2022春操作系统实验三实验报告

PB21111681 朱炜荣

实验目标

1. **实验第一部分**
    - 以显式空闲链表的方式实现一个64位的堆内存分配器
        - 实现最先匹配和最优匹配的两种搜索算法，并比较两者的优缺点
        - 实现对堆内存的动态分配
        - 实现实时统计用户申请内存和系统申请内存的大小
    - 学会以动态链接库的形式制作库并使用
    - 体会系统在分配内存时所做的处理和后续操作之间的联系
2. **实验第二部分**
    - 了解真实Linux系统的内存管理方式，包括：
        - 了解Linux系统虚拟内存的管理方式
        - 了解什么是VMA，并遍历、统计VMA；
        - 了解Linux的多级页表机制，并自己完成一个从虚拟地址到物理地址的转换函数
        - 了解Linux的页面置换算法，统计页面冷热并输出至图表观察
        - 了解用户内存地址空间的分布，并将用户进程中的数据转储至外存。
    - 鼓励同学了解如模块的编写与使用、 sysfs的编写与使用、内核线程的使用、shell脚本的控制/循环语句的相关知识

实验环境

- OS: Ubuntu 20.04.4 LTS
- Linux内核版本：5.15.0
- 虚拟机：VirtualBox
- 虚拟机CPU：6核
- 虚拟机内存大小：4096MB

实验过程

1. **实验第一部分**
    - 实验的最开始我先通读了一遍助教给出的实验文档，大致了解了实验具体是希望我们做什么。然后下载解压好文件夹后，阅读了我们要补充大部分函数的mm.c文件，同时在阅读的过程中思考了宏定义对于传入变量的要求。
    - 然后从最开始的memlib.c开始填充函数：mem_init函数的功能是初始化用于堆空间维护的三个指针，并申请5MB的内存。而mem_sbrk是用来增长分配用户内存的函数，执行时需要判断当前剩余空间是否还够分配用户请求的空间大小。
    - 再然后是让人想要debug却不能的mm.c。

        - 首先是两种搜索算法的实现，借助着函数维护的空间链表，我们得以很方便地遍历所有空闲块。
            - find_fit_first函数的实现逻辑为：从头开始遍历链表，当我们找到第一个块大小大于等于用户请求的空间大小的块时，就停止遍历，并将对应块的指针传回。
            - find_fit_best函数则是需要我们完全地遍历一整个链表，在遍历的过程中需要不断筛选块大小与用户请求空间大小接近的块（当然前提是自身大小大于请求大小），等遍历结束后返回该块的指针。
        - 借助函数实现对堆内存的动态分配。
            - 首先是用于合并空闲块的coalesce函数。这个函数会在最开始时收集前一块和后一块的是否被分配的信息。并将情况分成不同的四类：前后都分配、前分配后空闲、前空闲后分配和前后都空闲。这些情况需要分别处理空闲块的删除、新块的加入、设置合并块的新大小，之后返回新的空闲块。
            - 再者就是用于分配新块的place函数。借助着上面实现好的搜索算法，我们找到了一块可以安置用户申请空间大小的空闲块，现在我们要做的就是把他改设成分配块。主要的情况分为两种：剩余空间能分成一个新块和无法再细分为一个新块。处理上的不同是，在我们设置完块被分配、删除原空闲块、处理前后块自己被分配的消息之后（只处理头部信息就好），能细分的新块需要再进一步处理然后加入到空闲链表中。
        - 至于实时统计用户申请内存和系统申请内存的大小。则是需要我们在几个函数的合适地方修改两个全局变量的大小。如在堆增长的时候对应的增加heap_size，在mm_free和mm_malloc中对应减少的增加user_malloc_size（要注意增减时的大小要对应的上，不然会出现负数的可能）。
    - 然后我习惯的是写一个自己的脚本从而使我的实验流程最后只需要运行该脚本就好，不过这次脚本也没几条句子，逻辑上只有make，cd，然后执行助教提供的脚本。
    - 得到结果后与检查标准做对比，发现满足检查标准遂结束该实验。
2. **实验第二部分**
    - 尝试读懂助教给出的实验文档，但是感觉各个章节之间跳跃性太强了，难度又很大就感觉有些不知从何下手。
    - 于是转换策略开始一步步的理解每个函数的想要完成的功能（实际上就是在折磨助教），同时看了几个实验文档中推荐的拓展阅读知识，大概有了了解后开始写各个函数。
    - 开始处理5个function函数的填充：
        - func1：较为简单，是为数不多看完要求就能把代码写出来的函数。我们只需依靠维护的mm->map链表来遍历所有的vma即可。
        - func2：核心代码如下：`struct mm_struct *mm = get_task_mm(my_task_info.task);`

```
    if(mm){
        struct vm_area_struct *index = mm->mmap;
        unsigned long vm_flags;
        while(index){
            unsigned long addr = index->vm_start;
            struct page *page;
            int freq;
            while(addr < index->vm_end){
                page = mfollow_page(index, addr, FOLL_GET);
                if(!IS_ERR_OR_NULL(page)){
```

```
                    freq = mpage_referenced(page, 0, (struct
mem_cgroup *)(page->memcg_data), &vm_flags);
                    if(freq > 1){
                        if(addr + PAGE_SIZE < index->vm_end){
                            record_one_data(addr);
                        }
                        else{   //特殊处理还行
                            sprintf(str_buf, "%lu\n", addr);
                            sprintf(buf + curr_buf_length, "%lu\n",
addr);
                            curr_buf_length += strlen(str_buf);
                            if ((curr_buf_length + (sizeof(unsigned
long) + 2)) >= PAGE_SIZE)
                                flush_buf(0);
                        }
                    }
                }
                addr += PAGE_SIZE;
            }
            index = index->vm_next;
        }
    }
    flush_buf(1);
```

