

虚拟内存与页表管理

1 实验目标

最初Xinu采用了最基础的段式内存管理，所有的进程全部都运行在同一个地址空间之中，进程之间可以相互访问数据，存在一定的风险。因此，在这次实验之中，要使用二级页表实现页式内存管理，使得每一个进程看到的内存地址空间都是相同的。

2 实验步骤与过程

2.1 页表与页目录结构设计

从无到有从实现虚拟内存，一开始给我一种无从下手的感觉，整体的页式内存系统乍一想感觉非常庞大，不知道从哪里开始。所以，跟据老师的实验指导，首先我定义了页表和页目录所使用的结构体，规定页目录和页表的结构如下所示：

```
struct pd_t{
    unsigned int pd_pres : 1;          /* page table present? */
    unsigned int pd_write : 1;         /* page is writable? */
    unsigned int pd_user : 1;          /* is use level protection? */
    unsigned int pd_pwt : 1;           /* write through cachine for pt? */
    unsigned int pd_pcd : 1;           /* cache disable for this pt? */
    unsigned int pd_acc : 1;           /* page table was accessed? */
    unsigned int pd_mbz : 1;           /* must be zero */
    unsigned int pd_fmb : 1;           /* four MB pages? */
    unsigned int pd_global: 1;         /* global (ignored) */
    unsigned int pd_avail : 3;         /* for programmer's use */
    unsigned int pd_base : 20;        /* location of page table? */
};

struct pt_t {
    unsigned int pt_pres : 1;          /* page is present? */
    unsigned int pt_write : 1;         /* page is writable? */
    unsigned int pt_user : 1;          /* is use level protection? */
    unsigned int pt_pwt : 1;           /* write through for this page? */
    unsigned int pt_pcd : 1;           /* cache disable for this page? */
    unsigned int pt_acc : 1;           /* page was accessed? */
    unsigned int pt_dirty : 1;         /* page was written? */
    unsigned int pt_mbz : 1;           /* must be zero */
    unsigned int pt_global: 1;         /* should be zero in 586 */
    unsigned int pt_avail : 3;         /* for programmer's use */
    unsigned int pt_base : 20;        /* location of page? */
};
```

如上所示，使用了C语言之中struct结构体中位域的概念，两个结构体的大小都是32位4个字节，将结构体中的各个部位分别指定到这32中的固定位置，用于实现页表项和页目录项所可以记录的信息。需要注意的是，由于在intel之中使用的是小端法，也就是低地址对应了低位，所以整个结构体的定义需要和直观感觉倒过来，才是正确的。

我的页表项和页目录项的结构的设计是参考了xv6和Linux的架构。他们的内容大致是相似的：

- base，就是我们所需要的物理页表项下标（或者说就是真实物理地址的前20位）
- avail：这三位是预留给内核的，内核可以在这里保存自己所需要的信息；
- global：忽略，没有实际意义

- mbz: 校验位, 必须是0, 否则表示可能出现了bug或者被恶意修改;
- dirty: 脏页标志位, 代表这一页是否已经被修改过, 计划后续在cache之中保存页的时候需要标识cache中的这一页相对于内存是否修改过
- acc: 访问位, 这一页是否曾经被访问过, 这是计划后续在cache中保存页的时候可能需要寻找victim page进行驱逐
- pcd: 标识这一页是否被放入到cache之中;
- pwt: 标识这一页的修改方式, 当cache中的页发生了修改之中, 可有两种内存修改方式: 写回 (write back) 或者是直写 (write through)
- usr: 标识这一页的访问权限, 是用户态还是内核态, 在后续实现内核态的时候需要

对于我们的实验来说, 无论是页表项还是页目录项, 需要关注的地方只有:

- 存在位P: 最后一位present, 代表这个页表或者这个页是否真实存在;
- 读写位W: 倒数第二位writable, 代表这个位置是否是可写的;

与此同时, 还需要实验一些比较基础的页表页目录读写函数, 方便后面直接调用, 减少代码复写量。下面的代码可以实现将一个页的相关信息填写到一个页表项之中, 需要注意的一点是, struct 的位域设计, 是从低位开始截取的而不是从高位开始, 所以为了得到地址的高20位存入到pt_base字段之中, 必须要把实际的物理地址addr右移12位才行。

```
syscall add_page_to_table(struct pt_t* pt, uint32* addr, int offset){
    (pt + offset)->pt_pres = 1;
    (pt + offset)->pt_write = 1;
    (pt + offset)->pt_user = 1;
    (pt + offset)->pt_pwt = 1;
    (pt + offset)->pt_pcd = 1;
    (pt + offset)->pt_acc = 1;
    (pt + offset)->pt_dirty = 1;
    (pt + offset)->pt_mbz = 1;
    (pt + offset)->pt_global = 0;
    (pt + offset)->pt_avail = 1;
    (pt + offset)->pt_base = (uint32)addr >> PTXSHIFT;
    return OK;
}
```

