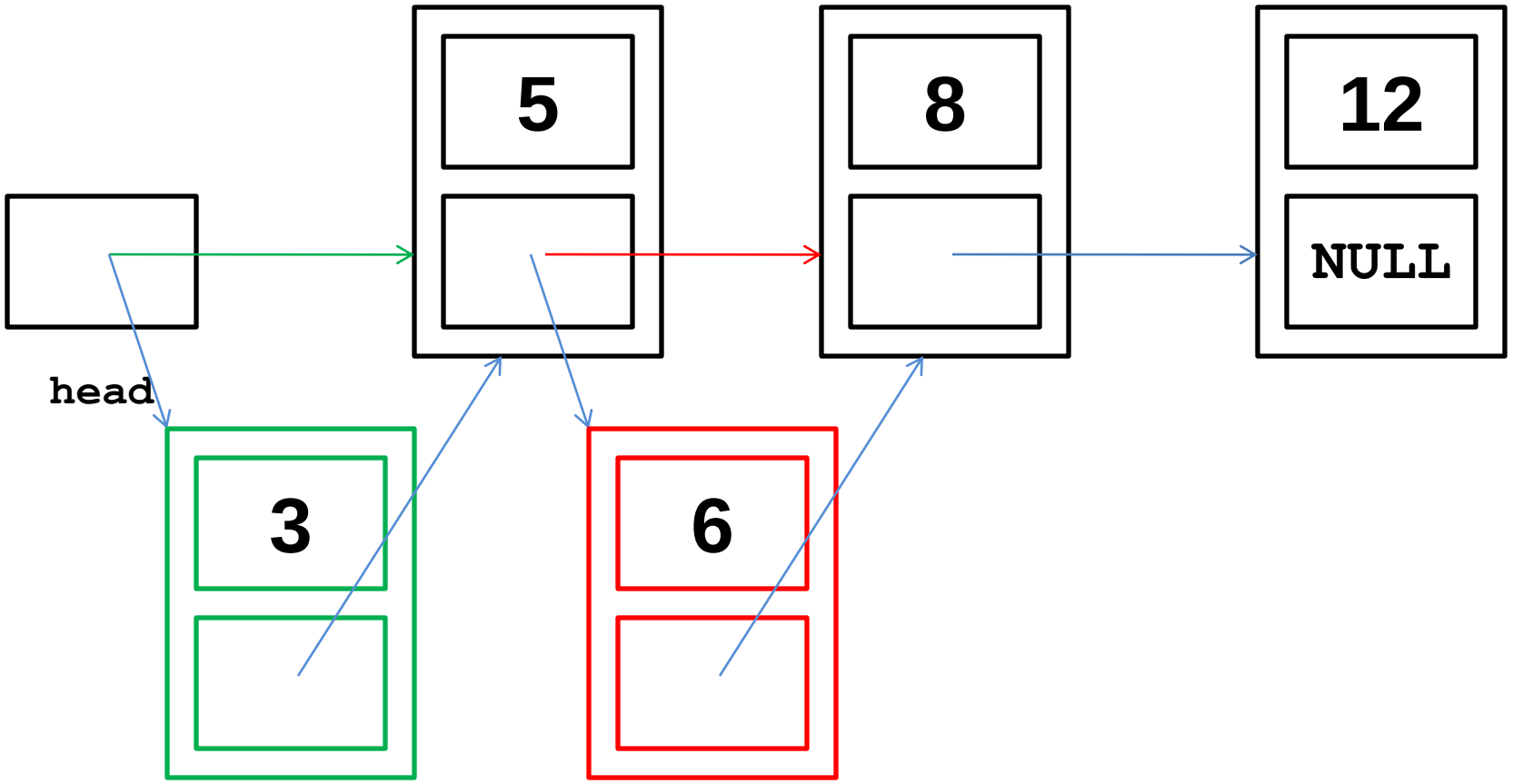
CMSC 216
Introduction to Computer Systems
Lecture 11
Linked Lists (cont) &
Assembly Language Programming

Administrivia

- Read Chapter 11, Reek

Deleting from a linked list



Tracing deletion

```
int delete(Node **head, int value) {
  Node *prev = NULL, *current = *head;
  while (current != NULL && current->data != value) {
    prev = current;
    current = current->next;
  }
  if (current == NULL)
    return 0;  /* not found */
  if (prev != NULL)
    prev->next = current->next;
  else
    *head = current->next;  /* deleted first item */
  free(current);
  return 1;
}
```

