北京交通大学实验报告

课程名称 ： **操作系统**
实验题目 ： **Linux进程控制的实现机制**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **何永忠**
报告日期 ： **2023年11月6日**

目录

---

1. 开发运行环境和工具

- **代码阅读环境**：Windows11 + VS Code
- **开发测试环境**：Bochs

2. 实验过程、分析和结论

2.1. fork系统调用和内核函数

首先在`init/main.c`-->`main()`中看到了`fork()`函数

```
if (!fork()) {      /* we count on this going ok */
    init();
}
```

即为如果`fork()`没有发现父进程（后面会讲到），则初始化系统，创建系统初始进程。

追踪`fork()`，发现在`main.c`开头有一行：

```
static inline _syscall0(int,fork)
```

即调用`fork()`时，编译器会将`fork()`内嵌为`_syscall0(int,fork)`，即在编译时直接替换为`_syscall0(int,fork)`的函数代码，以免调用堆栈。

追踪`_syscall0()`，在`include/unistd.h`中：

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

即，调用`fork()`就是在调用

```
"int $0x80" : "=a" (__res) : "0" (__NR_fork)
```

- `"int $0x80"`：表示触发软中断。在x86体系结构中，软中断0x80用于进入内核态进行系统调用。
- `: "=a" (__res)`：对输出操作数的约束。`=a`表示将寄存器`eax`的值分配给`__res`，并且这个值会在系统调用结束后存储系统调用的返回值。
- `: "0" (__NR_fork)`：对输入操作数的约束。`0`表示使用与前面输出操作数相同的寄存器，这里是`eax`。`__NR_fork`是系统调用号的宏，被定义为`2`，用于标识调用的具体系统调用。

那么调用`fork()`时，就是调用了2号系统软中断。

接下来到`kernel/system_call.s`中查看软中断函数`_system_call`代码是如何执行的：首先进行了一系列参数的检查和原寄存器入栈保存操作，这里不予赘述。然后执行了

```
call _sys_call_table(,%eax,4)
```

按照传进来的参数，实际上调用了`_sys_call_table(,2,4)`

由于这是汇编文件，调用`_sys_call_table()`实际上就是在调用C语言代码中的`sys_call_table()`函数，因为在编译时，编译器会在C代码函数前加上`_`作为其在汇编代码中的命名（实验证明见附录）。

那么追踪`sys_call_table()`，在`include/linux/sys.h`中发现了系统调用表：

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid,sys_setregid };
```

其中变量类型`fn_ptr`是函数指针。所以实际上`_sys_call_table(,2,4)`调用了第3个`sys_fork`函数。`sys_fork`函数在汇编后会被命名成`_sys_fork`。

我在`kernel/system_call.s`中找到了对其汇编后的调用`_sys_fork`，即系统调用`fork`的函数原型就是：

```
_sys_fork:
    call _find_empty_process
    testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
    addl $20,%esp
1:  ret
```

其中依次调用了`_find_empty_process`和`_copy_process`函数。这两个函数的C代码`find_empty_process()`和`copy_process()`在`kernel/fork.c`中：

- `find_empty_process()`：寻找一个空闲的pid，并为其在任务数组中为新任务寻找一个空闲项，并返回项号。
- `copy_process()`：复制父进程的各项值并初始化（这么做是因为复制比新建快）。

在这之中，代码为新进程创建了进程控制块`struct task_struct *p`，到`include/linux/sched.h`中跟踪其定义：

```
struct task_struct {
/* these are hardcoded - don't touch */
    long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    // 表示进程的状态，-1 表示不可运行，0 表示可运行，大于0 表示已停止。
    long counter;   // 计时器，用于调度，当计时器减至0时，进程可能被调度出去。
    long priority;  // 进程的优先级。
    long signal;    // 当前进程正在处理的信号。
    struct sigaction sigaction[32]; // 存储信号处理程序的数组。
    long blocked;   /* bitmap of masked signals */
    // 用于表示被阻塞的信号的位图。
/* various fields */
```