对于页目录项的填写函数也是同理可得:

```
syscall add_table_to_dir(struct pd_t * pd, struct pt_t* pt, int offset){
    (pd + offset)->pd_pres = 1;
    (pd + offset)->pd_write = 1;
    (pd + offset)->pd_user = 0;
    (pd + offset)->pd_pwt = 0;
    (pd + offset)->pd_pcd = 0;
    (pd + offset)->pd_acc = 0;
    (pd + offset)->pd_mbz = 0;
    (pd + offset)->pd_fmb = 0;
    (pd + offset)->pd_global = 0;
    (pd + offset)->pd_avail = 1;
    (pd + offset)->pd_base = (uint32)pt >> PTXSHIFT;
    return OK;
}
```

2.2 虚拟内存初始化

为了完成虚拟内存的初始化操作，首先要为“内核进程”进行操作，分配它所拥有的页表和页目录项。设置内核页表的页目录项地址为全局变量kpgdir，kpgdir既是内核页目录项的物理地址，同时也是内核页目录项的虚拟地址。由于Xinu的文本段(.text)和数据段(.data)以及块起始符号段(.bss)会从1MB的位置开始，并用符号end来标识结束，所以我们的内核页目录就设置在&end上取整到4kb的地址位置，并从这个位置开始，连续分配：

- 内核页目录, 1个页
- 用于映射低32MB的内核页表, 需要8个页
- 内核栈页表, 需要1个页

由于映射低32MB的内核页表需要填写完8个页，并且每一个页的1024个页表项全都都需要填写，所以为了减少代码复用可能导致的bug，专门设置了一个函数用于填写映射低32MB内核页的8个页表：

```
syscall init_core_pt(struct pt_t *pt, int n){
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < NPTENTRIES; j++){
            pt->pt_pres = 1;
            pt->pt_write=1;
            pt->pt_user=1;
            pt->pt_pwt=1;
            pt->pt_pcd=1;
            pt->pt_acc=1;
            pt->pt_dirty=1;
            pt->pt_mbz=1;
            pt->pt_global=0;
            pt->pt_avail=1;
            pt->pt_base = i * NPTENTRIES + j;
            pt++;
        }
    }
    return OK;
}
```

在这个函数的所有调用之中，n默认都是取8（因为32MB的内核空间需要8个页来完成映射），每一个外层循环会将一个完整的页表填写完成。

接下来，需要找到所有空闲的物理地址空间，方便之后分配内存页，为了实现这个功能，我直接在原有的minit函数上进行修改，去掉了原来用来串连起所有的memregion的链表结构memlist，同时也去掉了其节点结构体membblk的定义，而是重新设置了一个全局变量framelist，作为一个巨大的数组，其中的每一项存储的都是一个空闲的4kb大小的块的起始地址。同时，由于4kb对齐的地址最后12位一定是0，所以我将最后一位设置成为标志位，如果这一位是1，代表这这一个内存页已经被分配；如果这一位是0，代表这一个内存页还没有被分配，处于空闲可分配状态。另外使用两个全局变量framenum和usedframe，分别表示所有4kb初始空闲内存页的数量，以及已经使用了的内存页的数量。

在修改xinu之前首先运行了一遍，发现按照原来的架构，只有一个巨大的memregion结构，所以所有的空闲内存是一个巨大的整块。基于此，我设计了一个函数，可以将这一整块内存切割成为4kb大小的页：

```

void init_framelist(uint32 base,uint32 length){
    uint32 round_base = PGROUNDUP(base);
    length = base + length - round_base;
    while(length>=4096){
        framelist[framenum] = round_base;
        framenum++;
        round_base += 4096;
        length -= 4096;
    }
}

```

此外还要注意一点，一开始初始化“内核进程”的页目录以及页表结构时，是默认了从 PGROUNDUP (&end) 开始往后的10个页全部都是空闲的（在Xinu上这个事实可以被保证），所以在进行完这些初始化操作之后就需要在framelist之中将这些页全部都设置为已经使用，并在usedframe数值上进行修改。

为了方便对frame进行操作，专门实现了两个函数，可以搜索所有的空闲页，完成页的分配与回收，函数实现如下所示：

```

uint32 get_free_frame(){
    static int last = 0;
    if(usedframe==framenum)
        panic("free frame ran out\n");
    while(1){
        if(!(framelist[last]&1)){
            framelist[last] |= 1;
            usedframe++;
            return framelist[last] & -2;
        }
        last++;
        if(last>=framenum)
            last %= framenum;
    }
}

```

其中，last变量特地使用了static类型，可以记录上一次找到空闲页的位置，这一处的设计借鉴了分配进程号时所使用的getpid()函数，避免了每一次分配都要从头找起，提高了效率。

释放frame的函数原理类似，代码如下所示：

```

syscall release_frame(uint32 addr){
    int i;
    for(i=0;i<framenum;i++){
        if(framelist[i]-1==addr){
            framelist[i]-=1;
            usedframe--;
            //kprintf("release 0x%08X\n",addr);
            return OK;
        }
    }
    return SYSERR;
}

