

操作系统 第四章 进程管理

4.7 进程的动态内存及其管理策略

动态内存

- Unix V6++进程的虚地址空间中，bss段之下、初始用户栈之上的部分是进程可用的动态内存。
- 应用程序刚开始运行时，动态内存没有数据，与之对应的PTE为NULL。访问这个区域，CPU会抛出14#缺页异常，除非扩展堆栈，否则进程会用SIGSEGV信号杀死进程。这种内存访问地址是“野指针”。访问野指针，shell报段错误。
- 动态内存，先分配，后使用。与之相对的，代码段和数据段是进程刚生出来就有的，是虚空间中的静态内存。



动态内存分配

(1) 隐式分配。根据子程序调用的嵌套深度，进程的用户栈（stack）自动扩展。栈扩展是缺页异常引发的，进程不需要执行显式的系统调用。

(2) 显式分配。应用程序使用**malloc函数**为进程申请动态内存。需要时，malloc函数执行系统调用**brk**扩展用户数据段，将整数个虚拟页面追加在数据段尾部。扩展出的所有空间组成**堆（heap）**，后者是一块长度可变的连续内存区域。

- 堆空间的维护：Unix V6++系统的动态内存分配器（malloc函数和free函数）使用可变分区动态内存分配算法管理堆空间（与内核物理空间分配算法一致）。
- 如果找不到足够尺寸的空闲内存片，malloc函数执行brk系统调用扩展堆空间，一次 3 页（PAGE_SIZE 12288字节）



动态内存回收

- (1) 扩展出的栈空间不回收。
- (2) free()函数释放malloc()申请的动态内存。倘若free操作导致堆底部（高地址端）出现大量连续空闲页面（6页），函数执行sbrk系统调用缩小数据段。

- 进程运行过程中
 - 代码段尺寸不变，内容不变
 - 只读数据段尺寸不变，内容不变
 - 数据段尺寸不变（带初值的全局变量 和 不带初值的全局变量）
 - 堆（heap）按需扩展，容纳应用程序运行所需的全部动态内存
 - 栈（stack）按需扩展，每调用一个子程序，压入一个栈帧，栈就长长一截；子程序返回，弹出栈帧，栈缩短。
- 堆，向高地址端扩展；栈，向低地址方向延申。
- 堆底碰到栈顶，用户空间耗尽，OOM（out of memory）



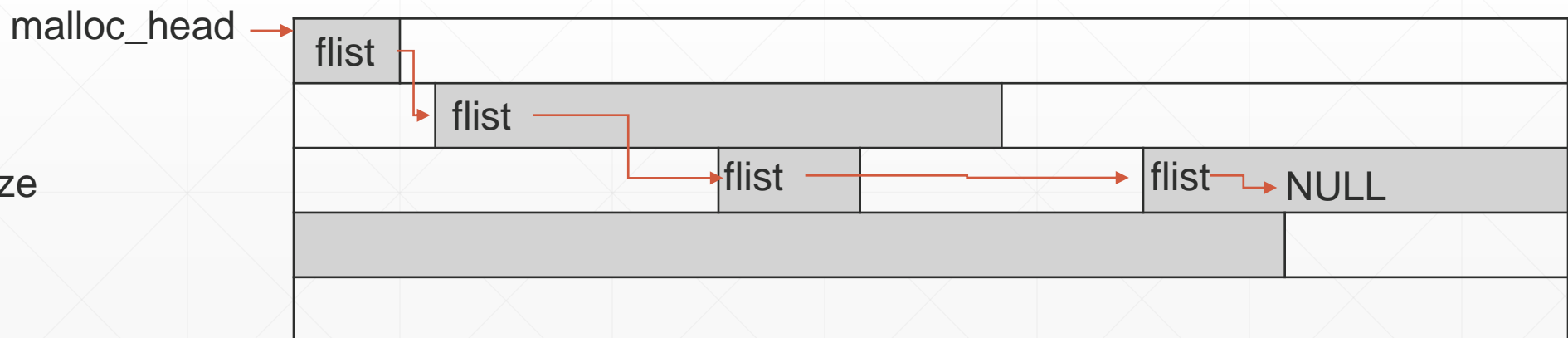
堆空间管理

- 堆空间，内存片、空闲区交错。
- 内存片是已分配的动态内存。按起始地址，形成单向链表。内存片控制块 flist，8字节，存放在内存片首部。

```
typedef struct flist {
    unsigned int size; // 长度
    struct flist *nlink; // 下个内存片的首地址
};
```



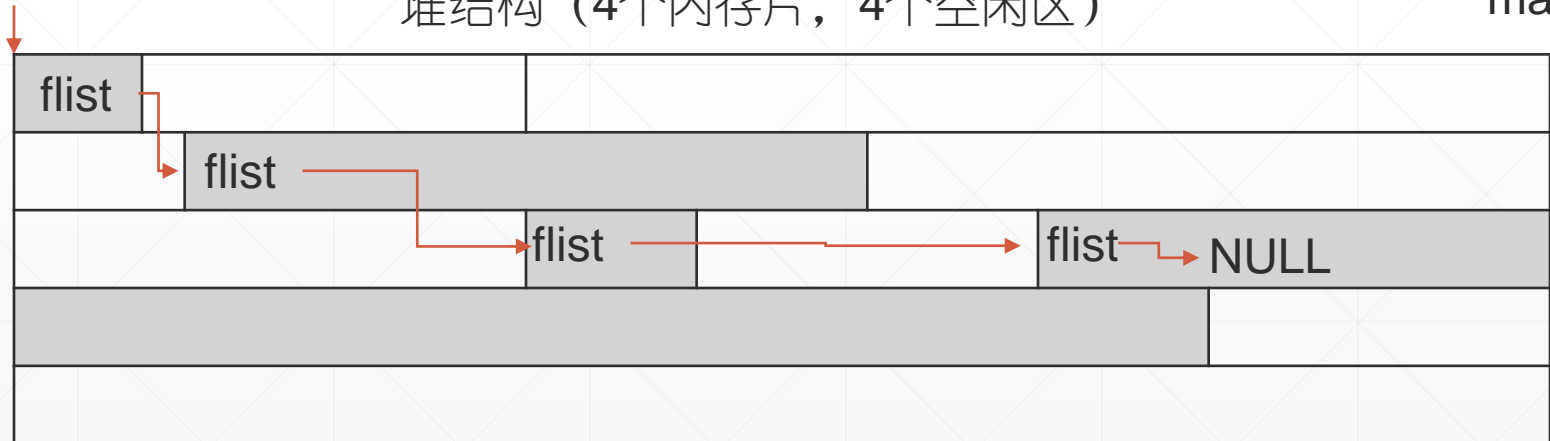
堆结构（4个内存片，4个空闲区）



堆空间

malloc_begin

堆结构 (4个内存片, 4个空闲区)



