华东师范大学数据科学与工程学院实验报告

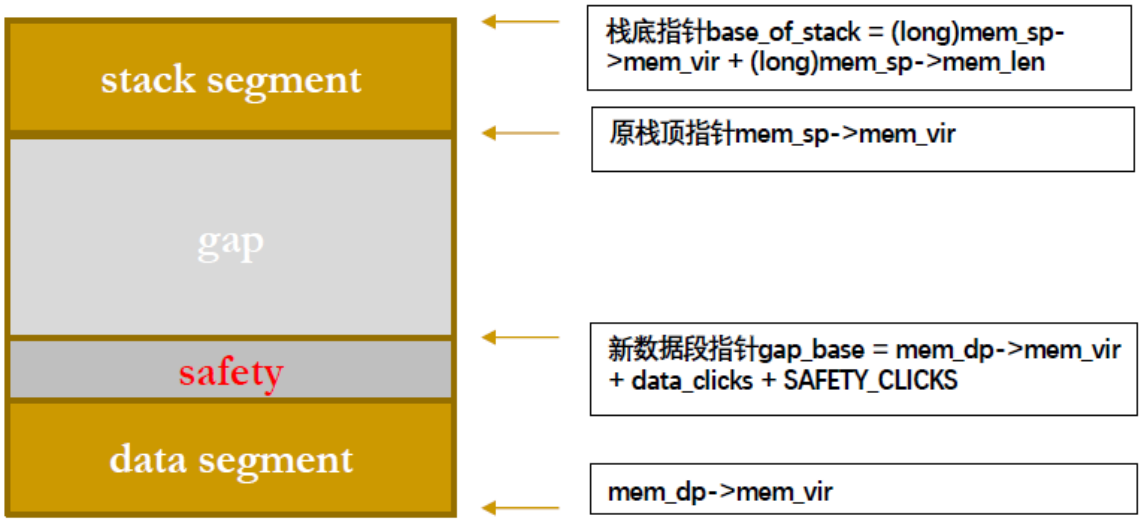
课程名称： 操作系统	**年级：** 21 级	**上机实践成绩：**
指导教师： 翁楚良	**姓名：** 杨茜雅	**学号：** 10215501435
上机实践名称： Lab04		**上机实践日期：** 2023.5.25
上机实践编号： 4	**组号：**	**上机实践时间：** 2023.5.25

一、 实验目的

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、 实验任务

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

VMware Workstation Pro
Visual studio code
MobaXterm

四、 实验注意事项

- PM 是用户进程，printf 结果显示在虚拟机下，可以不需要串口输出。
- 相关的函数
    - alloc_mem 分配内存
    - sys_abscopy 拷贝内存内容
    - free_mem 释放内存
    - sys_newmap 通知内核，注册内存段

