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

2

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

```
pine:asem hiroaki$ /usr/local/core/bin/m2cc -.o 1.s
pine:asem hiroaki$ mmvm -m a.out
 AX   BX   CX   DX   SP   BP   SI   DI  FLAGS IP
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0000:bb0000       mov bx, 0000
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0003:cd20         int 20
<write(1, 0x0020, 6)hello
 => 6>
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0005:bb1000       mov bx, 0010
0000 0010 0000 0000 ffdc 0000 0000 0000 ---- 0008:cd20         int 20
<exit(0)>
```

What is mov?

We know mov bx, 0000 -> bx = 0000

But how to know officially?

3

Refer to the specification – Chapter 3

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16*.
REX.W + 8C /r	MOV *r/m64,Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r/m64*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move lower 16 bits of *r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8**,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb ib*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8*.
REX + B0+ *rb ib*	MOV *r8***, imm8*	OI	Valid	N.E.	Move *imm8* to *r8*.
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16*.
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32*.

In case of MOV

REX + B0+ *rb ib*	MOV *r8***, imm8*	OI	Valid	N.E.	Move *imm8* to *r8*.
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16*.
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32*.

Vol2b 4-35

Official Behavior

Ignore 32bit and 64bit!

4

Refer to the specification - continue

Operation

DEST ← SRC;

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

```
IF SS is loaded
    THEN
        IF segment selector is NULL
            THEN #GP(0); FI;
        IF segment selector index is outside descriptor table limits
        or segment selector's RPL ≠ CPL
        or segment is not a writable data segment
        or DPL ≠ CPL
            THEN #GP(selector); FI;
        IF segment not marked present
            THEN #SS(selector);
            ELSE
                SS ← segment selector;
                SS ← segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector
THEN
    IF segment selector index is outside descriptor table limits
    or segment is not a data or readable code segment
    or ((segment is a data or nonconforming code segment)
    or ((RPL > DPL) and (CPL > DPL))
        THEN #GP(selector); FI;
    IF segment not marked present
        THEN #NP(selector);
        ELSE
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with NULL selector
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;
```

Flags Affected

None

Vol2b 4-37

More detailed descriptions

Pseudo code

```
pine:asem hiroaki$ /usr/local/core/bin/m2cc -.o 1.s
pine:asem hiroaki$ mmvm -m a.out
 AX   BX   CX   DX   SP   BP   SI   DI  FLAGS IP
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0000:bb0000       mov bx, 0000
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0003:cd20         int 20
<write(1, 0x0020, 6)hello
 => 6>
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0005:bb1000       mov bx, 0010
0000 0010 0000 0000 ffdc 0000 0000 0000 ---- 0008:cd20         int 20
<exit(0)>
```

Flag Behavior

Only CF, SF, OF, ZF

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

6

Let’s see the execution log of 1.s

```
pine:asem hiroaki$ /usr/local/core/bin/m2cc -.o 1.s
pine:asem hiroaki$ mmvm -m a.out
 AX   BX   CX   DX   SP   BP   SI   DI  FLAGS IP
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0000:bb0000       mov bx, 0000
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0003:cd20         int 20
<write(1, 0x0020, 6)hello
 => 6>
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0005:bb1000       mov bx, 0010
0000 0010 0000 0000 ffdc 0000 0000 0000 ---- 0008:cd20         int 20
<exit(0)>
```

System call

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/src/lib/other/syscall.c

```
#include <lib.h>

PUBLIC int _syscall(who, syscallnr, msgptr)
int who;
int syscallnr;
register message *msgptr;
{
  int status;

  msgptr->m_type = syscallnr;
  status = _sendrec(who, msgptr);
  if (status != 0) {
        /* 'sendrec' itself failed. */
        /* XXX - strerror doesn't know all the codes */
        msgptr->m_type = status;
  }
  if (msgptr->m_type < 0) {
        errno = -msgptr->m_type;
        return(-1);
  }
  return(msgptr->m_type);
}
```

