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

同济大学计算机系

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**sbrk**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sbrk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）

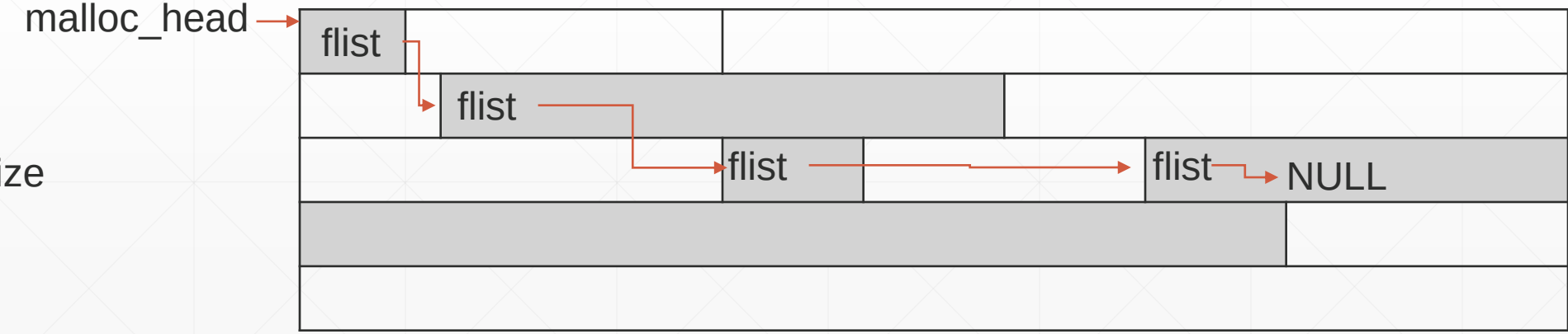
堆空间管理

- 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。

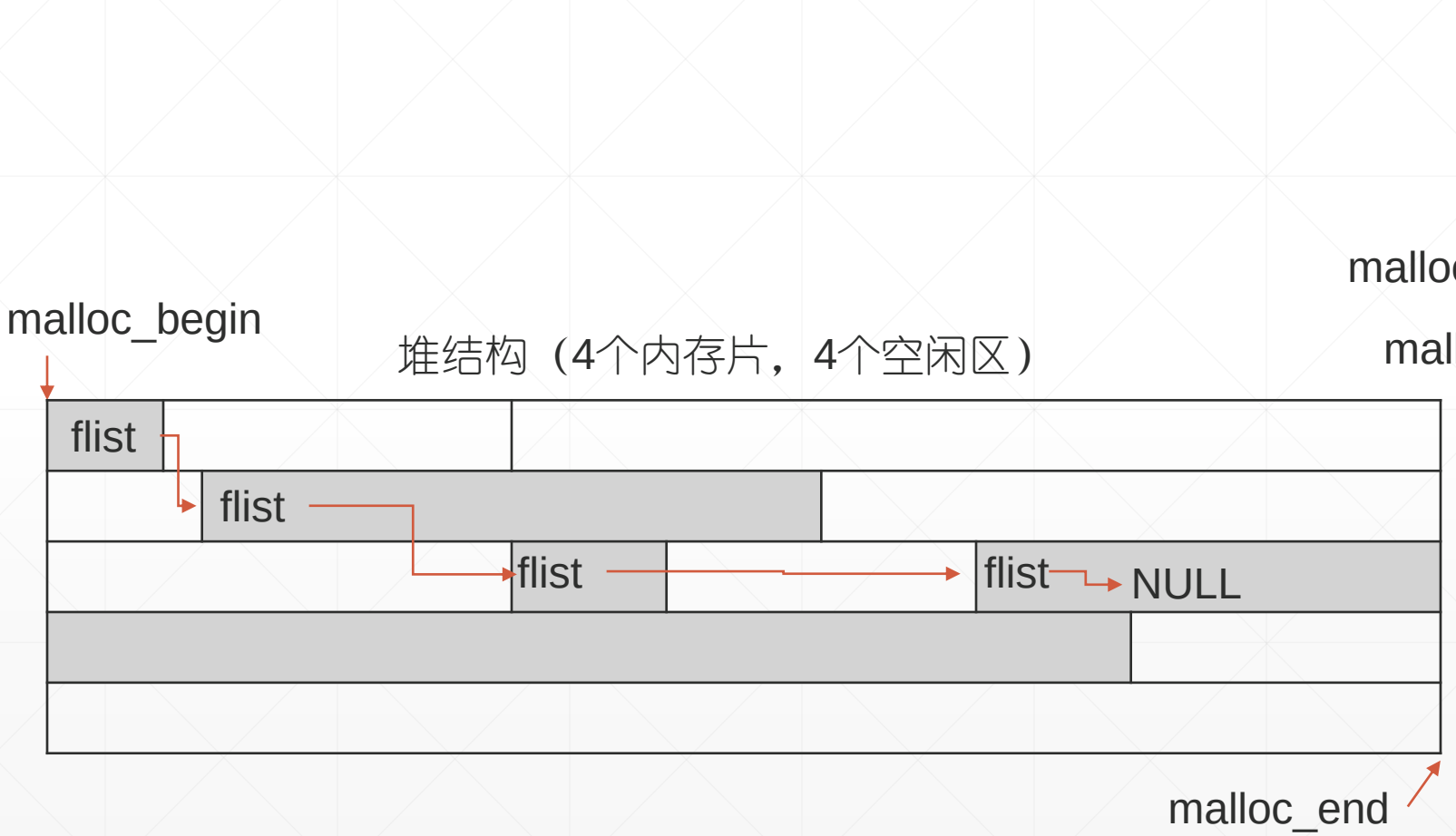
```
typedef struct flist {
   unsigned int size;  // 长度
   struct flist *nlink;  // 下个内存片的首地址
};
```