malloc_begin

malloc_end

malloc_end

进程的用户空间



首次 malloc 启用堆空间

- 例：现运行进程 PA，1页代码，1页数据，没有只读数据 和 bss，1页堆栈。代码段起始 0x401000。

char *p = malloc(4); (1) 情景分析 (2) 指针 p 的值是多少？

(1) 初始化堆空间

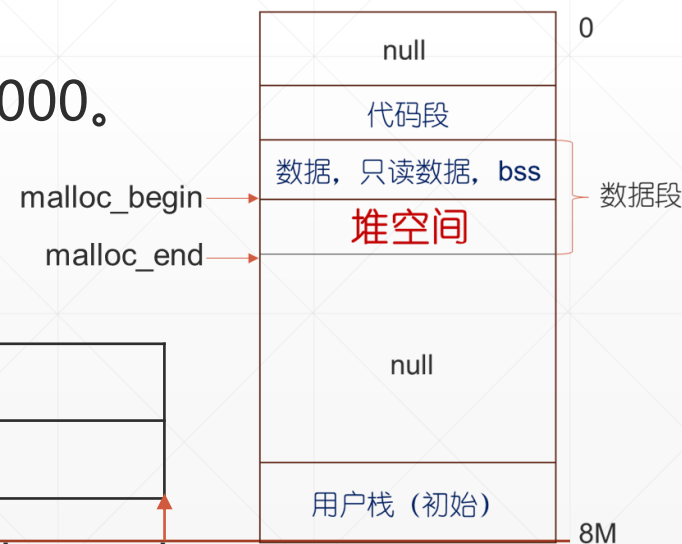
执行brk系统调用，为数据段追加 12k 字节。

malloc_begin = malloc_head = 0x403000, malloc_end = 0x406000。

malloc_begin指向的位置，写一个flist哑元 (dump) 。

```
typedef struct flist {
    unsigned int size; // 8字节
    struct flist *nlink; // null
};
```

malloc_begin , malloc_head

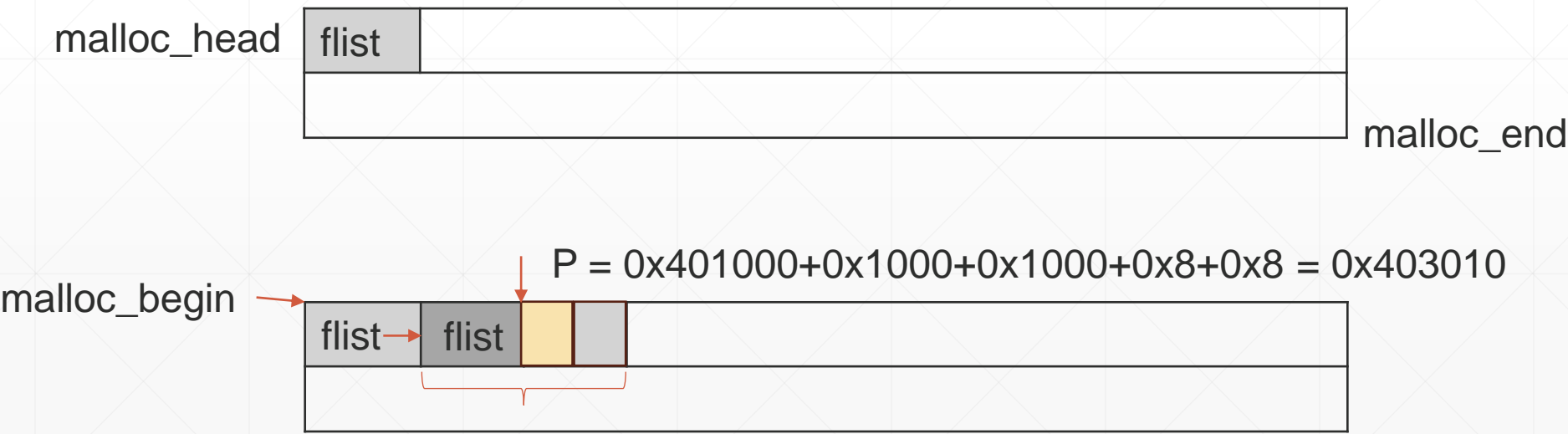


(2) 动态内存分配

遍历内存片单向链表，搜索可以容纳 净容量是4字节的空闲区，其尺寸是：8+4 = 12字节。

8字节对齐，size = 16字节。

内存片链表尾部追加一个元素，长16字节。P指针是malloc()的返回值。



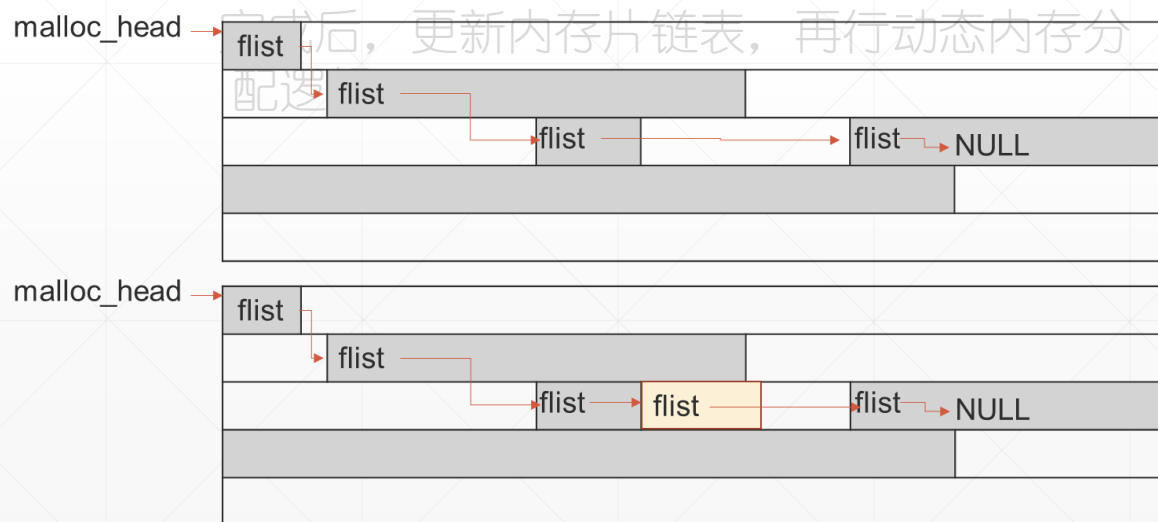
malloc (动态内存分配) , 更一般的逻辑 1

记 所需内存片的尺寸是 **size**

搜索内存片链表, 找足够大的空闲区 **temp**

(1) 找到, 空闲区首部写一个 **flist**, 插入内存片链表
返回分配给应用程序的内存块首地址:
temp + sizeof(struct flist)。