• 用户调用 brk 函数针对的是虚拟地址，而 minix 最底层内存管理是物理地址，不能混淆。

• 需要小心处理 clicks 和 bytes 的单位换算和对齐。

• 本实验是修改 brk 系统调用的实现，所以不需要更改 brk 系统调用的接口，brk 函数调用的参数和返回值请遵循原定义，请不要修改。

• 程序运行错误会可能产生体积很大的 core 文件，建议在磁盘空间较大的/home 下做实验，否则可能会占满磁盘，导致无法开机。

• 如果 break 修改后导致程序的编译都无法正常进行（因为在程序编译的过程中可能需要 brk 系统调用）。在这种情况下，需从未修改过的内核中对程序进行编译。

• CC 编译器版本很老，语法检查很严格，编写 C 代码时不要引入非英文字符，或者使用高级语法。

• 本 project 使用的 Minix 版本号是 3.1.2。

五、 实验过程

1.修改内存分配：

1.1 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。

该函数使用首次适应算法从空闲列表中分配内存块。它会在空闲列表中遍历，寻找满足请求内存量的空闲块。

首先声明了一些变量：

hp、prev_ptr 和 best_hp 是指向 struct hole 结构的指针，用于遍历和操作空闲列表中的空闲块。

- hp：指向空闲列表中的当前空闲块的指针。
- prev_ptr：指向前一个空闲块的指针，用于在删除空闲块时更新链表。
- best_hp：指向最接近请求内存量的空闲块的指针。
- old_base 是保存分配的空闲块的起始位置。
- best_size 用于记录最接近请求内存量的空闲块的大小。
- flag 用于标记是否找到了比请求内存量大但最接近的空闲块。
- Clicks 请求的内存量，以 click 为单位，click 是 1024 字节。

```
register struct hole *hp, *prev_ptr, *best_hp;
phys_clicks old_base;
phys_clicks best_size;
int flag=0;
```

然后，函数进入一个循环，重复尝试分配内存的过程。

在每次循环中，首先将 prev_ptr、hp、best_hp 和 best_size 初始化为初始值。

```
do {
        prev_ptr = NIL_HOLE;
        hp = hole_head;
        best_hp = NIL_HOLE;
        best_size = 0xFFFFFFFF;
        flag = 0;
```

接下来，函数通过遍历空闲内存列表来查找合适的空闲块。
在遍历过程中，对于每个空闲块，进行以下判断：
如果空闲块的大小正好等于请求的内存量，表示找到了一个足够大的空闲块，直接使用它。将 old_base 设置为该块的起始位置，然后通过调用 del_slot(prev_ptr, hp)函数从空闲列表中删除该块，并返回 old_base。

```
while (hp != NIL_HOLE && hp->h_base < swap_base) {
  if (hp->h_len == clicks) {
    /* We found a hole that is big enough.  Use it. */
    old_base = hp->h_base;  /* remember where it started */
    del_slot(prev_ptr, hp);
    /*printf("0\n");*/
        return(old_base);
```

如果空闲块的大小大于请求的内存量，并且比当前记录的最接近大小 best_size 要小，更新 best_hp 为指向该块的指针，更新 best_size 为该块的大小，并将 flag 设置为 1，表示找到了比请求内存量大但最接近的空闲块。

```
}else if(hp->h_len > clicks && hp->h_len < best_size){
  best_hp = hp;
  best_size = hp->h_len;
  flag = 1;
}
```

在遍历结束后，如果 flag 为 1，表示找到了比请求内存量大但最接近的空闲块。这时，将 old_base 设置为 best_hp 指向的块的起始位置，将 best_hp 的起始位置增加请求内存量，并将 best_hp 的大小减去请求内存量，最后返回 old_base。

```
if(flag == 1){
  old_base = best_hp->h_base;
  best_hp->h_base += clicks;
  best_hp->h_len -= clicks;
/*printf("1\n");*/
  return(old_base);
}
```

如果在空闲列表中无法找到足够大的空闲块，则尝试通过调用 swap_out()函数将其他进程换出。如果成功换出进程，则再次尝试分配内存。这个过程会持续进行，直到成功分配内存或无法继续换出其他进程。

```
} while (swap_out());    /* try t
  return(NO_MEM);
}
```