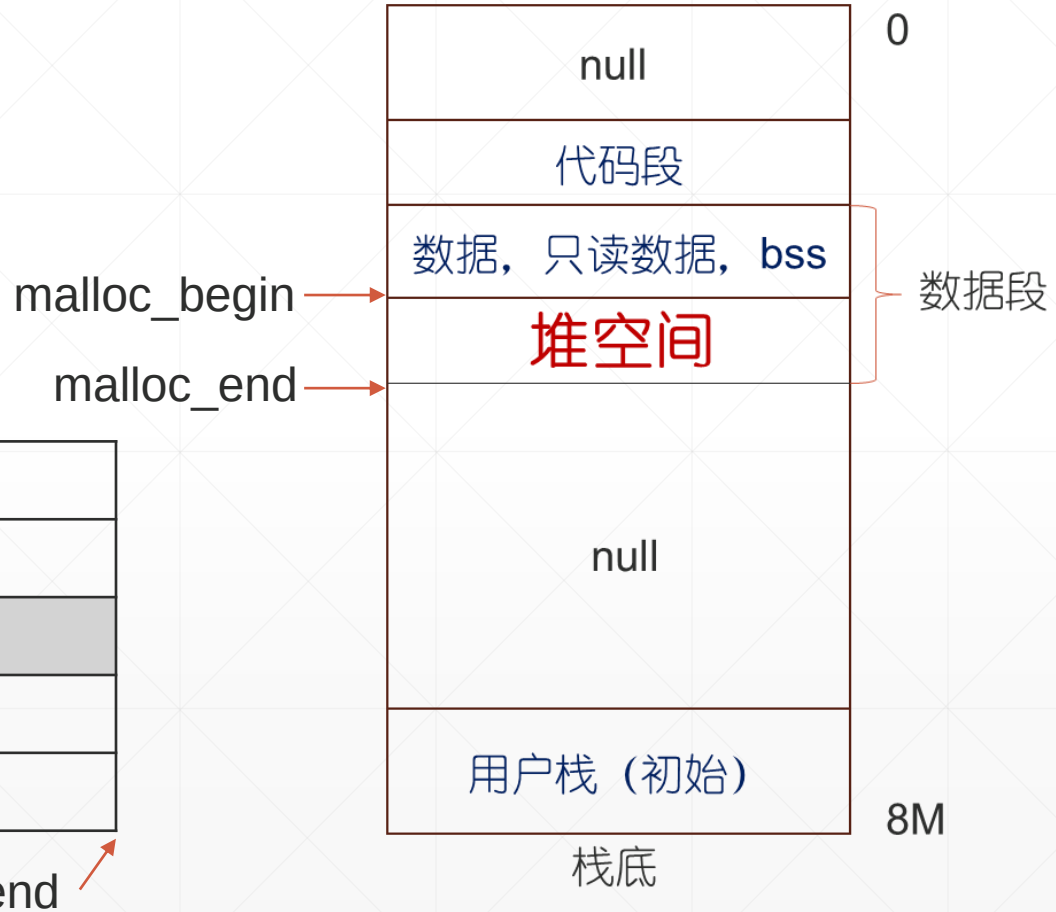
内存片结构

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）
flist
flist
flist
flist
NULL
malloc_end


