《操作系统实验》Lab5 实验报告

朱恬骅

09300240004 计算机科学与技术

1 实验目标

1. 了解Linux的内存管理
2. 熟悉fork() 的机制

2 实验要求

创建一个系统调用myfork()，使得子进程可以和父进程共享数据内容。

3 实验原理

3.1 Linux 的fork()

Linux使用fork()函数来创建子进程。由fork创建的新进程被称为子进程。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是新进程（子进程）的进程id。将子进程id返回给父进程的理由是：因为一个进程的子进程可以多于一个，没有一个函数使一个进程可以获得其所有子进程的进程id。对子进程来说，之所以fork返回0给它，是因为它随时可以调用getpid()来获取自己的pid；也可以调用getppid()来获取父进程的id。（进程id 0总是由交换进程使用，所以一个子进程的进程id不可能为0）。

执行fork之后，操作系统会复制一个与父进程完全相同的子进程，虽说是父子关系，但是在操作系统看来，他们更像兄弟关系，这2个进程共享代码空间，但是数据空间是互相独立的，子进程数据空间中的内容是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行到的位置（两进程的程序计数器pc值相同，也就是说，子进程是从fork返回处开始

执行的），但有一点不同，如果fork成功，子进程中fork的返回值是0，父进程中fork的返回值是子进程的进程号，如果fork不成功，父进程会返回错误。

fork是在system_call.s中定义的。通过汇编代码，程序可以得到全部用户寄存器的值，并且程序的指令地址寄存器可以通过call语句和栈操作得到。首先通过调用find_empty_process获取任务数组中还没有被使用的空项。因为Linux 0.11 只能同时运行64个进程，如果已经有64个进程在运行，则fork会因为没有空项而出错返回。然后系统为新建的进程在内存区中申请一页内存来存放其相关数据，并复制当前进程任务数据结构中的所有内容，即运行copy_process。为了保存下各种寄存器的数值，这些数值被作为copy_process的参数传递进去，然后copy_process利用这些参数的数值初始化新的进程的任务数据结构PCB，并在下次操作系统调度的时候真正运行。

copy_process用fork传入的参数初始化新进程的PCB，并且进行写时复制。这一复制过程是使用copy_mem实现的。而具体到和本次实验有关的数据复制，则通过调用copy_page_tables进行实质的复制过程。因而我们所需要修改的也是copy_page_tables。

3.2 内存管理

在Linux 0.11 内核中，在进行地址映射时，我们需要首先分清3 种地址以及它们之间的变换概念：a. 程序（进程）的逻辑地址；b. CPU 的线性地址；c. 实际物理内存地址。逻辑地址（Logical Address）是指有程序产生的与段相关的偏移地址部分。在Intel 保护模式下即是指程序执行代码段限长内的偏移地址（假定代码段、数据段完全一样）。应用程序员仅需与逻辑地址打交道，而分段和分页机制对他来说是完全透明的，仅由系统编程人员涉及。线性地址（Linear Address）是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386 的线性地址空间容量为4G。

物理地址（Physical Address）是指出现在CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。虚拟内存（Virtual Memory）是指计算机呈现出要比实际拥有的内存大得多的内存量。因此它允许程序员编制并运行比实际系统拥有的内存大得多的程序。这使得许多大型项目也能够在具有有限内存资源的系统上实现。一个很恰当的比喻