如果无法成功分配内存，函数将返回 NO_MEM 常量，表示内存不足。

总体来说，该函数使用首次适应算法在空闲列表中寻找足够大的空闲块来分配内存，并进行必要的操作和标记。在没有合适的空闲块时，会尝试通过换出其他进程来释放内存。
swap 机制其实就是将外存(如硬盘)当内存使用，怎么可以把外存当内存使用呢？原理就是当系统内存不够用的时候，内核会选择某些进程，把其使用较少的内存的内容交换(swap)到外存中，然后把内存让给需要的进程使用.

```
/*===========================================================================*
 *        alloc_mem           *
 *===========================================================================*/
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks;   /* 请求的内存量，以click为单位，click是1024字节 */
{
/* Allocate a block of memory from the free list using first fit. The block
 * consists of a sequence of contiguous bytes, whose length in clicks is
 * given by 'clicks'.  A pointer to the block is returned.  The block is
 * always on a click boundary.  This procedure is called when memory is
 * needed for FORK or EXEC.  Swap other processes out if needed.
 */
  register struct hole *hp, *prev_ptr, *best_hp;//是指向struct hole结构的指针，用于遍历和操作空闲列表中的空闲块
  phys_clicks old_base;//保存分配的空闲块的起始位置
  phys_clicks best_size;//记录最接近请求内存量的空闲块的大小
  int flag=0;//用于标记是否找到了比请求内存量大但最接近的空闲块
// 重复进行，直到成功分配内存或无法继续换出其他进程
  do {
    //每次循环都要初始化为初始值
        prev_ptr = NIL_HOLE; // 用于记录前一个空闲块的指针
        hp = hole_head;// 指向空闲列表的头部
        best_hp = NIL_HOLE;// 用于记录找到的最接近大小的空闲块
        best_size = 0xFFFFFFFF;// 最接近大小的空闲块的大小
        flag = 0;// 标志是否找到了合适的空闲块
    // 遍历空闲内存列表来查找合适的空闲块
    while (hp != NIL_HOLE && hp->h_base < swap_base) {//遍历开始
    if (hp->h_len == clicks) {
      //空闲块的大小正好==请求的内存量，说明找到了一个足够大的空闲块，可直接使用
      /* We found a hole that is big enough.  Use it. */
      old_base = hp->h_base;//记住起始位置
      del_slot(prev_ptr, hp);//从空闲列表中删除该块
      /*printf("0\n");*/
          return(old_base);
      }else if(hp->h_len > clicks && hp->h_len < best_size){
        //如果空闲块的大小大于请求的内存量，并且比当前记录的最接近大小best_size要小
        // 找到一个比请求的内存量大但最接近的空闲块
        best_hp = hp;//更新best_hp为指向该块的指针
        best_size = hp->h_len;//更新best_size为该块的大小
        flag = 1;//找到了比请求内存量大但最接近的空闲块
      }
    prev_ptr = hp;// 更新前一个空闲块的指针
  hp = hp->h_next;// 指向下一个空闲块
  }//遍历结束
  if(flag == 1){//如果flag为1，表示找到了比请求内存量大但最接近的空闲块
     // 分配最接近请求内存量的空闲块
    old_base = best_hp->h_base; // 记住起始位置
    best_hp->h_base += clicks; // 更新最接近大小的空闲块的起始位置
    best_hp->h_len -= clicks;// 更新最接近大小的空闲块的大小
  /*printf("1\n");*/
    return(old_base);// 返回分配的内存块的起始位置
  }
//如果flag!=1，说明无法找到足够大的空闲块
} while (swap_out());//尝试通过调用swap_out()函数将其他进程换出
  return(NO_MEM);
}
```

1.2 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。

• 可参考课本 4.7 和 4.8 节

修改前 adjust 函数：

```
gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
if (lower < gap_base) return(ENOMEM); /* data and stack collided */
```

lower 栈顶指针，gap_base 数据段的边界。原来的 adjust 函数中当堆栈段地址和数据段地址有重叠时直接返回 ENOMEM，do_brk 不能继续为数据段增加空间，所以需要新增一个 allocate_new_mem 局部函数在 adjust 函数中调用来申请新的足够大的内存空间。

allocate_new_mem 函数：

该函数的主要功能是为进程分配一个足够大的新内存空间，并将原始数据段和堆栈段的内容分别拷贝到新的内存空间的底部和顶部。它还更新进程的内存映射表以反映新的数据段和堆栈段的地址和大小。如果新的大小不适合或需要过多的页/段寄存器，则恢复原始状态并释放新分配的内存空间。

allocate_new_mem 函数中调用的相关函数和作用：

- alloc_mem 分配内存
- sys_abscopy 拷贝内存内容
- free_mem 释放内存
- sys_newmap 通知内核，注册内存段

• 保存原始数据段和堆栈段的物理地址和长度

```
// 保存原始数据段和堆栈段的物理地址和长度
 old_base = mem_dp->mem_phys;
 old_clicks = mem_sp->mem_vir + mem_sp->mem_len - mem_dp->mem_vir;
```

