操作系统原理实验报告

杨明 201605130117

一、简介

操作系统实验是对Linux 0.11的可视化。我们小组（一共两个人）选择的是内存管理（MM）部分的可视化工作。我们小组不只是可视化了Linux 0.11的内存管理，还把高版本Linux的部分高级内存管理策略移植到了Linux 0.11中，并对他们进行了可视化。我负责提供数据提取方案、提取数据、移植高级内存管理策略，并解决可能遇到的一些问题（下文有提到这方面的工作）。

二、编译、运行、调试Linux 0.11

由于Linux 0.11是上古时期的操作系统，无法在现有的机器上运行，而且，在物理机上运行也不方便调试，因此，使用虚拟机是个必然的选择。然而搭建这样的环境也是十分不容易的，需要交叉编译内核、系统自带工具（各种命令、gcc等等），还要制作根文件系统。还好由于Linux 0.11十分经典，研究的人非常多，已经有人制作好了编译、运行、调试（简单调试，无法满足实验要求）的环境：linux-0.11-lab。

该环境支持`QEMU`和`Bochs`模拟器，由于我需要添加新的指令，我选择了可扩展性、代码可读性都非常强的`Bochs`模拟器。下面我将介绍我添加新指令的目的。

三、更强大的数据提取方案

1. 其他解决方案的不足

1. 无法在汇编代码中提取数据
2. 只能提取内核数据，不能提取在Linux 0.11中运行的进程的数据
3. 无法提取特殊寄存器的数据，无法监控“物理事件"（操作系统有很多功能是由硬件实现的，例如地址映射，脏位更新等等，我的实现方案是能监控所有软件及硬件信息的）
4. 其他解决方案大多数需要在内核中添加、运行很多代码，可能会影响实验，例如影响CPU时间片（我的解决方案，数据提取是在虚拟机外进行的）。

2. 终极解决方案

我一直追求"完备"的解决方案。为了解决上述问题，我想到了一种方案：添加新的指令，我给你取名为`emit`指令，`emit`指令在执行的时候，虚拟机就会停下来，根据`eax`的值（ID），执行数据提取工作。通过查看`Intel`指令手册，找到一个空闲的机器码：`0F 3C`，于是我就令它为`emit`指令的机器码。下面是一个使用`emit`指令的实例。

```
mov eax, id
.byte 0x0F
.byte 0x3C
```

添加新的指令

修改`bochs`源码，可以添加新的指令，主要修改一下几个文件：

1. `cpu/decoder/ia_opcodes.def`

   这个文件定义了所有`opcode`与实现他们的C++函数的对应关系。

2. `cpu/cpu.h`

   这里定义了CPU类，CPU类里包含所有寄存器和指令的实现函数。想添加寄存器和添加新的指令都要修改这里

3. `cpu/decoder/fetchdecode32.cc`

   这个文件主要定义了`actual opcode numbers`与`opcode structure`的对应关系。

在虚拟机外提取数据能获取整个虚拟机所有的数据，包括软件和硬件。

Linux内运行的程序也能使用新的指令，因此能够捕获内部程序的数据，而不只是内核数据。这对于我们组来说非常重要，因为我们在演示`mmap`的过程中，需要提取我们自己写的程序（运行在Linux 0.11内）的数据。

3. 还需要完善的地方

由于时间问题，我没有实现对符号表的利用。利用上符号表信息，我们提取数据将变得更加方便。如果有人想进一步开发此框架，可以从添加对符号表的支持做起。

四、Linux 0.11 plus

1. `swap`

主要参考`Linux 0.12`

添加`swap`功能需要改动的地方比较少。

首先增加`init_swapping()`，进行`swap_bitmap`/`blk_size`的初始化。`init_swapping()`的调用是放在`sys_setup`这个系统调用中的（块设备初始化完毕后调用）。

然后就是分别在这几个地方考虑内存也是在物理内存内存中还是在`swap_dev`中:

1. `free_page_tables`:

```
int free_page_tables(unsigned long from,unsigned long size)
```

```
for (nr=0 ; nr<1024 ; nr++) {
  if (1 & *pg_table)
    free_page(0xfffff000 & *pg_table);
  *pg_table = 0;
  pg_table++;
}
```

```
for (nr=0 ; nr<1024 ; nr++) {
  if (*pg_table) {
    if (1 & *pg_table)
      free_page(0xfffff000 & *pg_table);
    else
      swap_free(*pg_table >> 1);
    *pg_table = 0;
  }
  pg_table++;
}
```

2. `copy_page_tables`:

```
int copy_page_tables(unsigned long from,unsigned long to,long size)
```

```
for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
  this_page = *from_page_table;
  if (!(1 & this_page))
    continue;
  this_page &= ~2;
  *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]++;
  }
}
```

```
for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
  this_page = *from_page_table;
  if (!this_page)
    continue;
  if (!(1 & this_page)) {
    if (!(new_page = get_free_page()))
      return -1;
    read_swap_page(this_page>>1, (char *) new_page);
    *to_page_table = this_page;
    *from_page_table = new_page | (PAGE_DIRTY | 7);
    continue;
  }
  this_page &= ~2;
  *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]++;
  }
}
```