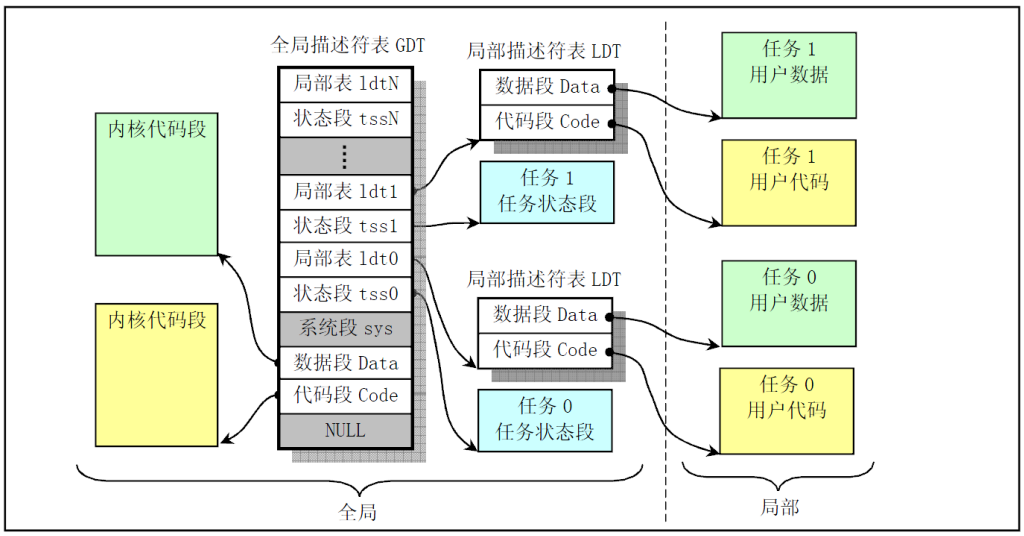

Figure 1: Linux 系统中虚拟地址空间分配图

是：你不需要很长的轨道就可以让一列火车从上海开到北京。你只需要足够长的铁轨（比如说3 公里）就可以完成这个任务。采取的方法是把后面的铁轨立刻铺到火车的前面，只要你的操作足够快并能满足要求，列车就能象在一条完整的轨道上运行。这也就是虚拟内存管理需要完成的任务。在Linux 0.11 内核中，给每个程序（进程）都划分了总容量为64MB 的虚拟内存空间。因此程序的逻辑地址范围是0x0000000 到0x4000000。有时我们也把逻辑地址称为虚拟地址。因为与虚拟内存空间的概念类似，逻辑地址也是与实际物理内存容量无关的。

Intel CPU 使用段（Segment）的概念来对程序进行寻址。每个段定义了内存中的某个区域以及访问的优先级等信息。而每个程序都可有若干个内存段组成。程序的逻辑地址（或称为虚拟地址）即是用于寻址这些段和段中具体地址位置。在Linux 0.11 中，程序逻辑地址到线性地址的变换过程使用了CPU 的全局段描述符表GDT 和局部段描述符表LDT。由LDT 映射的地址空间称为全局地址空间，由LDT 映射的地址空间则称为局部地址空间，而这两者构成了虚拟地址的空间。具体的使用方式见图1所示。

3.2.1 Linux中的内存管理

当进程A 使用系统调用fork 创建一个子进程B 时，由于子进程B 实际上是父进程A 的一个拷贝，因此会拥有与父进程相同的物理页面。也即为了达到节约内存和加快创建速度的目标，fork()函数会让子进程B 以只读方式共

享父进程A 的物理页面。同时将父进程A 对这些物理页面的访问权限也设成只读。详见memory.c 程序中的copy_page_tables()函数。这样一来，当父进程A 或子进程B 任何一方对这些以共享的物理页面执行写操作时，都会产生页面出错异常(page_fault int14)中断，此时CPU 会执行系统提供的异常处理函数do_wp_page()来试图解决这个异常。do_wp_page()会对这块导致写入异常中断的物理页面进行取消共享操作（使用un_wp_page()函数）, 为写进程复制一新的物理页面，使父进程A 和子进程B 各自拥有一块内容相同的物理页面。这时才真正地进行了复制操作(只复制这一块物理页面)。并且把将要执行写入操作的这块物理页面标记成可以写访问的。最后，从异常处理函数中返回时，CPU 就会重新执行刚才导致异常的写入操作指令，使进程能够继续执行下去。因此，对于进程在自己的虚拟地址范围内进行写操作时，就会使用上面这种被动的写时复制操作，也即：写操作→ 页面异常中断→ 处理写保护异常→ 重新执行写操作。而对于系统内核，当在某个进程的虚拟地址范围内执行写操作时，例如，进程调用某个系统调用，而该系统调用会将数据复制到进程的缓冲区域中，则会主动地调用内存页面验证函数write_verify()，来判断是否有页面共享的情况存在，如果有，就进行页面的写时复制操作。

4 实验步骤

4.1 添加sys_myfork函数

在文件kernel/system_call.s中：

```
.align 2
sys_myfork:
        call find_empty_process
        testl %eax,%eax
        js 1f
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call mycopy_process
        addl $20,%esp
1:      ret
```

