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**：2019 级　　**上机实践成绩**：

指导教师：翁楚良　　**姓名**：叶秋雨

上机实践名称：内存管理　　**学号**：10184102103　　**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

物理机：Windows10

虚拟机：Minix3.1.2

虚拟机软件：Vmware

终端控制软件：MobaXterm

物理机与虚拟机文件传输：FileZilla

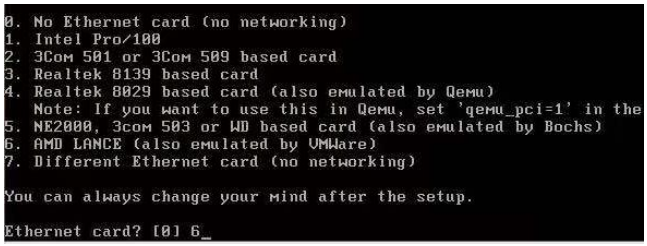
四、 实验过程

（1）Minix3.1.2 实验环境配置

• 下载 Minix3.1.2 镜像文件并创建虚拟机

• 修改虚拟机兼容性，选择 5.x 版本

• 启动虚拟机，在 setup 步骤中，网卡选择 AMD LANCE 以便获得 IP 地址。安装完成后，输入 shutdown，然后输入 boot d0p0 重启。



• 输入 mv /etc/rc.daemons.dist /etc/rc.daemons。在网络模式设置为 NAT 模式时，重启后可以看到 IP 地址。

```
192.168.106.128 login: root
Password:
```

• 在虚拟机终端输入 packman 安装额外的软件包，为了方便，选择全部安装（400MB 左右）

```
# packman
/dev/c0d2p2 is read-only mounted on /mnt
There are 46 CD packages.
Please choose:
 1  Install all 46 binary packages (408 MB uncompressed) from CD
 2  Install all 46 binary packages + sources from CD (1210 MB uncompressed)
 3  Display the list of packages on CD
 4  Let me select individual packages to install from CD or network.
 5  Exit.
Choice: [4] 1
```

（2）实现 alloc.c 函数

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

• 函数实现思路：

遍历空闲块的链表，best_ptr 记录当前最佳适配的空闲块，若找到满足条件且更小的空闲块，那么则对 best_ptr 进行更新，直至遍历结束。

```
hp=hole_head;
best_ptr=NIL_HOLE;//记录当前找到的最佳适配块
best_prev_ptr=NIL_HOLE;
prev_ptr=NIL_HOLE;
while(hp!=NIL_HOLE && hp->h_base < swap_base){
    if(hp->h_len >= clicks && best_ptr== NIL_HOLE){
        best_ptr=hp;
        best_prev_ptr=prev_ptr;
    }
    if(hp->h_len>=clicks && best_ptr!=NIL_HOLE){
        //如果块大小小于当前的最佳适配块，则更新当前最佳适配块
        if(hp->h_len < best_ptr->h_len){
            best_ptr=hp;
            best_prev_ptr=prev_ptr;
        }
    }
    prev_ptr= hp ;
    hp=hp->h_next;
}
```

（3）实现 break.c 函数

• 修改/usr/src/servers/pm/break.c 中的 adjust 函数，完成函数 allocate_new_mem 的定义并增加一个 allocate_new_mem 局部函数在 adjust 函数中调用。

break.c 函数的作用：do_brk 函数计算数据段新的边界，然后调用 adjust 函数计算程序当前的空闲空间是否足够分配：

(a)若足够，则调整数据段指针，堆栈指针并通知内核程序的映像发生了变化，返回 do_brk 函数。

(b)若不够，则调用 allocate_new_mem 函数申请新的足够大的内存空间。

```
  //计算栈底指针=栈顶指针+栈大小
  base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
  //byte 与 click 单位转换，得到新的栈顶指针
  sp_click = sp >> CLICK_SHIFT;
  if (sp_click >= base_of_stack) return(ENOMEM);

  /* Compute size of gap between stack and data segments. */
  delta = (long) mem_sp->mem_vir - (long) sp_click;
  lower = (delta > 0 ? sp_click : mem_sp->mem_vir);
  /* Add a safety margin for future stack growth. Impossible to do right. */
#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
  gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
  //lower 为栈顶指针，gap_base 为数据段的边界
  //如果 lower<gap 则说明栈段，说明发生了冲突
  //调用 allocate_new_mem 函数，分配新的空间
  if(lower < gap_base){
    res=allocate_new_mem(rmp,data_clicks,delta,(phys_clicks)(rmp-
>mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir + rmp->mp_seg[S].mem_len));
    if(res==OK){
      return (OK);
    }
    if(res== ENOMEM){
      return (ENOMEM);
    }
  }
```