(2) 没找到, 执行 **brk** 系统调用, 扩展堆空间 3 个页面。



```
struct flist* iter = malloc_head;
while(iter->nlink)
{
    if ((int)(iter->nlink) - (int)iter - iter->size >= size)
    { // 找到
        struct flist *temp = (char *)iter + (iter->size);
        temp->nlink = iter->nlink;
        iter->nlink = temp;
        temp->size = size;
        return (char *)temp + sizeof(struct flist);
    }
    iter = iter->nlink;
}
```

malloc (动态内存分配) , 更一般的逻辑 2

记 所需内存片的尺寸是 size

搜索内存片链表, 找足够大的空闲区 temp

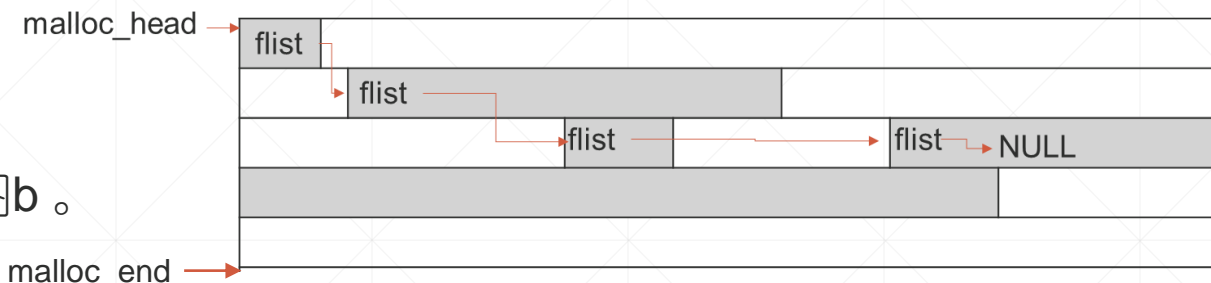
(1) 找到, 空闲区首部写一个 flist, 插入内存片链表

返回分配给应用程序的内存块首地址:

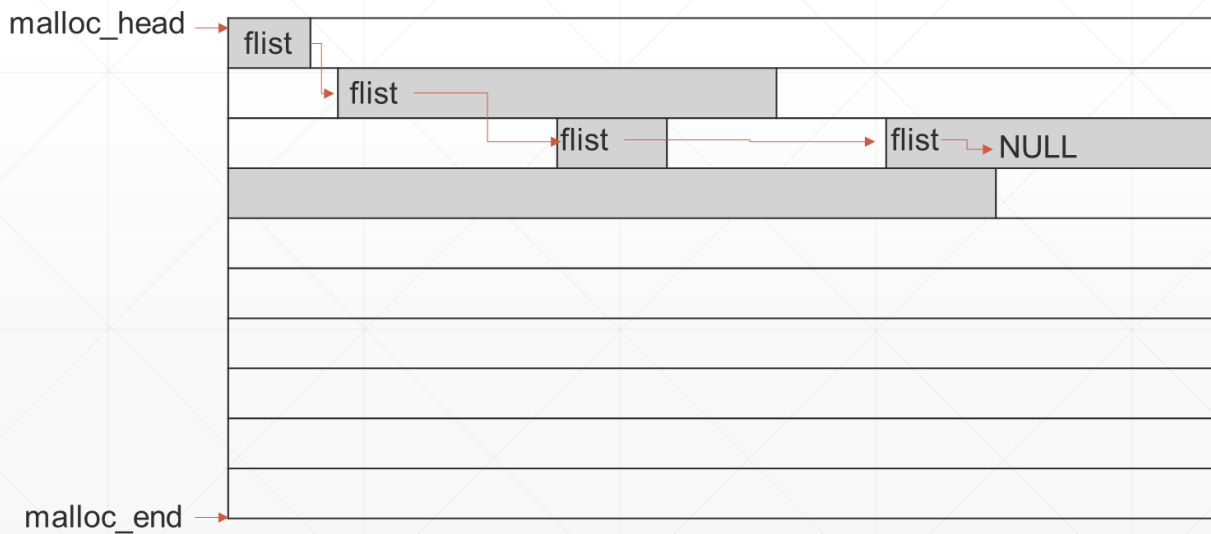
temp + sizeof(struct flist).