• 分配新的内存空间，大小为原始数据段和堆栈段的两倍，涉及函数 alloc_mem 分配内存

```
// 分配新的内存空间，大小为原始数据段和堆栈段的两倍
 new_base = alloc_mem(2*old_clicks);
 if(new_base == NO_MEM) {
   printf("no more space\n");
   return(ENOMEM);
 }else{
   printf("alloc mem in addr %d\n",new_base);
   }
```

• 计算需要拷贝的原始数据段和堆栈段的起始地址和长度

```
// 计算需要拷贝的原始数据段和堆栈段的起始地址和长度
  old_d_tran = mem_dp->mem_phys << CLICK_SHIFT;
  new_d_tran = new_base << CLICK_SHIFT;
  d_len = mem_dp->mem_len << CLICK_SHIFT;
  old_s_tran = mem_sp->mem_phys << CLICK_SHIFT;
  new_s_tran = (new_base + 2*old_clicks - mem_sp->mem_len) << CLICK_SHIFT;
  s_len = mem_sp->mem_len << CLICK_SHIFT;
```

• 将原始数据段拷贝至新的数据段底部，涉及函数 sys_abscopy 拷贝内存内容（源地址，目的地址，字节数）

```
 // 将原始数据段拷贝至新的数据段底部
 d = sys_abscopy(old_d_tran, new_d_tran, d_len);
 if(d < 0)
    panic(__FILE__,"can't copy data segment in allocate_new_mem", d);
```

• 将原始堆栈段拷贝至新的堆栈段顶部，涉及函数 sys_abscopy 拷贝内存内容（源地址，目的地址，字节数）

```
// 将原始堆栈段拷贝至新的堆栈段顶部
d = sys_abscopy(old_s_tran, new_s_tran, s_len);
if(d < 0)
   panic(__FILE__,"can't copy stack segment in allocate_new_mem", d);
```

• 更新进程控制块中数据段和堆栈段的信息

```
// 更新进程控制块中数据段和堆栈段的信息
mem_dp->mem_phys = new_base;
mem_sp->mem_phys = new_base + 2*old_clicks - (mem_sp->mem_vir+mem_sp->mem_len-sp_click);
mem_dp->mem_len = data_clicks;
changed |= DATA_CHANGED;
mem_sp->mem_len = mem_sp->mem_vir+mem_sp->mem_len-sp_click;
changed |= STACK_CHANGED;
mem_sp->mem_vir = mem_dp->mem_vir + 2*old_clicks - mem_sp->mem_len;
```

• 检查新的数据段和堆栈段的大小是否适合进程的地址空间

```
  // 检查新的数据段和堆栈段的大小是否适合进程的地址空间
  /* Do the new data and stack segment sizes fit in the address space? */
  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
```

• 如果新的大小适合，更新内存映射表，涉及函数 sys_newmap 通知内核，注册内存段

```
// 如果新的大小适合，更新内存映射表
  if (r == OK) {
  int r2;
  if (changed && (r2=sys_newmap(rmp->mp_endpoint, rmp->mp_seg)) != OK)
      panic(__FILE__,"couldn't sys_newmap in allocate_new_mem", r2);
  return(OK);
  }
```

• 新的大小不适合或需要过多的页/段寄存器，恢复原始状态。

```
/* 新的大小不适合或需要过多的页/段寄存器。恢复原始状态。*/
 /* New sizes don't fit or require too many page/segment registers. Restore.*/
  if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
  if (changed & STACK_CHANGED) {
  mem_sp->mem_vir += delta;
  mem_sp->mem_phys += delta;
  mem_sp->mem_len -= delta;
  }
```

• 释放之前分配的新内存空间（起始位置，空间大小），涉及函数 free_mem 释放内存

```
// 释放之前分配的新内存空间
free_mem(old_base, old_clicks);
```

do_brk 函数计算数据段新的边界，根据给定的新虚拟地址 addr，计算出新的数据段长度，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。