8

Find FS and WRITE

$(minix}/usr/include/lib.h

```
#include <minix/const.h>
#include <minix/type.h>
#include <minix/callnr.h>

#define MM                      0
#define FS                      1
```

$(minix}/usr/include/minix/callnr.h

```
#define NCALLS          78      /* nu
#define EXIT            1
#define FORK            2
#define READ            3
#define WRITE           4
#define OPEN            5
#define CLOSE           6
```

return _syscall(1, 4, &m)

9

Read _sendrec

$(minix}/usr/src/lib/i86/rts/_sendrec.s

```
.define __send, __receive, __sendrec

! See ../h/com.h for C definitions
SEND = 1
RECEIVE = 2
BOTH = 3
SYSVEC = 32

!*===========================================================================*
!                          _send and _receive                               *
!*===========================================================================*
! _send(), _receive(), _sendrec() all save bp, but destroy ax, bx, and cx.
.extern __send, __receive, __sendrec
__send: mov cx,*SEND            ! _send(dest, ptr)
        jmp L0

__receive:
        mov cx,*RECEIVE         ! _receive(src, ptr)
        jmp L0

__sendrec:
        mov cx,*BOTH            ! _sendrec(srcdest, ptr)
        jmp L0

  L0:   push bp                 ! save bp
        mov bp,sp               ! can't index off sp
        mov ax,4(bp)            ! ax = dest-src
        mov bx,6(bp)            ! bx = message pointer
        int SYSVEC              ! trap to the kernel
        pop bp                  ! restore bp
        ret                     ! return
```

bx points to message's address

```
pine:asem hiroaki$ /usr/local/core/bin/m2cc -.o 1.s
pine:asem hiroaki$ mmvm -m a.out
 AX   BX   CX   DX   SP   BP   SI   DI  FLAGS IP
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0000:bb0000       mov bx, 0000
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0003:cd20         int 20
<write(1, 0x0020, 6)hello
 => 6>
0000 0000 0000 0000 ffdc 0000 0000 0000 ---- 0005:bb1000       mov bx, 0010
0000 0010 0000 0000 ffdc 0000 0000 0000 ---- 0008:cd20         int 20
<exit(0)>
```

10

Inside Kernel

$(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) {
       get_work();               /* sets who and fs_call */

       fp = &fproc[who];         /* pointer to proc table struct */
       super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */

       /* Call the internal function that does the work. */
       if (fs_call < 0 || fs_call >= NCALLS)
               error = ENOSYS;
       else
               error = (*call_vec[fs_call])();

       /* Copy 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 */
  }
}
```

$(minix}/usr/src/fs/table.c

```
PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
        no_sys,         /*  0 = unused  */
        do_exit,        /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        do_read,        /*  3 = read    */
        do_write,       /*  4 = write   */
        do_open,        /*  5 = open    */
```

```
PUBLIC void reply(whom, result)
int whom;                       /* process to reply to */
int result;                     /* result of the call (usually OK or error #) */
{
/* 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.
 */

 reply_type = result;
 send(whom, &m1);
}
```

get_work()

```
    who = m.m_source;
    fs_call = m.m_type;
```

11

What is reply_type?

$(minix}/usr/src/fs/param.h

```
/* The following names are synonyms for the vari
#define reply_type      m1.m_type
#define reply_l1        m1.m2_l1
#define reply_i1        m1.m1_i1
#define reply_i2        m1.m1_i2
#define reply_t1        m1.m4_l1
#define reply_t2        m1.m4_l2
#define reply_t3        m1.m4_l3
#define reply_t4        m1.m4_l4
#define reply_t5        m1.m4_l5
```

$(minix}/usr/include/minix/type.h

```
typedef struct {
  int m_source;                 /* who sent the message */
  int m_type;                   /* what kind of message is it */
  union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_6 m_m6;
  } m_u;
} message;
```

$(minix}/usr/src/fs/glo.h

```
/* 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 */
```

12

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

13