XV6实例分析:虚拟内存的分配

组员：赵心培 杜汨鑫 党程睿

malloc函数

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "param.h"

// Memory allocator by Kernighan and Ritchie,
// The C programming Language, 2nd ed.  Section 8.7.

typedef long Align;

union header {
  struct {
    union header *ptr;
    uint size;
  } s;
  Align x;
};

typedef union header Header;

static Header base;
static Header *freep;
```

链表header：作为整个内存分配的单位，记录某个空间的可用空间大小并指向另一块空间

Base：记录第一块空间的位置

freep：记录下一块未分配空间的位置

```
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;

  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

实际分配空间大小

首次分配

```
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;

  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

首次分配，此时freep还没有被赋值，需要从base开始分配空间

分配空间

```
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;

  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

正常分配情况

```
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;

  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

空间不足（p->s.size < nunits），调用morecore函数，在该函数中调用sbrk函数

sys_sbrk()中switchuvm()的作用

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

sys_sbrk()函数的作用是将进程的内存空间大小增加n，同时调用growproc()为之申请相应的物理内存（避免page fault），switchuvm()函数的调用发生在分配物理内存的过程中

```
int growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

Switchuvm()

```
void switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

xv6让每个进程都有独立的页表结构，在切换进程时总是需要切换页表。

kpgdir是cr3寄存器的首地址，kpgdir仅仅在scheduler内核线程中使用。

页表和内核栈都是每个进程独有的，xv6使用结构体proc将它们统一起来，在进程切换的时候，他们也往往随着进程切换而切换，内核中模拟出了一个内核线程，它独占内核栈和内核页表kpgdir，它是所有进程调度的基础。

switchuvm通过传入的proc结构负责切换相关的进程独有的数据结构，其中包括TSS相关的操作，然后将进程特有的页表载入cr3寄存器，完成设置进程相关的虚拟地址空间环境。

morecore函数

```
static Header*
morecore(uint nu)
{
  char *p;
  Header *hp;

  if(nu < 4096)
    nu = 4096;
  p = sbrk(nu * sizeof(Header));
  if(p == (char*)-1)
    return 0;
  hp = (Header*)p;
  hp->s.size = nu;
  free((void*)(hp + 1));
  return freep;
}
```

通过sbrk函数扩张，若可以成功扩张，调用free函数

free函数

```
void
free(void *ap)
{
  Header *bp, *p;

  bp = (Header*)ap - 1;
  for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
      break;
  if(bp + bp->s.size == p->s.ptr){
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
    bp->s.ptr = p->s.ptr;
  if(p + p->s.size == bp){
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
```

bp为要释放对象

首先定位bp与空闲空间的相对位置，找到它的的最近的上一个和下一个空闲空间（p与p->s.ptr），或是当它在整个空闲空间的前面或后面时找到空闲链表的首尾元素。

```
void
free(void *ap)
{
  Header *bp, *p;

  bp = (Header*)ap - 1;
  for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
      break;
  if(bp + bp->s.size == p->s.ptr){
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
    bp->s.ptr = p->s.ptr;
  if(p + p->s.size == bp){
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
```

判断是否超过一个完整的空间大小

确定下一个空闲空间位置

Linux中的内存分配

- Linux中的动态储存管理采用了**伙伴系统**的方法

  引入几个重要的概念:

  1)某组page:
  假设order是2,则page_idx是0-3的是一组,4-7是一组.组中最低地址的那个页面代表这个组连接进area的list链表，并且组内其他page不会存在于其他任何area的list。

  2)组对(相关组):
  area中有两个重要的数据结构:
  (1)list : 属于此order(area)的pgae,或者page组的链表
  (2)*map : 每一位代表一对相关组(buddy)。所谓相关组就是可以合并成更高order的两个连续page组,我们也称之为相关组.

当没有低order的page满足用户需求时，将高order的page拆分为低order的page。
当两个相关组的低order的page被释放后，可以合并成为高order的page。

有图为将两个page合并的expand()函数

Linux和XV6中内存管理系统数据结构的主要区别在于Linux中的某些链表表项之间是可以合并的，而XV6不能。

```
static inline void expand(struct zone *zone, struct page *page,
    int low, int high, struct free_area *area,
    int migratetype)
{
    unsigned long size = 1 << high;

    while (high > low) {
        area--;
        high--;
        size >>= 1;
        VM_BUG_ON_PAGE(bad_range(zone, &page[size]), &page[size]);

        /*
         * Mark as guard pages (or page), that will allow to
         * merge back to allocator when buddy will be freed.
         * Corresponding page table entries will not be touched,
         * pages will stay not present in virtual address space
         */
        if (set_page_guard(zone, &page[size], high, migratetype))
            continue;

        add_to_free_area(&page[size], area, migratetype);
        set_page_order(&page[size], high);
    }
}
```