OS lab 3

PB21081601 张芷苒

1 实验目的

- 动态内存分配器 malloc。使用显式空闲链表实现一个64位堆内存分配器。
  - 实现两种基本搜索算法
  - 实现堆内存的动态扩容
  - 实现实时的分配器内存使用情况统计
  - 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

- Linux 进程内存信息统计。目的是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。需要了解的内容有：
  - 了解Linux系统虚拟内存的管理方式（ch7_part2.pdf, P3~17）；
  - 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（ch7_part2.pdf, P18~46）；
  - 了解Linux的页面置换算法，统计页面冷热并输出至图表观察（ch7_part2.pdf, P82~86）；
  - 了解用户内存地址空间的分布（ch7_part1.pdf, P4~23），并将用户进程中的数据转储至外存 。

2 实验环境

- 动态内存分配器 malloc
  - OS: Ubuntu 20.04.4 LTS
  - 无需在QEMU下调试

- Linux 进程内存信息统计
  - OS: Ubuntu 20.06 LTS

- Linux内核版本: 不限（但建议5以上）

3 实验要求

3.1 动态内存分配器 malloc

补全 mm.c 和 memlib.c 文件中的部分函数。待补全的函数如下：

- `void mem_init(void)`
  - 调用 sbrk ， 初始化 mem_start_brk 、 mem_brk 、以及 mem_max_addr 。
  - 此处初始增长空间大小为 MAX_HEAP 。
- `void *mem_sbrk(int incr)`
  - 模拟堆空间的增长。
  - 参数 incr 表示上层函数请求的空间大小。
  - 返回新分配空间的基址。
- `static void *find_fit_first(size_t asize)`
  - 针对某个内存分配请求，该函数在空闲链表中执行首次适配搜索。
  - 参数 asize 表示请求块的大小。 返回值为满足要求的空闲块的地址。
  - 若返回值为 NULL ，表示当前堆块中没有满足要求的空闲块。
- `static void *find_fit_best(size_t asize)`
  - 针对某个内存分配请求，该函数在空闲链表中执行首次适配搜索。
  - 参数 asize 表示请求块的大小。 返回值为满足要求的空闲块的地址。
  - 若返回值为 NULL ，表示当前堆块中没有满足要求的空闲块。
- `static void place(void* bp, size_t asize)`
  - 在空闲块中分配一个块给用户。
  - 参数 bp 表示以定位的空闲块指针， asize 为待分配块的总空间要求。
  - 对于分配后的剩余空闲空间，需要转换为新的空闲块。
- `static void* coalesce(void* bp)`
  - 释放空闲块时，判断相邻块是否空闲，合并空闲块。

  - 根据相邻块的的分配状态，共有四种不同情况待补充，具体参见合并步骤这一节。

- 为内存分配器添加实时统计内存使用情况和分配时间的功能：
  - user_malloc_size —— 用户需要的内存量。
  - heap_size —— 内存分配器实际使用的内存量。
  - double get_utilization() ——返回 user_malloc_size 除以 heap_size 的值。

3.2 Linux 进程内存信息统计

完成以下函数：

- func = 1：每隔5s统计测试程序的VMA数量
- func = 2：每隔5s统计测试程序的页面冷热信息
- func = 3：遍历测试程序的页表，并打印所有页面物理号
- func = 4：dump 测试程序的数据段
- func = 5：dump 测试程序的代码段

4 实验过程

4.1 动态内存分配器 malloc

4.1.1 内存分配器与内核的交互模块

本部分主要要补充两个功能函数，即初始化堆1和分配堆2的两个函数：

1. 堆1的初始化 `mem_init()`
   调用 sbrk() 获取5MB内存块，并记录对应的 mem_start_brk 、 mem_max_addr 和 mem_brk 。

```
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_start_brk = sbrk(MAX_HEAP); // 通过系统调用 sbrk 请求分配 MAX_HEAP 大小的堆空间
    if(mem_start_brk == (void * )-1){ //
        exit(-1);
```

```
    }
    mem_brk = mem_start_brk;
    mem_max_addr = mem_start_brk + MAX_HEAP;
}
```

2. 堆2的分配 `*mem_sbrk()`

   首先要记录旧的 mem_brk ，作为分配首地址返回。

   然后分两种情况：如果加上incr之后 mem_brk 的值超过了当前的 mem_max_addr ，则证明堆1不够用了，需要再次调用 sbrk() 来增加堆1的大小，并修改相关变量，然后增加 mem_brk 。

   如果没有超过，就直接给 mem_brk 增加一个incr，增加堆2，即分配堆2。

   最后返回旧的 mem_brk 。

```
/*
 * mem_sbrk - simple model of the sbrk function. Extends the heap
 *    by incr bytes and returns the start address of the new area. In
 *    this model, the heap cannot be shrunk.
 */
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
    */
    mem_brk += incr;
    if (mem_brk > mem_max_addr) {
        if (sbrk(MAX_HEAP) == (void *)-1) {
            return (void *)-1;
        }
        mem_max_addr += MAX_HEAP;
    }
    return (void *)old_brk;
}
```

