;

#ifndef CONFIG_SPARSEMEM
	/*
	 * Flags for a pageblock_nr_pages block. See pageblock-flags.h.
	 * In SPARSEMEM, this map is stored in struct mem_section
	 */
	unsigned long		*pageblock_flags;
#endif /* CONFIG_SPARSEMEM */

#ifdef CONFIG_NUMA
	/*
	 * zone reclaim becomes active if more unmapped pages exist.
	 */
	unsigned long		min_unmapped_pages;
	unsigned long		min_slab_pages;
#endif /* CONFIG_NUMA */

	/* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
	unsigned long		zone_start_pfn;

	/*
	 * spanned_pages is the total pages spanned by the zone, including
	 * holes, which is calculated as:
	 * 	spanned_pages = zone_end_pfn - zone_start_pfn;
	 *
	 * present_pages is physical pages existing within the zone, which
	 * is calculated as:
	 *	present_pages = spanned_pages - absent_pages(pages in holes);
	 *
	 * managed_pages is present pages managed by the buddy system, which
	 * is calculated as (reserved_pages includes pages allocated by the
	 * bootmem allocator):
	 *	managed_pages = present_pages - reserved_pages;
```


```
	 * bootmem allocator):
	 *	managed_pages = present_pages - reserved_pages;
```


```
	 *
	 * So present_pages may be used by memory hotplug or memory power
	 * management logic to figure out unmanaged pages by checking
	 * (present_pages - managed_pages). And managed_pages should be used
	 * by page allocator and vm scanner to calculate all kinds of watermarks
	 * and thresholds.
	 *
	 * Locking rules:
	 *
	 * zone_start_pfn and spanned_pages are protected by span_seqlock.
	 * It is a seqlock because it has to be read outside of zone->lock,
	 * and it is done in the main allocator path.  But, it is written
	 * quite infrequently.
	 *
	 * The span_seq lock is declared along with zone->lock because it is
	 * frequently read in proximity to zone->lock.  It's good to
	 * give them a chance of being in the same cacheline.
	 *
	 * Write access to present_pages at runtime should be protected by
	 * mem_hotplug_begin/end(). Any reader who can't tolerant drift of
	 * present_pages should get_online_mems() to get a stable value.
	 *
	 * Read access to managed_pages should be safe because it's unsigned
	 * long. Write access to zone->managed_pages and totalram_pages are
	 * protected by managed_page_count_lock at runtime. Idealy only
	 * adjust_managed_page_count() should be used instead of directly
	 * touching zone->managed_pages and totalram_pages.
	 */
	unsigned long		managed_pages;
	unsigned long		spanned_pages;
	unsigned long		present_pages;

	const char		*name;
```

```
#ifdef CONFIG_MEMORY_ISOLATION
	/*
	 * Number of isolated pageblock. It is used to solve incorrect
	 * freepage counting problem due to racy retrieving migratetype
	 * of pageblock. Protected by zone->lock.
	 */
	unsigned long		nr_isolate_pageblock;
#endif

#ifdef CONFIG_MEMORY_HOTPLUG
	/* see spanned/present_pages for more description */
	seqlock_t		span_seqlock;
#endif

	/*
	 * wait_table		-- the array holding the hash table
	 * wait_table_hash_nr_entries	-- the size of the hash table array
	 * wait_table_bits	-- wait_table_size == (1 << wait_table_bits)
	 *
	 * The purpose of all these is to keep track of the people
	 * waiting for a page to become available and make them
	 * runnable again when possible. The trouble is that this
	 * consumes a lot of space, especially when so few things
	 * wait on pages at a given time. So instead of using
	 * per-page waitqueues, we use a waitqueue hash table.
	 *
	 * The bucket discipline is to sleep on the same queue when
	 * colliding and wake all in that wait queue when removing.
	 * When something wakes, it must check to be sure its page is
	 * truly available, a la thundering herd. The cost of a
	 * collision is great, but given the expected load of the
	 * table, they should be so rare as to be outweighed by the
	 * benefits from the saved space.
	 *
	 * __wait_on_page_locked() and unlock_page() in mm/filemap.c, are the
```

__wait_on_page_locked() and unlock_page() in mm/filemap.c, are the

