操作系统 Linux 系统调用扩充实验报告

小组：202002001021 李天豪 | 202002001037 谭承浩

一、实验目的及分工

扩充 linux0.11 内核 0 版本的系统调用库，并使用测试样例测试所写的系统调用。

实现的系统调用为：

```
execve2, getdents, sleep, getcwd
```

小组分工	实现函数	GitHub用户名	commit log 署名
李天豪	sleep, getcwd	yizimi-yuanxin	yizimi
谭承浩	execve2, getdents	TCH-202002001037	TCH

二、实验环境

Windows系统，UbuntuKylin14.10虚拟机，bochs虚拟机，linux0.11内核，VSCode，GitHub

GitHub地址：https://github.com/yizimi-yuanxin/Exper1_ExSysCall

最终版本的分支：dev

三、设计思路

1. execve2

考虑在Linux-0.11已有的系统调用execve的基础上，执行进程时将代码段和数据段都放入内存之中，避免缺页异常的出现。

2. getdents

通过打开文件表filp，找到fd的目录项inode节点号，从而读取几个linux_dirent结构到dirp所指向的缓冲区。

3. sleep

考虑调用alarm来实现。在alarm超时时间时，接收到一个SIGALRM的信号，来判断是否结束，且使用SIG_IGN忽略发出信号时所调用的函数。

4. getcwd

对于getcwd的功能，我们可以从当前目录开始，每次记录他本身的目录项 inode 节点号，然后在该目录中找到 ".." 来到达父目录，并在父目录中找到刚刚目录的相同 inode 节点号的目录项，并记录目录名称。依此递归到根节点。

