Compiler 2017

Linux x86-64 Assembly in NASM

Lequn Chen

March 16, 2017

Operating System

- User Space
  - Your program runs in user space
  - Limited capability (simply put: an advanced calculator)
  - Calls some “magic function” to obtain super power! (e.g. read, write, …)

- Kernel Space
  - Dirty work: Deal with hardwares (e.g. raw I/O, interrupt, …)
  - Supervise: Kill user programs when unauthorized action
  - Service: Provide easy-to-use APIs
    - The most low level API: system call

C Library

- libc: some most basic high level operations
  - scanf / printf
  - memset / memcpy
  - strlen / strcmp
  - …
- in user space
- libc calls syscall for you, so you don’t feel their existence

Operating System

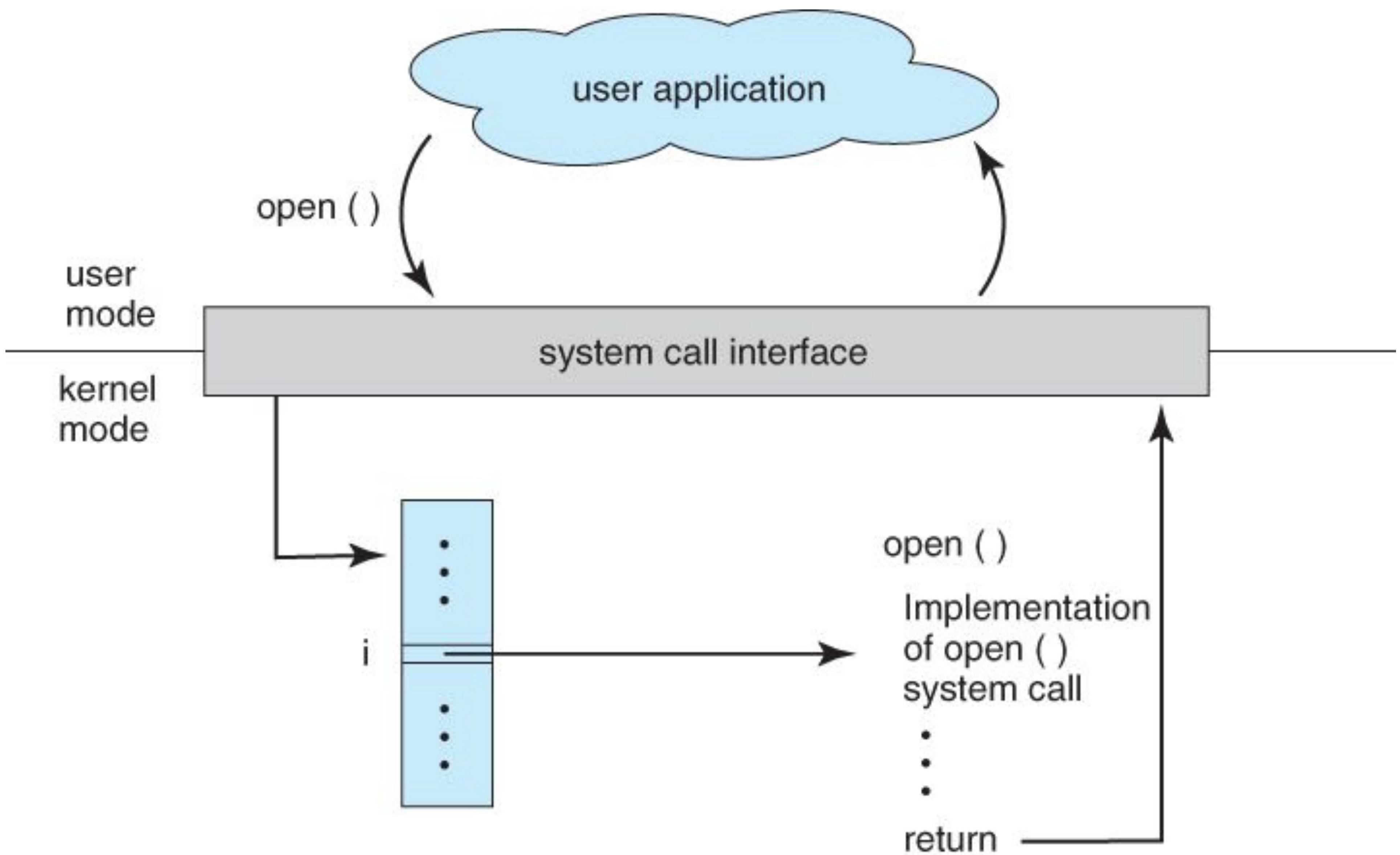


Image: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/2_Structures.html

```c
#include <stdio.h>

int main(void) {
    char name[10];
    scanf("%s", name);
    printf("Hello, %s\n", name);
    return 0;
}
```

Libc

```c
#include <unistd.h>


int main(void) {
    char name[10];
    ssize_t n = read(0, name, 10);
    write(1, "Hello, ", 8);
    write(1, name, n);
    write(1, "\n", 2);
    return 0;
}
```

Linux API

```c
#include <unistd.h>
#include <sys/syscall.h>

int main(void) {
    char name[10];
    long n = syscall(3, 0, name, 10);
    syscall(4, 1, "Hello, ", 8);
    syscall(4, 1, name, n);
    syscall(4, "\n", 2);
    return 0;
}
```

Linux System Call

Hello World in NASM

```
; ----------------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ----------------------------------------------------------------------------------------

        global  _start

        section .text
_start:
        ; write(1, message, 13)
        mov     rax, 1                  ; system call 1 is write
        mov     rdi, 1                  ; file handle 1 is stdout
        mov     rsi, message            ; address of string to output
        mov     rdx, 13                 ; number of bytes
        syscall                         ; invoke operating system to do the write

        ; exit(0)
        mov     eax, 60                 ; system call 60 is exit
        xor     rdi, rdi                ; exit code 0
        syscall                         ; invoke operating system to exit
message:
        db      "Hello, World", 10      ; note the newline at the end
```

```
mov x, y:   x := y
xor x, y:   x := x xor y
syscall : invoke system call
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Structure of a NASM Program

Instructions

Directives, such as global and extern

Operands

One or more sections

```
        global  _start

        section .text
_start:
        ; write(1, message, 13)
        mov     rax, 1
        mov     rdi, 1
        mov     rsi, message
        mov     rdx, 13
        syscall

