OS LAB3

刘芷辰　PB21111728

2023年5月29日

1. 实验题目

- part1： 动态内存分配器malloc的实现
- part2： Linux进程内存信息统计

2. 实验目的

- 通过代码填空的方式，让学生了解真实Linux系统的内存管理方式
- 使用显式空闲链表实现一个64位堆内存分配器
- 学会以动态链接库的形式制作库并使用
- 体会系统实验数据的测量和分析过程

3. 实验环境

- OS: Ubuntu 20.04.4 LTS
- 无需在QEMU下调试

4. 实验内容

> PART1 动态内存分配器malloc的实现

1. **void mem_init(void)**

```c
void mem_init(void)
{
    /*
        TODO:
        调用 sbrk，初始化 mem_start_brk、mem_brk、以及 mem_max_addr
        此处增长堆空间大小为 MAX_HEAP
    */
    mem_brk = mem_start_brk = sbrk(MAX_HEAP);
    mem_max_addr = mem_start_brk + MAX_HEAP;
}
```

- 初始化第一个字节地址和末尾字节地址都是sbrk的返回值，即堆上界brk的旧值，在初始化时即为起始地址
- 最大可用地址则为第一个字节地址加上分配的增长堆空间大小

2. void *mem_sbrk(int incr)

```c
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;

    /*
        TODO:
            模拟堆增长
            incr：申请 mem_brk 的增长量
            返回值：旧 mem_brk 值
        HINTS:
        1. 若 mem_brk + incr 没有超过实际的 mem_max_addr 值，直接推进 mem_brk 值即可
        2. 若 mem_brk + incr 超过实际的 mem_max_addr 值，需要调用 sbrk 为内存分配器掌管的内存扩容
        3. 每次调用 sbrk 时， mem_max_addr 增量以 MAXHEAP对齐
    */
    if(mem_brk + incr < mem_max_addr) mem_brk += incr;
    else {
       while(mem_brk + incr >= mem_max_addr){
        sbrk(MAX_HEAP);
        mem_max_addr += MAX_HEAP;
        }
       mem_brk += incr;
   }
    return (void *)old_brk;
}
```

- 增加量为incr，所以mem_brk + incr小于mem_max_addr时，可以直接增长，只需改变末尾字节地址mem_brk加上增长量incr即可
- 当mem_brk + incr大于mem_max_addr时，则需要扩容，调用sbrk每次扩容MAX_HEAP，直到满足扩容要求，调整最大地址，并改变末尾字节地址mem_brk加上增长量incr

3. static void *find_fit_first(size_t asize)

```
static void *find_fit_first(size_t asize)
{
    /*
        首次匹配算法
        TODO:
            遍历 freelist， 找到第一个合适的空闲块后返回

        HINT: asize 已经计算了块头部的大小
    */

    char *p = free_listp;

    for (; p; p = (void *)GET_SUCC(p))
    {
        if (GET_SIZE(HDRP(p)) > asize)
            return p;
    }
    return NULL; // 换成实际返回值
}
```

- 遍历freelist，找到size>asize的空闲块即返回该空闲块

4. static void *find_fit_best(size_t asize)

```
static void *find_fit_best(size_t asize)
{
    /*
        最佳配算法
        TODO:
            遍历 freelist， 找到最合适的空闲块，返回

        HINT: asize 已经计算了块头部的大小
    */
    char *p = free_listp;
    char *fit = NULL;
```

```
    int min = 200000000;
    while (p)
    {
        if (GET_SIZE(HDRP(p)) > asize)
        {
            if (GET_SIZE(HDRP(p)) - asize < min)
            {
                min = GET_SIZE(HDRP(p)) - asize;
                fit = p;

            }
        }
        p = (void *)GET_SUCC(p);
    }
    return fit;
}
```

- 遍历freelist，找到满足size>asize的空闲块时，和目前最小的差值比较，若更小则替换最小值，并记录当前的指针给fit，遍历完成后返回fit即为最小且合适的空闲块