```
    int exit_code;
    unsigned long start_code,end_code,end_data,brk,start_stack;
    long pid,father,pgrp,session,leader;
    // pid 进程ID
    // father 父进程ID
    // pgrp 进程组ID
    // session 会话ID
    // leader 会话的领导者ID
    unsigned short uid,euid,suid;   // 用户ID、有效用户ID、保存的用户ID
    unsigned short gid,egid,sgid;   // 组ID、有效组ID、保存的组ID
    long alarm; // 进程的闹钟定时器
    long utime,stime,cutime,cstime,start_time;
    // utime 用户态运行时间
    // stime 系统态运行时间
    // cutime 子进程的用户态运行时间
    // cstime 子进程的系统态运行时间
    // start_time 进程开始运行的时间
    unsigned short used_math;   // 表示是否使用了数学协处理器
/* file system info */
    int tty;    // 表示进程是否有终端
    /* -1 if no tty, so it must be signed */
    unsigned short umask;   // 文件创建的权限屏蔽掩码
    struct m_inode * pwd;   // 当前工作目录
    struct m_inode * root;  // 当前根目录
    struct m_inode * executable;    // 执行文件的指针
    unsigned long close_on_exec;    // 用于在执行新程序时关闭文件的标志位
    struct file * filp[NR_OPEN];    // 文件指针数组，用于表示打开的文件
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
    struct desc_struct ldt[3];  // 进程的局部描述符表，包含代码段、数据段、堆栈段的描述符
/* tss for this task */
    struct tss_struct tss;  // 任务状态段，包含了一些处理器的状态信息
};
```

最后，返回新建进程的pid。

于是，`fork`调用就成功创建了新的进程和pid，并把pid返回给了调用`fork`的进程。

2.2. exit系统调用和内核函数

首先在`init/main.c`-->`main()`中看到了`_exit()`函数：

```
if (!(pid=fork())) {
    close(0);
    if (open("/etc/rc",O_RDONLY,0))
        _exit(1);
    execve("/bin/sh",argv_rc,envp_rc);
    _exit(2);
}
```

追踪`_exit()`，到`lib/_exit.c`中看到定义：

```
volatile void _exit(int exit_code)
{
    __asm__("int $0x80"::"a" (__NR_exit),"b" (exit_code));
}
```

和 `fork()` 一样，`_exit()` 也是使用了系统中断来执行。

`__NR_exit` 的宏定义为系统调用号1，所以实际上：`__NR_exit` 的值被加载到 `%eax` 寄存器；`exit_code` 的值被加载到 `%ebx` 寄存器，作为 `exit` 系统调用的参数，即退出码。

于是在 `kernel/system_call.s` 中，软中断函数 `_system_call` 代码执行了 `call _sys_call_table(,1,4)`，即调用了系统调用表中的 `sys_exit`。

`sys_exit` 的定义在 `kernel/exit.c` 中：

```
int sys_exit(int error_code)
{
    return do_exit((error_code&0xff)<<8);
}
```

再追踪 `do_exit()` 的代码，主要执行以下任务：

- 释放当前进程的页表
- 处理与当前进程相关的其他进程
- 关闭当前进程打开的文件
- 释放当前进程的pwd、root和executable相关资源
- 处理当前进程是会话领导者等情况
- 通知父进程
- 调度下一个进程

至此，本进程结束退出，exit系统调用完毕。

2.3. 原子性实现

纵观代码，进程间的切换主要有两种方式，一种是主动切换，如 `pause()`，其作用就是主动将当前进程挂起，切换到下一个进程任务执行，需要调用 `schedule()`。在 `schedule()` 中会判断要操作的进程的状态是否是可中断的。

另一种是通过开关中断，主动允许/不允许中断打断当前进程。在 `include/asm/system.h` 中：

