华东师范大学数据学院上机实践报告

课程名称： 操作系统　　**年级：** 大二　　**上机实践成绩：**

指导教师： 翁楚良　　**姓名：** 沈小奇

上机实践名称：　　**学号：** 10185501401　　**上机实践日期：**

上机实践编号：

一、 目的

1.熟悉 Minix 操作系统的进程管理

2.学习 Unix 风格的内存管理

二、 内容与设计思想

修改 Minix3.1.2a 的进程管理器，改进 brk 系统调用的实现，使得分配给进程的数据段+栈段空间耗尽时，brk 系统调用给该进程 分配一个更大的内存空间，并将原来空间中的数据复制至新分 配的内存空间，释放原来的内存空间，并通知内核映射新分配 的内存段。

三、 使用环境

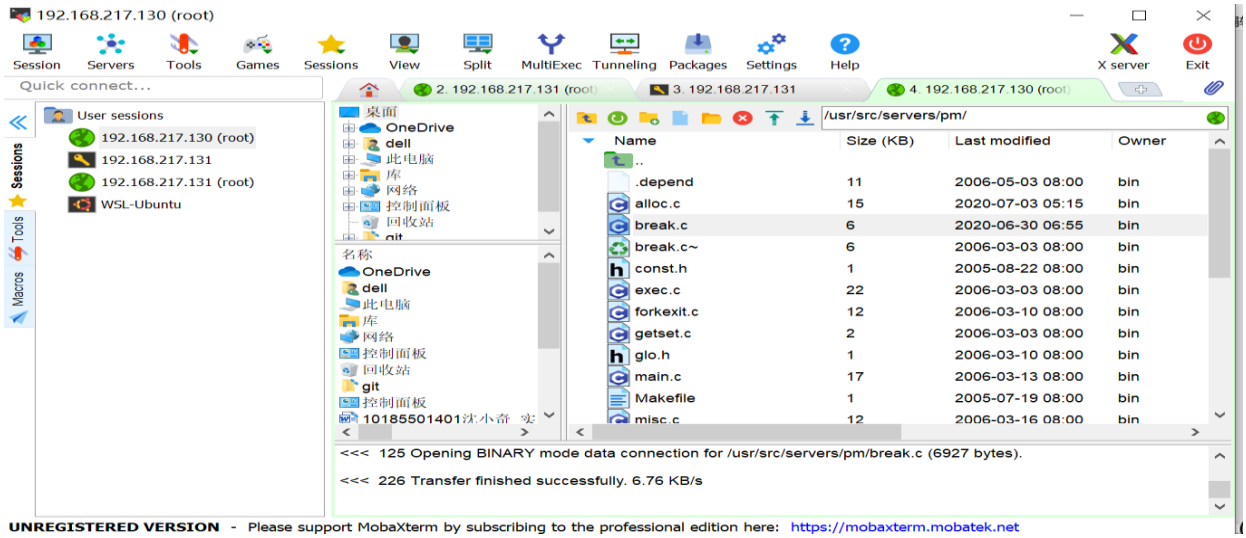
Minix3.1.2，mobaxterm

四、 实验过程

准备过程：

1.下载 Minix3.1.2 镜像文件，新建 VMware 虚拟机，具体过程详见 pdf 配置文档。

2.连接 Mobaxterm，这里由于 minix3.1.2 的 bug，导致 moba 连接上虚拟机后不显示左侧文件列表，因此这里需要用 ftp 来连接，即可修改文件。



实验步骤：

1.修改/usr/src/servers/pm/alloc.c 中的 alloc_mem 函数，把 first-fit 修改成 best-fit，即分配内存之前，先遍历整个空闲内存块列表，找到最佳匹 配的空闲块。

在上手代码前先了解一下每个空闲块的组成部分：

```
PRIVATE struct hole {
        struct hole * h_next;          /* 指针，指向链表的下一个结点 */
        phys_clicks h_base;            /* 空闲区的起始地址 */
        phys_clicks h_len;             /* 空闲区的大小 */
}hole[NR_HOLES];
```