```

初始化内核进程页表的操作全部都在vminit函数之中进行，并在vminit之中调用了mininit，找到了全部的空闲页，代码如下所示：

```

void vminit(){
    int i,j;
    uint32 *page_dir = (uint32 *)PGROUNDUP((uint32)&end);
    kpgdir = page_dir;
    memset(page_dir, 0, PGSIZE);
    uint32 *page_tab = (uint32 *)((uint32)page_dir + PGSIZE);
    init_core_pt(page_tab, 8);
    for (i = 0; i < 8;i++)
        add_table_to_dir(page_dir, (uint32*)((uint32)page_tab+i*PGSIZE), i);
    page_tab = (uint32 *)((uint32)page_dir + 36 * 1024);
    memset(page_tab, 0, PGSIZE);
    add_page_to_table(page_tab, (uint32 *)0x6000, 1023);
    add_table_to_dir(page_dir, page_tab, 1021);
    add_table_to_dir(page_dir,page_dir,1023);//page_dir is pa and va
    meminit();
    for (i = 0; i < 10;i++)
        framelist[i] |= 1;
    usedframe += 10;
    //invalidate();
}

```

完成vminit之后，由start.S指引，会将kpgdir放入到%eax寄存器，再放入到%cr3之中，同时设置cr0寄存器相应的位，开始页式内存管理：

```

call    vminit /* initialize virtual memory */
/* set page directory */
movl    kpgdir, %eax
movl    %eax, %cr3
/* turn on paging */
movl    %cr0, %eax
orl     $(CR0_PG | CR0_WP), %eax
movl    %eax, %cr0
/* reset ESP to the high virtual address */
addl    $0xFF7F9000, %esp
addl    $0xFF7F9000, %ebp

```

(其实，最后两步设置%esp和%ebp时没有必要的，因为在高端内存这些东西会被自动地映射)

2.3 创建新进程

在完成vminit之后，跟据start.S中的引导，会将cr3寄存器的值设置为kpgdir，也就是内核页目录表的起始位置，然后在进程nulluser之中，会调用create函数创建start up进程。所以接下来就需要相应地修改create函数，使之满足虚拟内存的要求。

在虚拟内存的模式下，创建新进程的时候需要为新进程分配和填写：

- 新进程的页目录表，1个页
- 专门映射低32MB内核空间的页表，8个页
- 用于映射新进程栈空间的栈页表，1个页
- 以及新进程的栈空间，大小不确定，最少1个页，最多1024个页（因为限制了单个进程的栈空间大小不能超过4MB

跟据我的设计，所有的进程，他们页目录表的各个页目录项的位置和作用都是比较相似的：

- 前八项填写用于映射低32MB的内核空间
- 从第九项开始用于映射堆空间，当然这一块映射不是在create函数之中设置的，创建进程的时候是不需要创建堆空间的，堆空间全部都是由进程自己通过专门的函数来实现的，会在下面专门介绍

- 第1020项备用，用于映射该进程创建的新进程的栈页表
- 第1021项，用于当前进程栈页表
- 第1022项，用于映射该进程创建的新进程的页目录表
- 第1023项，用于映射当前进程的页目录

下面是对我的页目录项结构以及设计理念的进一步解释：

首先是页目录自映射机制。众所周知，我们在填写页表项的时候，需要填写的是物理地址，而不是虚拟地址。在很多的情况下，我们需要知道自己的页目录表所在的虚拟地址，才方便在开启分页之后向其中填入数据。所以说，我设计将页目录的物理地址填入到页表的第1023项上去，这样的话就会形成从页目录表到自身的自映射，只需要访问 $(1023 \ll 22) | (1023 \ll 12) | 0$ 就可以找到当前进程的页目录表的虚拟地址。

其次，在创建新进程的时候，我们不仅需要操作自己的页表内容，同时还需要为新进程填写一系列的页表项和页目录项内容。只要我们要写入某一块空间，就必须获得这个空间在当前进程的逻辑地址。换句话说，就是需要在当前的页目录表和页表中找出某一块空闲区域进行临时映射，这个方法同xv6所使用的“高端内存动态映射”方法非常相像，也是我的灵感来源。在当前页目录项的1022项专门用于映射新进程的页目录，这样的话直接通过逻辑地址 $(1023) \ll 22 | (1022) \ll 12 | 0$ 就可以获得创建的新进程的页目录物理地址。有了这个逻辑地址，就可以操作新进程的页目录表，向其中填入启动一个新的进程所需要的必要信息。

通过 $(1022) \ll 22 | (x \ll 12) | 0$ 就可以获得新进程的第x个页，通过这种方法，就可以令x从0取到7，首先填写完映射低32MB内核空间的8个页表，然后通过 $(1022) \ll 22 | (1021 \ll 12) | 0$ 得到新进程的栈页表，根据新进程所要求的stack size，通过宏PGROUNDUP进行4kb取整，得到新进程的栈所需要的页数量，通过get_free_frame()获取并倒序填写在栈页表项之中（因为栈空间是向低地址增长，所以要先填栈页表的第1023项，不够的话再填1022项、1021项.....）