null
代码段
数据，只读数据，bss
堆空间
null
用户栈（初始）
0
8M
数据段
malloc_begin
malloc_end
栈底

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？

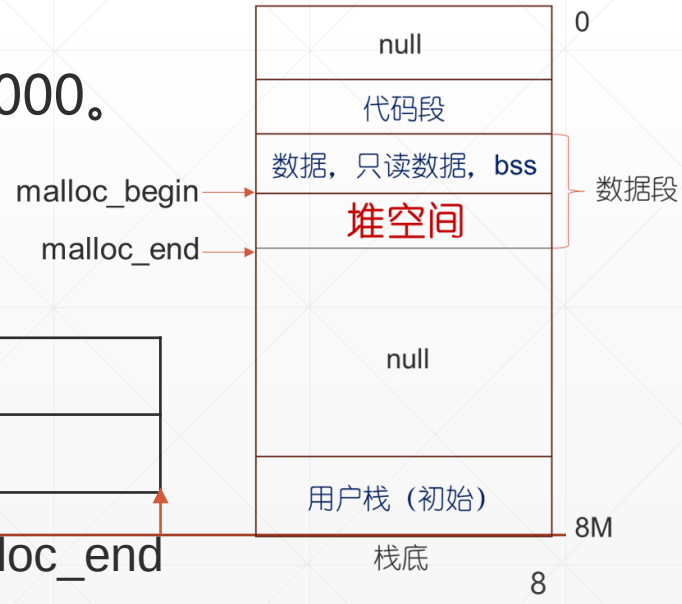
(1) 初始化堆空间

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
};
```



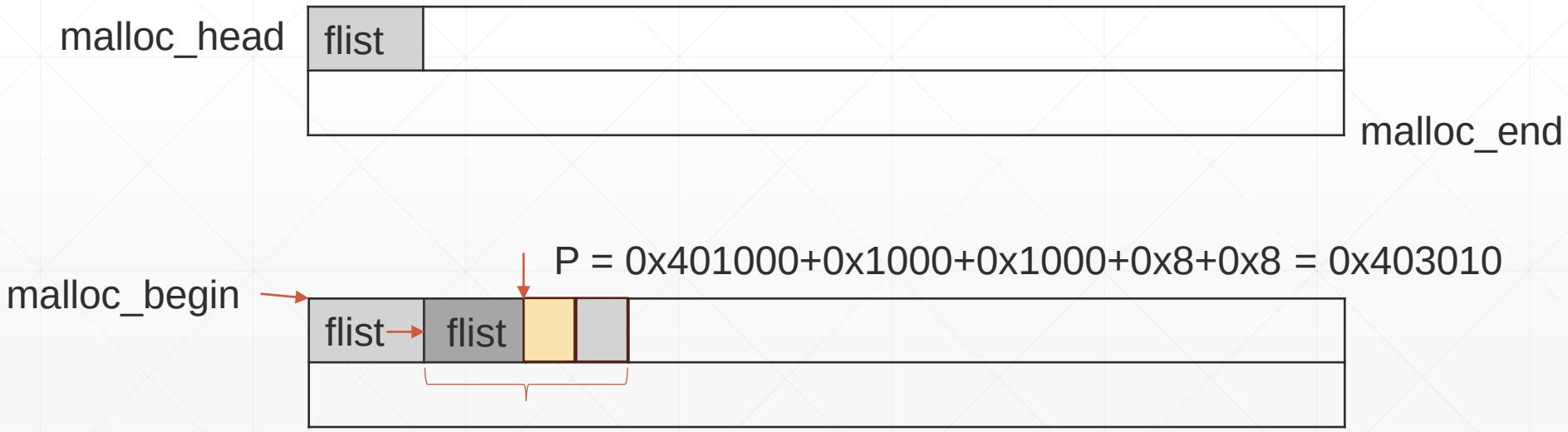
（2）动态内存分配

