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**：2018	**上机实践成绩**：
指导教师：翁楚良	**姓名**：池欣宁	**学号**：10185501409
上机实践名称：内存管理		**上机实践日期**：
上机实践编号：	**组号**：	**上机实践时间**：

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，该 brk 系统调用会

1. 给该进程一个更大的内存空间，
2. 并将原来空间中的数据复制到新分配的内存空间。
3. 释放原来的内存空间
4. 通知内核映射新分配的内存段。

三、 使用环境

Minix3.1.2

四、 实验过程

Task1.Minix3.1.2 环境的配置：

1. 下载源码安装配置 Vmware Fusion 虚拟机
2. 兼容性选择版本 6.
3. 进行虚拟机设置，要选择 AMD_LANCE
4. 进行虚拟机密码的设置
5. 更改配置文件，使虚拟机可以使用相应的 ftp 链接
6. packman 安装相关依赖
7. 注意：在实验过程中要使用 shutdown 命令和 boot d0p0 进行的开关与重启，否则容易出现镜像文件错误的问题。
8. filezilla 链接虚拟机使用 ftp 传文件

我在实验的过程中，一开始一直遇到无法使用 mac 进行传文件的问题，后来发现是由于我在 boot d0p0 后选择的内核为 custom 内核，导致无法链接，再将内核更换为 regular 以后，macos 也可以链接到相关虚拟机了。

Task2.修改内存分配函数

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

整体思路中，实在所有 hp->h_len>clicks 的空闲中，寻找到其中最小的，

我一开始，最小的使用的是 min=99999,后来发现 minix 操作系统和平时联系不一样，如果单纯的赋数，后很容易出现错误，与平时 C 语言的作业有很大的不同。

```
while (hp != NIL_HOLE && hp->h_base < swap_base) {
if ((hp->h_len >= clicks)&&flag==0) {
/* We found a hole that is big enough. Use it. */
min=hp->h_len;
flag=1;
}
if((hp->h_len >= clicks)&&flag==1){
if (hp->h_len<min)
min=hp->h_len;
}
prev_ptr = hp;
hp = hp->h_next;
}
```

找到了合适的 best-fit 空闲块后类比原 first-fit 进行赋值即可。

```
while (hp != NIL_HOLE && hp->h_base < swap_base) {
if(hp->h_len==min)
{
old_base = hp->h_base; /* remember where it started */
hp->h_base += clicks; /* bite a piece off */
hp->h_len -= clicks; /* ditto */

/* Remember new high watermark of used memory. */
if(hp->h_base > high_watermark)
high_watermark = hp->h_base;
```

```
/* Delete the hole if used up completely. */
if (hp->h_len == 0) del_slot(prev_ptr, hp);
```

```
/* Return the start address of the acquired block. */
return(old_base);
}
prev_ptr = hp;
hp = hp->h_next;

}
```

Task3.修改 brk 系统调用的主体，adjust 函数，实质上是新增 alloc_new_mem 函数进行功能的扩展。

在 minix 系统中，brk 系统调用，主要由 adjust 函数完成，所以修改 adjust 即是修改了 brk 系统调用。

brk 系统调用本身的逻辑为：

A. Do_brk 函数计算函数边界

B. 调用 adjust 函数，使用 adjust 函数来计算当前程序的空闲空间是否足够分配，

C. 如果足够分配，则进行相应的内存映射的修改。（原 adjust 函数的逻辑）

D. 如果不够分配，则调用 allocate_new_mem 函数申请新的足够大的内存空间），进行内容的拷贝和相应的内核映射的修改。

因为有两种情况，所以是在判断数据段和栈段是否 collide 处增加 if 的判断条件。

```
if (lower < gap_base) {
if (allocate_new_mem(rmp,(phys_clicks)(mem_sp-
>mem_vir+mem_sp->mem_len-mem_dp->mem_vir)) == NOT_OK)
return(ENOMEM);/* data and stack collided */
printf("allocate_new_mem succeed2!\n");
}
```