```
	 * primary users of these fields, and in mm/page_alloc.c
	 * free_area_init_core() performs the initialization of them.
	 */
	wait_queue_head_t	*wait_table;
	unsigned long		wait_table_hash_nr_entries;
	unsigned long		wait_table_bits;

	ZONE_PADDING(_pad1_)
	/* free areas of different sizes */
	struct free_area	free_area[MAX_ORDER];

	/* zone flags, see below */
	unsigned long		flags;

	/* Write-intensive fields used from the page allocator */
	spinlock_t		lock;

	ZONE_PADDING(_pad2_)

	/* Write-intensive fields used by page reclaim */

	/* Fields commonly accessed by the page reclaim scanner */
	spinlock_t		lru_lock;
	struct lruvec		lruvec;

	/* Evictions & activations on the inactive file list */
	atomic_long_t		inactive_age;

	/*
	 * When free pages are below this point, additional steps are taken
	 * when reading the number of free pages to avoid per-cpu counter
	 * drift allowing watermarks to be breached
	 */
	unsigned long percpu_drift_mark;
```

```c
#if defined CONFIG_COMPACTION || defined CONFIG_CMA
    /* pfn where compaction free scanner should start */
    unsigned long       compact_cached_free_pfn;
    /* pfn where async and sync compaction migration scanner should start */
    unsigned long       compact_cached_migrate_pfn[2];
#endif

#ifdef CONFIG_COMPACTION
    /*
     * On compaction failure, 1<<compact_defer_shift compactions
     * are skipped before trying again. The number attempted since
     * last failure is tracked with compact_considered.
     */
    unsigned int        compact_considered;
    unsigned int        compact_defer_shift;
    int         compact_order_failed;
#endif

#if defined CONFIG_COMPACTION || defined CONFIG_CMA
    /* Set to true when the PG_migrate_skip bits should be cleared */
    bool            compact_blockskip_flush;
#endif

