Warm-up and Booting

Youjip Won
Outline

- assembly instructions
- registers
- booting
xv6 memory layout
x86 Physical memory map

- The physical address space mostly looks like ordinary RAM.
- Some low-memory addresses actually refer to other things; VGA memory appear on the screen.
- Reset or power-on jumps to ROM at 0x000ffff0 (so must be ROM at top of BIOS)
Register: Instruction Pointer

- EIP: register that points to the instruction to be executed next.
- EIP is incremented after each instruction execution.
- Instructions are different length.
- EIP is modified by CALL, RET, JMP, and conditional JMP
8, 16, and 32 bit versions
- By convention some registers for special purposes.
- Example: ADD EAX, 10
- Other instructions: SUB, AND, etc
Instruction Modes

Register Mode

```assembly
movl %eax, %edx  // edx = eax;
```

Immediate Mode

```assembly
movl $0x123, %edx  // edx = 0x123;
```

Direct Mode

```assembly
movl 0x123, %edx  // edx = *(int32_t*)0x123;
```

Indirect Mode

```assembly
movl (%ebx), %edx // edx = *(int32_t*)ebx;
movl 4(%ebx), %edx // edx = *(int32_t*)(ebx+4);
```
Register: EFLAGS

- 32bit register
- Represent the state of the instruction.
- each bit has its own meaning.
- Test instructions

  TEST EAX, 0  // Perform EAX && 0x0000 and set ZF to 1 if it is zero.

- Conditional JMP instruction

  JNZ address  // jump to address if ZF of EFLAGS is non-zero.
Register: EFLAGS

- ID Flag (ID)
- Virtual Interrupt Pending (VIP)
- Virtual Interrupt Flag (VIF)
- Alignment Check (AC)
- Virtual-8086 Mode (VM)
- Resume Flag (RF)
- Nested Task (NT)
- I/O Privilege Level (IOPL)
- Overflow Flag (OF)
- Direction Flag (DF)
- Interrupt Enable Flag (IF)
- Trap Flag (TF)
- Sign Flag (SF)
- Zero Flag (ZF)
- Auxiliary Carry Flag (AF)
- Parity Flag (PF)
- Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
## Stack operations

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %eax</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl %eax, (%esp)</td>
</tr>
<tr>
<td>pop %eax</td>
<td>movl (%esp), %eax</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td>call 0x12345</td>
<td>pushl %eip(*)</td>
</tr>
<tr>
<td></td>
<td>movl $0x12345, %eip(*)</td>
</tr>
<tr>
<td>ret</td>
<td>pop %eip(*)</td>
</tr>
</tbody>
</table>
x86 instruction set

instructions classes:
- Data movement: MOV, PUSH, POP, ...
- Arithmetic: TEST, SHL, ADD, ...
- I/O: IN, OUT, ...
- Control: JMP, JZ, JNZ, CALL, RET
- String: REP, MOVSB, ...
- System: IRET, INT, ...

Intel architecture manual Volume 2
- Intel syntax: op dst, src
- AT&T (gcc/gas) syntax: op src, dst
xv6 Booting (Linux)

- what you need: qemu, gdb, xv6

- download qemu
  
  ```bash
  $ sudo apt-get install git wget qemu
  ```

- download xv6 source code
  
  ```bash
  $ git clone https://github.com/mit-pdos/xv6-public.git
  ```

- install gdb install (in Linux)
  
  ```bash
  $ sudo apt-get install gdb
  ```

- compile xv6 and run
  
  ```bash
  $ cd xv6-public && make qemu-nox
  ```
xv6 – Booting (Mac OS)

- Download package manager from https://www.macports.org

- Setting environment variable your mac to use package manager
  
  ```
  $ export PATH=/opt/local/bin:/opt/local/sbin:$PATH
  ```

- Install git
  
  ```
  $ sudo port install git wget
  ```

- Install qemu
  
  ```
  $ sudo port install qemu
  ```
xv6 – booting (Mac OS)