Alloc_new_mem 函数的具体逻辑:

将 rmp(which process)和 memory_clicks(original memory)

由下图可得：

原先的内存为：栈段底指针-数据段底指针，

即：

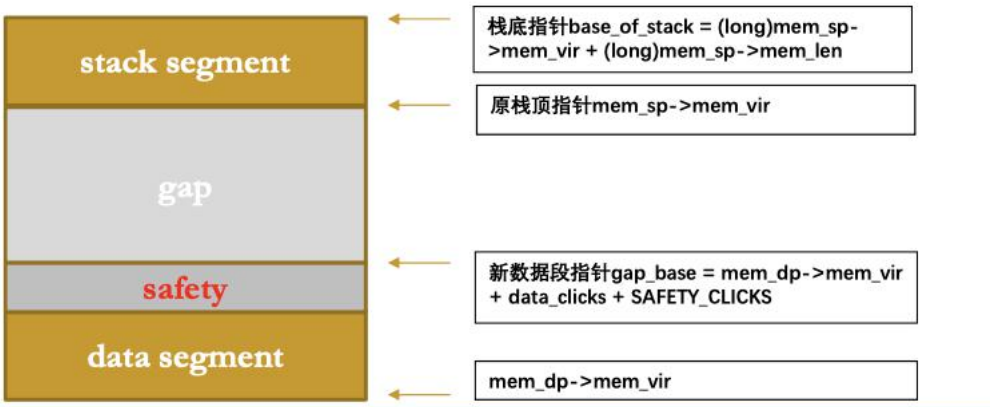
Memory_clicks=mem_sp->mem_vir+mem_sp->mem_len-mem_dp->mem_vir;

Mem_dp 和 mem_sp 结构体的结构类似，保存了栈/内存段的物理/虚拟起始地址和各自的长度：

```
struct mem_map {
```

```
vir_clicks mem_vir; /* virtual address */
phys_clicks mem_phys; /* physical address */
vir_clicks mem_len; /* length */
};
```

这里 phys_clicks 和 vir_clicks 都是 long int 的 typedef.



adjust 函数的核心逻辑为：

1. 申请足够大的内存空间(alloc_mem 函数)
2. 将程序现有的数据段和栈段的内容分别拷贝至新内存区域的底部（栈在顶）和顶部（数据段在底）
3. 通知内核程序的影像发生了变化；
4. 释放原的内存
5. Return adjust->return do_brk;

根据 adjust 函数的逻辑：

1. 申请足够大的内存空间(alloc_mem 函数)

足够大的内存空间：2 倍于原内存空间，原内存空间定义可见上文。

```
old_memory_size=memory_clicks;
new_memoey_size =2*memory_clicks;
```

首先要申请足够大的内存空间：其中 alloc_mem 函数返回值为申请到的空间的起始指针：

```
new_base_pointer=alloc_mem(new_memoey_size);
```

判断返回值，即判断是否申请成功：

```
if(new_base_pointer==NO_MEM)
{
printf("no mem!\n");
return(NOT_OK);
}
```

原栈段起始地址： （物理地址）

```
old_stack_base=mem_sp->mem_phys;
```

原数据段起始地址： （物理地址）

```
old_data_base=mem_dp->mem_phys;
```

要拷贝到的新的数据段的起始地址（物理地址），是 alloc_mem 返回的地址：

```
new_data_base=new_base_pointer;
new_stack_base=new_base_pointer+new_memoey_size-
(phys_clicks)mem_sp->mem_len;/*vir_clicks*/
```