        ; exit(0)
        mov     eax, 60
        xor     rdi, rdi
        syscall
message:
        db      "Hello, World", 10
```

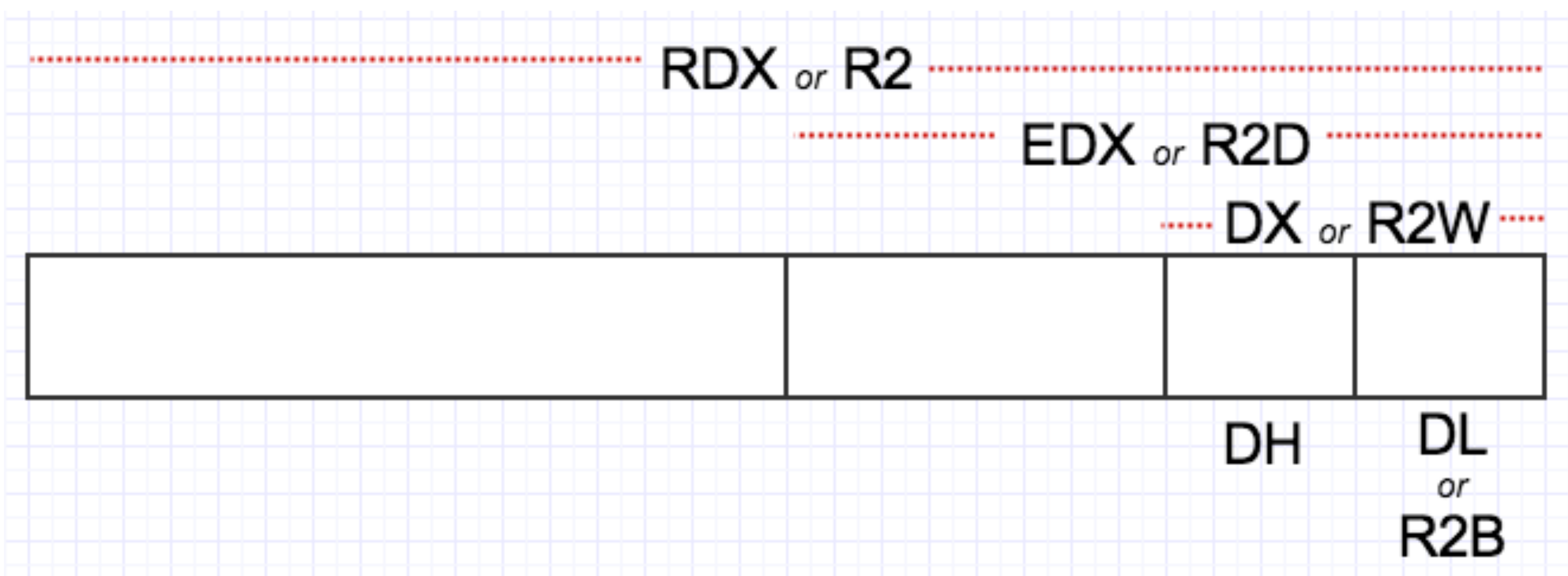
ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Register Operands

- 64-bit
  - RAX RCX RDX RBX RSP RBP RSI RDI R8 R9 R10 R11 R12 R13 R14 R15
    R0 R1 R2 R3 R4 R5 R6 R7
- 32-bit
  - EAX ECX EDX EBX ESP EBP ESI EDI R8D R9D R10D R11D R12D R13D R14D R15D
    R0D R1D R2D R3D R4D R5D R6D R7D
- 16-bit
  - AX CX DX BX SP BP SI DI R8W R9W R10W R11W R12W R13W R14W R15W
    R0W R1W R2W R3W R4W R5W R6W R7W
- 8-bit (low)
  - AL CL DL BL SPL BPL SIL DIL R8B R9B R10B R11B R12B R13B R14B R15B
    R0B R1B R2B R3B R4B R5B R6B R7B
- 8-bit (high)
  - AH CH DH BH

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Register Operands



ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Memory Operands

base
index
displacement

```
[ number ]
[ reg ]
[ reg + reg*scale ]     scale in {1,2,4,8}
[ reg + number ]
[ reg + reg*scale + number ]
```

```
[750]                    ; displacement only
[rbp]                    ; base register only
[rcx + rsi*4]            ; base + index * scale
[rbp + rdx]              ; scale is 1
[rbx - 8]                ; displacement is -8
[rax + rdi*8 + 500]      ; all four components
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Immediate Operands

```
200          ; decimal
0200         ; still decimal - the leading 0 does not make it octal
0200d        ; explicitly decimal - d suffix
0d200        ; also decimal - 0d prefex
0c8h         ; hex - h suffix, but leading 0 is required because c8h looks like a var
0xc8         ; hex - the classic 0x prefix
0hc8         ; hex - for some reason NASM likes 0h
310q         ; octal - q suffix
0q310        ; octal - 0q prefix
11001000b    ; binary - b suffix
0b1100_1000  ; binary - 0b prefix, and by the way, underscores are allowed
```