- install gcc
  
  $ sudo port install i386-elf-gcc

- modify Makefile
  
  $ cd xv6-public && vi Makefile

- locate the following line
  
  TOOLPREFIX = i386-jos-elf

- locate the following modification
  
  change i386-jos-elf to i386-elf-

- compile xv6 and execute
  
  $ make qemu-nox

```
# Cross-compiling (e.g., on Mac OS X)
TOOLPREFIX = i386-jos-elf
```

before the change

```
# Cross-compiling (e.g., on Mac OS X)
TOOLPREFIX = i386-elf-
```

after the change
xv6 - booting

```bash
gdbinit: .gdbinit.tmp
   sed "s/localhost:1234/localhost:$(GDBPORT)/" < $^ > $@

qemu-gdb: fs.img xv6.img .gdbinit
   @echo "*** Now run 'gdb'." 1>&2
   $(QEMU) -serial mon:stdio $(QEMUOPTS) -S $(QEMUGDB)

qemu-nox-gdb: fs.img xv6.img .gdbinit
   @echo "*** Now run 'gdb'." 1>&2
   $(QEMU) -nographic $(QEMUOPTS) -S $(QEMUGDB)
```

```bash
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512

qemu-memfs: xv6memfs.img
   $(QEMU) -drive file=xv6memfs.img,index=0,media=disk,format=raw -smp $(CPU) -m 256

qemu-nox: fs.img xv6.img
   $(QEMU) -nographic $(QEMUOPTS)
```

Youjip Won
executing a program in xv6

- xv6 is not full fledged OS.
- We cannot compile a program within xv6.
- We need to compile xv6 and the program that we like to run in xv6 together.
  - Create a file in xv6 folder. ex: test.c
    ```c
    int
    main(int argc, char *argv[]) {
        printf(1,"Hello world!
");
        exit();
    }
    ```
  - Add `_test` to `UPROGS` as in the right box.
- Compile xv6
  ```
  $ make qemu-nox
  ```
- execute program
example

- we need to use xv6 provided header file.
  - We cannot use `stdio.h`.
  - Use the following header files.
    - `types.h`: header file for variable types
    - `user.h`: header file for system calls
    - `fcntl.h`: header file for file IO
#include "type.h"
#include "user.h"

int main(int argc, char *argv[]) {
    int pid = fork();

    if(pid > 0) {
        printf(1,"parent: child=%d\n",pid);
        pid = wait();
        printf(1,"child %d is done\n",pid);
    } else if(pid == 0) {
        printf(1,"child exiting\n");
        exit();
    } else {
        printf(1,"fork error\n");
    }

    exit();
}
#include "types.h"
#include "user.h"

int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello";
    argv[2] = 0;

    exec("./echo", argv);

    printf(1,"exec error\n");

    exit();
}

#include "types.h"
#include "user.h"

int main(int argc, char *argv[]) {
    char buf[512];
    int n;
    for(;;){
        n = read(0,buf,sizeof buf);
        printf(1,"%d",n);
        if(n == 0)
            exit();
        if(n < 0) {
            printf(2,"read error!\n");
            exit();
        }
        if(write(1,buf,n) != n) {
            printf(2,"write error!\n");
            exit();
        }
    }
    exit();
}
```c
#include "types.h"
#include "user.h"

int main() {
    char *argv[2];
    argv[0] = "cat";
    argv[1] = 0;

    if(fork() == 0) {
        close(0);
        open("input.txt", 0); // 0 = O_RDONLY
        exec("cat", argv);
    } else {
        wait();
    }

    exit();
}
```
```c
#include "types.h"
#include "user.h"

int main() {
    if(fork() == 0) {
        write(1, "hello", 6);
        exit();
    } else {
        wait();
        write(1, "world\n", 6);
    }
    exit();
}
```
#include "types.h"
#include "user.h"

int
main() {
    int fd = dup(1);
    write(1, "hello ", 6);
    write(fd, "world\n", 6);
    exit();
}
#include "types.h"
#include "user.h"

int main () {
    int p[2];
    char *argv[2];
    argv[0] = "wc";
    argv[1] = 0;

    pipe(p);
    if(fork() == 0) {
        close(0);
        dup(p[0]);
        close(p[0]);
        close(p[1]);
        exec("./wc",argv);
    } else {
        close(p[0]);
        write(p[1], "hello world\n",12);
        close(p[1]);
        wait();
    }
    exit();
}
```c
#include "types.h"
#include "fcntl.h"
#include "user.h"

int main (int argc, char *argv[]) {

    mkdir("./dir");
    int fd = open("./dir/file", O_CREATE|O_WRONLY);
    close(fd);

    mknod("/console",1,1);

    exit();
}
```
GDB