但是仅仅这样的话，还有一个问题没有解决，那就是创建新进程的时候，需要往新进程的栈中填写STACKMAGIC、nargs、arg、INNIRET等信息，但是在我们目前的页目录架构之下，需要跳转三个表才能找得到新进程的栈空间(当前进程页目录 → 新进程页目录 → 新进程栈页表 → 新进程栈空间)而通过逻辑地址页表翻译最多只能跳转两层，所以在现行的架构下是不可能直接访问新进程的栈空间的。

为了解决这个问题，我创造性地将新进程的栈空间双重映射，新进程栈空间不仅按照常规被映射到了新进程的栈页表，同时还将新进程的栈页表填入到当前进程的页目录项的1020项，这样的话，相当于新进程的栈空间有了同当前进程栈空间相同的地理方位，只需要使用逻辑地址 $(1020 \ll 22) | (1023 \ll 12) | 0$ 就可以得到新进程的栈顶，从而得以往新进程的栈空间之中填写一系列的信息。

这一部分的代码如下所示：

```

/*****PAGE TABLE RELATED BEGIN*****/
uint32 newdir_p=get_free_frame();
uint32 curdir=(1023<<22)|(1023<<12)|0; //0xfffff000
add_table_to_dir(curdir,newdir_p,1022);
uint32 newdir_v=(1023<<22)|(1022<<12)|0;
for(i=0;i<8;i++)
    add_table_to_dir(newdir_v,get_free_frame(),i);
uint32 newtab_v=(1022<<22)|(0<<12)|0;
init_core_pt(newtab_v,8);
uint32 newstktab_p=get_free_frame();
add_table_to_dir(newdir_v,newstktab_p,1021);
add_table_to_dir(curdir,newstktab_p,1020);
add_table_to_dir(newdir_v,newdir_p,1023);
lcr3(proctab[currpid].pgdir);
/*****PAGE TABLE RELATED END*****/

```


此外，在填写新进程的栈时，尤其需要注意要弄清楚，究竟哪些地址是当前进程所看到的，哪些地址是需要给新进程看的，这两者之间会有一个前10位的页目录项下标的偏移。对于当前进程来说，页目录下标应当是1020，按照这个地址来填写才能填写到新进程的栈空间。但是对于新进程来说，他不应当看到1020开头的逻辑地址，这个地址对于新进程来说是错误的，它应当看到的是对应的，以1021开头的逻辑地址。因此，对于填入栈中的很多地址，需要处理一下，代码如下：

```

*--saddr = (long)funcaddr; /* Make the stack look like it's*/
/* half-way through a call to */
/* ctxsw that "returns" to the*/
/* new process */
/*--saddr = savsp; /* This will be register ebp */
*--saddr=(1021<<22)|(1023<<12)|4092;
/* for process exit */
savsp=(uint32) ((1021<<22)|((uint32) saddr & 0x3ffff));
//savsp = (uint32) ((1021<<22)|(1023<<12)|((uint32)saddr&0xffff));
*--saddr = 0x00000200; /* New process runs with */
/* interrupts enabled */

/* Basically, the following emulates an x86 "pushal" instruction*/
*--saddr = 0; /* %eax */
*--saddr = 0; /* %ecx */
*--saddr = 0; /* %edx */
*--saddr = 0; /* %ebx */
*--saddr = 0; /* %esp; value filled in below */
pushsp = saddr; /* Remember this location */
*--saddr = savsp; /* %ebp (while finishing ctxsw) */
*--saddr = 0; /* %esi */
*--saddr = 0; /* %edi */
saddr_trans=(uint32) ((1021<<22)|((uint32)saddr&0x3ffff));
*pushsp = (unsigned long) (prptr->prstkptr = (uint32 *)saddr_trans);
```

2.4 管理堆空间

管理堆空间的代码逻辑相对比较简单，首先用PGROUNDUP计算出申请的堆空间需要几个页，从当今进程的页目录的下标第8项处开始寻找（因为从0号页目录项一直到7号页目录项全部都是映射内核空间低32MB，已经填满了），如果遇到没有填写的页目录项，就立即通过get_free_frame()得到一个页作为页表填写进去，然后再在这个页表之中不断地填写4kb的内存页，直至满足请求的堆空间的大小。