图 4.35　空闲链表是一个 hole 结构体数组

因此，可参考源代码中的 first-fit 算法，实现 best-fit 算法，思想为：遍历所有空闲块，找寻能够装得下该进程的最小空闲块。

Best-fit：

```
1.  PUBLIC phys_clicks alloc_mem(clicks)
2.  phys_clicks clicks;     /* amount of memory requested */
3.  {
4.  /* Allocate a block of memory from the free list using first fit. The block
5.   * consists of a sequence of contiguous bytes, whose length in clicks is
6.   * given by 'clicks'.  A pointer to the block is returned.  The block is
7.   * always on a click boundary.  This procedure is called when memory is
8.   * needed for FORK or EXEC.  Swap other processes out if needed.
9.   */
10. register struct hole *hp, *prev_ptr,*best,*prev_best;
11. phys_clicks old_base,best_clicks;
12. int flag= 0;
13. do {
14. prev_ptr = NIL_HOLE;
15. hp = hole_head;
16. while (hp != NIL_HOLE && hp->h_base < swap_base) { //遍历空闲链表
17.     if (hp->h_len >= clicks) {
18.         //当找到合适的空闲链表时
19.
20.         if(!flag){//先暂时让 best 块等于这个比他大的空闲块
21.             best = hp;
22.             prev_best=prev_ptr;
23.             flag=1;
24.             best_clicks=best->h_len;
25.          }
26.
27.      else if(flag && hp->h_len<best_clicks){  //当找到更小的块时，best 块等于能够装的下该进程的
    最小的块
28.          best=hp;
29.          prev_best=prev_ptr;
30.          best_clicks=best->h_len;
31.
32.      }
33.
34.     }
35. prev_ptr = hp;
```

```
hp = hp->h_next;
}
} while (swap_out());       /* try to swap some other process out */

if (flag){  //当能够在空闲链表中找到这个 best 块时
    old_base = best->h_base;  //记录分配前块的起始位置
    best->h_base += clicks;  //best 块的起始位置发生改变，等于起始位置+分配出去的 clicks 大小
    best->h_len -= clicks; //块的大小改变了，等于原长度减去分配出去的长度

    if (best->h_base > high_watermark)  //high_watermark 记录分配的最大的内存
    high_watermark = best->h_base;

    if (best->h_len == 0) del_slot(prev_best,best);  //当这个块的长度为 0 时，就不再是空闲块了，所以将其释放。

    return(old_base);

}
return(NO_MEM);
}
```

2. 修改/usr/src/servers/pm/break.c 中的 adjust 函数，并增加了一个 allocate_new_mem 局部函数在 adjust 函数中调用。

brk 系统调用流程：

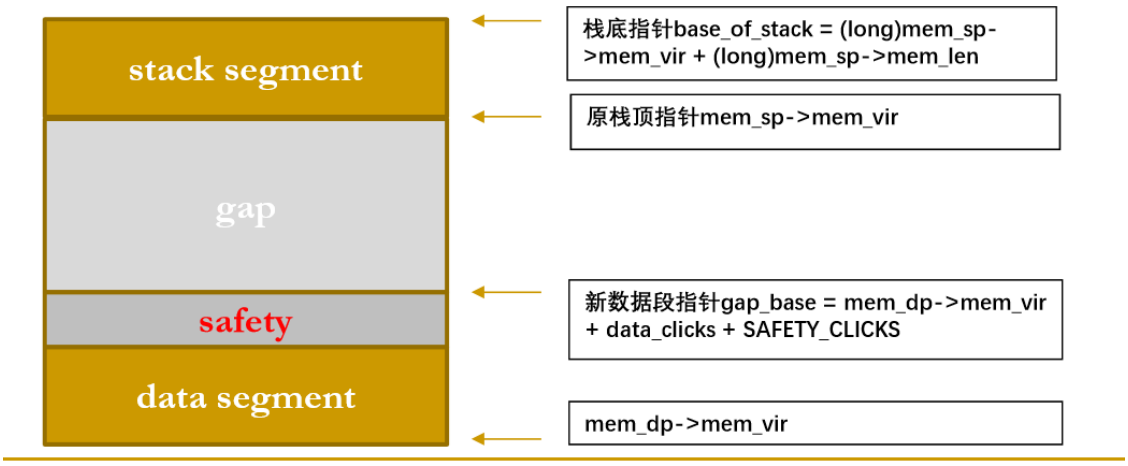
- do_brk 函数计算数据段新的边界，然后调用 adjust 函数，adjust 函数计算程序当前的空闲空间是否足够分配：
  - 若足够，则调整数据段指针，堆栈指针；通知内核程序的映像发生了变化，返回 do_brk 函数。
  - 若不够，调用 allocate_new_mem 函数申请新的足够大的内存空间；将程序现有的数据段和堆栈段的内容分别拷贝至新内存区域的底部(bottom)和顶部(top)；通知内核程序的映像发生 了变化；返回 do_brk 函数。