Doubly linked lists

- We can have references in both directions
- Allows use of only one pointer when performing some operations

```
typedef struct dl_node {
   struct dl_node *prev;
   struct dl_node *next;
   int val;
} DL_node;
```

Deletion from a doubly linked list

```
int delete(DL_node **root, int target) {
  DL_node *current;
  for (current = *root; current != NULL; current = current->next)
    if (current->val == target)
      break;
  if (current == NULL)
    return 0;
  if (current == *root)
    *root = current->next;
  else
    current->prev->next = current->next;
  if (current->next != NULL)
    current->next->prev = current->prev;
  free(current);
  return 1;
}
```

Chapters 3 and 4.1, Bryant and O'Hallaron

Assembly Language

Assembly language

- The CPU uses machine language to perform all its operations
- Assembly is a much more readable translation of machine language, and it is what we work with if we need to see what the computer is doing
- Many different kinds of assembly languages; we'll focus on the Y86 language defined in the text (with minor modifications)

An example of Y86 assembly

- The summation of all integers from 1 to 1000:

```
        irmovl $0,%eax     # sum = 0
        irmovl $1,%ecx     # num = 1
Loop:   addl   %ecx,%eax   # sum += num
        irmovl $1,%edx     # tmp = 1
        addl   %edx,%ecx   # num++
        irmovl $1000,%edx  # lim = 1000
        subl   %ecx,%edx   # if lim - num >= 0
        jge    Loop        # loop again

        wrint  %eax        # printf("%d", sum)
        irmovl $10,%edx    # ch = '\n'
        wrch   %edx        # printf("%c", ch)
        halt
```

Assembling assembly

- Machine code (pure numbers) is generated by translating each instruction into binary numbers that the CPU uses
- This process is called "assembling"; conversely, we can take assembled code and disassemble it into (mostly) human readable assembly language

Assembly language concepts

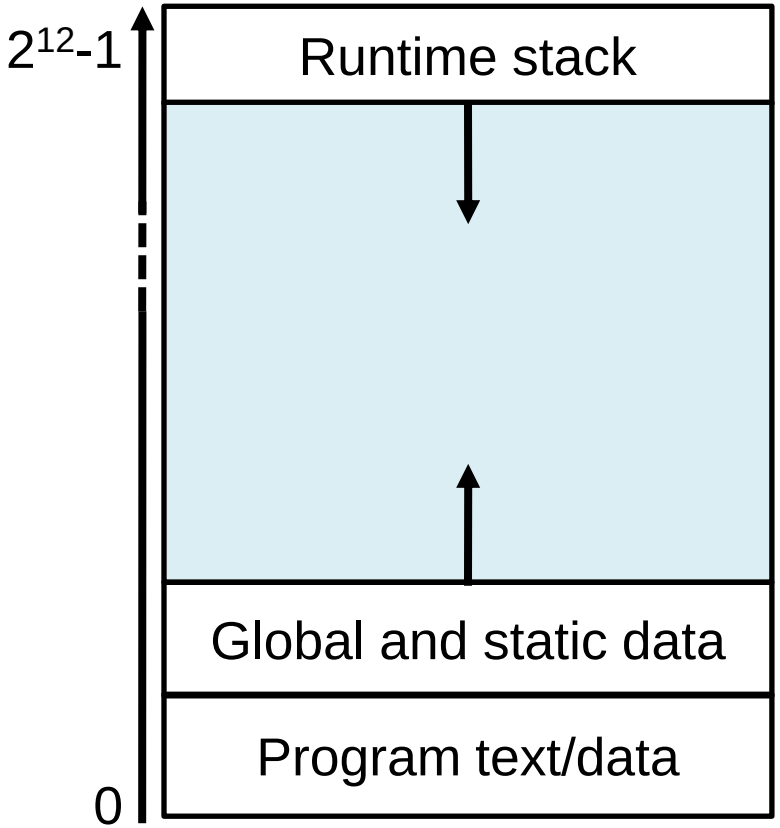
- Registers
  - Fast-access locations that the CPU uses, rather than storing all variables in memory
  - In the code we'll be examining, these are named by three-letter codes, and begin with `%`
    - **`%eax, %ebx`**
  - Y86 has 8 registers, listed on pg. 168 in Figure 3.2 (shown in another slide)
- Memory
  - Program text is here, along with any data which cannot fit in the registers
  - Since there are usually many different pieces of data involved in a program, and a limited number of registers, program variables are stored in memory when not in immediate use