• 如果不够，这就会涉及 allocate_new_mem 函数

```
  if (lower < gap_base) {
    z = allocate_new_mem(rmp, data_clicks, sp);
  if(z==ENOMEM) return(ENOMEM);
  return(OK);
    } /* data and stack collided */
```

计算数据段和堆栈段之间的安全间隙的空间，并将其存储在 gap_base 中。如果 lower 小于 gap_base，说明数据段和堆栈段发生了冲突。此时调用 allocate_new_mem 函数来为进程分配新的内存空间，以容纳扩展后的数据段和堆栈段。如果内存分配失败，返回 ENOMEM。如果分配成功，返回 OK。

• 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。

```
 /* Update data length (but not data orgin) on behalf of brk() system call. */
 old_clicks = mem_dp->mem_len;
 if (data_clicks != mem_dp->mem_len) {
 mem_dp->mem_len = data_clicks;
 changed |= DATA_CHANGED;
 }

 /* Update stack length and origin due to change in stack pointer. */
 if (delta > 0) {
 mem_sp->mem_vir -= delta;
 mem_sp->mem_phys -= delta;
 mem_sp->mem_len += delta;
 changed |= STACK_CHANGED;
 }

 /* Do the new data and stack segment sizes fit in the address space? */
 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 (changed && (r2=sys_newmap(rmp->mp_endpoint, rmp->mp_seg)) != OK)
    panic(__FILE__,"couldn't sys_newmap in adjust", r2);
 return(OK);
 }
```

2. 编译MINIX

2.1 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。

```
# ls
.ashrc   .ellepro.b1   .ellepro.e   .exrc   .profile   .vimrc
# cd ..
# ls
bin   boot   dev   etc   home   lib   mnt   root   sbin   tmp   usr   var
# cd home
# cc test1.c -o test1
# cc test2.c -o test2
"test2.c", line 20: (warning) implicit declaration of function exit
# ls
ast   beng   bin   test1   test1.c   test2   test2.c
# cd ..
# ls
bin   boot   dev   etc   home   lib   mnt   root   sbin   tmp   usr   var
# cd /usr/src/servers
# _
exec cc -c -I/usr/include  timers.c
exec cc -c -I/usr/include  table.c
exec cc -o fs -i main.o open.o read.o write.o pipe.o dmap.o \
        device.o path.o mount.o link.o super.o inode.o \
        cache.o cache2.o filedes.o stadir.o protect.o time.o \
        lock.o misc.o utility.o select.o timers.o table.o -lsys -lsysutil -ltime
rs
install -S 512w fs
cd ./rs && exec make - EXTRA_OPTS= build
exec cc -c -I/usr/include main.c
exec cc -c -I/usr/include manager.c
exec cc -o rs -i main.o manager.o  -lsys -lsysutil
install -S 16k rs
exec cc -c -I/usr/include service.c
exec cc -o service -i service.o -lsys
cd ./ds && exec make - EXTRA_OPTS= build
exec cc -c -I/usr/include main.c
exec cc -c -I/usr/include store.c
exec cc -o ds -i main.o store.o  -lsys -lsysutil
install -S 16k ds
cd ./init && exec make - EXTRA_OPTS= build
exec cc -c -I/usr/include -O -D_MINIX -D_POSIX_SOURCE init.c
exec cc -I/usr/include -O -D_MINIX -D_POSIX_SOURCE -o init -i init.o -lsysutil
install -S 192w init
# make install
```