栈的物理地址计算，同上，见图。(注意栈是向下生长）；

2.使用 sys_copy 进行拷贝：

在进行 sys_copy 拷贝之前，要使用 sys_memset 来对新开辟的内存进行初始化，否则会发生 memory_core_dump 的问题，估计是 C 语言访问空指针...

```
PUBLIC int sys_memset(unsigned long pattern, phys_bytes
base, phys_bytes bytes)
{
/* Zero a block of data. */
message mess;

if (bytes == 0L) return(OK);
```

```
mess.MEM_PTR = (char *) base;
mess.MEM_COUNT = bytes;
mess.MEM_PATTERN = pattern;
```

```
return(_taskcall(SYSTASK, SYS_MEMSET, &mess));
}
```

这里 sys_copy 函数的使用方法，可以类比 minix3.1.2 源码的使用方法：

在/servers/pm/forkexit.c 函数中：

```
/* 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);
```

这里,child_abs,是子进程的地址（dst)，parent_abs 是要被拷贝的父进程的地址(fork 函数的逻辑)

所以，要获取栈段的 dst/src 地址，也要获得数据段的 dst/src 地址：

```
data_bytes=(phys_bytes)mem_dp->mem_len<<CLICK_SHIFT;
stck_bytes=(phys_bytes)mem_sp->mem_len<<CLICK_SHIFT;
old_data_abs=(phys_bytes)old_data_base<<CLICK_SHIFT;
new_data_abs=(phys_bytes)new_data_base<<CLICK_SHIFT;
old_stack_abs=(phys_bytes)old_stack_base<<CLICK_SHIFT;
new_stack_abs=(phys_bytes)new_stack_base<<CLICK_SHIFT;
#define NONE 0
s=sys_abscopy(old_data_abs,new_data_abs,data_bytes);
if (s < 0 ) panic(__FILE__,"allocate_new_mem can't
copy",s);
```

因为这里如果不 define NONE 0，编译就过不去，头秃。。。

```
#define sys_abscopy(src_phys, dst_phys, bytes) \
sys_physcopy(NONE, PHYS_SEG, src_phys, NONE, PHYS_SEG,
dst_phys, bytes)
```

但看了 sys_abscopy 原型就这样啊 Orz

3.修改内核映射的相关变化：

整体变化的逻辑是：

A.数据段的物理起始地址变更为 new_data_base(new_base_pointer);

B.数据段的虚拟起始地址不变；

C.栈段的物理起始地址变更为 new_stack_basel

D.栈段的虚拟起始地址为 mem_dp->mew_vir+new_memory_size-mem_sp->len;

4. 释放原内存：

```
free_mem(old_data_base,old_memory_size);
```

5. 返回 OK（do_brk)

在原 adjust 函数中，对于数据段的更新，如果改成了这种分支形式，

则：

```
if (lower < gap_base) {
```

```
if (allocate_new_mem(rmp,(phys_clicks)(mem_sp-
>mem_vir+mem_sp->mem_len-mem_dp->mem_vir)) == NOT_OK)
return(ENOMEM);/* data and stack collided */
printf("allocate_new_mem succeed2!\n");
changed |= DATA_CHANGED;
changed |= STACK_CHANGED;
}
else{
```

则不会有相应的更新：

如果去掉了 if 分支，则测试结果正确：

```
ncremented by: 4, total: 7 , result: 4098
ncremented by: 8, total: 15 , result: 4102
ncremented by: 16, total: 31 , result: 4110
ncremented by: 32, total: 63 , result: 4126
ncremented by: 64, total: 127 , result: 4158
ncremented by: 128, total: 255 , result: 4222
截屏 ented by: 256, total: 511 , result: 4350
ncremented by: 512, total: 1023 , result: 4606
ncremented by: 1024, total: 2047 , result: 5118
ncremented by: 2048, total: 4095 , result: 6142
ncremented by: 4096, total: 8191 , result: 8190
ncremented by: 8192, total: 16383 , result: 12286
ncremented by: 16384, total: 32767 , result: 20478
ncremented by: 32768, total: 65535 , result: 36862
llocate_new_mem succeed2!
llocate_new_mem succeed2!
llocate_new_mem succeed2!
llocate_new_mem succeed2!
llocate_new_mem succeed2!
llocate_new_mem succeed2!
llocate_new_mem succeed2!
llocate_new_mem succeed2!
llocate_new_mem succeed2!
o mem!
```