lower 为栈顶指针，gap_base 为数据段的边界，如果 lower<gap_base 则说明栈段和数据段发生了冲突，当前空闲空间不够分配，因此调用 allocate_new_mem 函数分配新的空间。

• allocate_new_mem 函数实现思路：

调用 alloc_mem 函数申请一块大小为原空间两倍的新空间。调用 sys_abscopy 函数

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和顶部。修改进程表中记录的数据段和栈段的物理地址、虚拟地址。再调用 sys_newmap 函数更新虚拟地址到物理地址的映射。最后，调用 free_mem 函数释放掉原来的空间。

主要用到的三个函数 sys_abscopy、sys_newmap、free_mem，利用 source insight 查看三个函数的调用方式。

```
/* Create a copy of the parent's core image for the child. */
child_abs = (phys_bytes) child_base << CLICK_SHIFT;
parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
s = sys_abscopy(parent_abs, child_abs, prog_bytes);
if (s < 0) panic(__FILE__,"do_fork can't copy", s);
```

通过查看 sys_abscopy 函数在其他函数中被调用的方式，可知传入的第一个参数为对象的旧地址，第二个参数为对象的新地址，第三个参数为总的字节数。其中两个地址都是以 byte 为单位。

```
---- sys_newmap Matches (8 in 4 files) ----
Alloc.c (code\pm):      sys_newmap(rmp->mp_endpoint, rmp->mp_seg);
Alloc.c (code\pm):  sys_newmap(rmp->mp_endpoint, rmp->mp_seg);
Break.c (code\pm):  if (changed && (r2=sys_newmap(rmp->mp_endpoint, rmp->mp_seg)) != OK)
Break.c (code\pm):          panic(__FILE__,"couldn't sys_newmap in adjust", r2);
Exec.c (code\pm):  if((r2=sys_newmap(who_e, rmp->mp_seg)) != OK) {
Exec.c (code\pm):   panic(__FILE__,"sys_newmap failed", r2);
Forkexit.c (code\pm):  if((r=sys_newmap(rmc->mp_endpoint, rmc->mp_seg)) != OK) {
Forkexit.c (code\pm):   panic(__FILE__,"do_fork can't sys_newmap", r);
```

sys_newmap 传入的第一个参数为 rmp->mp_endpoint，第二个参数为 rmp->mp_seg。

```
free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
free_mem(rmp->mp_seg[D].mem_phys,
 rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);
```

free_mem 传入的第一个参数是释放空间的起始地址，第二个参数是空间大小。

函数头如图，其中传递的参数申明放在函数名和函数体大括号之间，是一种老式的参数传递方式。

```
PUBLIC int allocate_new_mem(rmp,data_clicks,delta,clicks)
register struct mproc *rmp;
phys_clicks data_clicks;
long delta;
phys_clicks clicks;
{
```

根据传入的 clicks 大小，计算出新空间的大小，按原来的 2 倍进行计算。调用 alloc_mem 函数申请一块新的空间。

```
new_clicks = clicks*2;
old_clicks = clicks;
if((new_address_data=alloc_mem(new_clicks)) == NO_MEM){
  return(ENOMEM);
}
```