四、实现方法及关键源码

1. execve2

在do_execve函数的基础上进行修改，在进程执行前通过函数immediate()将数据段和代码段都放入内存，从而避免进程执行时出现缺页异常。

```
int do_execve2(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    struct m_inode * inode;
    struct buffer_head * bh;
    struct exec ex;
    unsigned long page[MAX_ARG_PAGES];
    int i,argc,envc;
    int e_uid, e_gid;
    int retval;
    int sh_bang = 0;
    unsigned long phy_add_brk;
    unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4;

    if ((0xffff & eip[1]) != 0x000f)
```

```
        panic("execve called from supervisor mode");
    for (i=0 ; i<MAX_ARG_PAGES ; i++)   /* clear page-table */
        page[i]=0;
    if (!(inode=namei(filename)))       /* get executables inode */
        return -ENOENT;
    argc = count(argv);
    envc = count(envp);

restart_interp:
    if (!S_ISREG(inode->i_mode)) {  /* must be regular file */
        retval = -EACCES;
        goto exec_error2;
    }
    i = inode->i_mode;
    e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
    e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
    if (current->euid == inode->i_uid)
        i >>= 6;
    else if (current->egid == inode->i_gid)
        i >>= 3;
    if (!(i & 1) &&
        !((inode->i_mode & 0111) && suser())) {
        retval = -ENOEXEC;
        goto exec_error2;
    }
    if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
        retval = -EACCES;
        goto exec_error2;
    }
    ex = *((struct exec *) bh->b_data); /* read exec-header */
    if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
        char buf[1023], *cp, *interp, *i_name, *i_arg;
        unsigned long old_fs;

        strncpy(buf, bh->b_data+2, 1022);
        brelse(bh);
        iput(inode);
        buf[1022] = '\0';
        if (cp = strchr(buf, '\n')) {
```

```
            *cp = '\0';
            for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
        }
        if (!cp || *cp == '\0') {
            retval = -ENOEXEC; /* No interpreter name found */
            goto exec_error1;
        }
        interp = i_name = cp;
        i_arg = 0;
        for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
            if (*cp == '/')
                i_name = cp+1;
        }
        if (*cp) {
            *cp++ = '\0';
            i_arg = cp;
        }
        if (sh_bang++ == 0) {
            p = copy_strings(envc, envp, page, p, 0);
            p = copy_strings(--argc, argv+1, page, p, 0);
        }
        p = copy_strings(1, &filename, page, p, 1);
        argc++;
        if (i_arg) {
            p = copy_strings(1, &i_arg, page, p, 2);
            argc++;
        }
        p = copy_strings(1, &i_name, page, p, 2);
        argc++;
        if (!p) {
            retval = -ENOMEM;
            goto exec_error1;
        }
        old_fs = get_fs();
        set_fs(get_ds());
        if (!(inode=namei(interp))) {
            set_fs(old_fs);
            retval = -ENOENT;
            goto exec_error1;
        }
```

```
        set_fs(old_fs);
        goto restart_interp;
    }
    brelse(bh);
    if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
        ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
        inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
        retval = -ENOEXEC;
        goto exec_error2;
    }
    if (N_TXTOFF(ex) != BLOCK_SIZE) {
        printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
        retval = -ENOEXEC;
        goto exec_error2;
    }
    if (!sh_bang) {
        p = copy_strings(envc,envp,page,p,0);
        p = copy_strings(argc,argv,page,p,0);
        if (!p) {
            retval = -ENOMEM;
            goto exec_error2;
        }
    }
    if (current->executable)
        iput(current->executable);
    current->executable = inode;
    for (i=0 ; i<32 ; i++)
        current->sigaction[i].sa_handler = NULL;
    for (i=0 ; i<NR_OPEN ; i++)
        if ((current->close_on_exec>>i)&1)
            sys_close(i);
    current->close_on_exec = 0;
    free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
    free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    if (last_task_used_math == current)
        last_task_used_math = NULL;
    current->used_math = 0;
```

```
    p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
    p = (unsigned long) create_tables((char *)p,argc,envc);
    current->brk = ex.a_bss +
        (current->end_data = ex.a_data +
        (current->end_code = ex.a_text));
    current->start_stack = p & 0xfffff000;
    current->euid = e_uid;
    current->egid = e_gid;
    i = ex.a_text+ex.a_data;
    while (i&0xfff)
        put_fs_byte(0,(char *) (i++));
    eip[0] = ex.a_entry;
    eip[3] = p;

    phy_add_brk = 0;
    while (phy_add_brk < current->brk)
    {
        immediate(1, phy_add_brk + current->start_code);
        phy_add_brk += PAGE_SIZE;
    }

    return 0;
exec_error2:
    iput(inode);
exec_error1:
    for (i=0 ; i<MAX_ARG_PAGES ; i++)
        free_page(page[i]);
    return(retval);
}
/*----------------------------------------------这只是一条分隔线----------------------------------------------*/
void immediate(unsigned long error_code,unsigned long address)
{
    int nr[4];
    unsigned long tmp;
    unsigned long page;
    int block,i;

    address &= 0xfffff000;
    tmp = address - current->start_code;
    if (!current->executable || tmp >= current->end_data) {
```

```
        get_empty_page(address);
        return;
    }
    if (share_page(tmp))
        return;
    if (!(page = get_free_page()))
        oom();
    block = 1 + tmp/BLOCK_SIZE;
    for (i=0 ; i<4 ; block++,i++)
        nr[i] = bmap(current->executable,block);
    bread_page(page,current->executable->i_dev,nr);
    i = tmp + 4096 - current->end_data;
    tmp = page + 4096;
    while (i-- > 0) {
        tmp--;
        *(char *)tmp = 0;
    }
    if (put_page(page,address))
        return;
    free_page(page);
    oom();
}
```

2. getdents

通过打开文件表filp，找到fd的目录项inode节点号，通过bread函数读取内容，用while循环将读取到的几个linux_dirent结构放到dirp所指向的缓冲区。

```
int sys_getdents (unsigned int fd, struct linux_dirent *dirp, unsigned int count)
{
    struct m_inode *m_ino;
    struct buffer_head *buff_hd;
    struct dir_entry *dir;
    struct linux_dirent usr;
    int i, j, res;
    i = 0;
    res = 0;
    m_ino = current->filp[fd]->f_inode;
```

```
    buff_hd = bread(m_ino->i_dev, m_ino->i_zone[0]);
    dir = (struct dir_entry *)buff_hd->b_data;
    while (dir[i].inode > 0)
    {
        if (res + sizeof(struct linux_dirent) > count)
            break;
        usr.d_ino = dir[i].inode;
        usr.d_off = 0;
        usr.d_reclen = sizeof(struct linux_dirent);
        for (j = 0; j < 14; j++)
        {
            usr.d_name[j] = dir[i].name[j];
        }
        for(j = 0;j <sizeof(struct linux_dirent); j++){
            put_fs_byte(((char *)(&usr))[j],(char *)dirp + res);
            res++;
        }
        i++;
    }
    return res;
}
```