4.1.2 内存分配器空闲块管理

4.1.2.1 分配器内存使用率统计

这段代码实现了一个简单的分配器内存使用率统计功能，并在`mm_init()`函数中更新了相关变量。

代码解析：

1. 在全局作用域下声明了两个变量`user_malloc_size`和`heap_size`，用于记录用户申请的内存量和分配器占用的内存量。
2. 在`get_utilization()`函数中，计算使用率的公式为`user_malloc_size / heap_size`。将`user_malloc_size`转换为`double`类型进行除法运算，以得到精确的使用率。
3. 在`mm_init()`函数中，对变量进行初始化。`free_listp`被设置为`NULL`。
4. 使用`mem_sbrk()`函数从操作系统中请求一块内存，大小为4个字的倍数（4 * WSIZE）。如果请求失败，返回-1。
5. 设置初始化的堆结构。将堆的起始位置（`heap_listp`）与块的头部和尾部进行相应的设置。具体细节可参考实验文档。注意，这里假设了`extend_heap()`函数用于扩展堆空间。
6. 如果堆的初始化扩展失败，则返回-1。
7. 更新`heap_size`变量，增加了4个字的大小（用于堆结构的初始化）和扩展的堆空间的大小（CHUNKSIZE）。
8. 最后，返回0表示初始化成功。

```
/*
    TODO:
        完成一个简单的分配器内存使用率统计
        user_malloc_size: 用户申请内存量
        heap_size: 分配器占用内存量
    HINTS:
        1. 在适当的地方修改上述两个变量，细节参考实验文档
        2. 在 get_utilization() 中计算使用率并返回
*/
size_t user_malloc_size, heap_size;

double get_utilization() {
    return (double) ((user_malloc_size * 1.0) / heap_size);
}
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    free_listp = NULL;
```

```
21
22      if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1)
23          return -1;
24
25      PUT(heap_listp, 0);
26      PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1, 1));
27      PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1, 1));
28      PUT(heap_listp + (3 * WSIZE), PACK(0, 1, 1));
29      heap_listp += (2 * WSIZE);
30
31      if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
32          return -1;
33      /* mm_check(__FUNCTION__);*/
34      heap_size += 4 * WSIZE + CHUNKSIZE;
35      return 0;
36  }
```

4.1.2.2 合并操作

在 coalesce() 函数中实现，释放当前内存块时，根据相邻块的的分配状态，有如下四种不同情况：

1. 情况1：前面的块和后面的块都已分配；
2. 情况2：前面的块已分配，后面的块空闲；
3. 情况3：前面的块空闲，后面的块已分配；
4. 情况4：前后块都空闲。

合并的目的是将空闲块与相邻的块合并成更大的空闲块，以减少内存分配系统中的外部碎片。

1. 如果前一个块和后一个块都已分配：

```
1  if (prev_alloc && next_alloc)
2  {
3      add_to_free_list(bp);
4  }
```

在这种情况下，当前块 `bp` 不与任何相邻块合并，因为前一个和后一个块都已分配。该块只被添加到空闲块列表中。

2. 如果前一个块已分配而后一个块为空闲：

```
else if (prev_alloc && !next_alloc)
{
    delete_from_free_list(NEXT_BLKP(bp));
    PUT(HDRP(bp), PACK(size + GET_SIZE(HDRP(NEXT_BLKP(bp))), 1, 0));
    PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1, 0));
    add_to_free_list(bp);
}
```

在这种情况下，后一个块是空闲的，因此将 `bp` 与后一个块合并。更新合并块的大小并更新头部和尾部。然后将合并后的块添加到空闲块列表中。

3. 如果前一个块为空闲而后一个块已分配：

```
else if (!prev_alloc && next_alloc)
{
    delete_from_free_list(PREV_BLKP(bp));
    bp = PREV_BLKP(bp);
    PUT(HDRP(bp), PACK(GET_SIZE(HDRP(bp)) +
GET_SIZE(HDRP(NEXT_BLKP(bp))), 1, 0));
    PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1, 0));
    add_to_free_list(bp);
}
```

在这种情况下，前一个块是空闲的，因此将 `bp` 与前一个块合并。更新合并块的大小并更新头部和尾部，然后将 `bp` 指针移动到指向新合并的块。最后将合并后的块添加到空闲块列表中。