分别计算出栈段和数据段的长度以及栈段和数据段的旧地址、新地址，其中栈顶指针的位置等于数据段地址（起始地址）加上地址空间的长度减去栈段的长度，即new_address_stack=new_address_data+new_clicks - mem_sp->mem_len。将他们转化为以 byte 为单位。

```
    databytes=(phys_bytes)mem_dp->mem_len << CLICK_SHIFT;
    stackbytes=(phys_bytes)mem_sp->mem_len << CLICK_SHIFT;

    old_address_data=mem_dp->mem_phys;
    new_address_stack=new_address_data + new_clicks - mem_sp->mem_len;
    old_address_stack=mem_sp->mem_phys;

    new_address_data_byte=(phys_bytes)new_address_data << CLICK_SHIFT;
    old_address_data_byte=(phys_bytes)old_address_data << CLICK_SHIFT;
    new_address_stack_byte=(phys_bytes)new_address_stack << CLICK_SHIFT;
    old_address_stack_byte=(phys_bytes)old_address_stack << CLICK_SHIFT;
```

将栈段和数据段的内容复制到新的地址，并修改进程表中记录的栈段与数据段的虚拟地址和物理地址。其中数据段的虚拟地址不用进行修改，因为在 minix 中代码段与数据段和栈段是分开管理的，数据段虚拟地址是零。

```
d = sys_abscopy(old_address_data_byte,new_address_data_byte,databytes);
    if (d < 0)
        panic(__FILE__, " can't copy data segment in alloc_new_mem", d);
    s = sys_abscopy(old_address_stack_byte,new_address_stack_byte,stackbytes);
    if (s < 0)
        panic(__FILE__, " can't copy stack segment in alloc_new_mem", s);


    rmp->mp_seg[D].mem_phys = new_address_data;
    rmp->mp_seg[S].mem_phys = new_address_stack;
    rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir+new_clicks -mem_sp->mem_len;
```

类似 adjust 函数中记录下栈段和数据段的地址、长度改变。

```
    if(data_clicks != mem_dp->mem_len){
      mem_dp->mem_len = data_clicks;
      change |= DATA_CHANGED;
    }

    if(delta > 0){
      mem_sp->mem_vir -= delta;
      mem_sp->mem_phys -= delta;
      mem_sp->mem_len += delta;
      change |= STACK_CHANGED;
```

```
    }
```

判断新的栈段、数据段大小是否适合地址空间，若适合则调用 sys_newmap 函数更新虚拟地址到物理地址的映射，并调用 free_mem 释放旧的空间；若不适合则恢复栈段、数据段到旧的空间。

```
    ft = (rmp->mp_flags & SEPARATE);
#if (CHIP == INTEL && _WORD_SIZE == 2)
    r = size_ok(ft, rmp->mp_seg[T].mem_len,rmp->mp_seg[D].mem_len,
                rmp->mp_seg[S].mem_len,rmp->mp_seg[D].mem_vir, rmp-
>mp_seg[S].mem_vir);
#else
  r = (rmp->mp_seg[D].mem_vir + rmp->mp_seg[D].mem_len >
          rmp->mp_seg[S].mem_vir) ? ENOMEM : OK;
#endif
  if (r == OK) {
    int r2;
    if (change && (r2=sys_newmap(rmp->mp_endpoint, rmp->mp_seg)) != OK)
      panic(__FILE__,"couldn't sys_newmap in adjust", r2);
    free_mem(old_address_data, old_clicks);
    return(OK);
  }

  /* New sizes don't fit or require too many page/segment registers. Restore.*/
  if (change & DATA_CHANGED) mem_dp->mem_len = cur_data_clicks;
  if (change & STACK_CHANGED) {
  mem_sp->mem_vir += delta;
  mem_sp->mem_phys += delta;
  mem_sp->mem_len -= delta;
  }
  return(ENOMEM);
}
```

（4）编译 Minix

• 进入/usr/src/servers 目录，输入 make image，等编译成功之后输入 make install 安装新的 PM 程序。

• 进入/usr/src/tools 目录，输入 make hdboot，成功之后再键入 make install 命令安装新的内核程序。并记录下生成的新内核版本号。

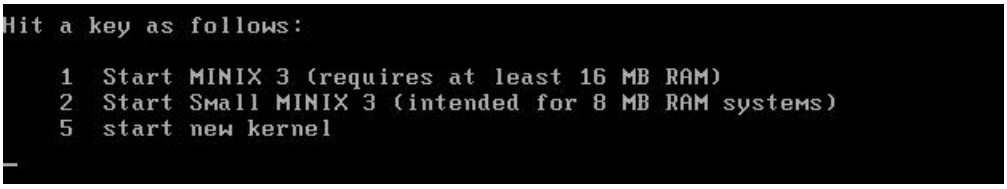
```
rm /dev/c0d3p0s0:/boot/image/3.1.2ar0
install image /dev/c0d3p0s0:/boot/image/3.1.2ar3
Done.
```