首先了解一下进程的结构：

内存中的进程

PM 的进程表称为 *mproc*，其定义位于 *src/servers/pm/mproc.h* 中。它包含了与进程内存分配有关的所有字段，还有其他一些额外信息。其中最重要的字段是 *mp_seg* 数组，它包含三个数组元素，分别用于代码段、数据段和栈段。每个数组元素是一个结构体类型，包含虚拟地址、物理地址和段长等信息，单位都是 **click** 而不是字节。在不同的实现中，click 的大小也是不同的。早期的 MINIX 版本把它设置为 256 个字节，而在 MINIX 3 中，一个 click 表示 1024 个字节。所有段的起始地址必须在 click 的边界上，而且段长必须是它的整数倍。

下图便于理解进程中各指针、地址的位置：



当要分配内存时候，有两种情况，分为空闲空间是否够分配，因此需要在 adjust 函数中作出判断，原先的代码定义为若不够分配直接返回 ENOMEM，现在要修改，使得内存块不够的时候分配更大的内存，需要调用 allocate_new_mem 函数。

```
1.  if (lower < gap_base) /* data and stack collided */
2.      {if (allocate_new_mem(rmp,
3.                    (phys_clicks)(mem_sp->mem_vir+mem_sp->mem_len-mem_dp-
    >mem_vir)) == ERROR)
4.          return(ENOMEM);}
```

allocate_new_mem 函数用于：

- 分配更大的内存
- 计算新分配的栈段和内存段的地址
- 将原来的数据段和栈段分别放入新内存块的底部和顶部
- 释放原来内存空间
- 通知内核映射新分配的内存段

首先有一大堆的定义，包括新分配的内存大小，旧内存块指针（由于数据段指针就是内存块指针，所以这里不出现 data_base，全部用 base 代替），新内存块指针，旧栈指针，新栈指针，数据段大小，栈段大小，旧内存块指针地址，旧栈指针地址，新栈指针地址，新内存块指针地址。

```
1.  register struct mem_map *mem_sp, *mem_dp;
2.      phys_clicks new_clicks, old_base,new_base;
3.      phys_clicks  old_stack_base,new_stack_base;
4.      phys_bytes data_bytes,stack_bytes,old_base_bytes;
5.      phys_bytes  old_stack_base_bytes,new_stack_base_bytes,new_base_bytes;
6.      mem_dp = &rmp->mp_seg[D];     /* pointer to data segment map */
7.      mem_sp = &rmp->mp_seg[S];  /* pointer to stack segment map */
```

注:在分配内存的时候单位为字节，这个我问了助教啊，是因为最后拷贝函数 sys_abscopy 里面的参数都是以字节为单位的。

原型是这样的`#define sys_abscopy(src_phys, dst_phys, bytes) \`

```
1.  old_base=mem_dp->mem_phys;
2.   old_stack_base=mem_sp->mem_phys;
```

```
3.
4.   data_bytes=(phys_bytes) mem_dp->mem_len << CLICK_SHIFT;  /*数据段长度，单位为字节 */
5.   stack_bytes=(phys_bytes) mem_sp->mem_len << CLICK_SHIFT;  //栈段长度，单位为字节
6.   old_base_bytes=old_base << CLICK_SHIFT;  //原始基地址，单位为字节
7.   old_stack_base_bytes=old_stack_base << CLICK_SHIFT;  //原始栈顶，单位为字节
8.
9.    //。。。。省略部分内容
10.
11. new_base_bytes = (phys_bytes) new_base << CLICK_SHIFT;//新的基地址，也是数据段基地址，单位为字节
12. new_stack_base_bytes=new_stack_base << CLICK_SHIFT;
```

分配更大的内存：

```
1.  new_clicks=2*old_clicks;
2.     new_base=alloc_mem(new_clicks);//分配 2 倍大的内存
3.     if(new_base==NO_MEM){
4.       return(ERROR);
5.     }
```