(2) 没找到, 执行brk系统调用, 扩展堆空间 3个页面, 子图b。

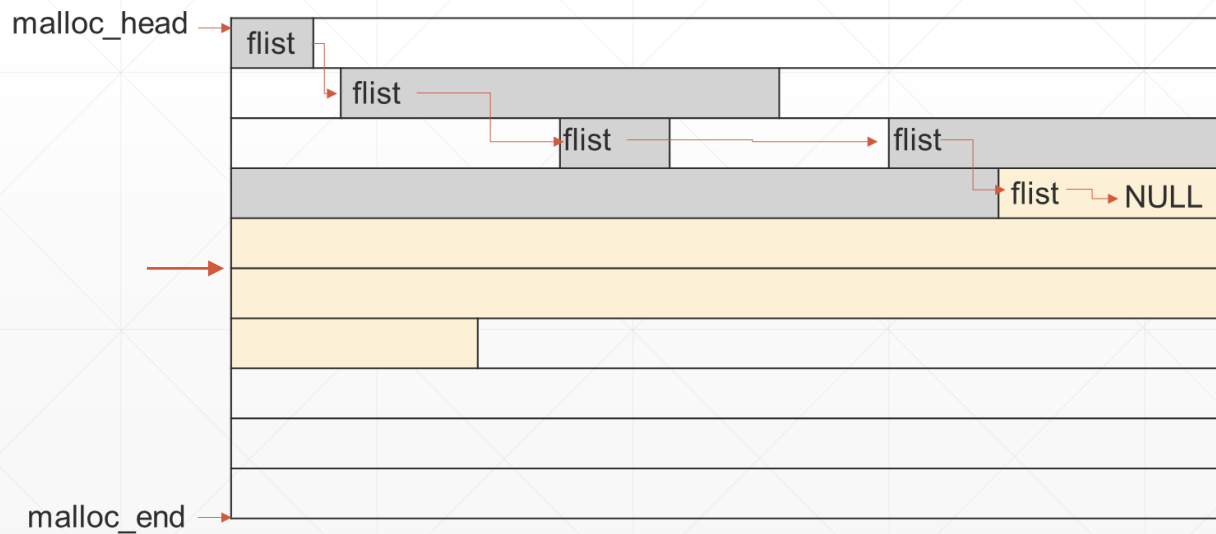
完成后, 再行动态内存分配逻辑, 子图c。



(a) 原有的堆



(b) 堆空间扩展 3个页面

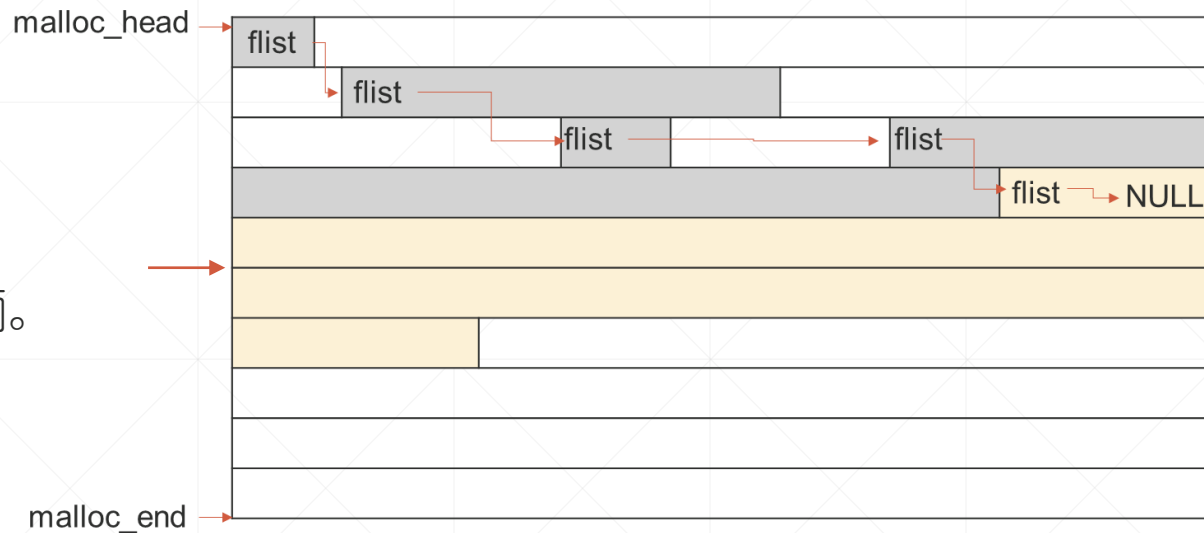


(c) 分配出去的新内存片紧贴原先的最后一个内存片

malloc (动态内存分配) , 更一般的逻辑 2

记 所需内存片的尺寸是 **size**

(2) 没找到, 执行**brk**系统调用, 扩展堆空间 3个页面。
完成后, 再行动态内存分配逻辑。



`int expand = size - (malloc_end - (char *)iter - (iter->size));` // iter, 最后一个内存片的首地址。红色, 堆尾部, 空闲区的尺寸

`expand = ((expand + PAGE_SIZE - 1) / PAGE_SIZE) * PAGE_SIZE;` // 3个页面的整数倍

`malloc_end = sbrk(expand);` // 扩展数据段, malloc_end是新数据段之后第一个字节的首地址

`iter->nlink = (char *)iter + (iter->size);`

`iter = iter->nlink;`

`iter->size = size;`

`iter->nlink = NULL;`

`printf("%u\n", iter);`

`return (char*)iter + sizeof(struct flist);`

新内存片, 插入内存片链表的尾部

// 新内存片紧贴、存放在最后一个内存片的后面。

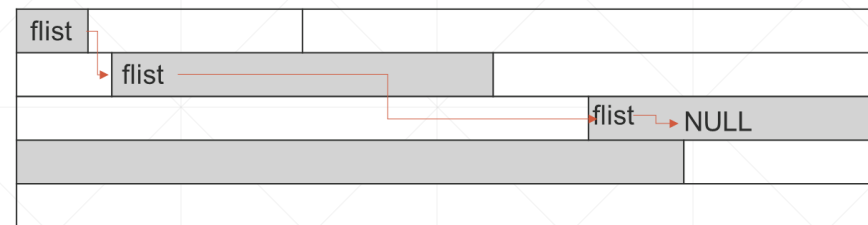
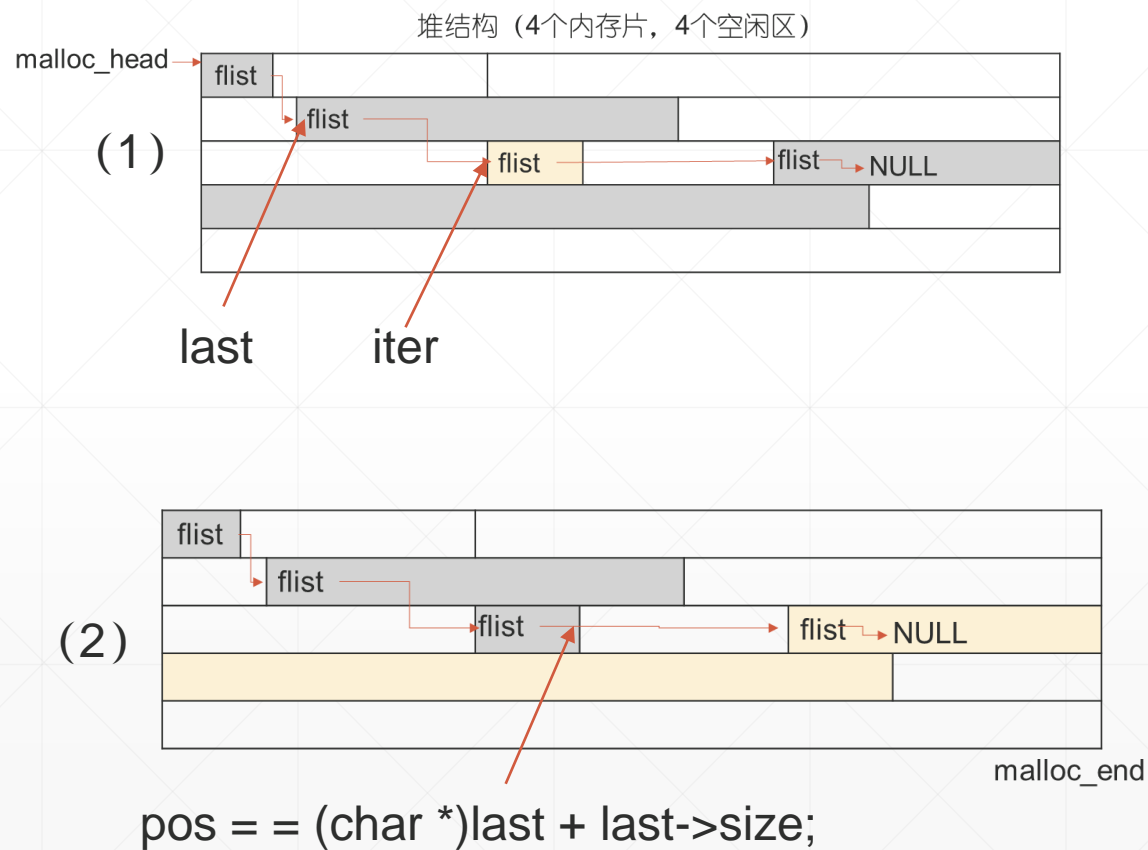
free (动态内存释放)

```
int free(void* addr) // addr, 内存片数据区的起始地址
{
    char * real_addr = addr - 8; // 内存片的起始地址
    struct flist* iter = malloc_head; // 当前内存片
    struct flist* last = malloc_head; // 前一个内存片

    if (addr == 0)
    {
        return -1;
    }
}
```