1.2 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。

```
cc  -I. -D_MINIX -o generic/tcp.o -c generic/tcp.c
cc  -I. -D_MINIX -o generic/tcp_lib.o -c generic/tcp_lib.c
cc  -I. -D_MINIX -o generic/tcp_recv.o -c generic/tcp_recv.c
cc  -I. -D_MINIX -o generic/tcp_send.o -c generic/tcp_send.c
cc  -I. -D_MINIX -o generic/ip_eth.o -c generic/ip_eth.c
cc  -I. -D_MINIX -o generic/ip_ps.o -c generic/ip_ps.c
cc  -I. -D_MINIX -o generic/psip.o -c generic/psip.c
cc  -I. -D_MINIX -o minix3/queryparam.o -c minix3/queryparam.c
cc  -I. -D_MINIX -o sha2.o -c sha2.c
cc -o inet  buf.o clock.o inet.o inet_config.o \
        mnx_eth.o mq.o qp.o sr.o stacktrace.o \
        generic/udp.o generic/arp.o generic/eth.o generic/event.o \
        generic/icmp.o generic/io.o generic/ip.o generic/ip_ioctl.o \
        generic/ip_lib.o generic/ip_read.o generic/ip_write.o \
        generic/ipr.o generic/rand256.o generic/tcp.o generic/tcp_lib.o \
        generic/tcp_recv.o generic/tcp_send.o generic/ip_eth.o \
        generic/ip_ps.o generic/psip.o \
        minix3/queryparam.o sha2.o version.c -lsys -lsysutil
install -c inet /usr/sbin/inet
# cd ..
# ls
LICENSE    boot       drivers   include   lib   servers   tools
Makefile   commands   etc       kernel    man   test
# cd tools
# make hdboot
```

```
        ../servers/rs/rs \
        ../servers/ds/ds \
        ../drivers/tty/tty \
        ../drivers/memory/memory \
        ../drivers/log/log \
        ../servers/init/init
     text     data      bss      size
    24176     3384    44384     71944  ../kernel/kernel
    21920     3304    93940    119164  ../servers/pm/pm
    41536     5224  5019704   5066464  ../servers/fs/fs
     6848      840    20388     28076  ../servers/rs/rs
     3280      464     1808      5552  ../servers/ds/ds
    27072     5696    48104     80872  ../drivers/tty/tty
     6144   287784     3068    296996  ../drivers/memory/memory
     5968      572    63280     69820  ../drivers/log/log
     7056     2412     1356     10824  ../servers/init/init
   ------   ------   ------   -------
   144000   309680  5296032   5749712  total
exec sh mkboot hdboot
install image /dev/c0d3p0s0:/boot/image/3.1.2ar0
Done.
# make install_
```

1.3 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/3.1.2ar1;boot;}

```
     5968      572    63280     69820  ../drivers/log/log
     7056     2412     1356     10824  ../servers/init/init
   ------   ------   ------   -------
   144000   309680  5296032   5749712  total
exec sh mkboot hdboot
install image /dev/c0d3p0s0:/boot/image/3.1.2ar1
Done.
# shutdown


Broadcast message from root@192.168.184.129 (console)
Sun Jun  4 23:49:12 2023...

The system will shutdown NOW

Local packages (down): sshd  done.
Sending SIGTERM to all processes ...
MINIX will now be shut down ...
d0p0s0>newminix(5,start new kernel){image=/boot/image/3.1.2ar1;boot;}
d0p0s0>save
d0p0s0>menu_
```

1.4 然后回车，键入save 命令保存设置。5 为启动菜单中的选择内核版本的键(数字键，可选其他数字键)，3.1.2ar1 为在/usr/src/tools 目录中输入make install 之后生成的内核版本号，请记得在/usr/src/tools 中执行make install 命令之后记录生成的新内核版本号。

1.5 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。

```
Broadcast message from root@192.168.184.129 (console)
Sun Jun  4 23:49:12 2023...

The system will shutdown NOW

Local packages (down): sshd  done.
Sending SIGTERM to all processes ...
MINIX will now be shut down ...
d0p0s0>newminix(5,start new kernel){image=/boot/image/3.1.2ar1;boot;}
d0p0s0>save
d0p0s0>menu

Hit a key as follows:

    1  Start MINIX 3 (requires at least 16 MB RAM)
    2  Start Small MINIX 3 (intended for 8 MB RAM systems)
    5  start new kernel
5

Loading  Boot image 3.1.2a revision 1.

     cs       ds     text     data      bss    stack
0001000  0007000    24176     3384    44384        0  kernel
```