4.2 在复制进程信息时，指定使用修改的内存复制函数

在文件kernel/fork.c中：

```
int mycopy_process(int nr,long ebp,long edi,long esi,
   long gs,long none,
                long ebx,long ecx,long edx,
                long fs,long es,long ds,
                long eip,long cs,long eflags,long esp,
                   long ss)
{
        struct task_struct *p;
        int i;
        struct file *f;

        p = (struct task_struct *) get_free_page();
        if (!p)
                return -EAGAIN;
        task[nr] = p;
        *p = *current;  /* NOTE! this doesn't copy the
           supervisor stack */
        p->state = TASK_UNINTERRUPTIBLE;
        p->pid = last_pid;
        p->father = current->pid;
        p->counter = p->priority;
        p->signal = 0;
        p->alarm = 0;
        p->leader = 0;          /* process leadership
           doesn't inherit */
        p->utime = p->stime = 0;
        p->cutime = p->cstime = 0;
        p->start_time = jiffies;
        p->tss.back_link = 0;
        p->tss.esp0 = PAGE_SIZE + (long) p;
        p->tss.ss0 = 0x10;
        p->tss.eip = eip;
        p->tss.eflags = eflags;
        p->tss.eax = 0;
        p->tss.ecx = ecx;
```

```
p->tss.edx = edx;
p->tss.ebx = ebx;
p->tss.esp = esp;
p->tss.ebp = ebp;
p->tss.esi = esi;
p->tss.edi = edi;
p->tss.es = es & 0xffff;
p->tss.cs = cs & 0xffff;
p->tss.ss = ss & 0xffff;
p->tss.ds = ds & 0xffff;
p->tss.fs = fs & 0xffff;
p->tss.gs = gs & 0xffff;
p->tss.ldt = _LDT(nr);
p->tss.trace_bitmap = 0x80000000;
if (last_task_used_math == current)
        __asm__("clts ; fnsave %0"::"m" (p->tss
            .i387));
if (mycopy_mem(nr,p)) { /* LAB5 */
        task[nr] = NULL;
        free_page((long) p);
        return -EAGAIN;
}
for (i=0; i<NR_OPEN;i++)
        if ((f=p->filp[i]))
                f->f_count++;
if (current->pwd)
        current->pwd->i_count++;
if (current->root)
        current->root->i_count++;
if (current->executable)
        current->executable->i_count++;
set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->
    tss));
set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->
    ldt));
p->state = TASK_RUNNING;        /* do this last
    , just in case */
return last_pid;
```

```
}
```

4.3 在复制内存信息时，指定使用修改过的内存页表复制函数

```
int mycopy_mem(int nr,struct task_struct * p)
{
        unsigned long old_data_base,new_data_base,
           data_limit;
        unsigned long old_code_base,new_code_base,
           code_limit;

        code_limit=get_limit(0x0f);
        data_limit=get_limit(0x17);
        old_code_base = get_base(current->ldt[1]);
        old_data_base = get_base(current->ldt[2]);
        if (old_data_base != old_code_base)
                panic("We don't support separate I&D");
        if (data_limit < code_limit)
                panic("Bad data_limit");
        new_data_base = new_code_base = nr * 0x4000000;
        p->start_code = new_code_base;
        set_base(p->ldt[1],new_code_base);
        set_base(p->ldt[2],new_data_base);
        if (mycopy_page_tables(old_data_base,
           new_data_base,data_limit)) {
                printk("free_page_tables: from copy_mem
                   \n");
                free_page_tables(new_data_base,
                   data_limit);
                return -ENOMEM;
        }
        return 0;
}
```

4.4 修改内存页表复制时的行为

在文件mm/memory.c中：首先定义变量i和pageCount。因为size 为代码

区、数据字节长度，而一个页表项对应内存4KB，因此$((unsigned)(size + 0xfff)) >> 12$，这些内存页无需设为只读。所以需要判断是否写操作针对这一页面进行；用pageCount保存这一数值。而i则记录了复制过程当前进行到哪一页面。修改的行体现了这一点。

```
int mycopy_page_tables(unsigned long from,unsigned long
    to,long size)
{
        unsigned long * from_page_table;
        unsigned long * to_page_table;
        unsigned long this_page;
        unsigned long * from_dir, * to_dir;
        unsigned long nr;
        unsigned long i = 0, pageCount = 0; /* LAB5 */

        if ((from&0x3fffff) || (to&0x3fffff))
                panic("copy_page_tables called with
                   wrong alignment");
        from_dir = (unsigned long *) ((from>>20) & 0
           xffc); /* _pg_dir = 0 */
        to_dir = (unsigned long *) ((to>>20) & 0xffc);
        pageCount = ((unsigned) (size+0xfff)) >> 12; /*
            LAB5 */
        size = ((unsigned) (size+0x3fffff)) >> 22;

        for( ; size--->0 ; from_dir++,to_dir++) {
                if (1 & *to_dir)
                        panic("copy_page_tables:
                           already exist");
                if (!(1 & *from_dir)) {
                        i += 1024; continue; /* LAB5 */
                }
                from_page_table = (unsigned long *) (0
                   xfffff000 & *from_dir);
                if (!(to_page_table = (unsigned long *)
                    get_free_page()))
                        return -1;      /* Out of
```

```
                        memory, see freeing */
                *to_dir = ((unsigned long)
                    to_page_table) | 7;
                nr = (from==0)?0xA0:1024;
                for ( ; nr--- > 0 ; from_page_table++,
                    to_page_table++) {
                        this_page = *from_page_table;
                        i++;
                        if (!(1 & this_page))
                                continue;
                        if (i >= pageCount) /* LAB5 */
                                this_page &= ~2; /*
                                    LAB5, this was not
                                    conditioned */
                        *to_page_table = this_page;
                        if (this_page > LOW_MEM) {
                                *from_page_table =
                                    this_page;
                                this_page -= LOW_MEM;
                                this_page >>= 12;
                                mem_map[this_page]++;
                        }
                }
        }
        invalidate();
        return 0;
}
```

