操作系统 实验报告4

温兆和 10205501432

实验背景

在本次实验中，我们需要改进 `brk` 系统调用的实现，使得进程原来的数据段+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。

实验过程

在本次实验中，我们修改了 `/usr/src/servers/pm/alloc.c` 和 `/usr/src/servers/pm/break.c` 这两个文件中的代码。

I. `alloc.c`

在 `alloc.c` 中，我们修改 `alloc_mem` 函数，在分配内存时找遍整个内存块列表，找到最优块（即所有大小超过需要的大小的内存块中最小的块）：

In [ ]:
```
/*===========================================================================*
 *                              alloc_mem                                    *
 *===========================================================================*/
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks;             /* amount of memory requested */
{
/* 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_ptr,*prev_of_best_ptr;
  phys_clicks old_base;

  do {
        prev_ptr = NIL_HOLE;
        best_ptr = NIL_HOLE;
        prev_of_best_ptr = NIL_HOLE;
    hp = hole_head;
    while (hp != NIL_HOLE && hp->h_base < swap_base) {
        if (hp->h_len >= clicks) {
            if(best_ptr==NIL_HOLE)
            {
                best_ptr = hp;
                prev_of_best_ptr = prev_ptr;
```

```
            }
            else
            {
                if(hp->h_len<best_ptr->h_len)
                {
                    best_ptr = hp;
                    prev_of_best_ptr = prev_ptr;
                }
            }
        }

        prev_ptr = hp;
        hp = hp->h_next;
    }
    if(best_ptr!=NIL_HOLE)
    {
        hp = best_ptr;
        prev_ptr = prev_of_best_ptr;
        /* We found a hole that is big enough.  Use it. */
        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);
    }
  } while (swap_out());/* try to swap some other process out */
  return(NO_MEM);
}
```

II. break.c

在 break.c 中，我们首先需要修改 adjust 函数并新增 allocate_new_mem 函数，使得当数据段的上界和栈段的下界碰撞时，系统不再报错，而是为进程申请新的、更大的内存块，将程序现有的数据段和堆栈段的内容拷贝至新内存区域并通知内核程序的映像发生 了变化。

在 adjust 函数中，如果调用 brk 后数据段和栈段即将发生重叠，则直接调用 allocate_new_mem ，为进程分配更大的内存空间：

In [ ]:
```
/*===========================================================================*
 *                                adjust                                     *
 *===========================================================================*/
PUBLIC int adjust(rmp, data_clicks, sp)
register struct mproc *rmp;/* whose memory is being adjusted? */
vir_clicks data_clicks;/* how big is data segment to become? */
vir_bytes sp;/* new value of sp */
```

```
{
/* See if data and stack segments can coexist, adjusting them if need be.
 * Memory is never allocated or freed.  Instead it is added or removed from the
 * gap between data segment and stack segment.  If the gap size becomes
 * negative, the adjustment of data or stack fails and ENOMEM is returned.
 */

  ......

  /* 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;
  if (lower < gap_base) /*return(ENOMEM);*/ /* data and stack collided */
  {
      res=allocate_new_mem(rmp,data_clicks,delta,(phys_clicks)(rmp->mp_seg[S].mem_v
      return res;
  }

  ......
}
```

在 `allocate_new_mem` 函数中，先把新的内存块的大小设置为新的数据段大小和可能的最大栈段大小之和，并获得 `rmp` 进程的栈段指针和数据段指针：

In [ ]:
```
/*===========================================================================*
 *                                allocate_new_mem
 *===========================================================================*/
 PUBLIC int allocate_new_mem(rmp,data_clicks,delta,clicks)
register struct mproc *rmp;
phys_clicks data_clicks;
long delta;
phys_clicks clicks;
{
    register struct mem_map *mem_sp,*mem_dp,*mem_cp;
    int change = 0,d,s,r,ft;
    phys_bytes databytes,stackbytes,new_address_data_byte,old_address_data_byte,new
    phys_clicks cur_data_clicks,new_address_data,old_address_data,old_clicks,new_cl
    /*set the size of new memory space and save the size of old memory space*/
    new_clicks = data_clicks+mem_sp->mem_len+delta;
    old_clicks = clicks;
    /*set pointer to data segment map & stack segment map*/
    mem_sp=&rmp->mp_seg[S];
    mem_dp=&rmp->mp_seg[D];
```

调用 `alloc_mem` 函数，重新分配一块大小为 `new_clicks` 的内存块，返回的地址正好就是新内存空间数据段的起点：

In [ ]:
```
/*allocate a new space of new_clicks for rmp*/
    if((new_address_data=alloc_mem(new_clicks)) == NO_MEM)
    {
      return(ENOMEM);
    }
```

把原来内存空间的数据段和栈段的大小的单位从click转为byte：

In [ ]:
```
/*save the size of old stack and data*/
    databytes=(phys_bytes)mem_dp->mem_len << CLICK_SHIFT;
    stackbytes=(phys_bytes)mem_sp->mem_len << CLICK_SHIFT;
```