首先借助链表进行遍历，然后再在每个vma中以PAGE_SIZE为颗粒度进行第二轮的遍历，依靠着遍历的地址获取所在的页最近是否被访问的信息，然后利用封装好的record_one_data进行写出，需要特殊判断在每个vma将要结束时不能再写入","，而是需要换行符。最后将结果再flush_buf写出。

- func3:核心代码如下：

```
struct vm_area_struct *index = mm->mmap;
        while(index){
            unsigned long addr = index->vm_start;
            while(addr <= index->vm_end){
                record_two_data(addr, virt2phys(task->mm, addr));
                addr += PAGE_SIZE;
            }
            index = index->vm_next;
        }
        mmput(mm);
```

其中我们还需要补充一个从pgd到page地址的函数：

```
pgd_t *pgd = pgd_offset(mm,virt);
    pud_t *pud = pud_offset((p4d_t)pgd,virt);      //不是很懂，这里编译时不加会报错
    pmd_t *pmd = pmd_offset(pud,virt);
    pte_t *pte = pte_offset_kernel(pmd,virt);
    struct page *page = pte_page(pte);
```

依旧是遍历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，在遍历的过程中，我们不断地去记录地址和物理页号。

- func4&5:核心代码如下：

```
struct vm_area_struct *index,before;
    unsigned long start,end;
    unsigned long v_addr;
    if(ktest_func == 4){
        start = mm->start_data;
        end = mm->end_data;
    }
    else{
        start = mm->start_code;
        end = mm->end_code;
    }
    unsigned long temp = start;
    while(temp < end){
        index = find_vma(mm, temp);
        if (index == before){
            temp += PAGE_SIZE;
            continue;
        }
        addr = index->vm_start;
        while(addr < index->vm_end - PAGE_SIZE){
            struct page *page = mfollow_page(index,addr,FOLL_GET);
            if(!IS_ERR_OR_NULL(page)){
                v_addr = kmap(page);
                kunmap_atomic(v_addr);

                if(addr <= end && (addr + PAGE_SIZE) >= start){
                    unsigned long from = (addr > start)?v_addr:v_addr + (start - addr);     //用于对齐页框到地址
                    unsigned long offset = ((addr+PAGE_SIZE > end)?end:(addr+PAGE_SIZE)) - ((addr > start)?addr:start);
```

```
                    memcpy(buf,from,offset);
                    curr_buf_length += offset;
                    flush_buf(0);
                }
            }
            addr += PAGE_SIZE;
        }
        temp += PAGE_SIZE;
        before = index;
    }
    mmput(mm);
```

由于其难度确实很大，这里只尽自己努力阐述自己的想法。首先func4和func5函数的处理逻辑是一致的，只需要在开始判断一下func的大小，然后给定义的start和end赋值。之后与其它func类似，开始遍历所有地址，找该地址对应的VMA，再找到页面描述符，之后根据代码注释将对应的数据写入到文件中。（经过舍友提醒，发现最开始会输出两遍的结果，于是开始思考为什么，得出的结论是两个地址可能对应到了同一个vma中，于是才将结果输出了两遍，遂在debug时追加判断了两次vma是否相同，结果便变得正常了）

