Ch. 3 Computer Memory

Memory mapping

- view memory as array of bytes, each at increasing address
- hardware mapping registers make processes appear to have access to the same logical addresses while they are actually accessing separate physical addresses
- hardware mapping registers on an x86-64 CPU can map pages of 2 different size
    - 4096 bytes
    - 2 MB
- Linux uses 2MB pages for the kernel and 4KB pages for other uses
- support for 1GB pages in more recent CPUs
- 4096 = 2^12, so an address is translated by combining the page number (leftmost 20 bits) with offset within page (rightmost 12 bits)
- generally can translate without slowdown
- provides OS protection from processes and protection between processes
- OS includes instructions for managing the hardware mapping registers
- segmentation faults occur when an address is accessed past the end of the areas of memory that have been mapped into a process by the OS/hardware
- from stack overflow:
    - segmentation fault:
        - process attempted to access an invalid or illegal memory address
    - page fault:
        - a pointer attempts to access a page of address space that's currently not mapped onto physical memory, so that the MMU needs to retrieve it from disk before it can be used
    - a segmentation fault is an illegal condition and the program will generally be aborted
    - a page fault is normal and the program won't even know about it
    - "segmentation" isn't at all related to the old "segmented memory model" used by early x86 processors

    - "segmentation" is an earlier use which just refers to a portion or segment of memory

Process memory model in Linux

- process memory divided into 4 logical regions
    - text
    - data
    - heap
    - stack
- stack is mapped to highest address of a process (0x7fffffffffff on x86-64 Linux)
- max number of bits allowed in logical addresses being 48 bits (bit 48 not used)
- text at 0, then data and bss (bss not used with C++ executables, initialized to 0)
- heap is dynamic and can grow to very large sizes, limited by physical memory and swap space
- stack size is limited by kernel (typically 16MB in Linux)
- gives range of stack from 0x7fffffffffff to 0x7fffff000000 (OS grows stack on a page fault)
- shared memory layout described here does not give fully accurate picture
    - shared libraries can be mapped into heap range
    - for security, some versions of Linux use randomized stack, data, and heap start addresses
- to examine memory of a process on Linux, read /proc//maps

Memory example

- gives example asm to examine memory

Examining memory with gdb

- p (print)
    - `p expression`
    - `p/FMT expression`
    - formats
        - d - decimal (default)
        - x - hexadecimal
        - t - binary
        - u - unsigned
        - f - floating point
        - i - instruction
        - c - character
        - s - string
        - a - address

- x (examine)
  - `x/NFS address`
  - N - number of items to print
  - F - format (see above)
  - S - size of each memory location
    - b - byte (1)
    - h - halfword (2)
    - w - word (4)
    - g - giant (8)

Ch. 4 Memory mapping 64 bit mode

The memory mapping register

- CR3 - control register 3
  - pointer to the top level of a hierachical collection of tables in memory which define the translation from virtual addresses to physical addresses
- PML4 - page map level 4
  - initial hierarchy of translation tables prepared by kernel

Page Map Level 4

- virtual address can be broken into fields

```
63–48 – unused
47–39 – PML4 index
38–30 – page directory pointer index
29–21 – page directory index
20–12 – page table index
11–0 – page offset
```

- more description of split by page size as above

Page Directory Pointer Table

- 2nd level
- collection of pointer tables that is array of 512 pointers
- points to an address of a page directory table

Page Directory Table

- 3rd level
- array of 512 pointers that points to a page table

Page Table

- 4th level
- array of 512 pointers that points to a page, which, when combined with the page offset, yields final physical address

Large pages

- usually 4096 bytes
- only uses 3 of the 4 levels of offset which leaves 2^21 bytes of addresses within a page

CPU Support for Fast Lookups

- a page lookup puts the result of a translation into the translation lookahead buffer (TLB)
- similar to hash table -> successfully produces physical address when presented with a virtual address or fails
  - lookup from TLB typically takes 1/2 cycle
  - full translation typically takes 10 - 100 cycles
  - miss rates are generally 0.01% to 1%
- each architecture has a set size for TLB level 1 and level 2
- Linux provides support for larger tables using the HUGETLB option

Ch. 5 Registers

- overview of registers here

Moving a constant into a register

```
mov rax, 100 ; Intel
movq $100, %rax ; AT&T
mov eax, 100 ; Intel
movl $100, %eax ; AT&T
```

Moving values from memory into registers

```
; move address of var a into rax
mov rax, a ; Intel
movq $a, %rax ; AT&T -> do not use with globals, will return absolute addressing/P
leaq a(%rip), %rax
; move contents of memory location var a into rax
mov rax, [a] ; Intel
movl a(%rip), %rax ; AT&T
```

Moving values from a register into memory

```
mov [sum], rax ; Intel
movl $20, sum(%rip) ; AT&T
```

Moving data from one register to another

```
mov rbx, rax ; Intel
movq %rax, %rbx ; AT&T
```

Ch. 6 A little bit of math

Negation

- neg - two's complement of operand (general purpose register or memory reference)
- requires a size specifier

```
neg rax ; negate the value in rax
neg dword [x] ; negate 4 byte int at x
neg byte [x]  ; negate byte at x
```

Addition

- `add` adds source and destination and stores results in destination
- source
  - immediate value (constant) of 32 bits
  - memory reference
  - register
- destination
  - memory reference
  - register
- only one operand can be a memory reference
- sets or clears flags in rflags
  - OF (overflow) set if addition overflows
  - SF (sign flag) set to sign bit of result
  - ZF (zero flag) set if result == 0
  - additional flags are set for decimal arithmetic
- no special add or subtract for signed vs. unsigned
- `inc` instruction can be used to add 1 to a memory reference or a register

```
add [a], rax   ; add rax to a
add rax, [a]   ; add contents of a to rax
add rax, 10    ; add 10 to rax
```

Subtraction

- `sub` instruction sets or clears
  - overflow flag (OF)
  - sign flag (SF)
  - zero flag (ZF)
  - other flags are set related to performing binary-coded-decimal arithmetic
- no special subtract for signed numbers versus unsigned numbers
- decrement instruction `dec` can be used to decrement either a register or a memory reference

```
sub   [a], rax    ; subtract 10 from a
sub   [b], rax    ; subtract 10 from b
mov   rax, [b]    ; move b into rax
sub   rax, [a]    ; set rax to b-a
```

Multiplication

- `mul` - unsigned integer multiplication
- `imul` - signed integer multiplication
- 3 forms with 1-3 operands

One operand imul

- multiplies value in rax by source operand

Size specifiers

- intel size specifiers
  - byte - 1
  - word - 2
  - dword - 4
  - qword - 8
- AT&T size specifier suffixes
  - b - byte (8–bit)
  - w - word (16–bit)
  - l - long (32–bit) (default)
  - q - quadword (64–bit)

- (suffixes for x87 floating-point instructions)
  - no suffix - instruction operands are registers only
  - l (long) - instruction operands are 64–bit
  - s (short) - instruction operands are 32–bit