```
#define sti() __asm__ ("sti"::)
#define cli() __asm__ ("cli"::)
```

因此，在代码中使用 `sti()` 和 `cli()` 就能实现开/关中断。

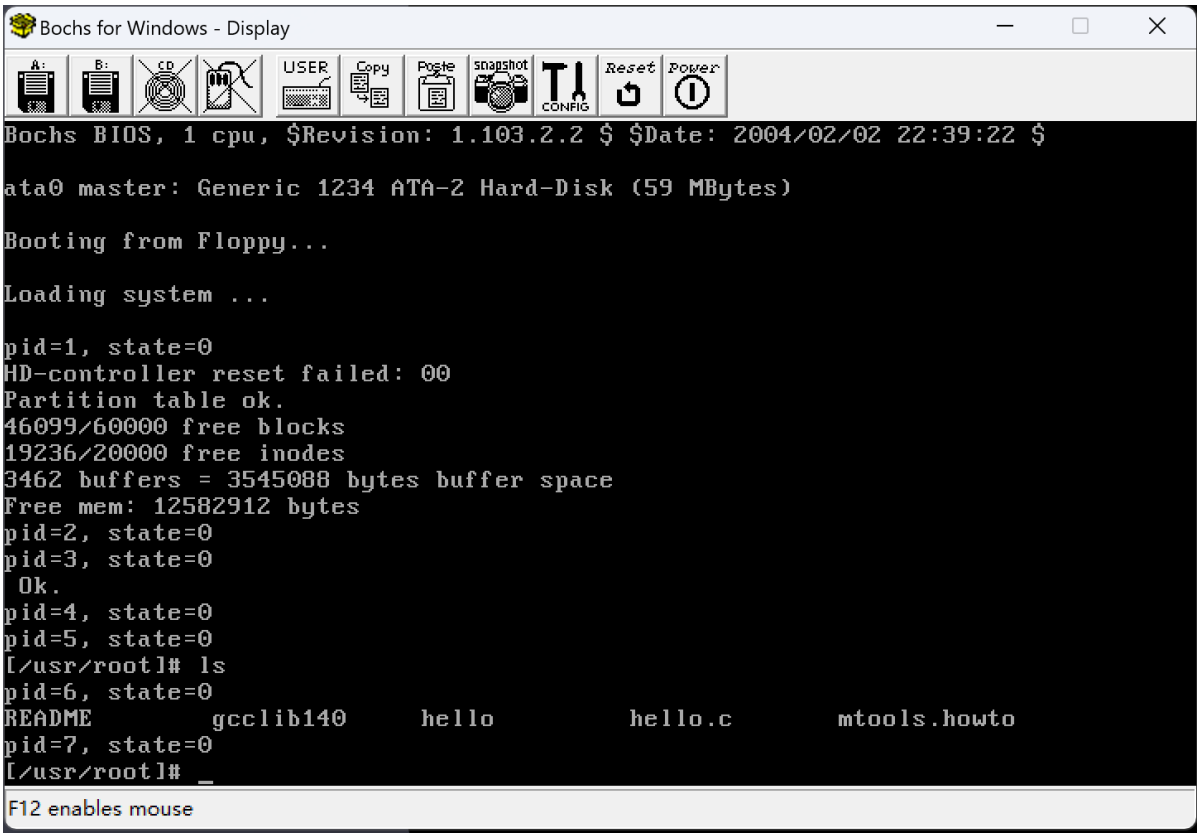
在 `fork` 和 `exit` 系统调用代码中，并没有调用 `pause()` 主动切换，也不会因为硬件中断而导致共享资源出错（对栈的使用得当），因此实现了原子性。

2.4. 修改fork系统调用

修改 `kernel/fork.c` 文件中 `copy_process()` 函数定义，在 `return` 前打印出新进程的pid和状态：

```
p->state = TASK_RUNNING;    /* do this last, just in case */
printk("pid=%d, state=%d", last_pid, p->state);
return last_pid;
```

运行结果如下：



`state`为0表示当前进程（在刚刚创建时）是就绪态和运行态。

2.5. Makefile文件

首先，文件定义了编译必要的工具和选项，以及根设备、目标文件等；

接下来定义了编译规则，如从`.c`文件编译出`.s`文件等；

然后定义最后生成`Image`镜像文件；

规定`Image`文件的构建规则、写入磁盘规则；

定义构建工具的规则；

定义编译完后要清理的文件；

创建备份；

规定项目源码文件之间构建的依赖关系。

附录

编译`kernel/fork.c`：

```
[usr/src/linux/kernel]# gcc -E fork.c -fork.i
[usr/src/linux/kernel]# gcc -S fork.i -fork.s
[usr/src/linux/kernel]# vi fork.s
```

则可以看到，函数`verify_area()`变成了`_verify_area`，函数`copy_mem()`变成了`_copy_mem`。