计算新的栈段和数据段地址：

```
1.  new_base_bytes = (phys_bytes) new_base << CLICK_SHIFT;//新的基地址，也是数据段基地址，单位为字节
2.  new_stack_base=new_base+new_clicks-mem_sp->mem_len;
3.      new_stack_base_bytes=new_stack_base << CLICK_SHIFT;
```

将原来的数据段和栈段分别放入新内存块的底部和顶部，sys_abscopy 是一个用于拷贝内存内容的函数，它的三个参数分别为：源地址、目的地址以及拷贝的字节数。

```
1.  x = sys_abscopy(old_base_bytes,new_base_bytes,data_bytes);//放数据段
2.     if (x < 0 ) panic(__FILE__,"allocate_new_mem can't copy",x);
3.     x = sys_abscopy( old_stack_base_bytes,new_stack_base_bytes,stak_bytes);//放栈段
4.     if ( x < 0 ) panic(__FILE__,"allocate_new_mem can't copy",x);
```

释放内存块：

```
1.  free_mem(old_base,old_clicks);
```

更新进程数据段和栈段的内存地址，以及栈段的虚拟地址：

```
1.   rmp->mp_seg[D].mem_phys = new_base;
2.   rmp->mp_seg[S].mem_phys = new_stack_base;
3.   rmp->mp_seg[S].mem_vir = mem_dp->mem_vir+new_clicks-mem_sp->mem_len;
```

以下为源代码：

```
1.  /*===========================================================================*
2.   *              allocate_new_mem                     *
3.   *===========================================================================*/
4.  PUBLIC int allocate_new_mem(rmp,old_clicks)
5.  register struct mproc *rmp;
```

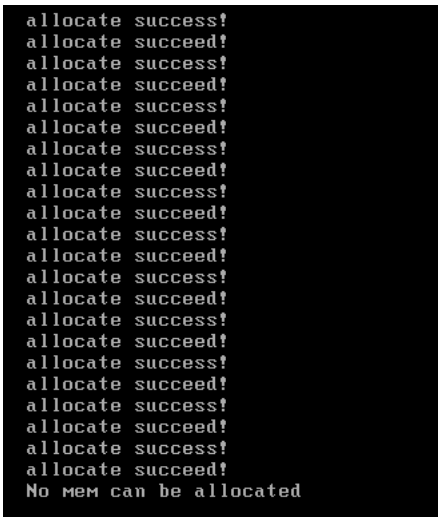
```
6.  phys_clicks old_clicks;
7.  {
8.      register struct mem_map *mem_sp, *mem_dp;
9.      phys_clicks new_clicks, old_base,new_base;
10.     phys_clicks  old_stack_base,new_stack_base;
11.     phys_bytes data_bytes,stack_bytes,old_base_bytes;
12.     phys_bytes  old_stack_base_bytes,new_stack_base_bytes,new_base_bytes;
13.    int x;
14.
15.     mem_dp = &rmp->mp_seg[D];    /* pointer to data segment map */
16.     mem_sp = &rmp->mp_seg[S];  /* pointer to stack segment map */
17.
18.     old_base=mem_dp->mem_phys;
19.     old_stack_base=mem_sp->mem_phys;
20.
21.     data_bytes=(phys_bytes) mem_dp->mem_len << CLICK_SHIFT;  /*数据段长度，单位为字节 */
22.     stack_bytes=(phys_bytes) mem_sp->mem_len << CLICK_SHIFT;  //栈段长度，单位为字节
23.     old_base_bytes=old_base << CLICK_SHIFT;  //原始基地址，单位为字节
24.     old_stack_base_bytes=old_stack_base << CLICK_SHIFT;  //原始栈顶，单位为字节
25.
26.     new_clicks=2*old_clicks;
27.     new_base=alloc_mem(new_clicks);//分配 2 倍大的内存
28.     if(new_base==NO_MEM){
29.       return(ERROR);
30.     }
31.
32.     new_base_bytes = (phys_bytes) new_base << CLICK_SHIFT;//新的基地址，也是数据段基地址，单
    位为字节
33.
34.     if ((x=sys_memset(0,new_base_bytes,(new_clicks<<CLICK_SHIFT)))!=OK){  //需要填充，不然会
    报错
35.       panic(__FILE__,"new mem can't be zero",x);
36. }
37.
38.     //得到新的栈段,表示成字节形式
39.     new_stack_base=new_base+new_clicks-mem_sp->mem_len;
40.     new_stack_base_bytes=new_stack_base << CLICK_SHIFT;
41.
42.
43.
44.     x = sys_abscopy(old_base_bytes,new_base_bytes,data_bytes);//放数据段
45.     if (x < 0 ) panic(__FILE__,"allocate_new_mem can't copy",x);
46.     x = sys_abscopy( old_stack_base_bytes,new_stack_base_bytes,stack_bytes);//放栈段
47.     if ( x < 0 ) panic(__FILE__,"allocate_new_mem can't copy",x);
48.
49.     //接着更新进程数据段和栈段的内存地址，以及栈段的虚拟地址：
50.     rmp->mp_seg[D].mem_phys = new_base;
51.     rmp->mp_seg[S].mem_phys = new_stack_base;
52.     rmp->mp_seg[S].mem_vir = mem_dp->mem_vir+new_clicks-mem_sp->mem_len;
53.     free_mem(old_base,old_clicks);
54.   return (1);
55.
56.
57.
58. }
```

