Advanced Operating System and Virtualization

Interpreter Implementation Hiroaki Fukuda

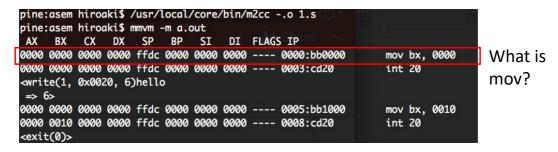
1

Contents

- Emulate operations
 - Refer to the specification provided by Intel
- Emulate system calls
 - Refer to the source code of minix2 operation system and posix system calls

า

Let's see again the execution log of 1.s

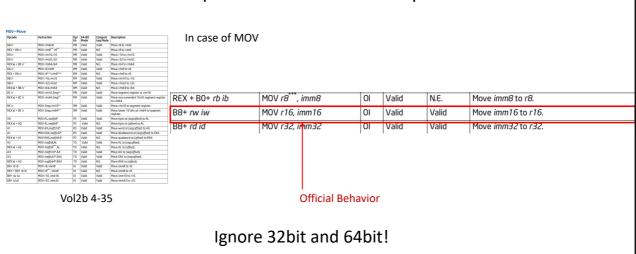


We know mov bx, 0000 -> bx = 0000

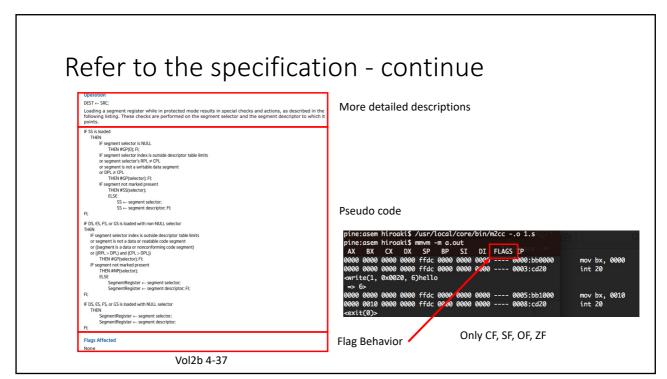
But how to know officially?

3

Refer to the specification – Chapter 3



/



5

Execute program

- 1. Extract text and data
- 2. Copy text and data to the memory
 - 1. Text and data are stored separately
- 3. Set registers to initial value (0)
- 4. Fetch/decode/execute/store

Let's see the execution log of 1.s

7

Let's see write system call

\$(minix): Depends on your environment (e.g., /usr/local/core/minix2)

```
$(minix}/usr/src/lib/posix/_write.c
#include <lib.h>
#define write _write
#include <unistd.h>
PUBLIC ssize_t write(fd, buffer, nbytes)
int fd;
_CONST void *buffer;
size_t nbytes;
{
   message m;

   m.m1_i1 = fd;
   m.m1_i2 = nbytes;
   m.m1_p1 = (char *) buffer;
   return(_syscall(FS, WRITE, &m));
}
```

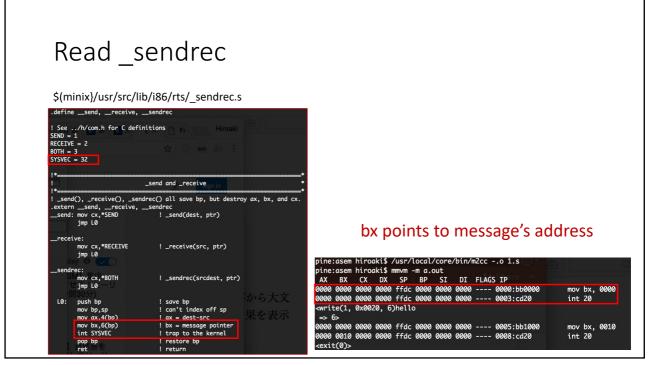
Return _systecall(1, 4, &m)

\$(minix)/usr/include/minix/callnr.h

- 21					Tilloan
	#define	NCALLS		78	/* nu
	10 数例				
	#define	EXIT		1	
	#define	FORK		2	
	#define	READ		3	
	#define	WRITE		4	X Seal
	#define	OPEN		5	
	#define	CLOSE		6	

return _syscall(1, 4, &m)

9



Inside Kernel \$(minix}/usr/src/fs/table.c \$(minix}/usr/src/fs/main.c PUBLIC void main() * This is the main program of the file system. The main loop consists of * three major activities: getting new work, processing the work, and sending * the reply. This loop never terminates as long as the file system runs. int error; fs_init(); /* This is the main loop that gets work, processes it, and sends replies. */ while (TRUE) { /* process to reply to */ /* result of the call (usually OK or error #) */ /* sets who and fs_call */ fp = &fproc[who]; /* pointer to proc table struct */ super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE); /* su? */ Send a reply to a user process. It may fail (if the process has just been killed by a signal), so don't check the return code. If the send fails, just ignore it. /* Call the internal function that does the work. */ if (fs_call < 0 || fs.call >= NCALLS) error = ENOSYS; reply_type = result; send(whom, &m1); error = (*call_vec[fs_call])(); /* (ony the results back to the user and send reply. */ if (error != SUSPEND) reply(who, error); if (rdahed_inode != NIL_INODE) read_ahead(); /* do block read ahead */ get_work() who = m.m_source; fs_call = m.m_type;

11

```
What is reply_type?
                                                                                   $(minix}/usr/include/minix/type.h
   $(minix}/usr/src/fs/param.h
  * The following names are synonyms for the var
                                                                                   typedef struct {
/* who sent the message */
/* what kind of message is it */
#define reply_l1
#define reply_i1
                                   m1.m2_l1
                                                                                     int m_type;
                                   m1.m1_i1
#define reply_i2
                                   m1.m1_i2
                                                                                             mess_1 m_m1;
mess_2 m_m2;
mess_3 m_m3;
#define reply_t1
#define reply_t2
                                   m1.m4_l1
m1.m4_l2
                                                                                             mess_4 m_m4;
#define reply_t3
                                   m1.m4_l3
                                                                                             mess 6 m m6:
#define reply_t4
                                   m1.m4_l4
#define reply_t5
                                   m1.m4_15
                                   $(minix}/usr/src/fs/glo.h
                                  /* The parameters of the call are kept here. */
                                  /* The parameters of the call are kept here. */
EXTERN message m; /* the input message itself */
EXTERN message m1; /* the output message used for reply */
EXTERN int who; /* caller's proc number */
EXTERN int fs_call; /* system call number */
EXTERN char user_path[PATH_MAX];/* storage for user path name */
```

To sum up for system call

- All information is encapsulated in *message* structure
- *Message.m_type* represents system call type(e.g., write, read)
- Message.m_type is also used to return value from system call to the caller (Note that, not always, it depends on system call kind, see source code in detail)
- Address of *Message* is stored in *BX* register

Let's continue interpreter implementation

1.s, 2.s, 1 ~6.c and nm.c