- GDB?
  - debugger
  - trace the execution of a program and examine the state of the execution

- compile a program for gdb debugging
  $ gcc -g gdb_test.c -o gdb_test

- execute gdb
  $ gdb [file_name]

- argument setting
  $ gdb [file_name]
    (gdb) set args argument
1. #include <stdio.h>
2.
3. void print() {
4.         printf("Hello World!\n");
5. }
6.
7. int main () {
8.
9.         int i;
10.        for (i=0; i < 10; i++) print();
11.        return 0;
12. }
GDB (Cont.)

```
Reading symbols from gdb_test...done.
(gdb) break 7
Breakpoint 1 at 0x40053f: file gdb_test.c, line 7.
(gdb) break print
Breakpoint 2 at 0x40052a: file gdb_test.c, line 4.
(gdb) run
Starting program: /home/sundoo/gdb_test

Breakpoint 1, main () at gdb_test.c:10
warning: Source file is more recent than executable.
10  for (i=0; i < 10; i++) print();
(gdb) continue
Continuing.

Breakpoint 2, print () at gdb_test.c:4
4    printf("Hello World!\n");
(gdb) next
Hello World!
5    }
(gdb) continue
Continuing.

Breakpoint 2, print () at gdb_test.c:4
4    printf("Hello World!\n");
(gdb) bt
#0  print () at gdb_test.c:4
#1  0x0000000000400552 in main () at gdb_test.c:10
(gdb) q
A debugging session is active.
Inferior 1 [process 20018] will be killed.
Quit anyway? (y or n) y
```
gdb - basic commands

- **continue(c):** continue executing a program
- **step(s):** execute a line. When a function is called, enter into the function.
- **next(n):** execute a line. When a function is called, execute a function and goes to the next line.
- **finish:** execute a program till the end and finish.

- **(gdb) list**
  - show the codes that are being executed.
- **(gdb) list 90 : print line 90**
- **(gdb) list badfunc**
  - print the source code of ‘badfunc’
- **(gdb) set listsize n**
  - set the number of lines that are displayed in the screen each time.
gdb - print

- examine the state of a variable.
  (gdb) whatis [var] : get the type of the variable.
  (gdb) print [var] : print the value of the variable.

- print
  (gdb) print a->member
  (gdb) print add(1,2)
  (gdb) print /x value / //x,u,o,c specifies the format of output.
gdb - setup break point

- suspend the execution at the designated position
  
  (gdb) break 31
  (gdb) break func
  (gdb) break hello.c:main
  (gdb) break util.c:300
  (gdb) info break
  (gdb) delete 1 : (delete the break point)

- print the function call trace
  
  (gdb) backtrace
  (gdb) backtrace n
gdb – stack frame

- call stack consists of the “stack frames”
- stack frame is a set of variables associated with a function call.
- stack frame contains parameters, local variables of the function, location of the return values
- (gdb) frame [args]
  - print the name of the function that is associated with the stack frame of [args]
- (gdb) select-frame [args]
  - select the frame with [args]
- (gdb) info frame
  - print the contents of ‘frame’

```
(gdb)
Stack level 0, frame at 0x7fffffffdf60:
  rip = 0x400587 in add (test.c:S); saved rip = 0x4005c0
called by frame at 0x7fffffffdf80
  source language c.
Arglist at 0x7fffffffdf50, args: a=1, b=2
Locals at 0x7fffffffdf50, Previous frame's sp is 0x7fffffffdf60
Saved registers:
  rbp at 0x7fffffffdf50, rip at 0x7fffffffdf58
```

we will explain the frame in detail in “calling convention section”.
Read the man page!!

- Manual for Unix command and system calls
- Executing man page
  - `$ man [command]`
- Numbers in the top left corner of the man page
  - ex) `man ls` —> we will see ‘ls(1)’
  - 1: Shell command
  - 2: System call
  - 3: C standard library
  - 4: Special file and the driver