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。

执行brk系统调用，请求内核扩展数据段，尾部追加3个逻辑页面。

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。



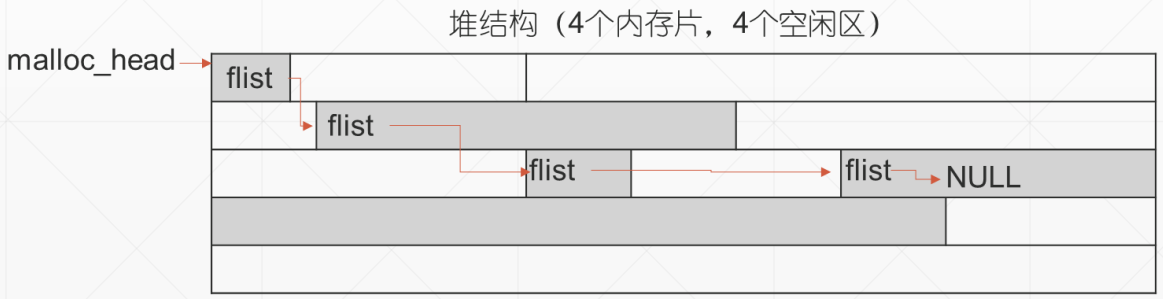
malloc（动态内存分配），更一般的逻辑 1

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。
完成后，更新内存片链表，再行动态内存分配逻辑。



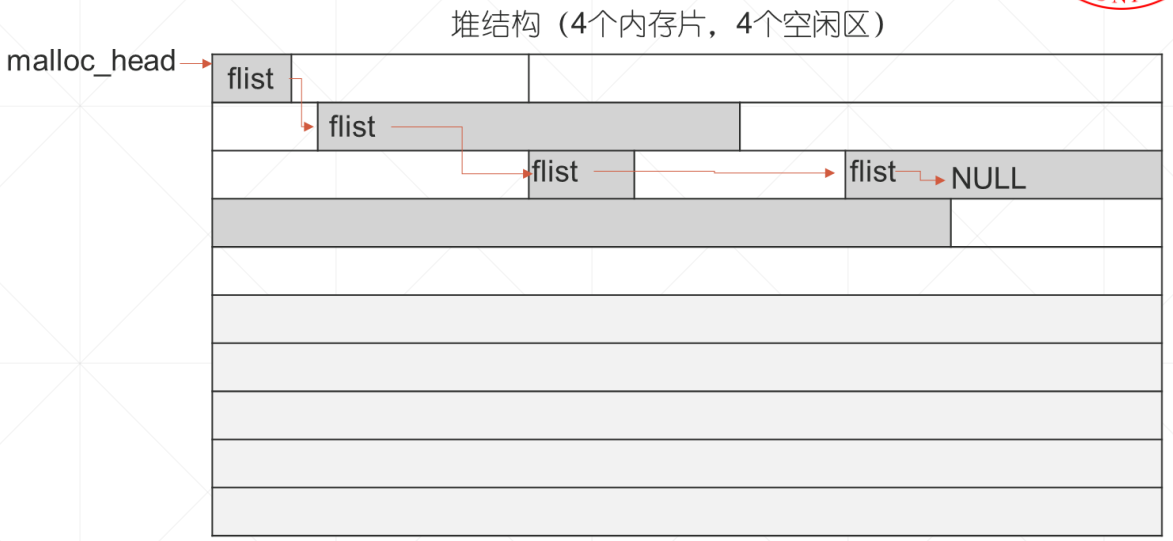