3. sleep

我们先调用 signal 函数，发送出 SIGALRM 信号，并传递 SIG_IGN 参数忽略信号处理函数。然后调用 sys_alarm，然后通过调用 sys_pause 实现进程调度，并在 alarm 超时时间时结束即可。

```
int sys_sleep(unsigned int seconds) {       // by yizimi
    sys_signal(SIGALRM, SIG_IGN, NULL);
    sys_alarm(seconds);
    sys_pause();
    return 0;
}
```

4. getcwd

current->pwd 记录了指向当前目录 inode 节点的指针，如果我们 current->pwd == current->root，即当前目录为根目录，直接返回 "/"。如果不是根目录，则一直跳到根目录。首先通过 find_father_dir(自定义) 来找到父目录的 inode 的节点号，记录下设备号(i_dev)，调用 iget 函数来访问父目录。再在父目录中调用 find_same_inode 来查找刚刚与目录 inode 节点号相同的目录项，然后将目录项中的 name 整合到目录中。

我们观察 find_entry 函数，find_entry 函数主要实现的是在该inode节点所指向的目录项中查找名字为 name 的目录项并可以返回目录项。我们可以以 find_entry 为原型，把 match 函数更改为相应的判断条件，实现 find_father_dir, find_same_inode，来满足上述操作。