4. 如果前一个块和后一个块都为空闲：

```
else
{
    delete_from_free_list(NEXT_BLKP(bp));
    PUT(HDRP(bp), PACK(size + GET_SIZE(HDRP(NEXT_BLKP(bp))), 0, 0));
    PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 0, 0));
    delete_from_free_list(PREV_BLKP(bp));
    bp = PREV_BLKP(bp);
    PUT(HDRP(bp), PACK(GET_SIZE(HDRP(bp)) +
GET_SIZE(HDRP(NEXT_BLKP(bp))), 1, 0));
    PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1, 0));
    add_to_free_list(bp);
}
```

在这种情况下，前一个块和后一个块都为空闲，因此将 `bp` 与这两个块都合并。首先，

合并后一个块，更新头部和尾部的大小。然后，合并前一个块，将 `bp` 指针更新为指向合并后的块，并更新头部和尾部的大小。最后，将合并后的块添加到空闲块列表中。

在执行适当的合并操作后，函数会返回指向合并后的块（或原始块，如果没有进行合并）的 `bp` 指针。

完整代码：

```
static void *coalesce(void *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*/
    {
        delete_from_free_list(NEXT_BLKP(bp));
        PUT(HDRP(bp), PACK(size + GET_SIZE(HDRP(NEXT_BLKP(bp))), 1, 0));
        PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1, 0));
        add_to_free_list(bp);
    }
    else if (!prev_alloc && next_alloc) /*前块空闲，后块已分配*/
    {
        delete_from_free_list(PREV_BLKP(bp));
        bp = PREV_BLKP(bp);
        PUT(HDRP(bp), PACK(GET_SIZE(HDRP(bp)) + GET_SIZE(HDRP(NEXT_BLKP(bp))), 1, 0));
        PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1, 0));
        add_to_free_list(bp);
    }
    else /*前后都是空闲块*/
    {
        delete_from_free_list(NEXT_BLKP(bp));
        PUT(HDRP(bp), PACK(size + GET_SIZE(HDRP(NEXT_BLKP(bp))), 0, 0));
        PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 0, 0));
        delete_from_free_list(PREV_BLKP(bp));
        bp = PREV_BLKP(bp);
        PUT(HDRP(bp), PACK(GET_SIZE(HDRP(bp)) + GET_SIZE(HDRP(NEXT_BLKP(bp))), 1, 0));
        PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1, 0));
        add_to_free_list(bp);
    }
    return bp;
```

```
}
```

4.1.2.3 查找第一个适合大小的空闲块

这段代码实现了首次匹配算法的内存分配函数`find_fit_first()`。该函数的作用是在空闲块链表中查找第一个适合大小的空闲块，并返回该空闲块的指针。

代码解析：

1. 函数首先声明一个指针变量`p`，并将其初始化为指向空闲块链表的起始位置（`free_listp`）。
2. 使用一个循环来遍历空闲块链表。循环会一直执行，直到指针`p`指向空（即链表的末尾）。
3. 在每次循环迭代中，函数检查指针`p`指向的空闲块的大小是否大于等于所需的大小`asize`。这里使用`GET_SIZE(HDRP(p))`来获取空闲块头部的大小，并与`asize`进行比较。
4. 如果找到一个大小合适的空闲块，函数立即返回该空闲块的指针`p`。
5. 如果当前空闲块不满足大小要求，则函数通过`(char *)GET_SUCC(p)`将指针`p`更新为下一个空闲块的地址。`GET_SUCC`是一个假设的宏，用于从空闲块头部中获取下一个空闲块的指针。
6. 如果遍历完整个空闲块链表仍未找到合适的空闲块，则函数返回`NULL`，表示没有可用的空闲块。

```c
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;
    while (p ≠ NULL) {
        if (GET_SIZE(HDRP(p)) ≥ asize) {
            return p;
        }
        p = (char *)GET_SUCC(p);
    }
    return NULL; // 换成实际返回值
}
```

4.1.2.4 查找最合适大小的空闲块

这段代码实现了最佳适配算法的内存分配函数`find_fit_best()`。该函数的作用是在空闲块链表中查找最合适大小的空闲块，并返回该空闲块的指针。

代码解析：

1. 函数首先声明一个指针变量`p`，并将其初始化为指向空闲块链表的起始位置（`free_listp`）。
2. 声明一个指针变量`best`，用于记录目前找到的最合适的空闲块。初始时将其设为`NULL`。
3. 使用一个循环来遍历空闲块链表。循环会一直执行，直到指针`p`指向空（即链表的末尾）。
4. 在每次循环迭代中，函数首先检查指针`p`指向的空闲块的大小是否大于等于所需的大小`asize`。这里使用`GET_SIZE(HDRP(p))`来获取空闲块头部的大小，并与`asize`进行比较。
5. 如果找到一个大小合适的空闲块，函数进一步判断是否是目前找到的最合适的空闲块。这里通过比较当前空闲块的大小与之前记录的最合适空闲块的大小来判断。如果`best`为`NULL`（表示目前还未找到合适的空闲块），或者当前空闲块的大小更小于之前记录的最合适空闲块的大小，就更新`best`为当前空闲块。
6. 如果当前空闲块不满足大小要求，函数通过`(char *)GET_SUCC(p)`将指针`p`更新为下一个空闲块的地址。`GET_SUCC`是一个假设的宏，用于从空闲块头部中获取下一个空闲块的指针。
7. 完成遍历整个空闲块链表后，函数返回记录的最合适空闲块的指针`best`。如果链表中没有合适的空闲块，则`best`的值为`NULL`。

```
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 *best = NULL;
    while (p ≠ NULL) {
        if (GET_SIZE(HDRP(p)) ≥ asize) {
            if (best == NULL || GET_SIZE(HDRP(p)) <
GET_SIZE(HDRP(best))) {
                best = p;
            }
        }
        p = (char*)GET_SUCC(p);
    }
    return best; // 换成实际返回值
}
```

