Computer Memory

Ray Seyfarth

June 29, 2012

Outline

Memory mapping

- Computer memory is an array of bytes from 0 to $n - 1$ where n is the memory size
- Programs perceive "logical" addresses which are mapped to physical addresses
- 2 people can run a program starting at logical address `0x4004c8` while using different physical memory
- CPU translates logical addresses to physical during instruction execution
- The CPU translation can be just as fast as if the software used physical addresses
- The x86-64 CPUs can map pages of sizes 4096 bytes and 2 megabytes
- Linux uses 2 MB pages for the kernel and 4 KB pages for programs
- Some recent CPUs support 1 GB pages

Translating an address

- Suppose an instruction references address `0x43215628`
- With 4 KB pages, the rightmost 12 bits are an offset into a page
- With `0x43215628` the page offset is `0x628`
- The page number is `0x43215`
- Let's assume that the computer is set up to translate page `0x43215` to physical addresses `0x7893000` - `0x7893fff`
- Then address `0x43215628` is mapped to `0x7893628`

Benefits of memory mapping

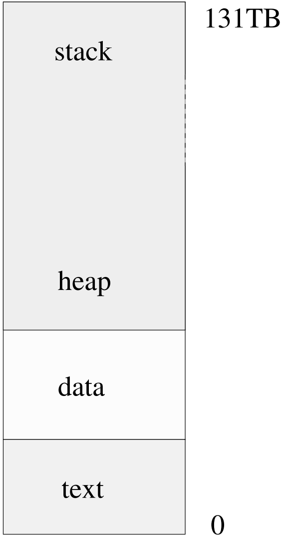
- User processes are protected from each other
  - Your `vi` process can't read my `vi`'s data
  - Your process can't write my data
- The operating system is protected from malicious or errant code
- It is easy for the operating system to give processes contiguous chunks of "logical" memory

Why study memory mapping?

- If you write programs, the mapping is automatic
- We will not discuss instructions for changing mapping tables
- So what difference does it make?
- It helps explain page faults
  - Suppose you allocate an array of 256 bytes at logical address `0x45678200`
  - Then all addresses from `0x45678000` to `0x45678fff` are valid
  - You can go well past the end of the array before you can get a segmentation violation
- Knowledge is power!

Process memory model in Linux

- A Linux process has 4 logical segments
  - text: machine instructions
  - data: static data initialized when the program starts
  - heap: data allocated by `malloc` or `new`
  - stack: run-time stack
    - return addresses
    - some function parameters
    - local variables for functions
    - space for temporaries
- In reality it is more complex
- 131TB is 47 bits of all 1's
- CPU could use 48 bit logical addresses
- Canonical addresses propagate bit 47 through 48-63 so Linux chose to use 47 bits to avoid the top stack address from appearing huge



Memory segments

- The text segment is named `.text` in `yasm`
  - `_start` and `main` are not actually at 0
  - The text segment does not need to grow, so the data segment can be placed immediately after it
- The data segment is in 2 parts
  - `.data` which contains initialized data
  - `.bss` which contains reserved data (initialized to 0)
  - "bss" stands for "Block Started by Symbol"
- The heap and the stack both need to grow
  - The heap grows up
  - The stack grows down
  - They meet in the middle and explode
- Use of heap and stack space in assembly does not involve using a named segment

Stack segment limits

- The stack segment is limited by the Linux kernel
- The typical size is 16 MB for 64 bit Linux
- This can be inspected using "`ulimit -a`"
- 16 MB seems fairly small, but it is fine until you start using large arrays as local variables in functions
- The stack address range is `0x7ffff000000` to `0x7fffffffffff`
- A fault to addresses in this range are recognized by the kernel to allow the stack to grow as needed

A few adjustments to the memory model

- It appears that the text segment starts at `0x400000` not 0
- Shared libraries map code and data into lots of addresses
- You can map shared memory regions into your programs
- Use "`cat /proc/$$/maps`" to see your shell's map
  - $$ is the shell's process id

Memory example source code

```
        segment .data
a       dd      4
b       dd      4.4
c       times   10 dd 0
d       dw      1, 2
e       db      0xfb
f       db      "hello world", 0

        segment .bss
g       resd    1
h       resd    10
i       resb    100
```

Memory example source code (2)

```
        segment .text
        global  main      ; let the linker know about main
main:
        push    rbp       ; set up a stack frame for main
        mov     rbp, rsp  ; set rbp to point to the stack frame
        sub     rsp, 16   ; leave some room for local variables
                          ; leave rsp on a 16 byte boundary
        xor     eax, eax  ; set rax to 0 for return value
        leave             ; undo the stack frame manipulations
        ret
```

Memory example listing file

```
                                     %line 1+1 memory.asm
                                     [section .data]
00000000 04000000                    a dd 4
00000004 CDCC8C40                    b dd 4.4
00000008 00000000<rept>              c times 10 dd 0
00000030 01000200                    d dw 1, 2
00000034 FB                          e db 0xfb
00000035 68656C6C6F20776F72-         f db "hello world", 0
00000035 6C6400
```

- Addresses are relative to start of `.data` in this file
- Notice that the 4 byte of 4 is at address 0 (backwards)
- b = 0x408ccccd = 0 10000001 00011001100110011001101
- Sign bit is 0, exponent field is 0x81 = 129, exponent = 2
- Fraction is 1.00011001100110011001101

Memory example listing file (2)

```
11                                  [section .bss]
12 00000000 <gap>                   g resd 1
13 00000004 <gap>                   h resd 10
14 0000002C <gap>                   i resb 100
```

- Notice that the addresses start again at 0
- The commands reserve space
- `resd 1` reserves 1 double word or 4 bytes
- `resd 10` reserves 10 double words or 40 bytes
- `resb 100` reserves 100 bytes

Memory example listing file (3)

```
16                                      [section .text]
17                                      [global main]
18                                      main:
19 00000000 55                           push rbp
20 00000001 4889E5                       mov rbp, rsp
21 00000004 4883EC10                     sub rsp, 16
22 00000008 31C0                         xor eax, eax
23 0000000A C9                           leave
24 0000000B C3                           ret
```

Examining memory with `gdb`

- Time to try some commands in `gdb`
- Use p for print
  - Print allows printing expressions
  - p/d for decimal
  - try format options t, u, i, c, s, f, a and x
- Examine requires a memory address
  - x/NFS
  - N is an optional count
  - F is a format like print
  - S is a size character: b=1, h=2, w=4, g=8

Examining memory with ebe

- Run program to a breakpoint
- Control-right-click on a variable name
- Fill in popup form
  - Variable name
  - Address - will the `&variable`
  - Format
    - floating point
    - decimal
    - hexadecimal
    - character
    - string
    - string array (like argv in main)
  - Size: 1, 2, 4 or 8 bytes
  - First and last indices
- Variable will be monitored in data window