六、 实验结果

修改前：

运行./test1

```
# ./test1
incremented by 1, total 1 , result + inc 761
incremented by 2, total 3 , result + inc 4098
incremented by 4, total 7 , result + inc 4102
incremented by 8, total 15 , result + inc 4110
incremented by 16, total 31 , result + inc 4126
incremented by 32, total 63 , result + inc 4158
incremented by 64, total 127 , result + inc 4222
incremented by 128, total 255 , result + inc 4350
incremented by 256, total 511 , result + inc 4606
incremented by 512, total 1023 , result + inc 5118
incremented by 1024, total 2047 , result + inc 6142
incremented by 2048, total 4095 , result + inc 8190
incremented by 4096, total 8191 , result + inc 12286
incremented by 8192, total 16383 , result + inc 20478
incremented by 16384, total 32767 , result + inc 36862
incremented by 32768, total 65535 , result + inc 69630
#
```

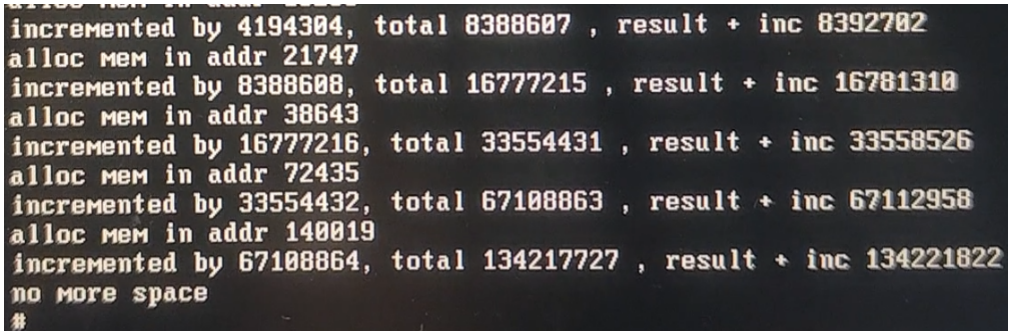
运行./test2

```
# ./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
# _
```