注意allocmem的开始需要使用disable()函数来关中断，等到函数结束之后才restore()

```

char *allocmem(
    uint32 nbytes /* Size of memory requested */
)
{
    uint32 npages;
    intmask mask;
    uint32* memstart;
    struct pd_t* curdir=(struct pd_t*)((1023<<22)|(1023<<12)|0);
    int i,j;
    int startflag=1;
    nbytes=PGROUNDUP(nbytes);
    npages=nbytes/PGSIZE;
    mask=disable();
    if(npages==0){
        restore(mask);
        return SYSERR;
    }
```

```

}
for(i=8;i<NPENTRIES;i++){
    if(npages==0)
        break;
    if(curdir[i].pd_pres){
        struct pt_t* tab=(struct pt_t*)((1023<<22)|(i<<12)|0);
        for(j=0;j<NPTENTRIES;j++){
            if(npages==0)
                break;
            if(tab[j].pt_pres==0){
                if(startflag){
                    memstart=(uint32*)((i<<22)|(j<<12)|0);
                    startflag=0;
                }
                add_page_to_table(tab,get_free_frame(),j);
                npages--;
            }
        }
    }else{
        add_table_to_dir(curdir,get_free_frame(),i);
        i--;
        continue;
    }
}
restore(mask);
return (char*)memstart;
}

```

dellocmem也是同理可得，接受一个指针作为输入，这个指针应当指向请求堆空间开始的地址。一个页一个页地释放，直至将该指针所申请的所有空间全部释放完毕。跟据我的设计，如果一个页表的4MB空间全部都被释放了，这个页表本身并不会被释放，会被留到下一次allocmem的时候再次使用。这一部分的代码如下所示：

```

syscall deallocmem(
    char      *addr, /* Pointer to memory block */
    uint32     nbytes /* Size of block in bytes */
)
{
    intmask mask;
    int i,j;
    struct pd_t* curdir=(struct pd_t*)((1023<<22)|(1023<<12)|0);
    int dir_index=(uint32)addr>>22;
    int tab_index=((uint32)addr>>12) & 0x3ff;
    nbytes=PGROUNDUP(nbytes);
    int npages=nbytes/PGSIZE;
    mask = disable();
    if ((nbytes == 0)) {
        restore(mask);
        return SYSERR;
    }
    for(i=dir_index;i<NPENTRIES;i++){
        if(npages==0)
            break;
        struct pt_t* tab=(struct pt_t*)((1023<<22)|(i<<12)|0);
        for(j=tab_index;j<NPTENTRIES;j++){
            if(npages==0)
                break;

```



```

        if(release_frame(tab[j].pt_base<<PTXSHIFT)==SYSERR)
            panic("can't find the frame to release\n");
        memset(tab+j,0,4);
        npages--;
    }
    tab_index=0;
}
restore(mask);
return OK;
}

```

2.5 进程回收

当一个进程被杀死或者是因为种种原因自杀时，会进入到kill函数之中启动回收过程。如果是被其他进程杀死的，没有特殊的问题可以直接进行；但是如果是自杀，就需要考虑栈空间的回收问题。要想成功地完成回收，必须要使用当前进程的栈空间存储临时变量prptr以及在当前栈帧之中调用函数resched(),但是，一旦使用resched()将当前进程调度出去之后，就不可能再次调度回来，也就没有办法回收当前进程的栈空间了。为了解决这一问题，我再次设计使用了双重映射：

首先将当前进程的页目录表复制到逻辑地址（同时也是物理地址）的0x0位置处，然后将进程栈页表复制到物理地址0x100处，也就分别是第一个页和第二个页，然后将这些栈页全部都释放掉，释放完成之后调用lcr3宏定义，重新加载cr3，设置PDBR为0x0位置处，来利用这个临时的栈空间映射完成函数调用，从而正常地结束一个进程。而这两个页则无需额外处理，因为下一次杀死进程的时候自然会覆盖掉这两个页。

相关代码如下所示,专门实现了一个函数deallocvm，用于释放和清除当前进程的页表和页目录，同时还检测了内存泄漏情况，如果此时该进程所申请的堆空间也有未被释放的区域，也会被一并释放。

```

syscall deallocvm(char* addr,uint32 nbytes){
    struct pd_t* curdir=(struct pd_t*)((1023<<22)|(1023<<12)|0);
    memcpy((void*)0x0,(void*)curdir,PGSIZE);
    struct pt_t*stktab=(struct pt_t*)((1023<<22)|(1021<<12)|0);
    memcpy((void*)0x1000,(void*)stktab,PGSIZE);
    add_table_to_dir((struct pd_t*)0x0,(struct pt_t*)0x1000,1021);
    int i,j;
    for(i=8;i<NPENTRIES;i++){
        if(curdir[i].pd_pres==0)
            break;
        struct pt_t* tab=(struct pt_t*)((1023<<22)|(i<<12)|0);
        for(j=0;j<NPTENTRIES;j++){
            if(tab[j].pt_pres==0)
                break;
            if(release_frame(tab[j].pt_base<<PTXSHIFT)==SYSERR){
                kprintf("1: can't find frame 0x%08X when release in
deallocstk\n",tab[j].pt_base<<PTXSHIFT);
                panic("");
            }
            memset(tab+j,0,4);
        }
        if(release_frame(curdir[i].pd_base<<PTXSHIFT)==SYSERR){
            kprintf("2: can't find frame 0x%08X when release in
deallocstk\n",curdir[i].pd_base<<PTXSHIFT);
            panic("");
        }
    }

    memset(curdir+i,0,4);
}