free (动态内存释放)

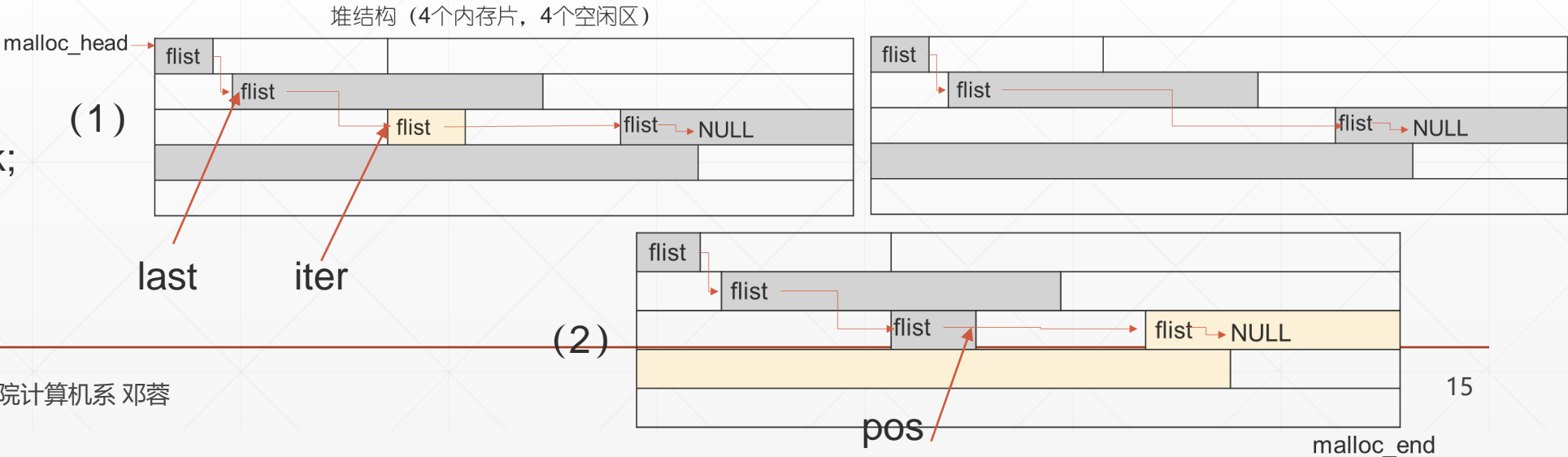


$malloc_end - pos \geq 6$ 页,
brk 缩小数据段, 堆空间缩小6页。



free (动态内存释放)

```
while(iter) // 遍历内存片链表，寻找释放的内存片
{
    if (iter == real_addr) // 找到啦
    {
        last->nlink = iter->nlink; // 删除释放的内存片
        if (last->nlink == NULL) // 如果回收的是堆尾部的内存片，需要考虑缩短数据段
        { // 计算内存片释放后，堆尾部空闲区的长度
            char *pos = (char *)last + last->size; // pos, 前一个内存片之后的第一个字节
            if (malloc_end - pos > PAGE_SIZE * 2) // 超过6个页面
            { // 执行brk系统调用，缩短数据段
                malloc_end = sbrk(-((malloc_end - pos) / PAGE_SIZE * PAGE_SIZE));
            }
        }
    }
    return 0;
}
last = iter;
iter = iter->nlink;
return -1;
}
```



brk系统调用，改变堆空间尺寸

- `brk(0)`，返回当前堆的尾部地址
- `brk(newEnd)`，改变堆空间尺寸。`newEnd`是新堆的尾部地址

进程的用户空间（虚空间）





动态内存分配器使用brk系统调用的包装函数， sbrk(increment)

- 动态内存分配器维护一个全局变量 fakeedata，跟踪堆尾部
- 用函数 sbrk 扩展 或 缩小 堆空间。入口参数 increment，堆尺寸的变化量。 扩大，正值；缩小，负值