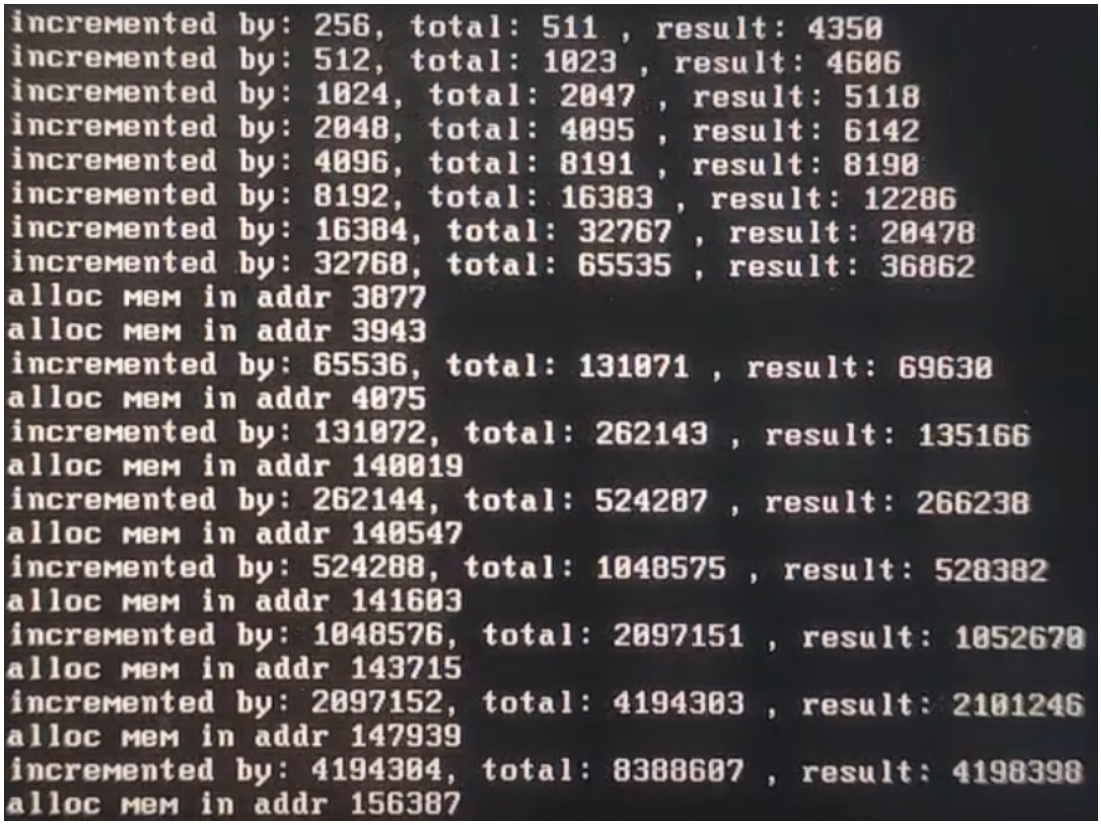
修改后：

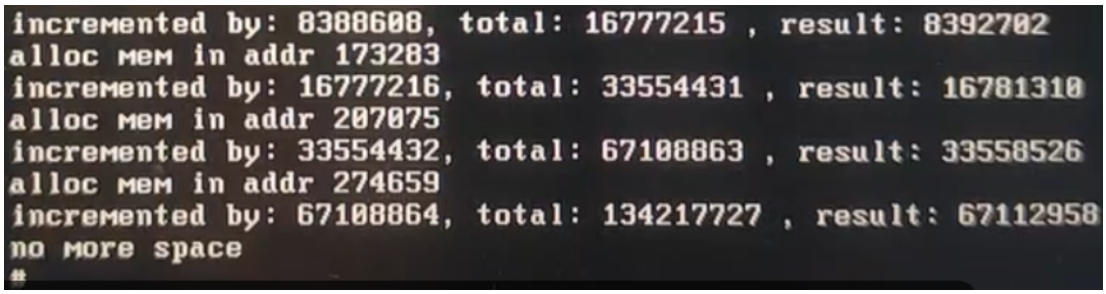
运行./test1

```
incremented by 64, total 127 , result + inc 4222
incremented by 128, total 255 , result + inc 4350
incremented by 256, total 511 , result + inc 4606
incremented by 512, total 1023 , result + inc 5118
incremented by 1024, total 2047 , result + inc 6142
incremented by 2048, total 4095 , result + inc 8190
incremented by 4096, total 8191 , result + inc 12286
incremented by 8192, total 16383 , result + inc 20478
incremented by 16384, total 32767 , result + inc 36862
incremented by 32768, total 65535 , result + inc 69630
alloc mem in addr 2852
alloc mem in addr 2918
incremented by 65536, total 131071 , result + inc 135166
alloc mem in addr 3050
incremented by 131072, total 262143 , result + inc 266238
alloc mem in addr 3314
incremented by 262144, total 524287 , result + inc 528382
alloc mem in addr 5907
incremented by 524288, total 1048575 , result + inc 1052670
alloc mem in addr 6963
incremented by 1048576, total 2097151 , result + inc 2101246
alloc mem in addr 9075
incremented by 2097152, total 4194303 , result + inc 4198398
alloc mem in addr 13299
```



运行./test2





七、 总结

本次实验是操作系统课程的最后一个课程设计，实验目的在于熟悉 Minix 操作系统的进程管理和学习 Unix 风格的内存管理。我修改了 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。 了解了其中需要修改的 alloc_men 函数、do_brk 函数、adjust 函数以及新增的 allocate_new_men 函数的实现逻辑。