```

```

    }
    for(i=NPTENTRIES-1;i>=0;i--){
        if(stktab[i].pt_pres==0)
            break;
        if(release_frame(stktab[i].pt_base<<PTXSHIFT)==SYSERR){
            kprintf("3: can't find frame 0x%08X when release in
deallocstk\n",stktab[i].pt_base<<PTXSHIFT);
            panic("");
        }
        memset(stktab+i,0,4);
    }
    release_frame(proctab[getpid()].pgdir);
    return OK;
}

```

2.6 shell 命令修改

shell的整体逻辑是，维护了一张命令表，对于用户在shell窗口输入的不同命令，对应地由字符串找到对应的函数指针，并创建进程。但是在创建进程的时候，尚未向栈中填入合适的参数，所以需要首先用一个tmpargs作为占位符先占据字符串指针应有的位置，然后通过调用addargs函数，不断地向新进程的栈空间中把命令行参数复制一份，并准备好填写各个参数的指针。因此，要想修改shell函数，使之支持echo等命令，关键还是在地址的偏移，也就是说哪些是为新进程填写栈时应用的地址，哪些是新进程应当看到和使用的地址，应该好好地区分开。

修改代码如下所示：

```

/* Find the second argument in process's stack */
uint32* trans_stkptr=(uint32*)(1020<<22|((uint32)(prptr->prstkptr) &
0x3fffff));
uint32* trans_stkbase=(uint32*)((1020<<22)|(1023<<12)|(4092));
for (search = (uint32 *)trans_stkptr;
    search < (uint32 *)trans_stkbase; search++) {

    /* If found, replace with the address of the args vector*/

    if (*search == (uint32)dummy) {
        *search = (uint32*)((1021<<22)|((uint32)argloc & 0x3fffff));
        restore(mask);
        //kprintf("finish addarg\n");
        return OK;
    }
}

```

3 实验效果截图

3.1 逻辑地址测试

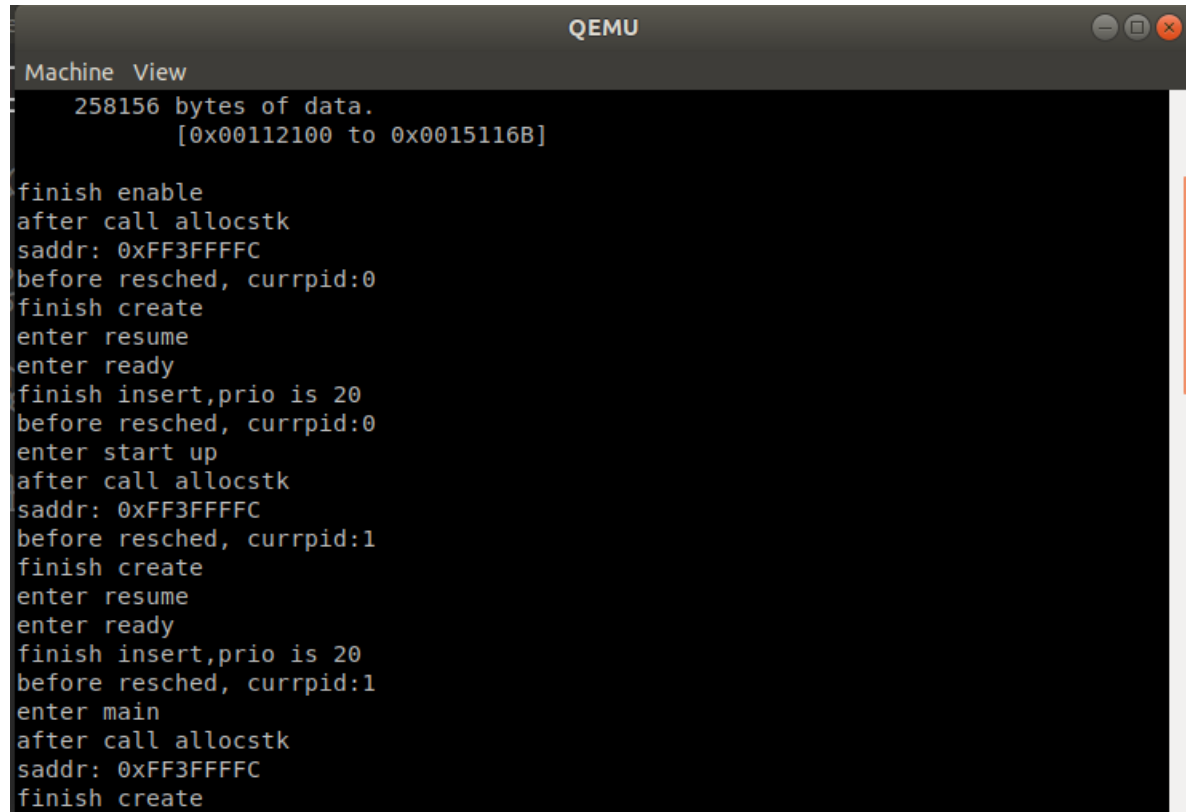
为了检验现在每一个进程看到的是否是虚拟地址，我编写了两个测试函数，在主函数之中分别创建和启动相应的进程，如下所示：

```

void test1(){
    int i,j,k;
    kprintf("test1: 0x%08X\n",&k);
}
void test2(){
    int i,j,k;
    kprintf("test2: 0x%08X\n",&k);
}