```
struct flist* iter = malloc_head;
while(iter->nlink)
  {
     if ((int)(iter->nlink) - (int)iter - iter->size >= size)
     {
        struct flist *temp = (char *)iter + (iter->size);
        temp->nlink = iter->nlink;
        iter->nlink = temp;
        temp->size = size;
        return (char *)temp + 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。
完成后，更新内存片链表，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);      // 新内存片紧贴、存放在最后一个内存片的后面。
```

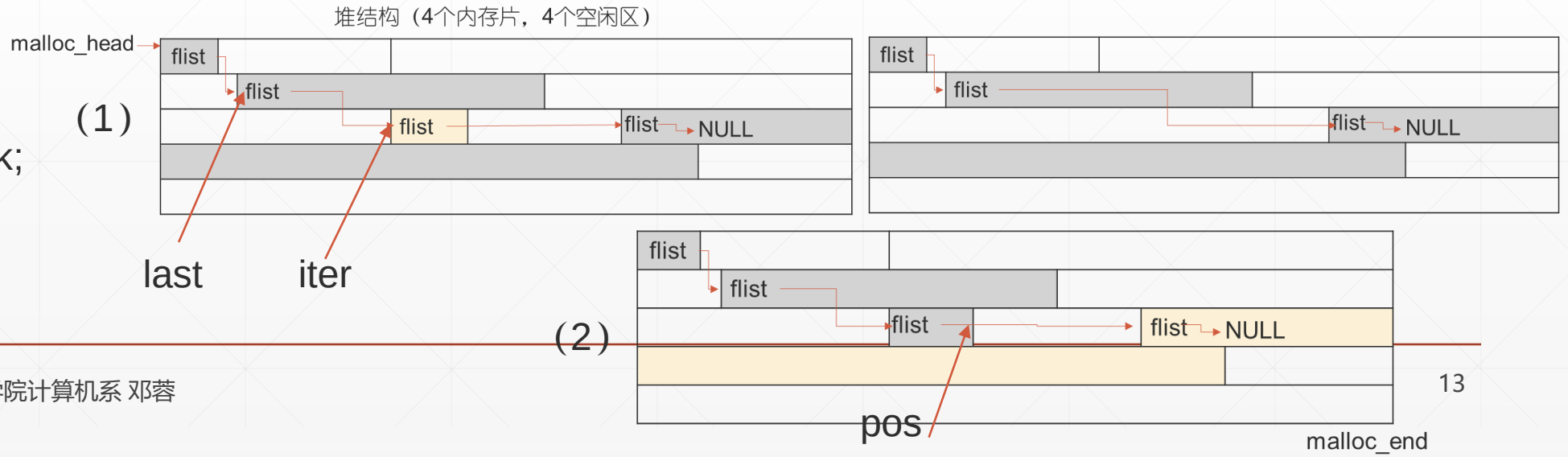
新内存片，插入内存片链表的尾部

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）