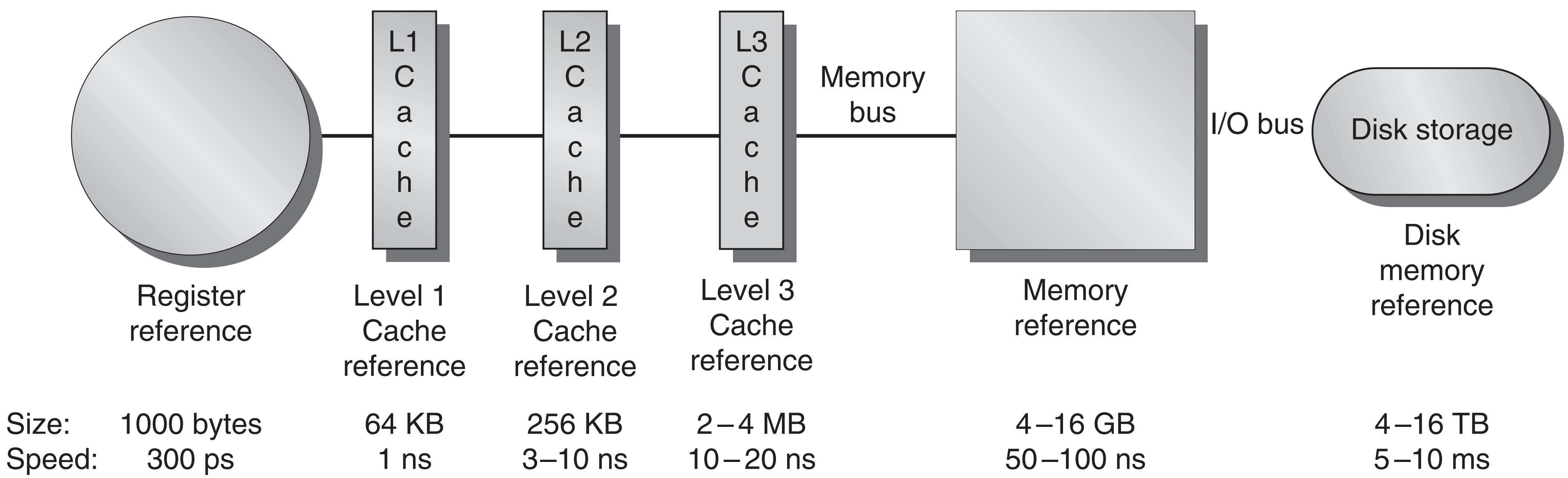
ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Instructions

- Instructions with two memory operands are extremely rare. Most of the basic instructions have only the following forms:
  - `add reg, reg`
  - `add reg, mem`
  - `add reg, imm`
  - `add mem, reg`
  - `add mem, imm`

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Memory Hierarchy



(a) Memory hierarchy for server

Image: Figure 2.1, Computer Architecture: A Quantitative Approach (5th Edition)

Defining Data

```
db    0x55                 ; just the byte 0x55
db    0x55,0x56,0x57       ; three bytes in succession
db    'a',0x55             ; character constants are OK
db    'hello',13,10,'$'    ; so are string constants
dw    0x1234               ; 0x34 0x12
dw    'a'                  ; 0x61 0x00 (it's just a number)
dw    'ab'                 ; 0x61 0x62 (character constant)
dw    'abc'                ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678           ; 0x78 0x56 0x34 0x12
dd    1.234567e20          ; floating-point constant
dq    0x123456789abcdef0   ; eight byte constant
dq    1.234567e20          ; double-precision float
dt    1.234567e20          ; extended-precision float
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Reserving Space

```
buffer:         resb    64              ; reserve 64 bytes
wordvar:        resw    1               ; reserve a word
realarray:      resq    10              ; array of ten reals
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Data Segments