4.1.2.5 分配一个块给用户

内存分配函数`place()`的部分实现，该函数负责将一个空闲块转变为已分配的块。函数接受一个指向空闲块的指针(`bp`)和请求分配的大小(`asize`)作为参数。

代码解析：

1. 调用`delete_from_free_list(bp)`函数将空闲块从空闲块链表中删除。这意味着该空闲块之前可能是一个分离的空闲块链表的一部分，但在给定的代码中没有展示出来。
2. 使用`GET_SIZE(HDRP(bp))`获取当前空闲块的大小。`GET_SIZE`假设是一个宏，用于从块头中提取大小字段。
3. 如果分配`asize`后剩余的空间小于最小块大小(`MIN_BLK_SIZE`)，则整个空闲块都被分配出去。在这种情况下，块的头部被更新为新的大小和分配状态（`1`表示已分配），并且`user_malloc_size`增加了原始块大小与分配块大小的差值（`size - WSIZE`）。注意，`WSIZE`是一个字（通常是4或8个字节）的大小，而`user_malloc_size`变量被假设用于跟踪用户分配的总大小。
4. 如果剩余空间足够形成一个新的空闲块，则将当前空闲块分割为一个已分配块和一个新的空闲块。已分配块的头部被更新为请求的大小和分配状态（`1`），而新空闲块的头部和尾部被更新为剩余大小和分配状态（头部为`1`，尾部为`0`）。调用`add_to_free_list()`函数将新的空闲块添加到适当的空闲块链表中。同样，`user_malloc_size`相应地进行更新。
5. 最后，使用`PUT`宏将下一个块的先前分配状态更新到其头部中，通过将先前分配位设置为`1`。`PUT_PREV_ALLOC`宏被假设将先前分配位打包到头部值中。

```
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
    */
    delete_from_free_list(bp);
    size_t size = GET_SIZE(HDRP(bp));
    if (size - asize < MIN_BLK_SIZE) {
        PUT(HDRP(bp), PACK(size, 1, 1));
        user_malloc_size += size - WSIZE;
```

```
    } else {
        PUT(HDRP(bp), PACK(asize, 1, 1));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(size - asize, 1, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size - asize, 1, 0));
        add_to_free_list(NEXT_BLKP(bp));
        user_malloc_size += asize - WSIZE;
    }
    PUT(HDRP(NEXT_BLKP(bp)),
PACK_PREV_ALLOC(GET(HDRP(NEXT_BLKP(bp))),1));
}
```

4.2 Linux 进程内存信息统计

4.2.1 func = 1：每隔5s统计测试程序的vma数量

功能：遍历给定进程的VMA（虚拟内存区域），并将VMA的个数记录到`my_task_info`结构体的`vma_cnt`变量中。

首先，通过调用`get_task_mm()`函数获取指定进程的`mm_struct`结构体，即进程的内存描述符。如果成功获取到了`mm_struct`结构体，就执行下面的操作。

接着，使用一个指针`vm_area`指向进程的`mmap`字段，该字段指向进程的第一个VMA的结构体。然后，通过一个循环遍历的方式，不断访问下一个VMA，直到`vm_area`指针为NULL为止。在每次循环迭代中，`vma_cnt`变量会递增，表示遍历到了一个VMA。

最后，使用`mmput()`函数释放之前获取的`mm_struct`结构体的引用计数。

```
// func == 1
static void scan_vma(void)
{
    struct mm_struct *mm;
    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
        struct vm_area_struct *vm_area = mm->mmap;
        while (vm_area != NULL) {
            my_task_info.vma_cnt++;
            vm_area = vm_area->vm_next;
        }
        mmput(mm);
    }
}
```

4.2.2 func = 2：每隔5s统计测试程序的页面冷热信息

功能：打印进程的活跃页面信息，并将被访问的页面的物理地址写入文件。

首先，通过调用`get_task_mm()`函数获取指定进程的`mm_struct`结构体，即进程的内存描述符。如果成功获取到了`mm_struct`结构体，就执行下面的操作。