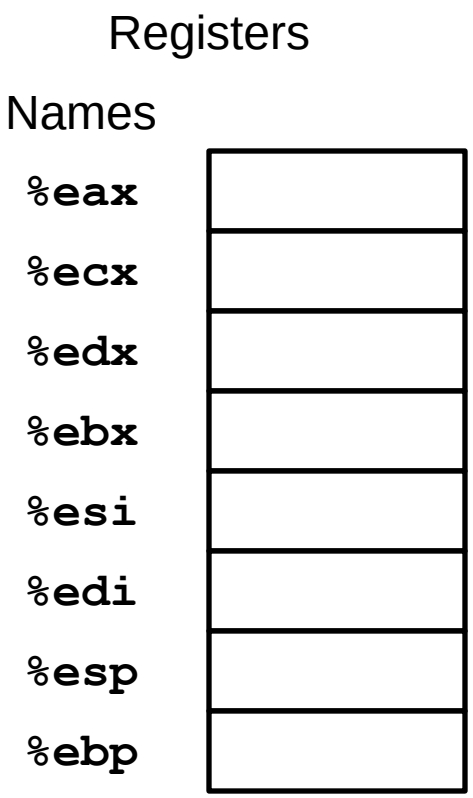
Memory Organization

- Word → 4 bytes
- Address → specifies first byte in word
- Big Endian vs. Little Endian
  - Assume word has value 0x01234567
  - Assume address for value is 0x300
  - Big Endian
    - Least significant byte has **highest** address
    - 01 → 0x300
    - 23 → 0x301
    - 45 → 0x302
    - 67 → 0x303
    - “Normal layout of data”
  - Little Endian
    - Least significant byte has **lowest** address
    - 67 → 0x300
    - 45 → 0x301
    - 23 → 0x302
    - 01 → 0x303
    - Reverse of “normal layout”

Y86 program state

- 2^{12} bytes of memory (4096 bytes)
- You can set the stack to start somewhere other than 0x1000 (4096), but you have to explicitly set it (%esp is 0 by default)





Y86 Registers

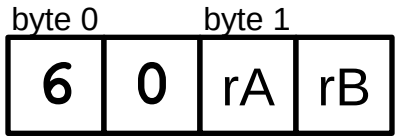
- %eax, %ecx, %edx, %ebx, %esi, %edi
  - For the most part are considered general-purpose registers
  - Corresponding (IDs) 0, 1, 2, 3, 6, 7
- %esp → stack pointer
  - ID → 4
- %ebp → frame pointer
  - ID → 5
- Register ID 8 indicates “no register”

Instruction encoding

- In any assembly language, each instruction name has a numeric opcode associated with it; Y86 opcodes use one byte each
- Registers also have a numeric mapping, as do labels (as we just saw)
- These numbers are used to encode each instruction into a number
- In Y86, instructions have variable lengths, and are not aligned; most take 6 bytes, but some use fewer

Encoding examples

- In Y86, the **`addl`** (add integers) instruction has the opcode of **`0x60`**
  - **`1 → "long" words (4-byte integers)`**
- The register names **`%edx`** and **`%ecx`** map to numbers 2 and 1, respectively
- The encoding for an **`addl`** instruction is specified by the Y86 instruction set as following this format (pgs. 259, 261):
  - **`addl`** rA, rB



- So what would the numeric encoding of **`addl %edx,%ecx`** be?
  - **`0x6021`**
- The **subl** (subtract integers)instruction works in a similar way

Working with Y86

- Source code is usually stored in **`*.ys`** files
- On the Grace systems, there are two programs available in **`~/216public/bin`** for working with Y86 programs
- **`yas`** is the Y86 assembler, which creates **`*.yo`** files
  - Run like this: **`yas prog.ys`**
- **`yis`** is the Y86 simulator, which operates on `*.yo` files
  - Run like this: **`yis prog.yo`**
- You can just type **`yas`** and **`yis`** (you don't need to access them in bin)