实验结果

1. **实验第一部分**
    - 首先是find_fit_first

```
before free: 0.971754; after free: 0.21819
time of loop 0 : 415ms
before free: 0.953877; after free: 0.21264
time of loop 1 : 564ms
before free: 0.948942; after free: 0.215493
time of loop 2 : 583ms
before free: 0.945729; after free: 0.213606
time of loop 3 : 606ms
before free: 0.942717; after free: 0.20887
time of loop 4 : 576ms
before free: 0.944327; after free: 0.211589
time of loop 5 : 577ms
before free: 0.941958; after free: 0.21027
time of loop 6 : 562ms
before free: 0.94511; after free: 0.21117
time of loop 7 : 545ms
before free: 0.939438; after free: 0.208905
time of loop 8 : 561ms
before free: 0.948793; after free: 0.210799
time of loop 9 : 531ms
before free: 0.946898; after free: 0.211666
time of loop 10 : 546ms
before free: 0.945307; after free: 0.214865
time of loop 11 : 545ms
before free: 0.948323; after free: 0.213318
time of loop 12 : 561ms
before free: 0.94815; after free: 0.210896
time of loop 13 : 558ms
before free: 0.947042; after free: 0.211872
time of loop 14 : 558ms
before free: 0.942315; after free: 0.211329
time of loop 15 : 538ms
before free: 0.938851; after free: 0.210143
time of loop 16 : 548ms
before free: 0.93879; after free: 0.210873
time of loop 17 : 531ms
before free: 0.939189; after free: 0.209751
time of loop 18 : 529ms
before free: 0.942652; after free: 0.211805
time of loop 19 : 536ms
```

我们不难发现首次匹配所需要的平均用时很短，但是对应的空间利用率就不是很乐观了。是一种速度快但是笨重的结果

- 然后是find_fit_best

```
before free: 0.971754; after free: 0.21819
time of loop 0 : 391ms
before free: 0.972011; after free: 0.21671
time of loop 1 : 1683ms
before free: 0.970978; after free: 0.220482
time of loop 2 : 1740ms
before free: 0.967858; after free: 0.218587
time of loop 3 : 1779ms
before free: 0.964811; after free: 0.213779
time of loop 4 : 1794ms
before free: 0.966365; after free: 0.216486
time of loop 5 : 1723ms
before free: 0.963814; after free: 0.215146
time of loop 6 : 1757ms
before free: 0.967057; after free: 0.216016
time of loop 7 : 1745ms
before free: 0.961424; after free: 0.213795
time of loop 8 : 1782ms
before free: 0.971064; after free: 0.215637
time of loop 9 : 1748ms
before free: 0.968794; after free: 0.216595
time of loop 10 : 1740ms
before free: 0.967507; after free: 0.219914
time of loop 11 : 1750ms
before free: 0.97048; after free: 0.218296
time of loop 12 : 1721ms
before free: 0.970281; after free: 0.215841
time of loop 13 : 1736ms
before free: 0.972174; after free: 0.217413
time of loop 14 : 1682ms
before free: 0.967305; after free: 0.216913
time of loop 15 : 1715ms
before free: 0.963665; after free: 0.21568
time of loop 16 : 1691ms
before free: 0.963823; after free: 0.216463
time of loop 17 : 1694ms
before free: 0.964156; after free: 0.215264
time of loop 18 : 1788ms
before free: 0.967491; after free: 0.217325
time of loop 19 : 1684ms
```

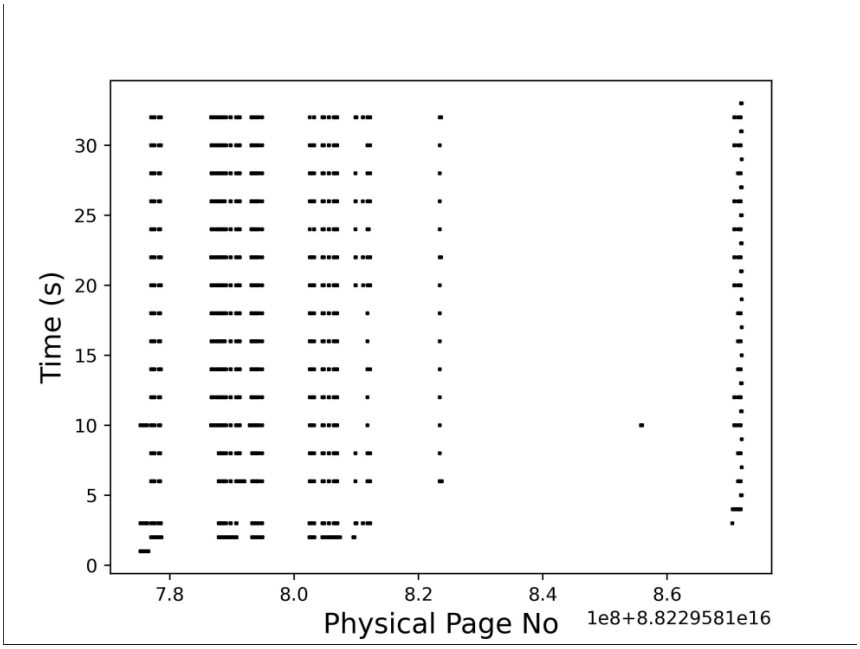
相较于首次匹配，最优匹配的优势在于空间利用率很高，但是因为需要遍历整个空闲链表，所以需要的耗时很大，花费的时间更多，是一个速度慢但是“灵活”的算法。