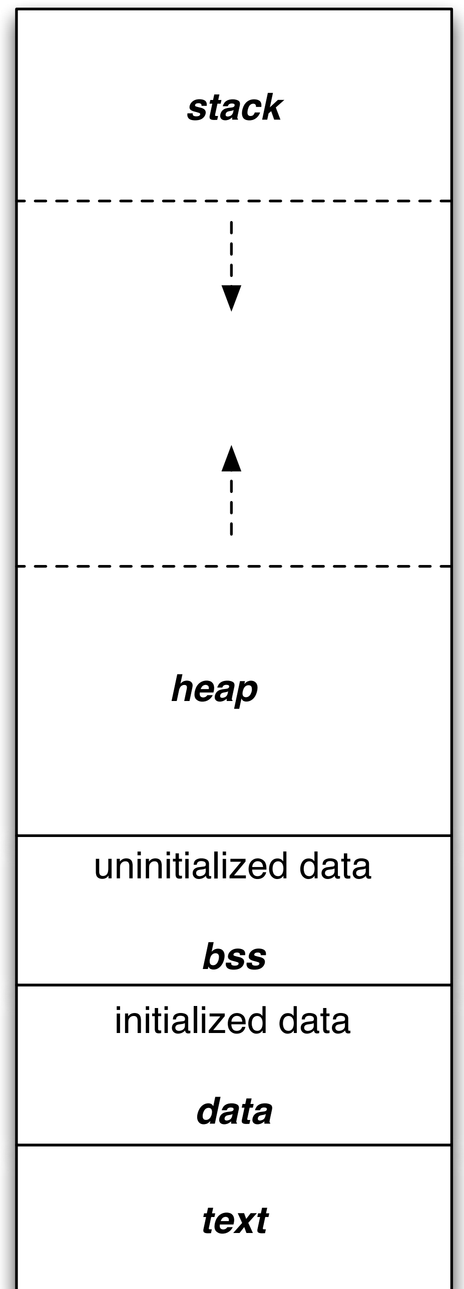


Image: https://en.wikipedia.org/wiki/Data_segment

Using a C Library

```
; ----------------------------------------------------------------------------------------
; Writes "Hola, mundo" to the console using a C library. Runs on Linux or any other system
; that does not use underscores for symbols in its C library. To assemble and run:
;
;     nasm -felf64 hola.asm && gcc hola.o && ./a.out
; ----------------------------------------------------------------------------------------

        global  main
        extern  puts

        section .text
main:                                   ; This is called by the C library startup code
        mov     rdi, message            ; First integer (or pointer) argument in rdi
        call    puts                    ; puts(message)
        ret                             ; Return from main back into C library wrapper
message:
        db      "Hola, mundo", 0        ; Note strings must be terminated with 0 in C
```

```
call:  call a function