5. static void* coalesce(void* bp)

```
static void *coalesce(void *bp)
{
    /*add_to_free_list(bp);*/
    size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    /*
        TODO:
            将 bp 指向的空闲块 与 相邻块合并
            结合前一块及后一块的分配情况，共有 4 种可能性
            分别完成相应case下的 数据结构维护逻辑
    */
    if (prev_alloc && next_alloc) /* 前后都是已分配的块 */
    {
        add_to_free_list(bp);
    }
    else if (prev_alloc && !next_alloc) /*前块已分配，后块空闲*/
    {
        char *next_bp = NEXT_BLKP(bp);
        delete_from_free_list(next_bp);
        size_t next_size = GET_SIZE(HDRP(next_bp));
```

```
22
23          PUT(HDRP(bp), PACK(size + next_size, 1, 0));
24          PUT(FTRP(next_bp), PACK(size + next_size, 1, 0));
25          add_to_free_list(bp);
26      }
27      else if (!prev_alloc && next_alloc) /*前块空闲，后块已分配*/
28      {
29          char *prev_bp = PREV_BLKP(bp);
30          delete_from_free_list(PREV_BLKP(bp));
31          size_t prev_size = GET_SIZE(HDRP(prev_bp));
32          size_t prev_alloc = GET_PREV_ALLOC(HDRP(prev_bp));
33
34          PUT(HDRP(prev_bp), PACK(size + prev_size, prev_alloc, 0));
35          PUT(FTRP(bp), PACK(size + prev_size, prev_alloc, 0));
36          bp = prev_bp;
37          add_to_free_list(bp);
38      }
39      else /*前后都是空闲块*/
40      {
41          char *prev_bp = PREV_BLKP(bp);
42          char *next_bp = NEXT_BLKP(bp);
43          delete_from_free_list(PREV_BLKP(bp));
44          delete_from_free_list(NEXT_BLKP(bp));
45
46          size_t prev_size = GET_SIZE(HDRP(prev_bp));
47          size_t next_size = GET_SIZE(HDRP(next_bp));
48          size_t prev_alloc = GET_PREV_ALLOC(HDRP(prev_bp));
49
50          PUT(HDRP(prev_bp), PACK(size + prev_size + next_size,
    prev_alloc, 0));
51          PUT(FTRP(next_bp), PACK(size + prev_size + next_size,
    prev_alloc, 0));
52          bp = prev_bp;
53          add_to_free_list(bp);
54      }
55      return bp;
56  }
```

- 前后都是已分配的块：直接把释放为空闲的块加入空闲列表
- 前块已分配，后块空闲：则需要合并释放为空闲的块和后块，首先将后块从空闲列表中删除，然后通过PUT修改合并后形成的空闲块的头部和脚部信息，包括size修改为两个块size之和，前块已分配所以为1，该块空闲所以为0，最后将合并后的块加入空闲列表

- 前块空闲，后块已分配：则需要合并释放为空闲的块和前块，首先将前块从空闲列表中删除，然后通过PUT修改合并后形成的空闲块的头部和脚部信息，包括size修改为两个块size之和，前块是否分配通过GET_PREV_ALLOC获取，该块空闲所以为0，最后将合并后的块加入空闲列表
- 前后块都空闲：则需要将三个块都合并，首先将前块和后块从空闲列表中删除，然后通过PUT修改合并后形成的空闲块的头部信息和脚部信息，包括size修改为三个块size之和，前块是否分配通过GET_PREV_ALLOC获取，该块空闲所以为0，最后将合并后的块加入空闲列表