3. `do_no_page`:

```
void do_no_page(unsigned long error_code,unsigned long address)
```

```
page = *(unsigned long *) ((address >> 20) & 0xffc);
if (page & 1) {
  page &= 0xfffff000;
  page += (address >> 10) & 0xffc;
  tmp = *(unsigned long *) page;
  if (tmp && !(1 & tmp)) {
    swap_in((unsigned long *) page);
    return;
  }
}
```

由于`swap`机制对于系统的其他部分来说是**透明的**，所以并不需要修改系统的其他部分。

2. `mmap`

`mmap`主要借鉴自`Linux 0.96a`

`mmap`功能非常强大。拥有`mmap`功能的内存管理系统，抽象程度更高，使得内存管理与物理内存完全解耦。内存管理的对象不再是只有物理内存，还能管理文件，而文件在`Linux`中抽象程度就非常的高，可以表示普通的文件、字符设备等等。因此，我认为，实现`mmap`功能，是`Linux`内存管理系统发展的一个里程碑。

`mmap`是一个新的系统调用，因此，需要先添加新的系统调用：

1）添加系统调用

1. `sys.h`:

   所有系统调用的实现函数都放到了一个叫`sys_call_table`的数组里，数组的下标代表系统调用的`id`，例如0号系统调用是：`sys_setup`:

```
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, sys_sigsuspend, sys_sigpending,
sys_sethostname, sys_setrlimit, sys_getrlimit, sys_getrusage,
sys_gettimeofday,  sys_settimeofday, sys_getgroups, sys_setgroups,
sys_select, sys_symlink, sys_lstat, sys_readlink, sys_uselib,
sys_swapon, sys_reboot, sys_readdir, sys_mmap, sys_munmap,
sys_truncate, sys_ftruncate, sys_fchmod, sys_fchown, sys_getpriority,
sys_setpriority, sys_profil, sys_statfs, sys_fstatfs, sys_ioperm,
sys_socketcall, sys_syslog };

/* So we don't have to do any more manual updating.... */
int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr);
```

`NR_syscalls`表示系统调用的数量。

2. `sys_call.s`：

```
60
61    nr_system_calls = 72
62
```

`nr_system_calls`表示系统调用的数量，代码通过这个过滤掉非法系统调用。由于我们新增了系统调用，就必须增加`nr_system_calls`。

3. `mmap.c`

   这部分是对系统调用`sys_mmap`的实现，主要借鉴自`Linux 0.96a`的代码。

2）修改内核数据结构

为了实现`mmap`功能，需要修改几个内核数据集结构：

1. `file`：

```
struct file {
    unsigned short f_mode;
    unsigned short f_flags;
    unsigned short f_count;
    unsigned short f_reada;
    struct inode * f_inode;
    struct file_operations * f_op;
    off_t f_pos;
};
```

2. `inode`:

```
struct m_inode {
    unsigned short i_mode;
    unsigned short i_uid;
    unsigned long i_size;
    unsigned long i_mtime;
    unsigned char i_gid;
    unsigned char i_nlinks;
    unsigned short i_zone[9];
/* these are in memory also */
    struct task_struct * i_wait;
    struct task_struct * i_wait2; /* for pipes */
    unsigned long i_atime;
    unsigned long i_ctime;
    unsigned short i_dev;
    unsigned short i_num;
    unsigned short i_count;
    unsigned char i_lock;
    unsigned char i_dirt;
    unsigned char i_pipe;
    unsigned char i_mount;
    unsigned char i_seek;
    unsigned char i_update;
};
```

```
struct inode {
    dev_t i_dev;
    ino_t i_ino;
    umode_t i_mode;
    nlink_t i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev;
    off_t i_size;
    time_t  i_atime;
    time_t  i_mtime;
    time_t  i_ctime;
    unsigned long i_data[16];
    struct inode_operations * i_op;
    struct super_block * i_sb;
    struct task_struct * i_wait;
    struct task_struct * i_wait2; /* for pipes */
    unsigned short i_count;
    unsigned char i_lock;
    unsigned char i_dirt;
    unsigned char i_pipe;
    unsigned char i_mount;
    unsigned char i_seek;
    unsigned char i_update;
};
```

3) 修正

上一步我们修改了内核数据结构，接下来就是工作量最大、最繁琐、最无聊的工作了，修正所有这些数据结构的代码，几乎文件系统、进程管理系统的所有函数都要修改，虽然改动都不是特别大，但是总的工作量巨大。