然后，使用一个指针`vm_area`指向进程的第一个VMA的结构体，并通过一个循环遍历的方式，依次访问每个VMA。

在每个VMA的循环中，使用`vm_area->vm_flags`打印VMA标志信息。接着，使用一个内部循环来遍历VMA的每个虚拟地址。在内部循环中，代码调用`mfollow_page()`函数，根据VMA的虚拟地址获取对应的`struct page`结构体。然后，通过`mpage_referenced()`函数检查页面是否被访问，并获取页面的物理地址。

如果页面被访问，则将其物理地址转换为字符串，并将其写入文件。代码中使用了一个字符串缓冲区`str_buf`来存储物理地址的字符串表示。通过`sprintf()`函数将每个页面的物理地址追加到`str_buf`中，然后调用`write_to_file()`函数将`str_buf`写入文件。

最后，通过`mmput()`函数释放之前获取的`mm_struct`结构体的引用计数。

```
// func == 2
static void print_mm_active_info(void)
{
    int first = 1;
    struct mm_struct *mm;
    unsigned long virt_addr;
    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);
    mm = get_task_mm(my_task_info.task);
    if (mm) {
        struct vm_area_struct *vm_area = mm->mmap;
        while (vm_area) {
```

```
            pr_info("%lx", vm_area->vm_flags);
            virt_addr = vm_area->vm_start;
            while (virt_addr != vm_area->vm_end) {
                struct page *page = mfollow_page(vm_area, virt_addr,
FOLL_GET | FOLL_TOUCH);
                if (!IS_ERR_OR_NULL(page)) {
                    unsigned long vm_flags;
                    int freq = mpage_referenced(page, 0, (struct
mem_cgroup *)(page->memcg_data), &vm_flags);
                    if (freq != 0) {
                        if (first) {
                            sprintf(str_buf, "%lu",
page_to_pfn(page));
                            first = 0;
                        } else {
                            sprintf(str_buf, ",%lu",
page_to_pfn(page));
                        }
                        write_to_file(str_buf, strlen(str_buf));
                    }
                }
                virt_addr += PAGE_SIZE;
            }
            vm_area = vm_area->vm_next;
        }
        sprintf(str_buf, "\n");
        write_to_file(str_buf, strlen(str_buf));
        mmput(mm);
    }
}
```

4.2.3 func = 3：遍历测试程序的页表，并打印所有页面物理号

功能：遍历给定进程的VMA，并以页表粒度逐个遍历每个VMA中的虚拟地址，然后进行页表遍历，并将虚拟地址、页帧号和物理地址的信息记录到文件中。

首先，通过调用`get_task_mm()`函数获取指定进程的`mm_struct`结构体，即进程的内存描述符。如果成功获取到了`mm_struct`结构体，就执行下面的操作。

然后，使用一个指针`vm_area`指向进程的第一个VMA的结构体，并通过一个循环遍历的方式，依次访问每个VMA。

在每个VMA的循环中，使用`vm_area->vm_start`作为起始地址，以`PAGE_SIZE`为步长递增虚拟地址，直到达到VMA的结束地址`vm_area->vm_end`为止。在每次循环迭代中，调用`mfollow_page()`函数，根据VMA的虚拟地址获取对应的`struct page`结构体。

如果成功获取到`page`结构体，则通过调用`virt2phys()`函数将虚拟地址转换为物理地址，并将虚拟地址、页帧号和物理地址的信息格式化为字符串，存储在`str_buf`中。最后，通过调用`write_to_file()`函数将`str_buf`写入文件。

最后，通过`mmput()`函数释放之前获取的`mm_struct`结构体的引用计数。

```
// func = 3
static void traverse_page_table(struct task_struct *task)
{
    struct mm_struct *mm;
    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 *vm_area = mm->mmap;
        while (vm_area) {
            unsigned long virt_addr = vm_area->vm_start;
            while (virt_addr != vm_area->vm_end) {
                struct page *page = mfollow_page(vm_area, virt_addr, FOLL_GET);
                if (!IS_ERR_OR_NULL(page)) {
                    sprintf(str_buf, "0x%lx--0x%lx--0x%lx\n", virt_addr, page_to_pfn(page),virt2phys(mm,virt_addr));
                    write_to_file(str_buf, strlen(str_buf));
                }
                virt_addr += PAGE_SIZE;
            }
            vm_area = vm_area->vm_next;
        }
        mmput(mm);
    }
    else
    {
        pr_err("func: %s mm_struct is NULL\n", __func__);
    }
}
```

```
static unsigned long virt2phys(struct mm_struct *mm, unsigned long virt)
{
    struct page *page = NULL;
    // TODO : 多级页表遍历: pgd->pud->pmd->pte，然后从pte到page，最后得到pfn
    pgd_t *pgd = pgd_offset(mm, virt); // 1级页表
    p4d_t *p4d = p4d_offset(pgd, virt); // 4级页表
```

```
    pud_t *pud = pud_offset(p4d, virt); // 3级页表
    pmd_t *pmd = pmd_offset(pud, virt); // 2级页表
    pte_t *pte = pte_offset_kernel(pmd, virt);

    if (pte_present(*pte)) // 判断pte是否存在
    {
        page = pte_page(*pte); // 获取pte对应的page
        if (page)
        {
            return page_to_phys(page); // 获取page对应的物理地址
        }
    }

    pr_err("func: %s page is NULL\n", __func__);
    return 0;
}
```