• 键入 shutdown 命令关闭虚拟机，进入 boot monitor 界面。设置启动新内核的选项，在提示符键入：

newminix(5,start new kernel) {image=/boot/image/3.1.2ar3;boot;}

注：5 为启动菜单中的选择内核版本的数字键，也可设置为其他数字；3.1.2ar3 为上一步中记录的新生成的内核版本号。

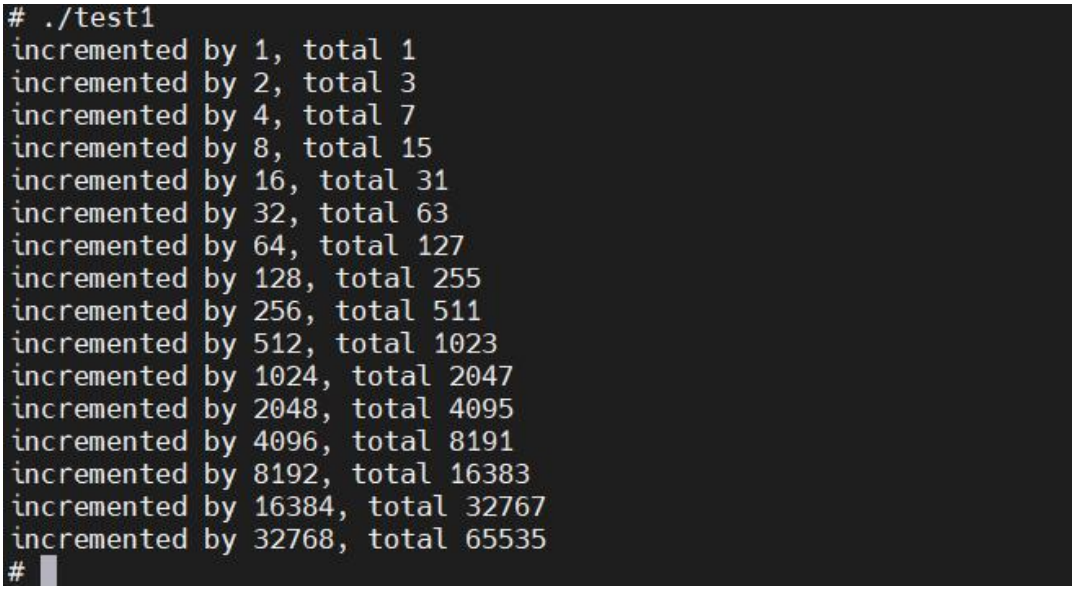
• 输入 menu 命令，然后敲数字键（上一步骤中设置的数字）启动新内核，登录进 minix 3 中测试。



五、实验结果

测试程序 test1 测试 sbrk 调用，不断地调整数据段的上界并未对新分配的内存空间进行访问；测试程序 test2 则对新分配的内存空间进行了访问。

测试结果如下：



```
# ./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
#
```

六、总结

通过这次试验，再次熟悉了虚拟机的安装与配置，熟悉了 Minix 操作系统系统的进程管理。通过修改函数的代码，更好地掌握了内存管理的最佳适配原则。对 break.c 的修改，首先阅读其原本的代码理解了从 do_brk 到 adjust 再到 allocate_new_mem 的层层调用并在此基础上完成了 allocate_new_mem 函数的实现，也让我对 minix 的内存管理的理解更加明晰。同时，也更加熟练地应用 source insight 来阅读源码，查看变量、函数的定义。在实验过程中，一定要仔细，否则就会犯很多小错误阻碍实验进度。如：在调用 sys_abscopy 时注意传入的参数是以 byte 为单位，要进行 click 到 byte 的转换；函数的定义方式，其中传递的参数申明放在函数名和函数体大括号之间，是一种老式的参数传递方式；在编译测试代码时，利用 cc 而不是 clang 进行编译等等。在测试时，遇到错误也要善于利用 printf 输出中间结果来定位出错的位置。