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Calling Conventions

- From left to right, pass as many parameters as will fit in registers. The order in which registers are allocated, are: `rdi, rsi, rdx, rcx, r8, r9`

- Additional parameters are pushed on the stack, right to left, and are to be removed by the caller after the call.

- After the parameters are pushed, the call instruction is made, so when the called function gets control, the return address is at `[rsp]`, the first memory parameter is at `[rsp+8]`, etc.

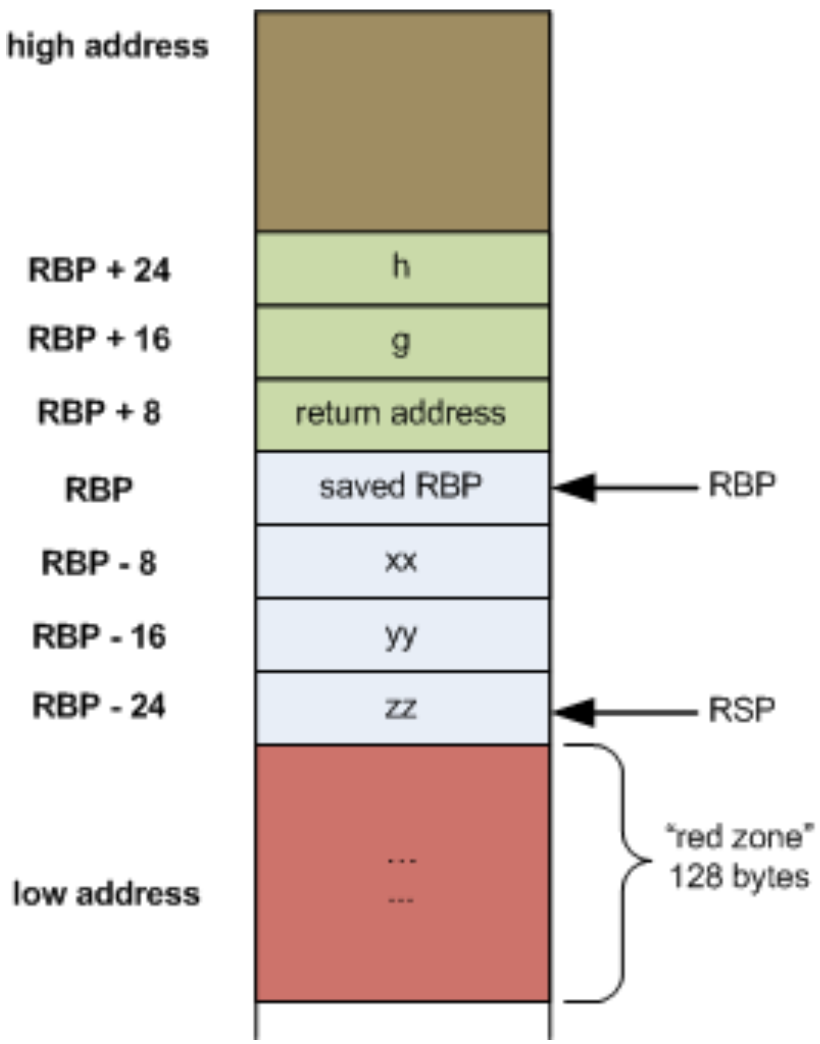
ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

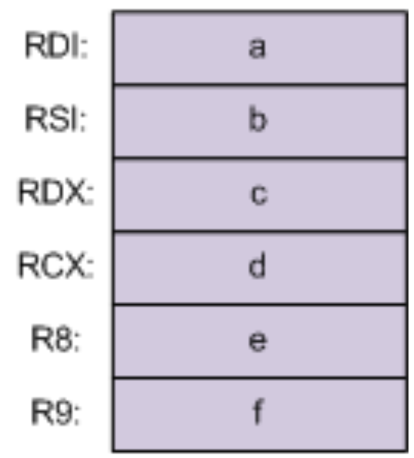
Calling Conventions

- The stack pointer `rsp` must be aligned to a 16-byte boundary before making a call.
- The only registers that the called function is required to preserve (the callee-save registers) are: `rbp`, `rbx`, `r12`, `r13`, `r14`, `r15`. All others are free to be changed by the called function.
- Integers are returned in `rax`

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Calling Conventions