文件编写完成以后，进行相关编译

1. 进入/usr/src/servers 目录，输入 make image /make install
2. 进入/usr/src/tools 目录，输入 make hdboot,/make install;

然后进行测试

测试结果如下：

```
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
截屏 te_new_mem succeed!
allocate_new_mem succeed!
incremented by: 65536, total: 131071 , result: 69630
allocate_new_mem succeed!
incremented by: 131072, total: 262143 , result: 135166
allocate_new_mem succeed!
incremented by: 262144, total: 524287 , result: 266238
allocate_new_mem succeed!
incremented by: 524288, total: 1048575 , result: 528382
allocate_new_mem succeed!
incremented by: 1048576, total: 2097151 , result: 105267
allocate_new_mem succeed!
incremented by: 2097152, total: 4194303 , result: 210124
allocate_new_mem succeed!
incremented by: 4194304, total: 8388607 , result: 419839
allocate_new_mem succeed!
incremented by: 8388608, total: 16777215 , result: 83927
no mem!
```

```
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
截屏 te_new_mem succeed!
allocate_new_mem succeed!
incremented by 65536, total 131071
allocate_new_mem succeed!
incremented by 131072, total 262143
allocate_new_mem succeed!
incremented by 262144, total 524287
allocate_new_mem succeed!
incremented by 524288, total 1048575
allocate_new_mem succeed!
incremented by 1048576, total 2097151
allocate_new_mem succeed!
incremented by 2097152, total 4194303
allocate_new_mem succeed!
incremented by 4194304, total 8388607
allocate_new_mem succeed!
incremented by 8388608, total 16777215
no mem!
```

五、总结

这次 project 的过程异常坎坷，感觉整体的思路逻辑比较清楚，但由于很多细节无法 debug，需要反复重装虚拟机。

一开始对于 macosfillizila 连接的问题，一直没有发现是因为选择了 custom 内核的原因，一直以为无法连接上去，导致我翻出了我充满广告弹窗的古早的 windows 并重新下了 vmware workstation 重新开始。

后来发现是因为内核选择的缘故，开始转战到原来的 mac 电脑。

一开始因为不是很熟悉 minix 关于内存管理的相关代码，所以先去修改了 firstfit 函数，一开始把 C 语言的求最小值思维偷懒的带入，却遗忘了系统中最大值 99999 还远远不够。

修改完 alloc.c 函数以后，开始了关于 break.c 函数的漫长修改。

一开始对于 cc 编译器不是很理解，经常报错，后来发现了 cc 编译器对于变量最初定义的规范要求很高，很多变量的初始化需要放在函数开始，不然很容易报错。

后来，又因为没有自信看指导书上的要求，把测试放在/root 文件夹下，经常导致 coredump 而需要重新装虚拟机和环境，而不是内核。

最后，将测试放在了/home 文件下才开始乖乖走上正规。

总结而言，内存管理对于 minix 来说，要注意栈指针的变化，因为数据段指针是自底向上增长的，地址逐渐增大，而栈指针则是自顶向下生长的，所以整体的逻辑需要倒过来想。

其次，很多函数的使用方法，可以直接参照 minix 源码其他部分的使用情况，很多宏定义也可以根据 vscode 进行搜索解决。

最后，自己反反复复一直报错的部分是处于如何通知内核映射相应的变化：

一开始的原内存，设置成 base_of_stack-mem_dp->vir 会由于变量类型的原因，而没有真正的开辟到新的内存，因为直接在 NO_MEM 处返回。

但是这次在 project 完成的过程中，反复实践迭代，使我受益匪浅。