4.2.4 func = 4: dump测试程序的数据段

4.2.5 func = 5: dump测试程序的代码段

`print_seg_info` 函数用于根据数据段或代码段的起始地址和终止地址获取其中的页面，并将页面的内容打印到文件中。

- 首先，获取当前任务的 `mm_struct` 结构指针 `mm`，如果获取失败，则打印错误信息并返回。
- 根据 `ktest_func` 的值，选择数据段或代码段的起始地址和终止地址，将其分别存储在 `start` 和 `end` 变量中。
- 遍历当前进程的虚拟内存区域链表，使用 `vm_area` 指针来迭代每个虚拟内存区域。
- 在每个虚拟内存区域内部，通过一个循环遍历虚拟地址范围。
- 根据起始地址和终止地址的关系，判断当前页是否位于数据段或代码段内，并根据需要拷贝页面的内容。
- 如果页面在数据段或代码段内，使用 `mfollow_page` 函数获取页面对应的 `struct page` 结构，并使用 `kmap_atomic` 将页面映射到可访问的虚拟地址。然后，使用 `memcpy` 将页面的内容拷贝到全局变量 `buf` 中，最后使用 `kunmap_atomic` 释放映射的虚拟地址，并将 `buf` 的内容写入文件。
- 循环结束后，完成了数据段或代码段的内容打印。
- 根据数据段或代码段的起始地址和终止地址获取其中的页面，并将页面的内容打印到文件中。

```
// func == 4 或者 func == 5
static void print_seg_info(void)
{
```

```
    struct mm_struct *mm;
    unsigned long addr;
    unsigned long start,end;
    struct vm_area_struct *vm_area;
    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;
    }

    if (ktest_func == 4) {
        start = mm->start_data;
        end = mm->end_data;
    } else {
        start = mm->start_code;
        end = mm->end_code;
    }
    vm_area = mm->mmap;
    while (vm_area) {
        unsigned long virt_addr = vm_area->vm_start;
        while (virt_addr < vm_area->vm_end) {
            if (start ≤ virt_addr && end ≥ virt_addr + PAGE_SIZE) {
                struct page *page = mfollow_page(vm_area, virt_addr,
FOLL_GET);
                if (!IS_ERR_OR_NULL(page)) {
                    addr = kmap_atomic(page);
                    memcpy(buf, addr, PAGE_SIZE);
                    kunmap_atomic(addr);
                    write_to_file(buf, PAGE_SIZE);
                }
            }
            else if (end ≥ virt_addr && end ≤ virt_addr + PAGE_SIZE)
{
                struct page *page = mfollow_page(vm_area, virt_addr,
FOLL_GET);
                if (!IS_ERR_OR_NULL(page)) {
                    addr = kmap_atomic(page);
                    memcpy(buf, addr, end - virt_addr);
                    kunmap_atomic(addr);
                    write_to_file(buf, end - virt_addr);
                }
            }
            else if (start ≥ virt_addr && start ≤ virt_addr +
PAGE_SIZE) {
                struct page *page = mfollow_page(vm_area, virt_addr,
FOLL_GET);
```

```
                if (!IS_ERR_OR_NULL(page)) {
                    addr = kmap_atomic(page);
                    memcpy(buf, addr + start - virt_addr, virt_addr + PAGE_SIZE - start);
                    kunmap_atomic(addr);
                    write_to_file(buf, virt_addr + PAGE_SIZE - start);
                }
            }
            virt_addr += PAGE_SIZE;
        }
        vm_area = vm_area->vm_next;
    }

    // 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。
    mmput(mm);
}
```