计算新的栈段的地址并保存旧的栈段、数据段地址。值得一提的是，这里的栈段地址是栈段最下端的地址，因为拷贝内容时的输入是最底端的地址。

In [ ]:
```
/*work out the address of the old and new data and stack segment*/
    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;
```

把所有地址的单位从click转为byte：

In [ ]:
```
/*change click into byte*/
    new_address_data_byte=(phys_bytes)new_address_data << CLICK_SHIFT;
    new_address_stack_byte=(phys_bytes)new_address_stack << CLICK_SHIFT;
    old_address_data_byte=(phys_bytes)old_address_data << CLICK_SHIFT;
    old_address_stack_byte=(phys_bytes)old_address_stack << CLICK_SHIFT;
```

把新的内存空间全部置为零，并将原有的数据段和栈段拷贝到新的内存空间中去：

In [ ]:
```
/*fill the new memory space with 0*/
    sys_memset(0, new_address_data_byte, (new_clicks << CLICK_SHIFT));
    /*copy the data and stack segment to the new memory space*/
    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);
    }
```

修改程序映像中的数据段和栈段地址：

In [ ]:
```
/*change the physical address of data and stack segment and the virtue address of t
    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;
```

判断并记录数据段和栈段的大小是否发生了变化：

In [ ]:
```
/*save current size of data segment*/
    cur_data_clicks= mem_dp->mem_len;
    /*adjust the size of data segment*/
    if(data_clicks != mem_dp->mem_len)
    {
```

```
      mem_dp->mem_len = data_clicks;
      change |= DATA_CHANGED;
    }
    /*adjust the size of stack segment*/
    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是合适的地址空间：

In [ ]:
```
/*judge whether the new memory size 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].me
#else
    r = (rmp->mp_seg[D].mem_vir + rmp->mp_seg[D].mem_len > rmp->mp_seg[S].mem_vir)
#endif
```

如果新的栈段数据段大小是合适的地址空间，通知内核注册新的内存段并释放原有地址空间：

In [ ]:
```
/*if the new memory size fit in the new address space,free the old memory space*/
    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);
    }
```

否则，把进程中的数据段大小和栈段地址恢复原样：

In [ ]:
```
/*if the new memory size don't fit in the new address space, restore the size and a
    else
    {
        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);
}
```

实验结果

随后，编译内核并执行以下两段测试代码。

I. `test1.c`

`test1.c` 测试代码是这样的：

In [ ]:
```c
#include <stdio.h>
#include <unistd.h>

int inc = 1;
int total = 0;
char *sbrk(int incr); /* should be in unistd, but isn't */
char *result;

int main(int argc, int **argv)
{
       while (((int)(result = sbrk(inc))) >= 0)
       {
              total += inc;
              printf("incremented by %d, total %d , result + inc %d\n", inc, total,
              inc += inc;
       }
       return 0;
}
```

其中，`inc` 变量从一开始每次迭代翻一倍。在每次迭代中，通过 `sbrk` 函数把数据段的上界上移 `inc`，并打印每次数据段上界的增量、数据段总增量和此时数据段上界的地址。值得注意的是，`sbrk` 函数返回的是调用 `sbrk` 函数前数据段的上界，所以 `inc + (int)result` 恰好就是当前数据段上界的地址。

运行结果如下图所示：

```
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
incremented by 65536, total 131071 , result + inc 135166
incremented by 131072, total 262143 , result + inc 266238
incremented by 262144, total 524287 , result + inc 528382
incremented by 524288, total 1048575 , result + inc 1052670
incremented by 1048576, total 2097151 , result + inc 2101246
incremented by 2097152, total 4194303 , result + inc 4198398
incremented by 4194304, total 8388607 , result + inc 8392702
incremented by 8388608, total 16777215 , result + inc 16781310
incremented by 16777216, total 33554431 , result + inc 33558526
incremented by 33554432, total 67108863 , result + inc 67112958
incremented by 67108864, total 134217727 , result + inc 134221822
#
```

II. `test2.c`

`test2.c` 测试代码是这样的：

In [ ]:
```c
#include <stdio.h>
#include <unistd.h>

int inc = 1;
int total = 0;
char *sbrk(int incr);
char *result;
int i;

int main(int argc, int **argv)
{
    while (((int)(result = sbrk(inc))) > 0)
    {
        for (i = 0; i < inc; i++)
            result[i] = 0x12;
        total += inc;
        printf("incremented by: %d, total: %d , result: %d\n", inc, total, (int)res
        inc += inc;
    }
    exit(0);
    return 0;
}
```

与第一个测试代码不同的是，这里每次要把新分配的数据段用 `0x12` 填满。打印的内容也有所不同：这里打印的第三个内容是 `sbrk` 函数的返回值 `(int)result` ，也就是调用 `sbrk` 函数前数据段的上界。

运行结果如下图所示：

```
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
incremented by: 67108864, total: 134217727 , result: 67112958
#
```

值得注意的是，当数据段的上界 `brk` 向上增加了8的时候，第一个测试代码返回的地址是4110，而第二个测试代码返回的地址是4102，刚好相差8，根据 `sbrk` 函数的原理，应该分别是调用 `sbrk` 函数之后和 `sbrk` 函数之前的数据段上界。