6. static void place(void* bp, size_t asize)

```
static void place(void *bp, size_t asize)
{
    /*
        TODO:
        将一个空闲块转变为已分配的块

        HINTS:
            1. 若空闲块在分离出一个 asize 大小的使用块后，剩余空间不足空闲块的最小大小，
                则原先整个空闲块应该都分配出去
            2. 若剩余空间仍可作为一个空闲块，则原空闲块被分割为一个已分配块+一个新的空闲块
            3. 空闲块的最小大小已经 #define，或者根据自己的理解计算该值
    */
    size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
    size_t old_size = GET_SIZE(HDRP(bp));


    delete_from_free_list(bp);

    if (old_size - asize > MIN_BLK_SIZE)//分配后分为分配块＋空闲块
    {

        PUT(HDRP(bp), PACK(asize, prev_alloc, 1));
        char *next_bp = NEXT_BLKP(bp);
        PUT(HDRP(next_bp), PACK(old_size - asize, 1, 0));
        PUT(FTRP(next_bp), PACK(old_size - asize, 1, 0));
        add_to_free_list(next_bp);
    }
    else//分配后为分配块
    {
        PUT(HDRP(bp), PACK(old_size, prev_alloc, 1));
```

```
        char *next_bp = NEXT_BLKP(bp);
        size_t next_size = GET_SIZE(HDRP(next_bp));
        size_t next_alloc = GET_ALLOC(HDRP(next_bp));
        if (!next_alloc)
            PUT(FTRP(next_bp), PACK(next_size, 1, 0));
        PUT(HDRP(next_bp), PACK(next_size, 1, next_alloc));
    }
}

```

- 首先从空闲列表中删除该块
- 当分离一个asize大小的分配块出去后，若剩下的空间满足空闲块的最小大小，则形成分配块+空闲块，首先修改分配块的头部信息，size修改为asize，前块是否被分配通过GET_PREV_ALLOC获取，该块已分配所以为1，然后修改后面空闲块的头部信息和脚部信息，size修改为原来的size减去asize，前块被分配所以为1，该块空闲所以为0，最后将该空闲块加入空闲列表
- 若分配asize后剩余空间不够空闲块的最小大小，则应该将该块全部分配出去，首先修改分配块的头部信息，size不变，前块是否分配通过GET_PREV_ALLOC获取，该块已分配所以为1，由于该块变为分配块所以需要修改后块信息，size不变，前块已分配所以为1，该块是否被分配通过GET_ALLOC获取，若是空闲块还需要修改脚部信息和头部信息一致

> PART2 Linux进程内存信息统计

- **func1**

```
// func == 1
static void scan_vma(void)
{
    printk("func == 1, %s\n", __func__);
    struct mm_struct *mm = get_task_mm(my_task_info.task);
    if (mm)
    {
        // TODO:遍历VMA将VMA的个数记录到my_task_info的vma_cnt变量中
        int cnt = 0;
        struct vm_area_struct* ptr = mm->mmap;
        while (ptr){
            cnt++;
            ptr = ptr->vm_next;
        }

```

```
        my_task_info.vma_cnt = cnt;
        mmput(mm);
    }
}
```

- 遍历vma，每个next将计数器加一，得到总数后传入vma_cnt中

- **func2**

```
// func == 2
static void print_mm_active_info(void)
{
    printk("func == 2, %s\n", __func__);
    // TODO: 1. 遍历VMA，并根据VMA的虚拟地址得到对应的struct page结构体（使用mfollow_page函数）
    // struct page *page = mfollow_page(vma, virt_addr, FOLL_GET);
    // unsigned int unused_page_mask;
    // struct page *page = mfollow_page_mask(vma, virt_addr, FOLL_GET, &unused_page_mask);
    // TODO: 2. 使用page_referenced活跃页面是否被访问，并将被访问的页面物理地址写到文件中
    // kernel v5.13.0-40及之后可尝试
    // unsigned long vm_flags;
    // int freq = mpage_referenced(page, 0, (struct mem_cgroup *)(page->memcg_data), &vm_flags);
    // kernel v5.9.0
    // unsigned long vm_flags;
    // int freq = mpage_referenced(page, 0, page->mem_cgroup, &vm_flags);
    struct mm_struct *mm = get_task_mm(my_task_info.task);
    if(mm) {
        struct vm_area_struct *vma = mm -> mmap;
        while (vma) {
            unsigned long vm_addr = vma->vm_start;
            int freq;
            for(vm_addr; vm_addr < vma->vm_end; vm_addr += page_add){
                struct page *page = mfollow_page(vma, vm_addr, FOLL_GET);
                if(!IS_ERR_OR_NULL(page)) {
                    unsigned long vm_flags;
                    freq = mpage_referenced(page, 0, (struct mem_cgroup *)(page->memcg_data), &vm_flags);
                    if(freq) {
                    record_one_data(page_to_pfn(page));
```

```
                    }
                } else {
                    continue;
                }

            }
            vma = vma -> vm_next;
        }
        flush_buf(1);
    }
}