    ZONE_PADDING(_pad3_)
    /* Zone statistics */
    atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
} « end zone »  ____cacheline_internodealigned_in_smp;
```

vmalloc ： vmalloc 分配的是虚拟地址连续，物理地址不一定连续的一片区域，一般为高端内存。需要释放该段内存是使用 vfree 函数。

伙伴系统： 内存初始化完成后，内存管理的工作由伙伴系统算法承担，伙伴算法采用叶框作为基本内存区，伙伴系统的调用既是为了获得存放新内存区所需的额外叶框，也是为了释放不再包含内存区的叶框。

slab : slab 层用于解决频繁分配和释放数据结构的问题。物理内存中有多个高速缓存，每个高速缓存中会有一个或多个 slab, slab 通常为一页，其中存放着数据结构类型的实例化对象。通过建立 slab 缓冲，内核能储备一些对象，供后续使用。slab 分配器将释放的内存块保存在一个内部列表中，而不是立刻返回给伙伴系统，这样内核不必使用伙伴系统算法，缩短了处理时间。

固定映射的线性地址： 它所对应的物理地址不是通过简单的线性转换得到的，而是人为强制指定的。每个固定线性地址都映射到任何一页物理内存。

永久内核映射： 允许内核建立高端页框到内核地址空间的长期映射。

2、参考图 2 解释内核层不同内存分配接口的区别，包括__get_free_pages，kmalloc，vmalloc 等，3 分。

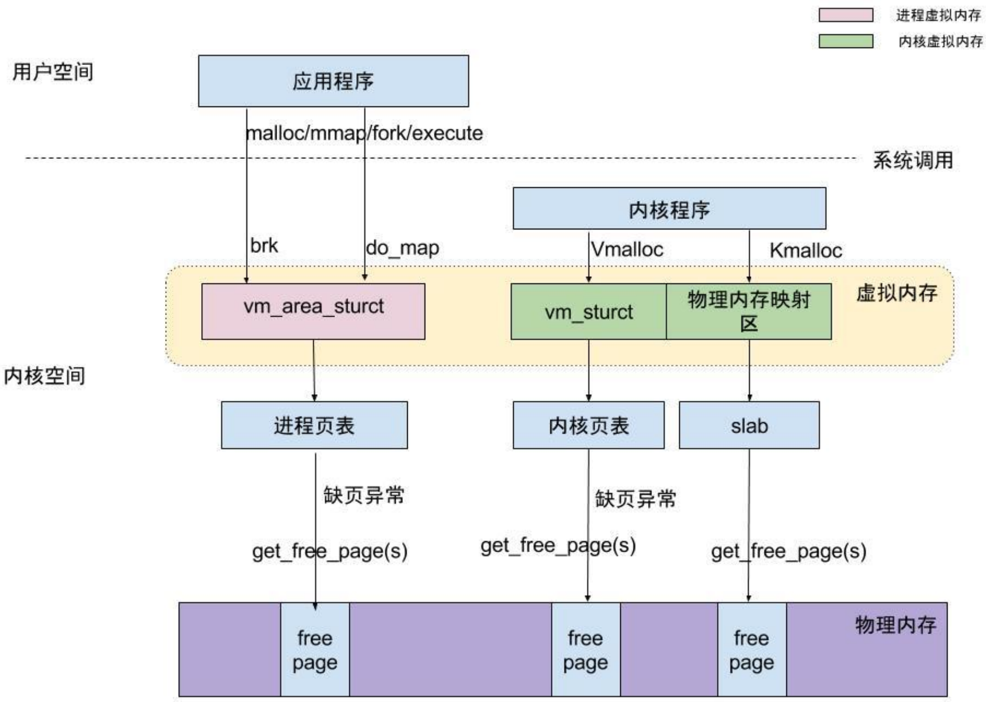


图 2. 用户空间和内核空间内存分配示意图

vmalloc： 在内存空间的 vmalloc 区（VMALLOC_START~4GB）中申请内存，并建立虚拟地址到物理地址映射，此时的虚拟地址连续而物理地址可能是不连续的，因为在申请内存时每个页面都是单独申请的，建立映射过程也是每个页面单独建立映射。在内核空间中调用 vmalloc()分配非物理连续空间，分配的地址成为内核虚拟地址。在分配过程中必须更新内核页表。

kmalloc： 分配的内核内存处于处于 3GB~high_memory 之间，这段内核空间与物理内存的映射一一对应，虚拟地址连续，物理地址也连续。在内核空间中调用 kmalloc()分配连续物理空间，分配的地址成为内核逻辑地址。kmalloc 分配内存是基于 slab。

malloc： 在用户内存进行分配，返回所分配内存块的虚拟地址。

_get_free_pages： 是最原始的内存分配方式，直接从伙伴系统中获取原始页框，返回值为第一个页框的起始地址。

3、参考 Anatomy of a Program in Memory 和 User-Level Memory Management 中例程，写一个实验程序 mtest.c，生成可执行程序 mtest；打印代码段、数据段、BSS，栈、堆等的相关地址；需要创建自己的例子，不允许简单照搬，8 分。

```
#include <stdio.h>
#include <malloc.h>
#include <unistd.h>
#include <alloca.h>

extern void show_stack(void); // the function to show stack growth

int bss_var;
int data_var1 = 40;
int data_var2 = 42;

int main(int argc, char **argv)
{
        char *stack_p, *h, *new_h;
        pid_t pid;
        pid = getpid();
        printf("THE pid is %d\n", pid);

        printf("Text Locations:\n");
        printf("\tAddress of main: %p\n", main);
        printf("\tAddress of show_stack: %p\n", show_stack);

        printf("Stack Locations:\n");
        show_stack();

        stack_p = (char *) alloca(48);
        if (stack_p != NULL){
                printf("\tStart of alloca() 'ed array: %p\n", stack_p);
                printf("\tEnd of alloca() 'ed array: %p\n", stack_p + 47);
        }

        printf("Data Locations:\n");
        printf("\tAddress of data_var1: %p\n", & data_var1);
        printf("\tAddress of data_var2: %p\n\n", & data_var2);