```
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，用户空间的钩子函数

```
unsigned int fakeedata = 0;      // 数据段底部（数据段之后第一个字节的地址）
int sbrk(int increment)     // increment，数据段长度变化量
{
    if (fakeedata == 0)   // 首次调用 sbrk 函数
    {
        fakeedata = brk(0);     // 得到数据段底部
    }
    unsigned int newedata = fakeedata + increment - 1;   // 新数据段最后一个字节的地址
    brk(((newedata >> 12) + 1) << 12);     // 执行 brk 系统调用，改变数据段的尺寸（以页为单位）
    fakeedata = newedata + 1;     // 新数据段底部
    return fakeedata;
}
```

```
int brk(void * newEndDataAddr)   // 17#系统调用，入口参数是新数据段之后第一个字节的地址
{
    int res;
    __asm__ volatile ("int $0x80":"=a"(res):"a"(17),"b"(newEndDataAddr));
    if ( res >= 0 )
        return res;
    errno = -1*res;
    return -1;
}
```

brk系统调用

```
int SystemCall::Sys_SBreak()
{
        User& u = Kernel::Instance().GetUser();
        u.u_procp->SBreak();

        return 0; /* GCC likes it ! */
}
```

入口函数

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( )函数扩展/缩小可交换部分长度[注]。

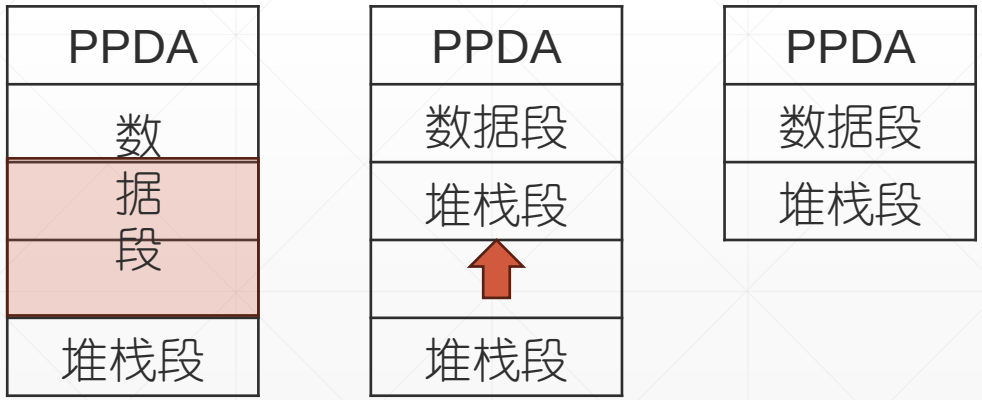
[注] 现代操作系统广泛使用基于请求调页的虚拟内存技术，所以进程图像扩展时，brk系统调用只更新进程虚地址空间，不需要为新空间分配物理内存，更无需移动已有进程图像。Unix V6++系统使用连续内存管理方式，brk系统调用需要在更新进程虚地址空间的同时，同步更新分配给进程的物理内存单元。

void Process::SBreak()