进程的用户空间（虚空间）



`unsigned int fakeedata = 0;` // 堆底部（其后，第一个字节的地址）。静态变量，函数返回，值保留。

`int sbrk(int increment)`

`{`

`if (fakeedata == 0) // 首次调用sbrk函数，初始化fakeedata`

`{`

`fakeedata = brk(0); // 执行brk(0)得到数据段底部地址`

`}`

`unsigned int newedata = fakeedata + increment - 1;` // 1、计算新数据段最后一个字节的地址

`brk(((newedata >> 12) + 1) << 12);` // 2、执行brk系统调用，改变数据段的尺寸（以页为单位）

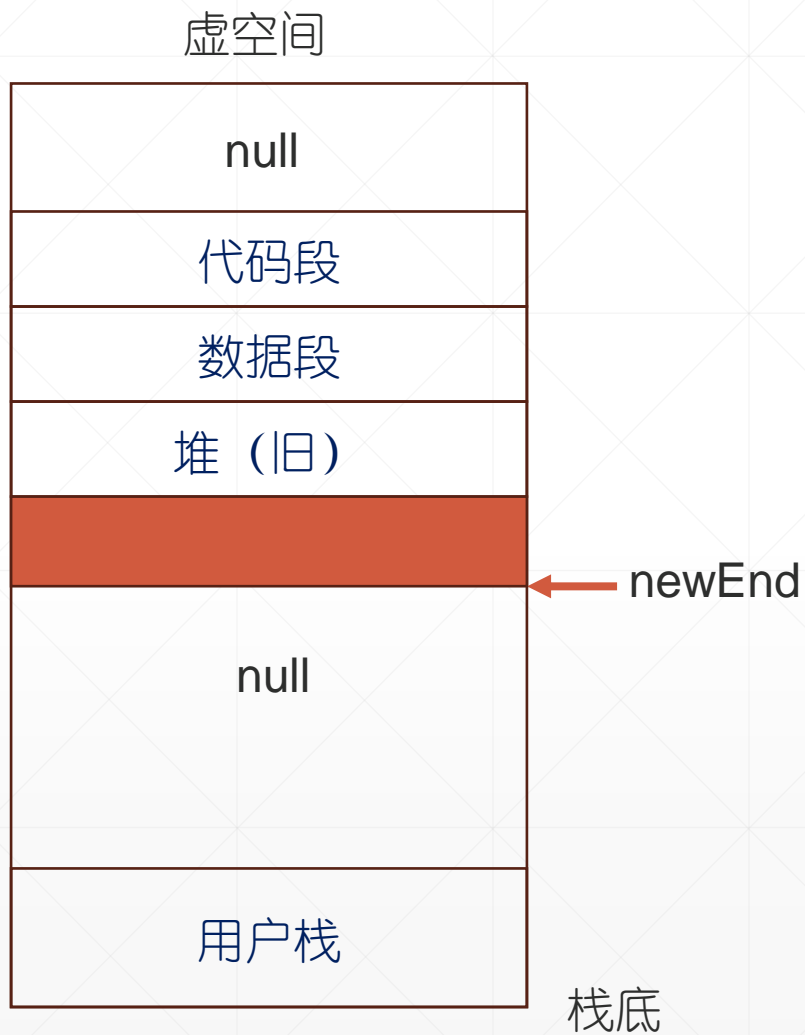
`fakeedata = newedata + 1;`

`return fakeedata;` // 3、返回新数据段底部

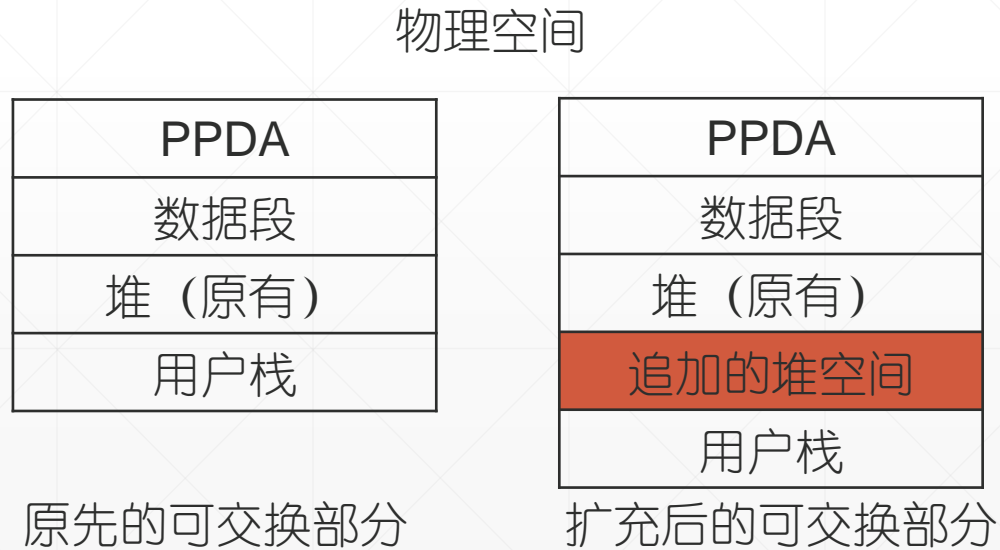
`}`

brk系统调用的执行过程

堆空间扩大



假设, 1页代码, 1页数据, 1页堆, 1页栈。追加1页堆



虚空间的变化



null			
0	RO	U	P
1	RW	U	P
2	RW	U	P
null			
null			
3	RW	U	P

null			
0	RO	U	P
1	RW	U	P
2	RW	U	P
3	RW	U	P
null			
4	RW	U	P

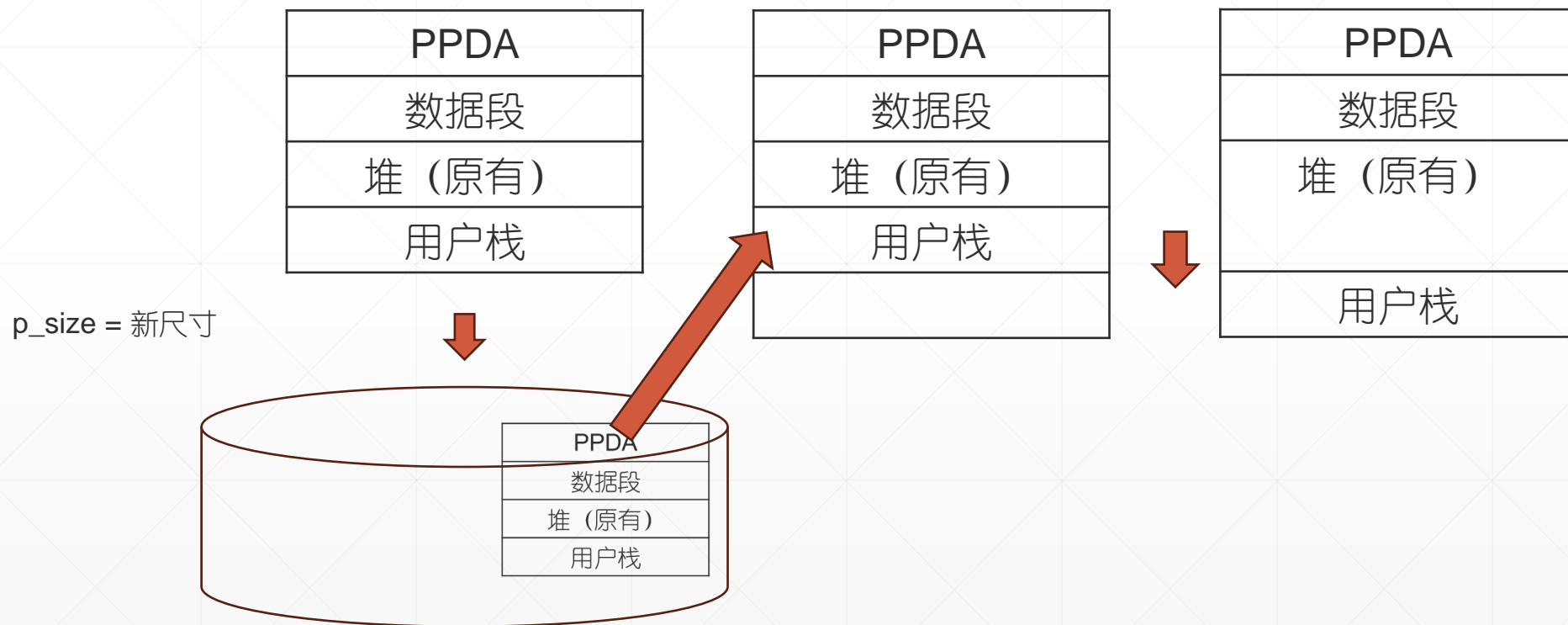
```
class MemoryDescriptor
{
    PageTable*
    unsigned long
    unsigned long
    unsigned long
    unsigned long
    unsigned long
}
```

```
m_UserPageTableArray;
m_TextStartAddress;
m_TextSize;
m_DataStartAddress;
m_DataSize; + 4096
m_StackSize;
```

进程图像在物理空间中的移动



进程图像在物理空间中的移动（内存不足）



brk系统调用的执行过程

堆空间缩小

假设，1页代码，1页数据，2页堆，1页栈。堆缩小1页。



物理空间



原先的可交换部分



扩充后的可交换部分

虚空间的变化

虚空间



← newEnd

栈底

null				
0	RO	U	P	
1	RW	U	P	
2	RW	U	P	
3	RW	U	P	
null				
4	RW	U	P	

null				
0	RO	U	P	
1	RW	U	P	
2	RW	U	P	
null				
null				
3	RW	U	P	

```
class MemoryDescriptor
{
```

```
    PageTable*
```

```
    unsigned long
    unsigned long
    unsigned long
    unsigned long
    unsigned long
```

```
    m_UserPageTableArray;
```

```
    m_TextStartAddress;
    m_TextSize;
    m_DataStartAddress;
    m_DataSize;      - 4096
    m_StackSize;
```


进程图像在物理空间中的移动





brk系统调用

SBreak(), brk系统调用的处理函数。

按新尺寸重写内存描述符 和 相对虚实地址映射表，
调用Expand()函数为新的可交换部分分配物理内存[注]。

```
int SystemCall::Sys_SBreak()  
{  
    User& u = Kernel::Instance().GetUser();  