2. **实验第二部分**
    - func1:

```
root@Whale: /home/auv/桌面/LAB_OS/lab3/lab3.2
3. load linux module.
4. compile workload.
workload.cc:27:20: warning: ISO C++ forbids converting a string constant to 'cha
r*' [-Wwrite-strings]
   27 | char *trace_data = "The early bird catches the worm. The early bird catc
hes the worm. The early bird catches the worm.";
      |                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
workload.cc: In function 'int workload_read(workload_base*)':
workload.cc:128:1: warning: no return statement in function returning non-void [
-Wreturn-type]
  128 | }
      | ^
workload.cc: In function 'void* workload_run(void*)':
workload.cc:162:1: warning: no return statement in function returning non-void [
-Wreturn-type]
  162 | }
      | ^
5. run workload.
6. run linux module, func=1
7. rename expr_result.txt
root@Whale:/home/auv/桌面/LAB_OS/lab3/lab3.2# cat /sys/kernel/mm/ktest/vma
0, 39
root@Whale:/home/auv/桌面/LAB_OS/lab3/lab3.2#
```

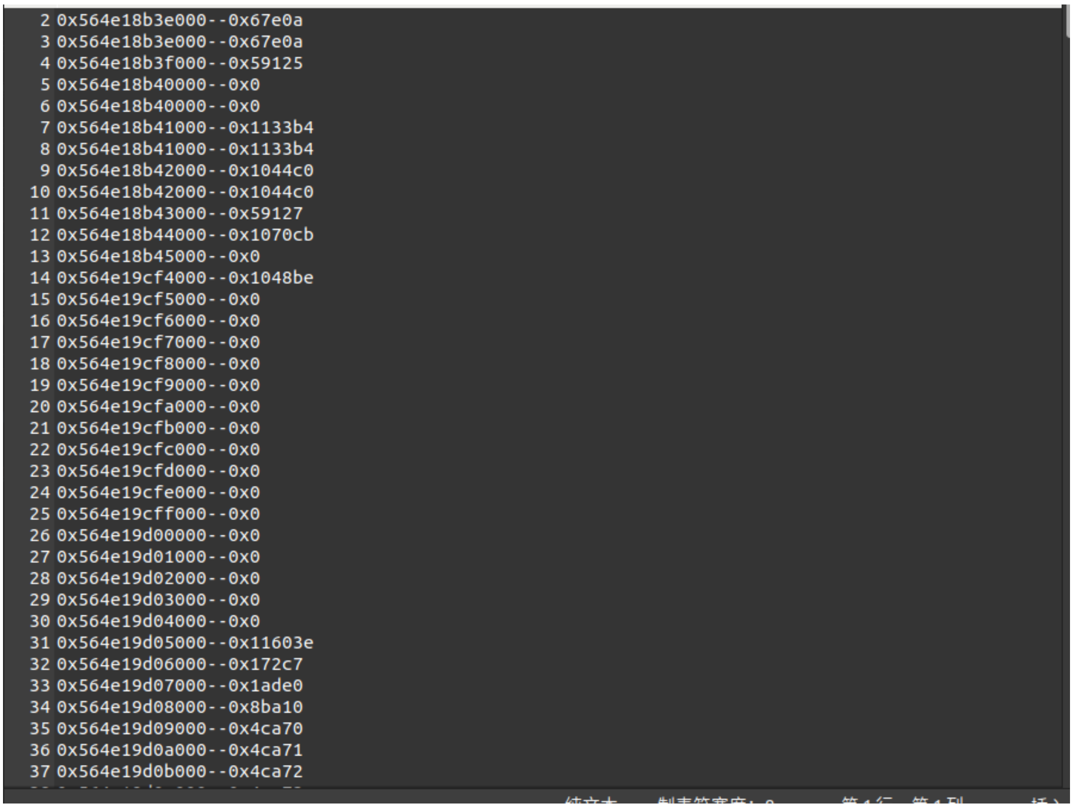
程序成功输出了vma的数量：39个

- func2：



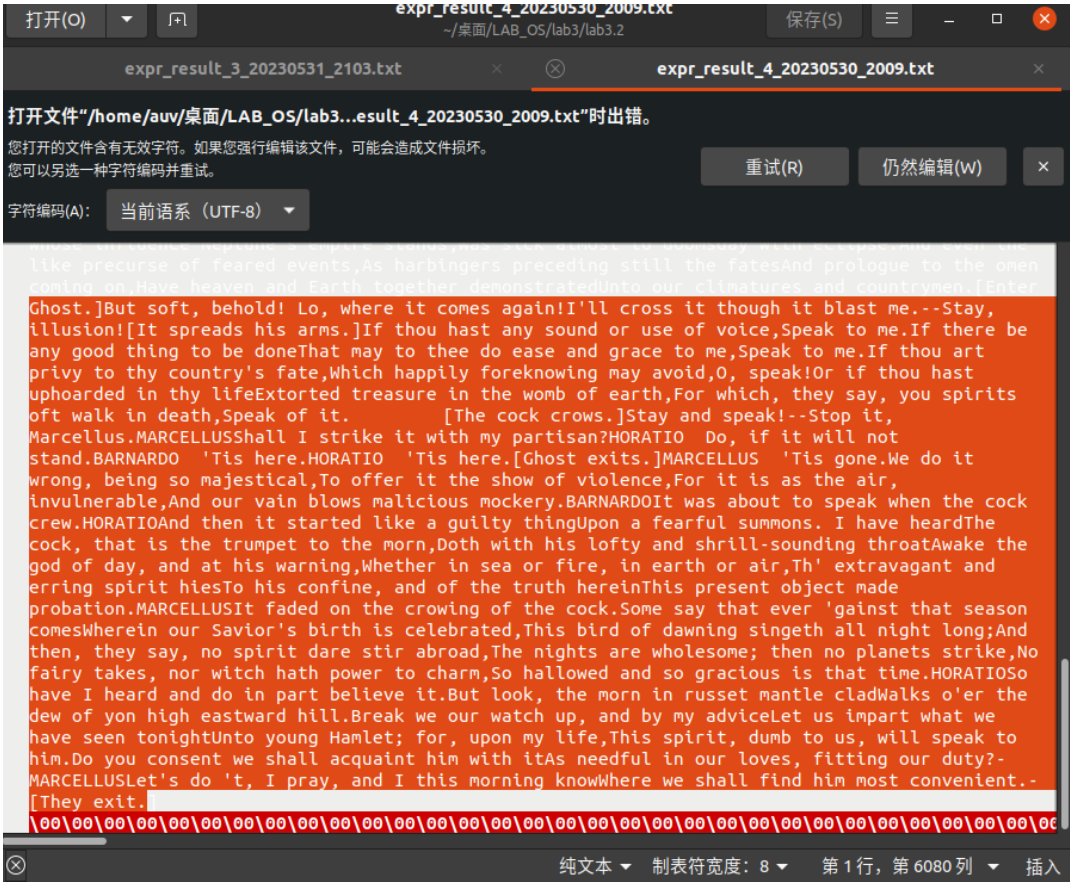
能够看出页面冷热信息分布，较为遵循我心目中的二八定律——即20%的数据被大概80%的概率被访问，而剩下的80%就很少被访问。

- func3:



较为成功的是，程序遍历测试程序的页表，并打印了地址和其对应页面物理号，由于篇幅原因不做全部展示。

- func4&5:



这里为了防止大家看不清，我先选中了输出的部分章节，然后才截图保存，并不是输出了一堆的乱码。

总结

1. 本次实验较高的难度进一步锻炼了我阅读框架和编写C语言代码的能力
2. 较难debug的part1某种程度上也要求了自己去做到逻辑严谨
3. 有惊无险的写完该实验加深了我在课上学到的内存管理相关的内容，比如虚拟地址、页表等
4. 对Linux操作系统的美妙有有了更深层次的理解，尤其是相较于Windows系统在内存分配，文件权限修改方面的提升
5. 理解了为什么需要内核空间来保护电脑，以及允许小白访问内核重要文件的危险程度。

吐槽和建议：助教们能否对第二部分的难度进行一定的修改，以及实验文档目前存在的指向不明也有待提高，实验本身给的时间是很充裕的，但是无奈遇到了机组的流水线CPU实验，大部分同学都会有写不完的感觉，更不用说尝试附加实验了。以及实验一中部分代码与真正框架中不符，很影响阅读者的体验（