```

- 遍历vma，对每一个vma首先获取其虚拟地址，然后通过虚拟地址使用mfollow_page函数获得每一页的struct page结构体
- 得到page结构体后，首先使用宏IS_ERR_OR_NULL判断该page是否非空且有效，在非空且有效的前提下再使用page_referenced活跃页面是否被访问，然后将被访问的页面的物理地址通过record_one_data写入文件中

- **func3**

```
// func = 3
static void traverse_page_table(struct task_struct *task)
{
    printk("func == 3, %s\n", __func__);
    struct mm_struct *mm = get_task_mm(my_task_info.task);
    if (mm)
    {
        // TODO:遍历VMA，并以PAGE_SIZE为粒度逐个遍历VMA中的虚拟地址，然后进行页表遍历
        // record_two_data(virt_addr, virt2phys(task->mm, virt_addr));
        struct vm_area_struct *vma = mm -> mmap;
        while (vma) {
            unsigned long vm_addr = vma->vm_start;
            int freq;
            for(vm_addr; vm_addr < vma->vm_end; vm_addr += page_add){
                unsigned long page_phys = virt2phys(task->mm, vm_addr);
                record_two_data(vm_addr, page_phys);
            }
            vma = vma -> vm_next;
        }
```

```
        flush_buf(1);
        mmput(mm);
    }
    else
    {
        pr_err("func: %s mm_struct is NULL\n", __func__);
    }
}
```

- 遍历vma，对每一个vma获取虚拟地址，然后遍历该vma的每一页，使用给出的record_two_data函数将页面物理号写入文件中

- **func4或func5**

```
// func == 4 或者 func == 5
static void print_seg_info(void)
{
    struct mm_struct *mm;
    unsigned long addr;
    printk("func == 4 or func == 5, %s\n", __func__);
    mm = get_task_mm(my_task_info.task);
    if (mm == NULL)
    {
        pr_err("mm_struct is NULL\n");
        return;
    }
    // TODO: 根据数据段或者代码段的起始地址和终止地址得到其中的页面，然后打印页面内容到文件中
    // 相关提示：可以使用follow_page函数得到虚拟地址对应的page，然后使用addr=kmap_atomic(page)得到可以直接
    //          访问的虚拟地址，然后就可以使用memcpy函数将数据段或代码段拷贝到全局变量buf中以写入到文件中
    //          注意：使用kmap_atomic(page)结束后还需要使用kunmap_atomic(addr)来进行释放操作
    //          正确结果：如果是运行实验提供的workload，这一部分数据段应该会打印出char *trace_data,
    //                   static char global_data[100]和char hamlet_scene1[8276]的内容。
    unsigned long start_data_addr = mm -> start_data;
    unsigned long end_data_addr = mm -> end_data;

    struct vm_area_struct *vma = mm -> mmap;

    while (vma) {
        unsigned long vm_addr = vma ->vm_start;
```

```
        for(vm_addr; vm_addr < vma -> vm_end; vm_addr += page_add)
{
            struct page *page = mfollow_page(vma, vm_addr,
FOLL_GET);
            if(!IS_ERR_OR_NULL(page)) {
                char* addr = NULL;
                addr = kmap_atomic(page);
                kunmap_atomic(addr);
                if(vm_addr >= start_data_addr)//开头在地址之前
                {
                    if(vm_addr + page_add <= end_data_addr)//结尾在
下一页地址之后
                    {
                        //输出当前页全部内容
                        memcpy(buf, addr, PAGE_SIZE);
                        curr_buf_length += PAGE_SIZE;
                        flush_buf(0);
                    }
                    else if(vm_addr <= end_data_addr)//结尾在下一页
地址前且在当前页地址后
                    {
                        //输出当前页前面一部分（有数据段的部分）
                        memcpy(buf, addr, end_data_addr -
vm_addr);
                        curr_buf_length += end_data_addr -
vm_addr;
                        flush_buf(0);
                    }
                }
                else//开头在当前页地址后
                {
                    if(vm_addr + page_add >= end_data_addr)//结尾在
下一页地址前
                    {
                        //输出当前页中间一部分（有数据段的部分）
                        memcpy(buf, addr + start_data_addr -
vm_addr, end_data_addr - start_data_addr);
                        curr_buf_length += end_data_addr -
start_data_addr;
                        flush_buf(0);
                    }
                    else if(vm_addr + page_add >=
start_data_addr)//结尾在下一页地址后且开头在下一页地址前
                    {
                        //输出当前页后面一部分（有数据段的部分）
```

```
                        memcpy(buf, addr + start_data_addr - vm_addr, vm_addr + page_add - start_data_addr);
                        curr_buf_length += vm_addr + page_add - start_data_addr;
                        flush_buf(0);
                    }
                }

            }
        }
        vma = vma -> vm_next;
    }
    mmput(mm);
}