        printf("BSS Locations:\n");
"mtest.c" 65L, 1515C
```

```
        printf("\tAddress of bss_var: %p\n\n", &bss_var);
        printf("\tBUSS is above data variables.\n");

        h = sbrk((ptrdiff_t)16);
        new_h = sbrk((ptrdiff_t) 0);

        printf("Heap Locations:\n");
        printf("\tInitial end of heap: %p\n", h);
        printf("\tHeap grow upward\n");
        printf("\tNew end of heap: %p\n", new_h);

        h = sbrk((ptrdiff_t) - 48);
        new_h = sbrk((ptrdiff_t) 0);
        printf("\tAddress space shrinks.\n");
        printf("\tFinal end of heap: %p\n\n", new_h);

        sleep(100);
}

void show_stack(void){
        static int level = 0;
        auto int stack_var;
        level++;
        if(level == 4)
                return;

        printf("\tStack level %d: Address of stack_var: %p\n",level, & stack_var);
        show_stack();
}
```

运行结果:

```
>> Welcome to the Linux VM for Operating System Concepts!
osc@ubuntu:~$ cd linux2
osc@ubuntu:~/linux2$ ./mtest
THE pid is 1180
Text Locations:
        Address of main: 0x4006b6
        Address of show_stack: 0x4008b5
Stack Locations:
        Stack level 1: Address of stack_var: 0x7ffcb08d86a4
        Stack level 2: Address of stack_var: 0x7ffcb08d8684
        Stack level 3: Address of stack_var: 0x7ffcb08d8664
        Start of alloca()'ed array: 0x7ffcb08d8680
        End of alloca()'ed array: 0x7ffcb08d86af
Data Locations:
        Address of data_var1: 0x601060
        Address of data_var2: 0x601064

BSS Locations:
        Address of bss_var: 0x601070

        BUSS is above data variables.
Heap Locations:
        Initial end of heap: 0xfbc000
        Heap grow upward
        New end of heap: 0xfbc010
        Address space shrinks.
        Final end of heap: 0xfbbfe0
```

4、参考 How The Kernel Manages Your Memory，通过/proc/pid_number/maps，分析 mtest 各个内存段（参考链接）。绘制图表，解释输出的每一段的各种属性，包括每一列的内容。为了让 mtest 程序驻留内存，可以在程序末尾加上长时睡眠，并将 mtest 在后台运行，即./mtest & 6 分。

```
#include <stdio.h>
#include <malloc.h>
#include <unistd.h>
#include <alloca.h>

extern void show_stack(void); // the function to show stack growth

int bss_var;
int data_var1 = 40;
int data_var2 = 42;

int main(int argc, char **argv)
{
        char *stack_p, *h, *new_h;
        pid_t pid;
        pid = getpid();
        printf("THE pid is %d\n", pid);

        printf("Text Locations:\n");
        printf("\tAddress of main: %p\n", main);
        printf("\tAddress of show_stack: %p\n", show_stack);

        printf("Stack Locations:\n");
        show_stack();

        stack_p = (char *) alloca(48);
        if (stack_p != NULL){
                printf("\tStart of alloca() 'ed array: %p\n", stack_p);
                printf("\tEnd of alloca() 'ed array: %p\n", stack_p + 47);
        }

        printf("Data Locations:\n");
        printf("\tAddress of data_var1: %p\n", & data_var1);
        printf("\tAddress of data_var2: %p\n\n", & data_var2);

        printf("BSS Locations:\n");
"mtest.c" 65L, 1515C
```

```
        printf("\tAddress of bss_var: %p\n\n", &bss_var);
        printf("\tBUSS is above data variables.\n");

        h = sbrk((ptrdiff_t)16);
        new_h = sbrk((ptrdiff_t) 0);

        printf("Heap Locations:\n");
        printf("\tInitial end of heap: %p\n", h);
        printf("\tHeap grow upward\n");
        printf("\tNew end of heap: %p\n", new_h);

        h = sbrk((ptrdiff_t) - 48);
        new_h = sbrk((ptrdiff_t) 0);
        printf("\tAddress space shrinks.\n");
        printf("\tFinal end of heap: %p\n\n", new_h);

        sleep(100);
}

void show_stack(void){
        static int level = 0;
        auto int stack_var;
        level++;
        if(level == 4)
                return;

        printf("\tStack level %d: Address of stack_var: %p\n",level, & stack_var);
        show_stack();
}
                                                                                65
```

运行结果:

```
cat /proc/1186/maps
00400000-00401000 r-xp 00000000 08:01 23393                              /home/osc/linux2/mtest
00600000-00601000 r--p 00000000 08:01 23393                              /home/osc/linux2/mtest
00601000-00602000 rw-p 00001000 08:01 23393                              /home/osc/linux2/mtest
00647000-00669000 rw-p 00000000 00:00 0                                  [heap]
7f913a558000-7f913a718000 r-xp 00000000 08:01 134524                     /lib/x86_64-linux-gnu/libc-
2.23.so
7f913a718000-7f913a918000 ---p 001c0000 08:01 134524                     /lib/x86_64-linux-gnu/libc-
2.23.so
7f913a918000-7f913a91c000 r--p 001c0000 08:01 134524                     /lib/x86_64-linux-gnu/libc-
2.23.so
7f913a91c000-7f913a91e000 rw-p 001c4000 08:01 134524                     /lib/x86_64-linux-gnu/libc-
2.23.so
7f913a91e000-7f913a922000 rw-p 00000000 00:00 0
7f913a922000-7f913a948000 r-xp 00000000 08:01 134521                     /lib/x86_64-linux-gnu/ld-2.
23.so
7f913ab3b000-7f913ab3e000 rw-p 00000000 00:00 0
7f913ab47000-7f913ab48000 r--p 00025000 08:01 134521                     /lib/x86_64-linux-gnu/ld-2.
23.so
7f913ab48000-7f913ab49000 rw-p 00026000 08:01 134521                     /lib/x86_64-linux-gnu/ld-2.
23.so
7f913ab49000-7f913ab4a000 rw-p 00000000 00:00 0
7ffef70d0000-7ffef70f1000 rw-p 00000000 00:00 0                          [stack]
7ffef7125000-7ffef7127000 r--p 00000000 00:00 0                          [vvar]
7ffef7127000-7ffef7129000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
osc@ubuntu:~/linux2$
```

分析表格在附件里

5、参考 A Malloc Tutorial 以及相关资料（如链接）回答以下问题：3 分

（1）用户程序的内存分配涉及 brk/sbrk 和 mmap 两个系统调用，这两种方式的区别是什么，什么时候用 brk/sbrk，什么时候用 mmap？

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk 和 mmap（不考虑共享内存）。

1、brk 是将数据段(.data)的最高地址指针_edata 往高地址推；

2、mmap 是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

在标准 C 库中，提供了 malloc/free 函数分配释放内存，这两个函数底层是由 brk，mmap，munmap 这些系统调用实现的。

情况一、当 malloc 小于 128k 的内存，使用 brk 分配内存，将_edata 往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)

情况二、malloc 大于 128k 的内存，使用 mmap 分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为 0)

（2）应用程序开发时，为什么需要用标准库里的 malloc 而不是直接用这些系统调用接口？malloc 额外做了哪些工作？

mmap 系统调用实现了更有用的动态内存分配功能，可以将一个磁盘文件的全部或部分内容映射到用户空间中，进程读写文件的操作变成了读写内存的操作。在 linux/mm/mmap.c 文件的 do_mmap_pgoff()函数，是 mmap 系统调用实现的核心。do_mmap_pgoff()的代码，只是新建了一个 vm_area_struct 结构，并把 file 结构的参数赋值给其成员变量 m_file，并没有把文件内容实际装入内存。

Linux 内存管理的基本思想之一，是只有在真正访问一个地址的时候才建立这个地址的物理映射。

（3）malloc 的内存分配，是分配的虚拟内存还是物理内存？两者之间如何转换？

虚拟内存，通过地址映射。