3. Kernel MM

这部分主要借鉴`Linux 0.99`和`Linux 2.2`的代码。

上学期的课程中，我们学习了针对内核的内存管理，主要方法是buddy、slab等内存分配方案。通过翻看所有老版本Linux内核代码，我发现`Linux 0.99`实现了`buddy`，`Linux 2.2`实现了`slab`。本想移植这两种算法，但是仔细考虑过后，发现不太现实。因为即使移植了，也不好演示效果，而且工作量十分巨大。最终，我们没有移植这两种算法。如果“后人”感兴趣的话，可以尝试在我工作的基础上移植一下。

五、遇到的问题及解决方案

1. `Bochs` + `GDB` 会遇到不断`page fault`中断

使用`Bochs` + `GDB`时，会不断地被`page fault`异常中断，这样让调试根本无法正常的进行下去。很多人遇到种情况后就抛弃`Bochs`，选择了`QEMU`。但是由于我被`Bochs`清晰的代码结构所吸引，所以我决定解决它。
`Bochs`与`gdb`相关的代码都集中在`gdbstub.cc`中。阅读代码后，发现只要屏蔽`GDBSTUB_STOP_NO_REASON`这个异常就可以了，修改`gdbstub.cc:487`：

```
/* gdbstub.cc:487 */
if (last_stop_reason == GDBSTUB_EXECUTION_BREAKPOINT ||
            last_stop_reason == GDBSTUB_TRACE)
{
  write_signal(&buf[1], SIGTRAP);
}
else if (last_stop_reason == GDBSTUB_STOP_NO_REASON)
{
  write_signal(&buf[1], SIGSEGV);
}
else
{
  write_signal(&buf[1], 0);
}
```

然后在`.kernel_gdbinit`中加入：

```
# .kernel_gdbinit
handle SIGSEGV nostop noprint ignore
```

这样，问题就解决了。

2. Linux 0.11 文件系统权限异常：`not owner`

这个问题是一个困扰着所有人的问题，例如，在系统内部，可以新建文件，但不能删除、移动等等。给大家带来了非常多的不便。我通过一番努力，解决了它。

以`rm`指令为突破口（Linux 0.11中，`rm`指令无法使用，删除任何文件都会提示:`ot owner`）。首先把`rm`这个指令从Linux 0.11中拷贝出来，使用`ida pro`逆向。`rm`指令是`coreutils`这个系统组件里面的，于是，下载`coreutils`源码，找到`rm`指令对应于`remove.c`里的`remove`函数。由于历史久远，源码中很多细节和Linux 0.11中的`rm`不一样，所以源码只要是帮助理解汇编代码，主要还是看`ida pro`逆向出来的汇编代码。

```
.text:000002C0 ; ---------------------------------------------------------------------------
.text:000002C0
.text:000002C0 loc_2C0:                                ; CODE XREF: sub_280+17↑j
.text:000002C0                                         ; sub_280+23↑j ...
.text:000002C0                 lea     eax, [ebp+var_20]
.text:000002C3                 push    eax             ; statbuf
.text:000002C4                 push    path            ; filename
.text:000002CA                 call    lstat
.text:000002CF                 add     esp, 8
.text:000002D2                 test    eax, eax
.text:000002D4                 jz      short loc_30C
.text:000002D6                 cmp     dword_6FD4, 2
.text:000002DD                 jnz     short loc_2EC
.text:000002DF                 cmp     dword_6F4C, 0
.text:000002E6                 jz      short loc_2EC
.text:000002E8                 xor     eax, eax
.text:000002EA                 jmp     short locret_339
.text:000002EC ; ---------------------------------------------------------------------------
```

如上图所示，找到一个系统调用`lstat`，Linux 0.11是没有这个系统调用的，导致`call lstat`恒返回0。Linux 0.12+才有这个系统调用。其实根本原因是，Linux 0.11不支持`link`。为了支持`link`，Linux 0.12实现了一套`l`开头的函数，例如：`lname/lseek/lstat`等等。但是我们没有必要全部移植，只需要这样做：

```
int sys_stat(char * filename, struct stat * statbuf)
{
  struct m_inode * inode;

  if (!(inode=namei(filename)))
    return -ENOENT;
  cp_stat(inode,statbuf);
  iput(inode);
  return 0;
}

int sys_lstat(char * filename, struct stat * statbuf)
{
  return sys_stat(filename, statbuf);
}
```

然后把`sys_lstat`加入系统调用（修改`sys.h` `sys_call.s`）就能完美解决无法`rm`的问题。然后，可以发现`chmod`等其他指令也能正常使用了。

六、 总结

通过这学期的操作系统实验，更深刻的认识了操作系统的实现过程。特别是在我移植`swap`、`mmap`、`slab`、`buddy`等高级内存管理功能时，实践了上学期学到的内存管理理论知识（原版Linux 0.11的内存管理太过简单，根本用不到上学期学习的知识）。通过实现我提出的新的数据提取方案，体验了一次定制指令集与寄存器的快感，同时，也学习到了很多虚拟机（模拟器）相关的知识。总之，收获满满。