```

- 遍历vma，并对每个vma进行页表遍历
- 当当前页整个都在数据段内部时，将整页输出到buf，修改缓冲偏移量后将buf输出
- 当当前页后面部分是数据段时（即start在addr后且在addr+page_add前，end在addr+page_add后），将当前页后面部分输出到buf，修改缓冲偏移量后将buf输出
- 当当前页前面部分是数据段时（即start在addr前，end在addr后且在addr+page_add前），将当前页前面部分输出到buf，修改缓冲偏移量后将buf输出
- 当当前页包含整个数据段（即start在addr后且end在addr+page_add前），将整个数据段输出到buf，修改缓冲偏移量后将buf输出

5. 实验结果

> PART1 动态内存分配器malloc的实现

- **find_fit_first**

```
before free: 0.999804; after free: 0.223745
time of loop 0 : 688ms
before free: 0.973545; after free: 0.216414
time of loop 1 : 921ms
before free: 0.967669; after free: 0.219073
time of loop 2 : 965ms
before free: 0.964393; after free: 0.217223
time of loop 3 : 939ms
before free: 0.96164; after free: 0.212361
time of loop 4 : 944ms
before free: 0.963092; after free: 0.215055
time of loop 5 : 966ms
before free: 0.960802; after free: 0.213814
time of loop 6 : 926ms
before free: 0.963977; after free: 0.214693
time of loop 7 : 912ms
before free: 0.958546; after free: 0.212467
time of loop 8 : 913ms
before free: 0.967839; after free: 0.21446
time of loop 9 : 947ms
before free: 0.96558; after free: 0.215257
time of loop 10 : 972ms
before free: 0.964289; after free: 0.218399
```

符合first-fit性能参考：使用workload测试，20次循环（loop），每次输出中before free的结果为 0.90~1.00之间，每次循环的时间需要小于1800ms

- **find_fit_best**

```
before free: 0.999804; after free: 0.223745
time of loop 0 : 686ms
before free: 0.999589; after free: 0.222049
time of loop 1 : 3067ms
before free: 0.999358; after free: 0.226104
time of loop 2 : 3236ms
before free: 0.996632; after free: 0.224357
time of loop 3 : 3376ms
before free: 0.993686; after free: 0.219366
time of loop 4 : 3420ms
before free: 0.995129; after free: 0.222096
time of loop 5 : 3313ms
before free: 0.992645; after free: 0.220798
time of loop 6 : 3504ms
before free: 0.995905; after free: 0.221641
time of loop 7 : 3507ms
before free: 0.9902; after free: 0.219388
time of loop 8 : 3364ms
before free: 0.999487; after free: 0.221265
time of loop 9 : 3250ms
before free: 0.997381; after free: 0.222116
time of loop 10 : 3374ms
before free: 0.995899; after free: 0.225492
```