    u.u_procp->SBreak();  
  
    return 0; /* GCC likes it ! */  
}
```

入口函数

u_arg[0], 新数据段之后的第一个字节的地址。

[注] 现代操作系统广泛使用基于请求调页的虚拟内存技术，所以进程图像扩展时，brk系统调用只更新进程虚地址空间，不需要为新空间分配物理内存，更无需移动已有进程图像。Unix V6++系统使用连续内存管理方式，brk系统调用需要在更新进程虚地址空间的同时，同步更新分配给进程的物理内存单元。



void Process::SBreak()

```
{
```

```
    User& u = Kernel::Instance().GetUser();
```

```
    unsigned int newEnd = u.u_arg[0]; // 堆底部，第一个字节的地址
```

```
    MemoryDescriptor& md = u.u_MemoryDescriptor; // 内存描述符
```

```
    unsigned int newSize = newEnd - md.m_DataStartAddress; // 新数据段的长度
```

```
    if (newEnd == 0)
```

```
    { // 获取当前堆底部（数据段之后第一个字节的地址）
```

```
        u.u_ar0[User::EAX] = md.m_DataStartAddress + md.m_DataSize;
```

```
        return;
```

```
    }
```

```
    // 1、按数据段的新尺寸，重写相对虚实地址映射表，并且加载系统页表
```

```
    if ( false == u.u_MemoryDescriptor.EstablishUserPageTable(md.m_TextStartAddress,  
                                                                md.m_TextSize, md.m_DataStartAddress, newSize, md.m_StackSize) )
```

```
        return; // 超过8M字节。OOM，动态空间溢出，malloc失败
```



```
int change = newSize - md.m_DataSize; // 数据段长度的变化量
md.m_DataSize = newSize; // 数据段新尺寸写入内存描述符
newSize += ProcessManager::USIZE + md.m_StackSize; // 计算可交换部分的新尺寸
```

```
if ( change < 0 )
{
    /* 数据段缩小，可交换部分无需移动。堆栈段向低地址方向平移 */
}
else if ( change > 0 )
{
    /* * 数据段增大。（1）按新尺寸分配物理内存
                        (2) 将可交换部分复制到新空间的低地址端
                        (3) 堆栈段向高地址方向平移
    */
}
```

```
u.u_ar0[User::EAX] = md.m_DataStartAddress + md.m_DataSize; /* 系统调用的返回值，数据段新的尾部 */
```

```
}
```

/* 数据段缩小 (1) 堆栈段向低地址方向平移 (2) 释放高端物理内存。同步更新内存描述符 和 相对表 */

if (change < 0)

{

int dst = u.u_procp->p_addr + newSize - md.m_StackSize; // 堆栈段新的起始地址

int count = md.m_StackSize; // 堆栈段的长度

while(count--)

{

Utility::CopySeg(dst - change, dst);

dst++;