测试结果：

```
# cd /home
# ./test1
incremented by 1, total 1
incremented by 2, total 3
incremented by 4, total 7
incremented by 8, total 15
incremented by 16, total 31
incremented by 32, total 63
incremented by 64, total 127
incremented by 128, total 255
incremented by 256, total 511
incremented by 512, total 1023
incremented by 1024, total 2047
incremented by 2048, total 4095
incremented by 4096, total 8191
incremented by 8192, total 16383
incremented by 16384, total 32767
incremented by 32768, total 65535
incremented by 65536, total 131071
incremented by 131072, total 262143
incremented by 262144, total 524287
incremented by 524288, total 1048575
incremented by 1048576, total 2097151
incremented by 2097152, total 4194303
incremented by 4194304, total 8388607
incremented by 8388608, total 16777215
incremented by 16777216, total 33554431
incremented by 33554432, total 67108863
#
```

```
# ./test2
incremented by: 1, total: 1 , result: 760
incremented by: 2, total: 3 , result: 4096
incremented by: 4, total: 7 , result: 4098
incremented by: 8, total: 15 , result: 4102
incremented by: 16, total: 31 , result: 4110
incremented by: 32, total: 63 , result: 4126
incremented by: 64, total: 127 , result: 4158
incremented by: 128, total: 255 , result: 4222
incremented by: 256, total: 511 , result: 4350
incremented by: 512, total: 1023 , result: 4606
incremented by: 1024, total: 2047 , result: 5118
incremented by: 2048, total: 4095 , result: 6142
incremented by: 4096, total: 8191 , result: 8190
incremented by: 8192, total: 16383 , result: 12286
incremented by: 16384, total: 32767 , result: 20478
incremented by: 32768, total: 65535 , result: 36862
incremented by: 65536, total: 131071 , result: 69630
incremented by: 131072, total: 262143 , result: 135166
incremented by: 262144, total: 524287 , result: 266238
incremented by: 524288, total: 1048575 , result: 528382
incremented by: 1048576, total: 2097151 , result: 1052670
incremented by: 2097152, total: 4194303 , result: 2101246
incremented by: 4194304, total: 8388607 , result: 4198398
incremented by: 8388608, total: 16777215 , result: 8392702
incremented by: 16777216, total: 33554431 , result: 16781310
incremented by: 33554432, total: 67108863 , result: 33558526
#
```

然后再看一下函数执行情况



五、**总结**

这次的实验难度有点大，但主要思想应该不是最难的，更多的是在语法吧，很多函数不太清楚怎么用，然后会出现各种语法错误，core_dump，整体代码量大，一个地方写错可能要找好久，浪费很长时间，然后各种栈啊、数据段地址如果算错就很难找，不知道错哪里，所以其实相比改正问题，寻找问题更耗时。但是通过本次实验，使得我对内存的分配有了更深的认识，整个操作流程用到的函数 do_brk,adjust,allocate_new_mem,alloc_mem, size_ok 等，亲自过一遍会记得更深。