5　5 实验结果

5.1　动态内存分配器 malloc

按照实验文档要求运行`workload`得到结果如下。

first-fit:

```
time of malloc 0 : 724ms
before free: 0.971754; after free: 0.21819
time of loop 0 : 999ms
time of malloc 1 : 862ms
before free: 0.953877; after free: 0.21264
time of loop 1 : 1135ms
time of malloc 2 : 920ms
before free: 0.948942; after free: 0.215493
time of loop 2 : 1187ms
time of malloc 3 : 902ms
before free: 0.945729; after free: 0.213606
```

```
time of loop 3 : 1174ms
time of malloc 4 : 945ms
before free: 0.942717; after free: 0.20887
time of loop 4 : 1228ms
time of malloc 5 : 828ms
before free: 0.944327; after free: 0.211589
time of loop 5 : 1029ms
time of malloc 6 : 530ms
before free: 0.941958; after free: 0.21027
time of loop 6 : 694ms
time of malloc 7 : 516ms
before free: 0.94511; after free: 0.21117
time of loop 7 : 683ms
time of malloc 8 : 527ms
before free: 0.939438; after free: 0.208905
time of loop 8 : 682ms
time of malloc 9 : 503ms
before free: 0.948793; after free: 0.210799
time of loop 9 : 663ms
time of malloc 10 : 503ms
before free: 0.946898; after free: 0.211666
time of loop 10 : 663ms
time of malloc 11 : 510ms
before free: 0.945307; after free: 0.214865
time of loop 11 : 666ms
time of malloc 12 : 526ms
before free: 0.948323; after free: 0.213318
time of loop 12 : 686ms
time of malloc 13 : 532ms
before free: 0.94815; after free: 0.210896
time of loop 13 : 693ms
time of malloc 14 : 533ms
before free: 0.947042; after free: 0.211872
time of loop 14 : 697ms
time of malloc 15 : 510ms
before free: 0.942315; after free: 0.211329
time of loop 15 : 678ms
time of malloc 16 : 507ms
before free: 0.938851; after free: 0.210143
time of loop 16 : 670ms
time of malloc 17 : 509ms
before free: 0.93879; after free: 0.210873
time of loop 17 : 666ms
time of malloc 18 : 495ms
before free: 0.939189; after free: 0.209751
time of loop 18 : 659ms
time of malloc 19 : 506ms
before free: 0.942652; after free: 0.211805
```

```
time of loop 19 : 663ms
```

best_fit:

```
time of malloc 0 : 717ms
before free: 0.971754; after free: 0.21819
time of loop 0 : 997ms
time of malloc 1 : 3604ms
before free: 0.972011; after free: 0.21671
time of loop 1 : 3906ms
time of malloc 2 : 3563ms
before free: 0.970978; after free: 0.220482
time of loop 2 : 3742ms
time of malloc 3 : 2103ms
before free: 0.967858; after free: 0.218587
time of loop 3 : 2276ms
time of malloc 4 : 2175ms
before free: 0.964811; after free: 0.213779
time of loop 4 : 2336ms
time of malloc 5 : 2085ms
before free: 0.966365; after free: 0.216486
time of loop 5 : 2250ms
time of malloc 6 : 2062ms
before free: 0.963814; after free: 0.215146
time of loop 6 : 2230ms
time of malloc 7 : 2038ms
before free: 0.967057; after free: 0.216016
time of loop 7 : 2209ms
time of malloc 8 : 2054ms
before free: 0.961424; after free: 0.213795
time of loop 8 : 2219ms
time of malloc 9 : 1989ms
before free: 0.971064; after free: 0.215637
time of loop 9 : 2177ms
time of malloc 10 : 2071ms
before free: 0.968794; after free: 0.216595
time of loop 10 : 2234ms
time of malloc 11 : 2041ms
before free: 0.967507; after free: 0.219914
time of loop 11 : 2209ms
time of malloc 12 : 2044ms
before free: 0.97048; after free: 0.218296
time of loop 12 : 2206ms
time of malloc 13 : 2105ms
before free: 0.970281; after free: 0.215841
time of loop 13 : 2272ms
time of malloc 14 : 2066ms
before free: 0.972174; after free: 0.217413
time of loop 14 : 2229ms
```

```
time of malloc 15 : 2060ms
before free: 0.967305; after free: 0.216913
time of loop 15 : 2232ms
time of malloc 16 : 2098ms
before free: 0.963665; after free: 0.21568
time of loop 16 : 2262ms
time of malloc 17 : 2020ms
before free: 0.963823; after free: 0.216463
time of loop 17 : 2184ms
time of malloc 18 : 2110ms
before free: 0.964156; after free: 0.215264
time of loop 18 : 2277ms
time of malloc 19 : 2003ms
before free: 0.967491; after free: 0.217325
time of loop 19 : 2170ms
```

根据提供的输出结果，可以总结出以下观察结果：

first-fit 策略：

- 内存分配时间相对较短，每次分配内存所需的时间都在 500ms - 900ms 之间。
- 内存释放后，剩余的可用内存（即空闲块）的比例（after free）在 0.2 - 0.22 之间波动。
- 内存分配时间随着循环次数的增加而保持稳定。

best-fit 策略：

- 内存分配时间相对较长，每次分配内存所需的时间在 2000ms - 3600ms 之间，比 first-fit 策略的时间更长。
- 内存释放后，剩余的可用内存的比例（after free）在 0.21 - 0.22 之间波动，与 first-fit 策略相似。
- 内存分配时间随着循环次数的增加而增加，呈线性增长。