```

最终的全部测试输出如下所示:



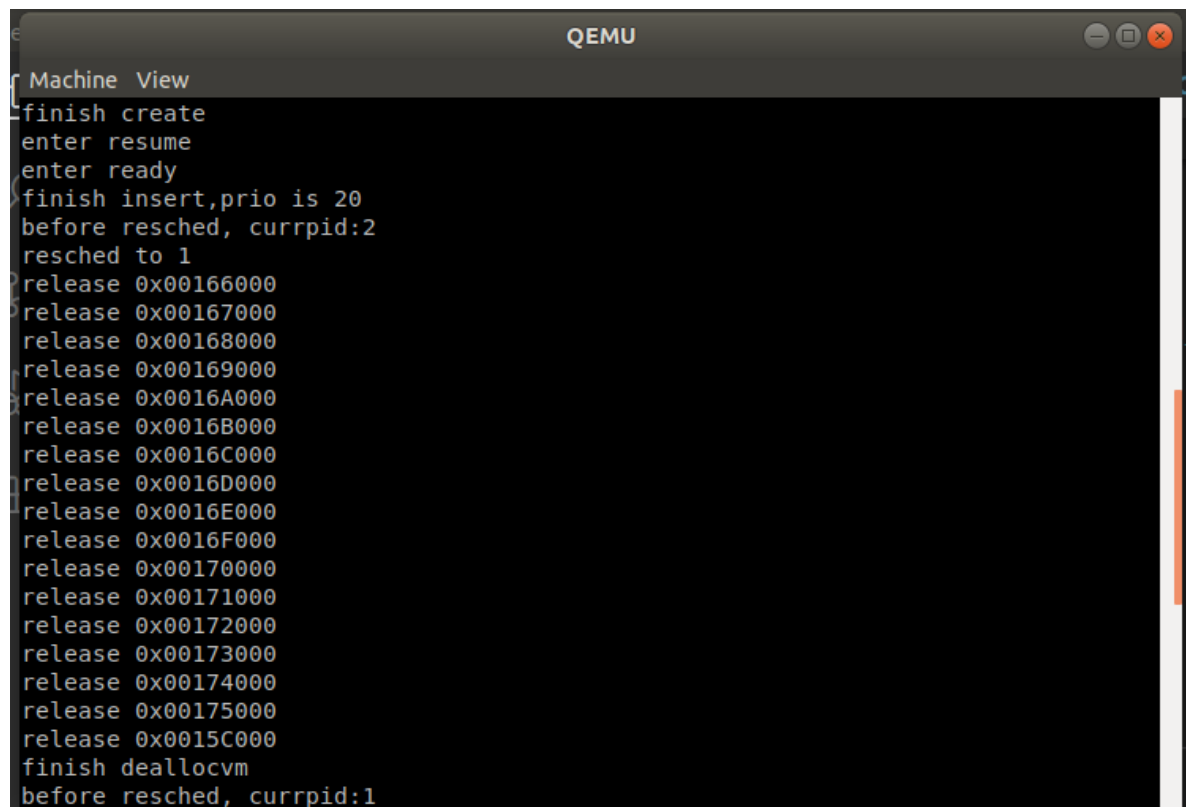
A screenshot of the QEMU Machine View window. The window title is "QEMU". The content shows the following text:

```

Machine View
258156 bytes of data.
[0x00112100 to 0x0015116B]

finish enable
after call allocstk
saddr: 0xFF3FFFFC
before resched, currpri:0
finish create
enter resume
enter ready
finish insert,prio is 20
before resched, currpri:0
enter start up
after call allocstk
saddr: 0xFF3FFFFC
before resched, currpri:1
finish create
enter resume
enter ready
finish insert,prio is 20
before resched, currpri:1
enter main
after call allocstk
saddr: 0xFF3FFFFC
finish create

```



A screenshot of the QEMU Machine View window. The window title is "QEMU". The content shows the following text:

```

Machine View
finish create
enter resume
enter ready
finish insert,prio is 20
before resched, currpri:2
resched to 1
release 0x00166000
release 0x00167000
release 0x00168000
release 0x00169000
release 0x0016A000
release 0x0016B000
release 0x0016C000
release 0x0016D000
release 0x0016E000
release 0x0016F000
release 0x00170000
release 0x00171000
release 0x00172000
release 0x00173000
release 0x00174000
release 0x00175000
release 0x0015C000
finish deallocvm
before resched, currpri:1