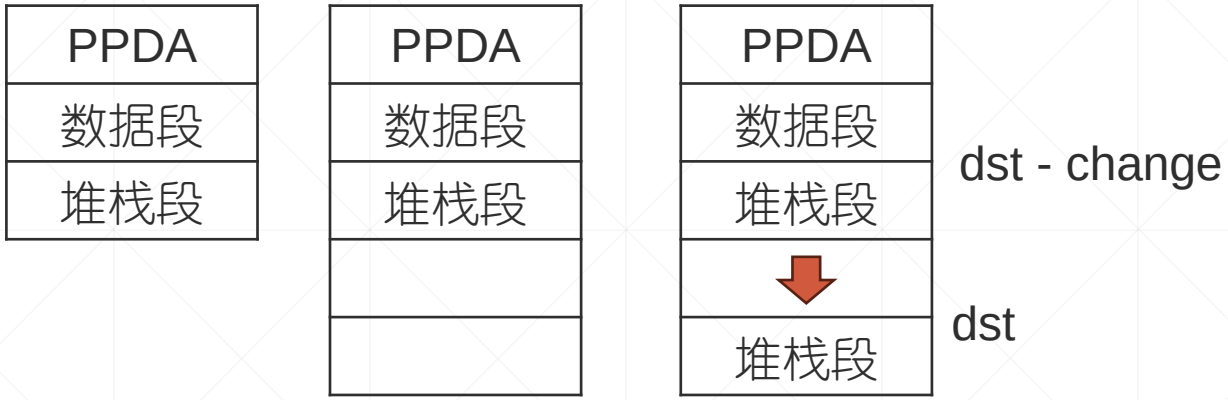
```
{
        User& u = Kernel::Instance().GetUser();
        unsigned int newEnd = u.u_arg[0]; // 新数据段之后，第一个字节的地址
        MemoryDescriptor& md = u.u_MemoryDescriptor;  // 内存描述符
        unsigned int newSize = newEnd - md.m_DataStartAddress; // 新数据段的长度

        if (newEnd == 0)
        {   // 获取当前数据段底部（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;
        }

        // 按数据段的新尺寸，重写相对虚实地址映射表
        if ( false == u.u_MemoryDescriptor.EstablishUserPageTable(md.m_TextStartAddress,
                                md.m_TextSize, md.m_DataStartAddress, newSize, md.m_StackSize) )
                return;  // 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

/* 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 */
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（4K的整数倍）
        while(count--)   // 复制堆栈段
        {
                Utility::CopySeg(dst - change, dst);
                dst++;
        }
        this->Expand(newSize);  // 缩短可交换部分，也就是直接释放高地址端物理内存
}
/* 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 */
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)
```

```
int change = newSize - md.m_DataSize;   // 数据段长度的变化量。也就是堆栈段的平移量
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

/* 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--)    // 1、复制堆栈段
    {
            Utility::CopySeg(dst - change, dst);
            dst++;
    }
    this->Expand(newSize);   // 2、释放高端物理内存
}
```



红色，需要释放的空间



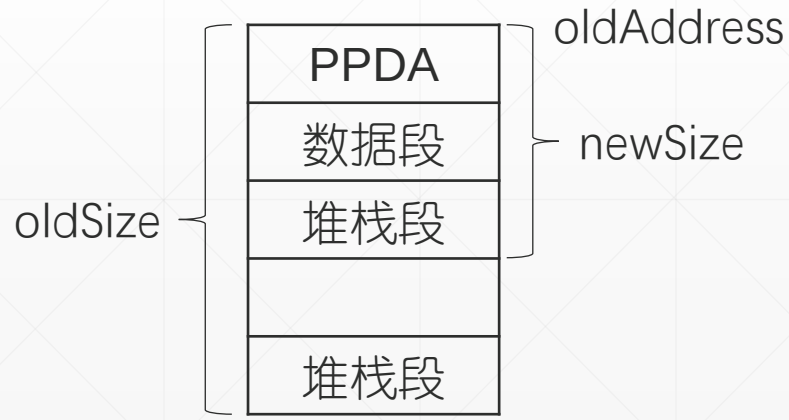

```
/* 数据段增大。（1）按新尺寸分配物理内存（2）将可交换部分复制到新空间的低地址端（3）堆栈段向高地址方向平移。同步更新内存描述符 和 相对表 */
        else if ( change > 0 )
        {
                this->Expand(newSize);  // (1)  (2)
                int dst = u.u_procp->p_addr + newSize;   // 可交换部分新的尾部
                int count = md.m_StackSize;
                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 ) // 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);
        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();
        SwtchUStruct(pProcess);   // 可交换部分已移动，需要更新PPDA区的地址映射关系
        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();
}
```