**Alos文档**

1.**进程管理:**

进程切换时最大的问题就是保存当前进程上下文，并在再次切换到此进程时，可以恢复上下文，以达到好像未被中断过的效果，linux早期版本使用的是直接切换tss的方式，但是这种方式开销大，并且由于硬件限制，导致最大进程数受到制约，我使用的办法是:在task数据结构中增加两个数据结构--pause\_stack和pause\_ip，用来保存任务切换时的堆栈和恢复时的ip，当切换发生时，将所有寄存器到压入当前进程的内核堆栈中，并将此时的esp和下条指令的地址保存到task中，只改变tss的esp0，使进程拥有自己的内核堆栈，恢复任务时，就根据内核堆栈和pause\_stack、pause\_ip来恢复上下文。代码如下:

void switch\_to(int next)

{

/\* 保存现场 \*/

asm("pusha\n"

"pushf\n"

"mov %%esp,%0\n"

"movl $1f,%1\n"

:"=m"(current->pause\_stack),"=m"(current->pause\_eip):);

/\* 切换任务 \*/

current=task[next]; //current

current->tss->esp0=(u32)(current)+PAGE\_SIZE; //tss

asm("mov %0,%%cr3\n" //cr3

"mov %1,%%esp\n" //esp

"jmp \*%2\n" //eip

::"g"(current->cr3),"m"(current->pause\_stack),"m"(current->pause\_eip));

/\* 恢复现场 \*/

asm("1:popf\n"

"popa\n");

return ;

}

2**.内存管理**:

主要分为物理页内存管理和虚拟内存管理两部分，关于物理内存，建立一个数组char mem\_map [ PAGING\_PAGES ]，每一项表示一页内存，数值代表共享的次数，这作为写时复制的依据。关于虚拟内存，linux早期版本只用到了一个页表，每个任务只能使用页表的一部分，对进程数量造成了限制，我使用的办法是：每个进程都会拥有自己的页表，切换进程时，切换了页表便可以完全隔离所有进程的用户空间，每个进程都映射了3G~4G的内核空间，访问用户空间时，再分配物理页，然后映射到对应地址上，并且新进程复制父进程的用户空间，也用到了写时复制技术。代码如下：

if(!(tmp=get\_free\_page()))

return -1;

newTask->cr3=tmp; //新进程获得页表

pde\_new=(u32 \*)(tmp+VM\_START); //新页目录表首地址

memset(pde\_new,0,PAGE\_SIZE);

pde\_cur=(u32 \*)(current->cr3+VM\_START); //当前进程页目录表首地址

/\* 内核空间:0xc0000000~0xffffffff+1

复制pde表中的对应的表项，与之共享指向内核的pte表，从而共享内核空间 \*/

for(i=VM\_START>>22;i<1024;i++)

pde\_new[i]=pde\_cur[i];

/\* 用户空间:0~0xc0000000(VM\_START)

将当前进程的pte表中的权限改为只读，新进程建立新的pte表复制表项，并增加页面引用次数

注意:与共享内核页面不同，共享用户空间需要建立新的pte复制原内容 \*/

for(i=0;i<VM\_START>>22;i++){

if(!(pde\_cur[i] & 1))//页目录表项不存在

continue ;

pte\_cur=(u32 \*)((pde\_cur[i] & 0xfffff000)+VM\_START);

//页目录指向新建立的页表，复制页表内容

if(!(tmp=get\_free\_page()))

return -1;

pde\_new[i]=tmp | P\_ATTR\_USER\_RDWR;

pte\_new=(u32 \*)(tmp+VM\_START);

memset(pte\_new,0,PAGE\_SIZE);

for(n=0;n<1024;n++)

{

pte\_cur[n] &= ~0x2; //只读权限

pte\_new[n] = pte\_cur[n];

mem\_map[ADDR2MAP(pte\_cur[n])]++;//低12位对页号无影响

}

}

内核起始虚拟地址3G对应物理地址0，后面的内存都呈线性映射关系，因此，内核可以直接加上3G的偏移量，访问1G以内的任意物理地址，内核和磁盘缓冲区只使用0~1M的物理内存，其余全部用于自由分配。

由于，我参考的linux0.11内核所需的数据结构较少用到动态分配的内存，即使用到，比如新建进程时的task\_struct也是直接需要正好一页内存，所以直接得到一页物理内存，加上3G偏移量，便可直接访问，文件结点，文件描述符等数据结构，直接用数组表示，在编译时，便放在数据区。所以，简化了内存分配的难度，如果，物理内存大于1G或者想动态分配所有数据结构，那么内存分配模型还需大的改变，可以使用slab等分配算法。

写时复制的原理，运用到了页表的读写属性，当与其他进程共享物理页时，将各自页表对应的项改为只读，如果，有一个进程想要写时，便申请一个新的物理页，映射为可读写，并且复制页内容。代码如下:

if(mem\_map[ADDR2MAP(oldPage)]>1){//页面共享

if(!(newPage=get\_free\_page()))

panic("no memory");

pte[(err\_virAddr>>12)&0x3ff]=newPage | P\_ATTR\_USER\_RDWR;//可读写页

//复制页面内容

memcpy((void \*)((newPage&0xfffff000)+VM\_START),\

(void \*)((oldPage&0xfffff000)+VM\_START),PAGE\_SIZE);

mem\_map[ADDR2MAP(oldPage)]--;

}

else{ //未共享，则取消写保护

pte[(err\_virAddr>>12)&0x3ff] = oldPage | 0x2;

}

3.**文件管理**：

Linux早期版本用的是minix文件 系统，我实现的是ext2的文件系统，ext2文件系统主要包含superblock、group、inode等模块。Ext2最大的改进就是分组管理，可以更快的检索，也支持更大的容量，文件系统的结构也相对比较清楚。在定位结点时，先得到group信息，然后加上group的偏移量就可以快速的得到指定结点。代码如下：

if (!(sb=get\_super(inode->i\_dev)))

panic("trying to read inode without dev");

//结点所在组号

group=(inode->i\_num-1) / sb->s\_inodes\_per\_group;

if(!(bh=get\_group(sb,group,&gd)))

panic("unable to read group");

//结点所在物理块号

block=gd->bg\_inode\_table+(inode->i\_num-1) % sb->s\_inodes\_per\_group /INODES\_PER\_BLOCK;

brelse(bh);

if (!(bh=bread(inode->i\_dev,block)))

panic("unable to read i-node block");

//根据偏移读结点

\*(struct d\_inode \*)inode =

((struct d\_inode \*)bh->b\_data)

[(inode->i\_num-1)%sb->s\_inodes\_per\_group%INODES\_PER\_BLOCK];