```

```
QEMU

Machine View
before resched, currpid:1
test1: 0xFF7FFFE8
release 0x0019A000
release 0x00190000
finish deallocvm
before resched, currpid:3
resched to 2
after call allocstk
saddr: 0xFF3FFFC
finish create
enter resume
enter ready
finish insert,prio is 20
before resched, currpid:2
test2: 0xFF7FFFE8
release 0x001A4000
release 0x0019A000
finish deallocvm
before resched, currpid:4
resched to 2

All user processes have completed.
```

可以看到，对于两个结构完全相同的函数test1和test2，他们的变量k的地址全部都是0xFF7FFFE8，也就是页目录1021项所对应的页表的第1023项所对应的内存页，同实现逻辑相吻合。

通过qemu中自带的info mem命令，可以清晰地看到当前所有的映射内存页，如下所示：

```
(qemu) info mem
0000000000000000-0000000002000000 0000000002000000 - rw
00000000ff7f0000-00000000ff800000 0000000000010000 - rw
00000000ffc00000-00000000ffc08000 0000000000008000 - rw
00000000ffffd000-00000000ffffe000 0000000000001000 - rw
00000000fffff000-0000000100000000 0000000000001000 - rw
(qemu)
```

其中第一项是低23MB内核空间的——映射；

第二项是栈空间，在页目录1021项对应页表的1023项；

第三项是由于自映射，对页目录之中前八个页表的映射

第四项是由于自映射，对栈页表的映射

第五项是由于自映射，对页目录自身的映射

3.2 分配和释放堆测试

使用如下的main主函数，多次申请和释放堆空间：

```
process main(void)
{
    kprintf("enter main\n");
    char* m1=alloca(4096);
    char* m2=alloca(4*1024*1024);
    char* m3=alloca(8192);
    char* m4=alloca(4096);
```

```

kprintf("allocmem:\nm1: 0x%08X\nm2: 0x%08X\nm3: 0x%08X\nm4:
0x%08X\n",m1,m2,m3,m4);
deallocmem(m1,4096);
deallocmem(m2,4*1024*1024);
deallocmem(m3,8192);
deallocmem(m4,4096);
}

```

运行输出效果如下图所示:



```

QEMU
Machine View
enter start up
after call allocstk
saddr: 0xFF3FFFC
before resched, currpri:1
finish create
enter resume
enter ready
finish insert,prio is 20
before resched, currpri:1
enter main
allocmem:
m1: 0x02000000
m2: 0x02001000
m3: 0x02401000
m4: 0x02403000
before resched, currpri:2
resched to 1
finish deallocvm
before resched, currpri:1
resched to 2

All user processes have completed.

```

使用gdb调试工具,在第二个kprintf语句的位置停下来,通过qemu的info mem命令,分别在分配和释放之后查看所有的映射页,结果如下所示:

```

(qemu) info mem
0000000000000000-0000000002404000 0000000002404000 -rw
00000000ff7f0000-00000000ff800000 0000000000010000 -rw
00000000ffc00000-00000000ffc0a000 000000000000a000 -rw
00000000ffffd000-00000000ffffe000 0000000000001000 -rw
00000000fffff000-0000000100000000 0000000000001000 -rw
(qemu) info mem
0000000000000000-0000000002000000 0000000002000000 -rw
00000000ff7f0000-00000000ff800000 0000000000010000 -rw
00000000ffc00000-00000000ffc0a000 000000000000a000 -rw
00000000ffffd000-00000000ffffe000 0000000000001000 -rw
00000000fffff000-0000000100000000 0000000000001000 -rw
(qemu)

```

结果显示,堆空间从32MB低内存的位置开始分配,符合实验逻辑。

3.3 shell 命令测试

使用shell中的echo命令对shell的功能进行测试,测试结果如下图所示:

A screenshot of a QEMU window titled "QEMU" with standard window controls. The main area is labeled "Machine View" and shows a black terminal with red text. The text indicates memory addresses [0x00100000 to 0x0010EFCC] and [0x001110E0 to 0x0015016B], followed by "258188 bytes of data." and "enter main". A large "XINU" logo is displayed in red, composed of dashed lines. Below the logo, it says "Welcome to Xinu!". Then, a shell prompt "xsh \$" is shown, followed by the command "echo hello world" and its output "hello world". Another command "echo I love Xinu" is entered, resulting in the output "I love Xinu". The prompt "xsh \$" appears again at the bottom.

```
Machine View
[0x00100000 to 0x0010EFCC]
258188 bytes of data.
[0x001110E0 to 0x0015016B]

enter main

XINU

Welcome to Xinu!

xsh $ echo hello world
hello world
xsh $ echo I love Xinu
I love Xinu
xsh $
```

可以看到shell的echo命令可以正常运行，符合实验的要求。