最后，我们需要把目录项通过 put_fs_byte 将内核态中缓存内容拷贝入用户态的 buf 中。

```c
// a part of kernel/sys.c
#define BUF_MAX 4096

long sys_getcwd(char * buf, size_t size) {  // by yizimi
    // printk("getcwd");
    char buf_name[BUF_MAX];
    char *nowbuf;
    struct dir_entry * de;
    struct dir_entry * det;
    struct buffer_head * bh;
    nowbuf = (char *)malloc(BUF_MAX * sizeof(char));
    struct m_inode *now_inode = current->pwd;
    int idev, inid, block;
    int prev_inode_num = now_inode->i_num;
    if (now_inode == current->root)
        strcpy(nowbuf, "/");
    while (now_inode != current->root) {
        bh = find_father_dir(&now_inode, &det);
        idev = now_inode->i_dev;
        inid = det->inode;
        now_inode = iget(idev, inid);
        bh = find_same_inode(&now_inode, &de, prev_inode_num);
        prev_inode_num = det->inode;
        strcpy(buf_name, "/");
        strcat(buf_name, de->name);
```

```
        strcat(buf_name, nowbuf);
        strcpy(nowbuf, buf_name);
    }
    int chars = size;
    char *p1 = nowbuf, *p2 = buf;
    ++chars;
    while (chars-- > 0)
        put_fs_byte(*(p1++), p2++);
    return (long)buf;
}
```

```
// a part of namei.c
struct buffer_head * find_father_dir(struct m_inode ** dir, struct dir_entry ** res_dir) {
    int entries;
    int block,i;
    struct buffer_head * bh;
    struct dir_entry * de;
    struct super_block * sb;
    int namelen = 2;
    const char name[] = "..";
#ifdef NO_TRUNCATE
    if (namelen > NAME_LEN)
        return NULL;
#else
    if (namelen > NAME_LEN)
        namelen = NAME_LEN;
#endif
    entries = (*dir)->i_size / (sizeof (struct dir_entry));
    *res_dir = NULL;
    if (!namelen)
        return NULL;
    if (namelen==2 && get_fs_byte(name)=='.' && get_fs_byte(name+1)=='.') {
        if ((*dir) == current->root)
            namelen=1;
        else if ((*dir)->i_num == ROOT_INO) {
            sb=get_super((*dir)->i_dev);
            if (sb->s_imount) {
                iput(*dir);
                (*dir)=sb->s_imount;
```

```
                (*dir)->i_count++;
            }
        }
    }
    if (!(block = (*dir)->i_zone[0]))
        return NULL;
    if (!(bh = bread((*dir)->i_dev,block)))
        return NULL;
    i = 0;
    de = (struct dir_entry *) bh->b_data;
    while (i < entries) {
        if ((char *)de >= BLOCK_SIZE+bh->b_data) {
            brelse(bh);
            bh = NULL;
            if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
                !(bh = bread((*dir)->i_dev,block))) {
                i += DIR_ENTRIES_PER_BLOCK;
                continue;
            }
            de = (struct dir_entry *) bh->b_data;
        }
        // The only changed code there...
        if ((de->name[0] == '.' && de->name[1] == '.' && de->name[2] == '\0')) {
            *res_dir = de;
            return bh;
        }
        de++;
        i++;
    }
    brelse(bh);
    return NULL;
}
struct buffer_head * find_same_inode(struct m_inode ** dir, struct dir_entry ** res_dir, int yizimi) {
    int entries;
    int block,i;
    struct buffer_head * bh;
    struct dir_entry * de;
    struct super_block * sb;
```

```c
    int namelen = 7;
    const char name[] = "yizimi";
#ifdef NO_TRUNCATE
    if (namelen > NAME_LEN)
        return NULL;
#else
    if (namelen > NAME_LEN)
        namelen = NAME_LEN;
#endif
    entries = (*dir)->i_size / (sizeof (struct dir_entry));
    *res_dir = NULL;
    if (!namelen)
        return NULL;
    if (namelen==2 && name[0]=='.' && name[1]=='.') {
        if ((*dir) == current->root)
            namelen=1;
        else if ((*dir)->i_num == ROOT_INO) {
            sb=get_super((*dir)->i_dev);
            if (sb->s_imount) {
                iput(*dir);
                (*dir)=sb->s_imount;
                (*dir)->i_count++;
            }
        }
    }
    if (!(block = (*dir)->i_zone[0]))
        return NULL;
    if (!(bh = bread((*dir)->i_dev,block)))
        return NULL;
    i = 0;
    de = (struct dir_entry *) bh->b_data;
    while (i < entries) {
        if ((char *)de >= BLOCK_SIZE+bh->b_data) {
            brelse(bh);
            bh = NULL;
            if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
                !(bh = bread((*dir)->i_dev,block))) {
                i += DIR_ENTRIES_PER_BLOCK;
                continue;
            }
```

```c
            de = (struct dir_entry *) bh->b_data;
        }
        if (yizimi == de->inode) { // The only changed code there...
            *res_dir = de;
            return bh;
        }
        de++;
        i++;
    }
    brelse(bh);
    return NULL;
}
```