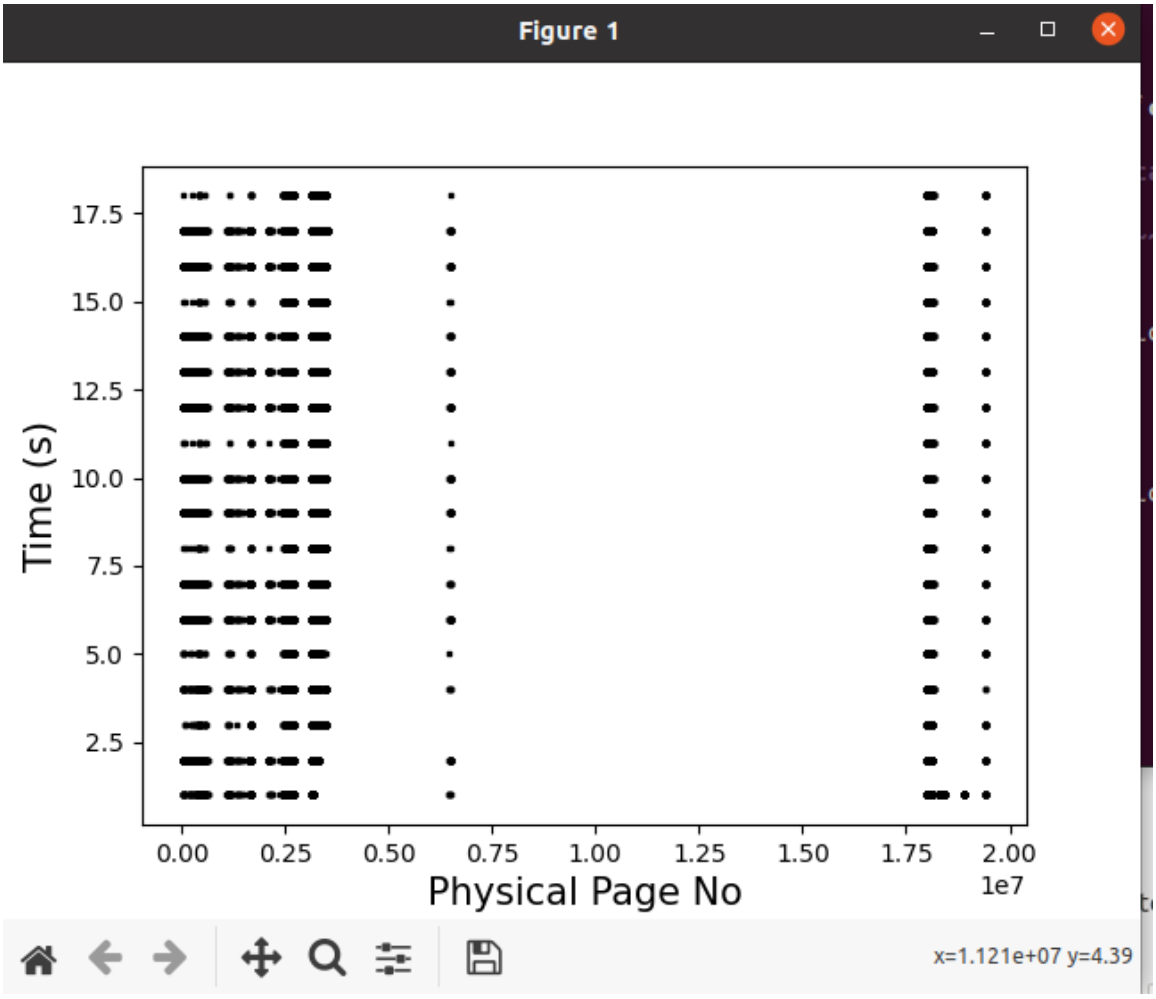
符合best-fit性能要求：使用workload测试，20次循环（loop），每次输出中before free的结果为 0.93~1.00之间，每次循环的时间需要小于4000ms