4.5 注册myfork的系统调用

在include/linux/sys.h中：

```
extern int sys_myfork(); /* LAB5 */

fn_ptr sys_call_table[] = { sys_setup, ... sys_myfork
    /* LAB5 */ };
```

在include/unistd.h中：

```
#define __NR_myfork        72        /* LAB5 */
```

4.6 测试程序

修改/usr/include/unistd.h使与修改过的程序保持一致，然后键入测试程序：

```
#define __LIBRARY__
#include <stdio.h>
#include <unistd.h>

_syscall0(int, myfork);

int data = 10;
int child_process(){
        printf("child process %d, data %d\n",getpid(),
           data);
        data = 20;
        printf("child process %d, data %d\n",getpid(),
           data);
}
int main(){
        if(fork()==0){
                child_process();
        }
        else{
                sleep(1);
                printf("parent process %d, data %d\n",
                   getpid(),data);
        }
}
```

屏幕输出如图2所示。
这就完成了实验。

5 遇到的问题和解决方法

1. 尝试在页错误处理函数判断是否为据区再进行写时复制，但实现较困

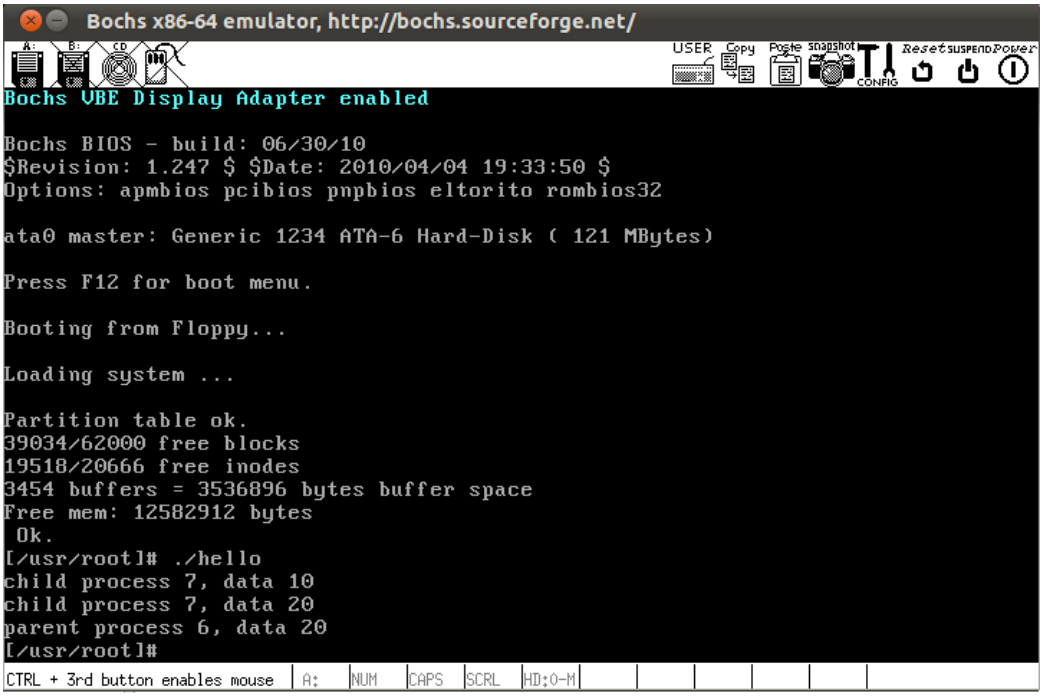

Figure 2: 测试程序运行时截图

难，无法确定数据区大小。
解决：在copy_page_tables中判断是否数据区再决定如何复制。

2. 在system_call.s中增加myfork后无法编译通过。
   解决：需要在.globl中定义。

3. 对目录项计数的变量i在!$this_page$时未累加，运行时出错。
   解决：将$i++$放在$if(!(1\&this_page))$之前。

4. 计算$pageCount$的语句放在了对$size$更新之后，导致运行出错。
   解决：将$pageCount$放在更新$size$之前。

6 实验收获

1. 了解Linux中fork的实现；

2. 学习了有关内存管理的内容。

2011年11月11日