Assembly language instructions

- These typically have an instruction name, a list of registers, and sometimes have a constant value too
- Example: simple.ys
  - Le’ts run yas *simple.ys*
  - Run the simulator yis *simple.yo*
    - PC → Program Counter
      - Address of instruction to be executed
    - CC → Condition codes (single-bit flags set by logical or arithmetic instruction
      - Z (Zero), S (Negative), O (Overflow)
    - Words (4-bytes) stored in little-endian
  - Let’s see the contents of simple.yo

halt

- Without a **`halt`** instruction, the simulator will attempt to read in possibly invalid instructions

Assembler directives

Directive	Effect
`.pos number`	Subsequent lines of code start at address **number**
`.align number`	Align the next line to a **number**-byte boundary
`.long number`	Put **number** at the current address in memory

- These can be used to set up memory in various places in the address space
- **`.pos`** can put sections of code in different places in memory
- **`.align`** should be used before setting up a static variable
- **`.long`** can be used to initialize a static variable

Y86 data movement instructions

Instruction	Effect	Description
`irmovl V,R`	Reg[R] ← V	Immediate-to-register move
`rrmovl rA,rB`	Reg[rB] ← Reg[rA]	Register-to-register move
`rmmovl rA,D(rB)`	Mem[Reg[rB]+D] ← Reg[rA]	Register-to-memory move
`mrmovl D(rA),rB`	Reg[rB] ← Mem[Reg[rA]+D]	Memory-to-register move

- **`irmovl`** is used to place known numeric values (labels or numeric literals) into registers
- **`rrmovl`** copies a value between registers
- **`rmmovl`** stores a word in memory
- **`mrmovl`** loads a word from memory
- **`rmmovl`** and **`mrmovl`** are the only instructions that access memory - Y86 is a load/store architecture

Examples of data movement

```
irmovl $55,%edx       # d = 55
rrmovl %edx,%ebx      # b = d
irmovl Array,%eax     # a = Array
rmmovl %ebx,4(%eax) # a[1] = 55
mrmovl 0(%eax),%ecx # c = a[0]
halt
   .align 4
Array:
   .long 0x6f
   .long 0x84
```

Data movement example, cont.

- Assembler output:

```
0x000: 308237000000 | irmovl $55,%edx     # d = 55
0x006: 2023         | rrmovl %edx,%ebx    # b = d
0x008: 30801c000000 | irmovl Array,%eax   # a = Array
0x00e: 403004000000 | rmmovl %ebx,4(%eax) # a[1] = 55
0x014: 501000000000 | mrmovl 0(%eax),%ecx # c = a[0]
0x01a: 10           | halt
                    |
0x01c:              |   .align 4
0x01c:              | Array:
0x01c: 6f000000     |   .long 0x6f
0x020: 84000000     |   .long 0x84
```

- Simulator output:

```
Stopped in 6 steps at PC = 0x1b.  Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x0000001c
%ecx:   0x00000000      0x0000006f
%edx:   0x00000000      0x00000037
%ebx:   0x00000000      0x00000037

Changes to memory:
0x0020: 0x00000084      0x00000037
```

Y86 input/output instructions

Instruction	Effect	Description
`rdch R`	`scanf("%c", &Reg[R])`	Read character
`rdint R`	`scanf("%d", &Reg[R])`	Read integer
`wrch R`	`printf("%c", Reg[R])`	Write character
`wrint R`	`printf("%d", Reg[R])`	Write integer

- All these instructions are extensions to Y86 we've added to the ones in the book
- These are what allow you to interact with the simulator and write more interesting programs

I/O example

- Assembler output:

```
0x000: f208           | rdint  %eax      # a = 65 (via scanf())
0x002: f038           | rdch   %ebx      # b = 'B' (via scanf())
0x004: f308           | wrint  %eax      # printf("%d", a)
0x006: f108           | wrch   %eax      # printf("%c", a)
0x008: f338           | wrint  %ebx      # printf("%d", b)
0x00a: f138           | wrch   %ebx      # printf("%c", b)
0x00c: 30810a000000   | irmovl $10,%ecx  # c = 10
0x012: f118           | wrch   %ecx      # printf("%c", c)
0x014: 10             | halt
```

- Simulator run:

```
$ echo 65B | yis io.yo
65A66B
Stopped in 9 steps at PC = 0x15.  Exception 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%eax:   0x00000000      0x00000041
%ecx:   0x00000000      0x0000000a
%ebx:   0x00000000      0x00000042

Changes to memory:
```