五、测试过程及画面

以下测试环境已配置好测试文件，文件夹位置为 /usr/root/1

测试环境：bochs + linux0.11

1. execve2

首先在相应位置打补丁（指令打补丁出现问题，故通过补丁文件自行添加补丁代码）

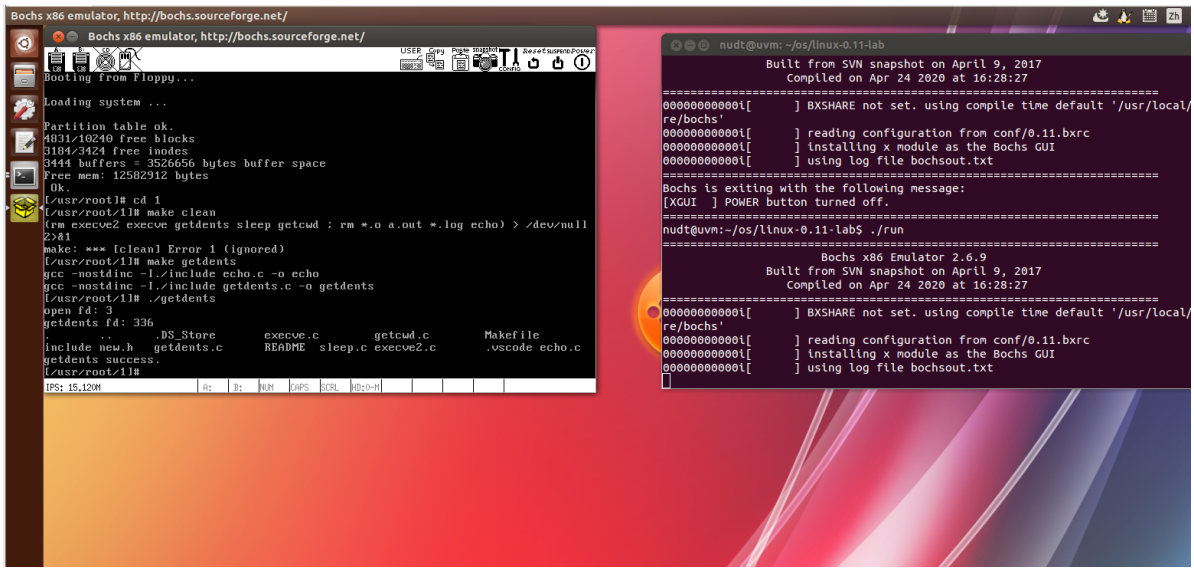
```
[/usr/root/1]# ./execve2
 --do_no_page: address=10000000, pid=40
 --do_no_page: address=10004000, pid=40
 --syscall: sid=87, pid=40
I am test_echo.
 --do_no_page: address=13ffdc40, pid=41
 --do_no_page: address=1005ac0c, pid=41
[/usr/root/1]#
```

2. getdents

指令（虚拟机开机后）

```
cd 1
make clean
make getdents
./getdents
```

效果：

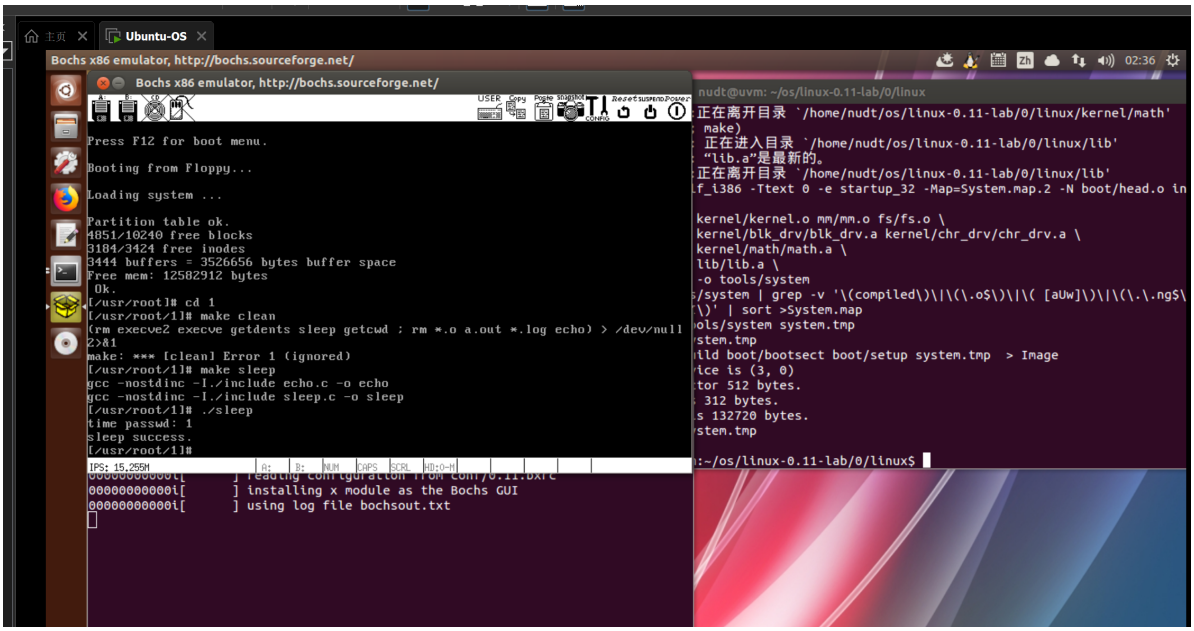


3. sleep

指令（虚拟机开机后）

```
cd 1
make clean
make sleep
./sleep
```

效果：



4. getcwd

指令（虚拟机开机后）

```
cd 1
make clean
make getcwd
./getcwd
```

效果：