> PART2 Linux进程内存信息统计

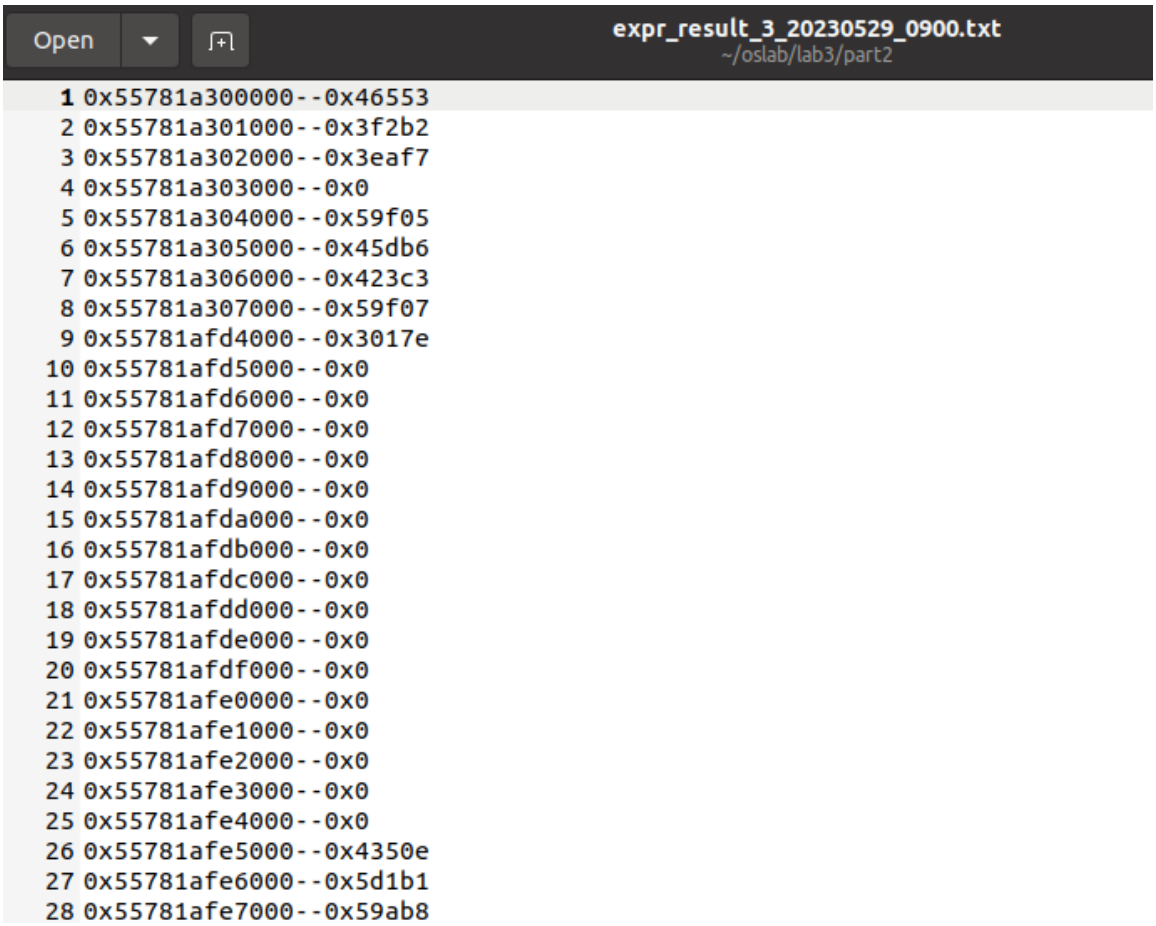
- **func1**

```
  162 | }
      | ^
5. run workload.
6. run linux module, func=1
7. rename expr_result.txt
lzcnutrition@ubuntu:~/oslab/lab3/part2$ cat /sys/kernel/mm/ktest/vma
0, 39
lzcnutrition@ubuntu:~/oslab/lab3/part2$
```

- **func2**



- **func3**



- **func4或func5**（以func4为例）

6. 实验总结

- 本次实验很好的帮助我理解了内存分配以及内存在操作系统中的实现方式，对于操作系统这门课的学习很有帮助
- 实验文档很详细，介绍清楚
- 助教很耐心，有问题能得到详细的解答