}

this->Expand(newSize); // 2、释放高端物理内存

}



/* 数据段增大。(1) 按新尺寸分配物理内存 (2) 将可交换部分复制到新空间的低地址端
(3) 堆栈段向高地址方向平移。同步更新内存描述符 和 相对表 */

```
else if ( change > 0 )
{
    this->Expand(newSize); // (1) (2)
    int dst = u.u_procp->p_addr + newSize; // 可交换部分新的尾部
    int count = md.m_StackSize;
    while(count--) // (3)
    {
        dst--;
        Utility::CopySeg(dst - change, dst);
    }
}
u.u_ar0[User::EAX] = md.m_DataStartAddress + md.m_DataSize; /* 系统调用的返回值，数据段新的尾部 */
}
```



Expand()

更新分配给可交换部分的物理内存单元。

- 如果长度缩小，释放高地址端多余的内存空间。
- 如果长度增加，按增长后的尺寸为进程的可交换部分分配内存空间，之后将原先的进程图像拷贝至新内存区低地址端。内存不足时，进程会用盘交换区暂存原先的可交换部分，等待内存紧张条件得到缓解后、0#进程为扩展后的进程图像分配新内存区、把磁盘上的可交换区复制到新内存区的低地址端。



```
void Process::Expand(unsigned int newSize)
```

```
{
```

```
    UserPageManager& userPgMgr = Kernel::Instance().GetUserPageManager();
```

```
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
```

```
    User& u = Kernel::Instance().GetUser();
```

```
    Process* pProcess = u.u_procp;
```

```
    unsigned int oldSize = pProcess->p_size; // 进程可交换部分, 原先的尺寸
```

```
    p_size = newSize; // 进程可交换部分的新尺寸
```

```
    unsigned long oldAddress = pProcess->p_addr; // 进程可交换部分, 原先的起始地址
```

```
    unsigned long newAddress; // 进程可交换部分的新地址
```

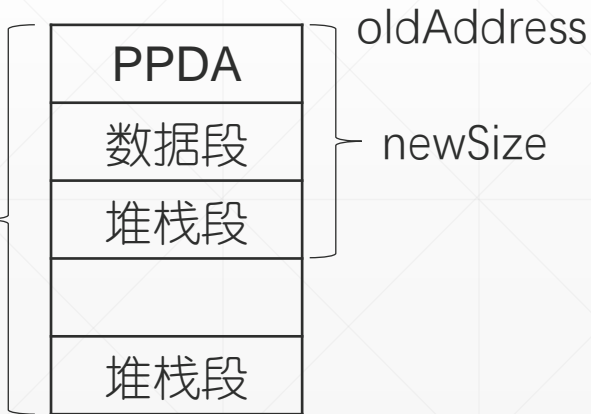
```
    if ( oldSize >= newSize ) // 1、如果进程图像缩小, 释放高地址端多余的空间
```

```
    {
```

```
        userPgMgr.FreeMemory(oldSize - newSize, oldAddress + newSize); oldSize
```

```
        return; // 完事, 返回
```

```
    }
```





```
/* 进程图像扩大，需要寻找一块大小为newSize的连续内存区 */  
SaveU(u.u_rsav);  
newAddress = userPgMgr.AllocMemory(newSize); // 按新尺寸为可交换部分分配内存  
  
/* 内存分配不成功，将可交换部分复制到盘交换区，释放原先占用的内存单元 */  
if ( NULL == newAddress )  
{  
    SaveU(u.u_ssav); // 保存Expand( )栈帧的顶部和基地址  
    procMgr.XSwap(pProcess, true, oldSize); // 将原可交换部分复制到盘交换区，释放内存  
    pProcess->p_flag |= Process::SSWAP; // 复制完成后，置SSWAP标识  
    procMgr.Swtch(); // 放弃CPU。Expand到此结束。  
    /* no return */  
}
```

未来，0#进程会将进程换入内存，随后进程接受调度、从Expand()返回。所以这里no return。



```
/* 内存分配成功，将可交换部分拷贝到新内存区，释放原先占用的内存单元 */  
pProcess->p_addr = newAddress; // 修改p_addr  
for ( unsigned int i = 0; i < oldSize; i++ )  
{  
    Utility::CopySeg(oldAddress + i, newAddress + i); //复制可交换部分至新空间低地址区  
}  
userPgMgr.FreeMemory(oldSize, oldAddress); // 释放可交换部分原先占用的内存单元  
  
X86Assembly::CLI();  
SwchUStruct(pProcess); // 可交换部分已移动，需要更新核心页表中的1023#PTE  
RetU();  
X86Assembly::STI();  
  
/* 基于新的相对虚实地址映射表重写系统用户页表，为现运行进程重建地址映射关系。*/  
u.u_MemoryDescriptor.MapToPageTable();  
  
}
```

栈自动扩展

虚空间



假设, 1页
代码, 1页
数据, 1页
堆, 1页栈。
追加1页栈。

	null
0	RO U P
1	RW U P
2	RW U P
	null
	null
3	RW U P

	null		
0	RO	U	P
1	RW	U	P
2	RW	U	P
	null		
3	RW	U	P
4	RW	U	P

```
class MemoryDescriptor
{
    PageTable*      m_UserPageTableArray;

    unsigned long    m_TextStartAddress;
    unsigned long    m_TextSize;
    unsigned long    m_DataStartAddress;
    unsigned long    m_DataSize;
    unsigned long    m_StackSize; - 4096
}
```

物理空间



栈底

原先的可交换部分

扩充后的可交换部分



```
void Process::SStack()
{
    User& u = Kernel::Instance().GetUser();
    MemoryDescriptor& md = u.u_MemoryDescriptor;
    unsigned int change = 4096;
    md.m_StackSize += change;
    unsigned int newSize = ProcessManager::USIZE + md.m_DataSize + md.m_StackSize;

    if ( false == u.u_MemoryDescriptor.EstablishUserPageTable(md.m_TextStartAddress,
        md.m_TextSize, md.m_DataStartAddress, md.m_DataSize, md.m_StackSize) )
    {
        u.u_error = User::ENOMEM;
        return;
    }

    this->Expand(newSize);
    int dst = u.u_procp->p_addr + newSize;
    unsigned int count = md.m_StackSize - change;

    while(count-- > 0) // 原堆栈段下移
    {
        dst--;
        Utility::CopySeg(dst - change, dst);
    }

    u.u_MemoryDescriptor.MapToPageTable();
}
```