根据观察结果可以得出以下结论：

first-fit 策略的优点：

- 内存分配时间相对较短，可以更快地完成内存分配操作。
- 在释放内存后，剩余的可用内存的比例保持稳定。

best-fit 策略的优点：

- 在内存分配时，可以选择最合适大小的空闲块，从而更好地利用可用内存空间。

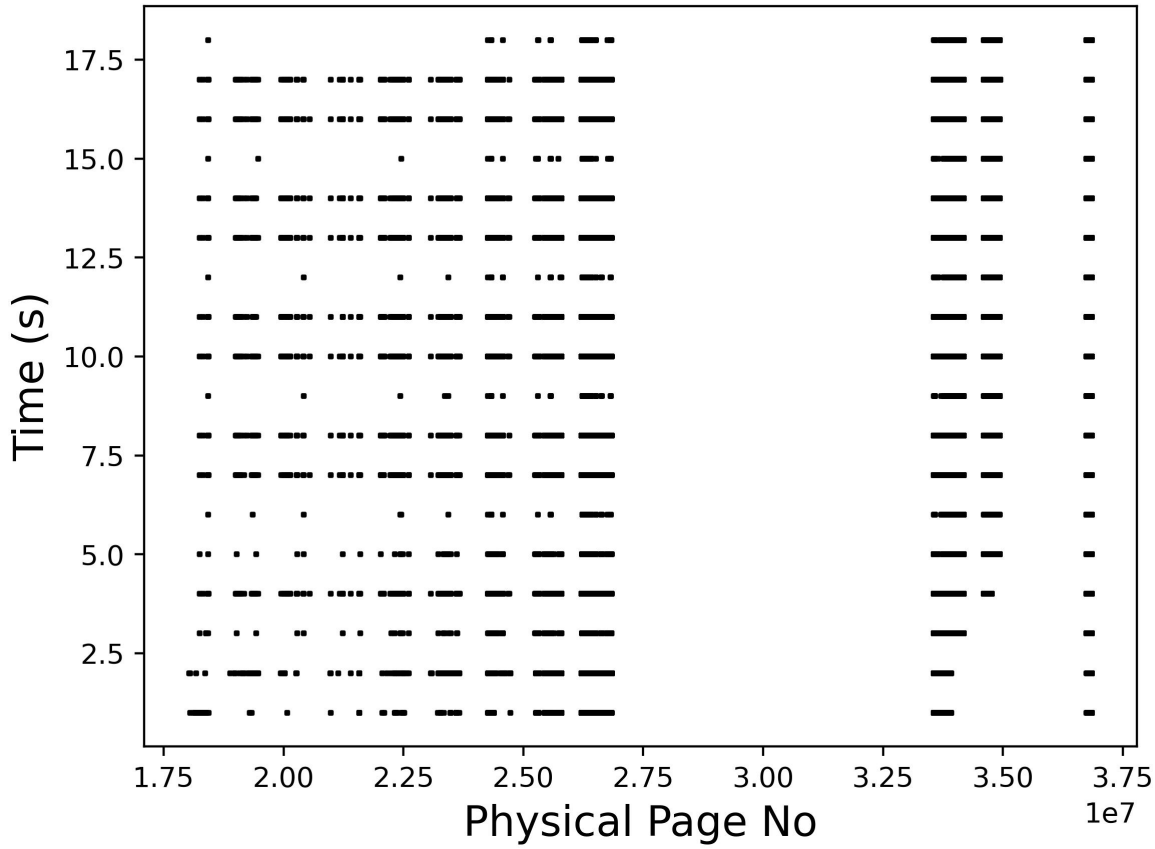
综合考虑，first-fit 策略在时间效率上更优，而 best-fit 策略在空间利用率上略有优势。选择适当的策略取决于具体的应用场景和需求。如果时间效率更为重要，则可以选择 first-fit 策略；如果空间利用率更为重要，则可以选择 best-fit 策略。

5.2 Linux 进程内存信息统计

5.2.1 func = 1

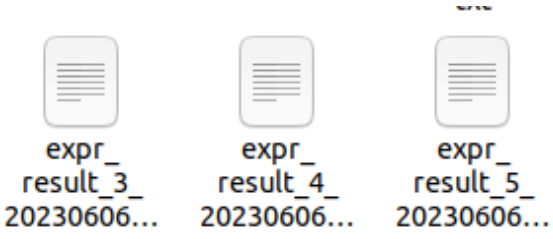
```
minerva@minerva-VirtualBox:~/oslab/lab3/test/part2$ cat /sys/kernel/mm/ktest/vma
0, 39
```

5.2.2 func = 2



5.2.3 其他

输出存在了输出文档里。



【补充说明】

上周阳了，和姚路路助教以及李老师都沟通过，老师同意延迟一周验收，助教哥哥做迟交标记的时候可以不要扣我的分嘛~）