```c
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```

ref: http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64

Register Usage

Register	Usage	Preserved across function calls
`%rax`	temporary register; with variable arguments passes information about the number of vector registers used; 1^{st} return register	No
`%rbx`	callee-saved register	Yes
`%rcx`	used to pass 4^{th} integer argument to functions	No
`%rdx`	used to pass 3^{rd} argument to functions; 2^{nd} return register	No
`%rsp`	stack pointer	Yes
`%rbp`	callee-saved register; optionally used as frame pointer	Yes
`%rsi`	used to pass 2^{nd} argument to functions	No
`%rdi`	used to pass 1^{st} argument to functions	No
`%r8`	used to pass 5^{th} argument to functions	No
`%r9`	used to pass 6^{th} argument to functions	No
`%r10`	temporary register, used for passing a function's static chain pointer	No
`%r11`	temporary register	No
`%r12-r14`	callee-saved registers	Yes
`%r15`	callee-saved register; optionally used as GOT base pointer	Yes

Image: Figure 3.4, System V Application Binary Interface: AMD64 Architecture Processor Supplement

Fibonacci Example

```
; ----------------------------------------------------------------------------
; A 64-bit Linux application that writes the first 90 Fibonacci numbers. To
; assemble and run:
;
;     nasm -felf64 fib.asm && gcc fib.o && ./a.out
; ----------------------------------------------------------------------------

        global  main
        extern  printf

        section .text
main:
        push    rbx                     ; we have to save this since we use it

        mov     ecx, 90                 ; ecx will countdown to 0
        xor     rax, rax                ; rax will hold the current number
        xor     rbx, rbx                ; rbx will hold the next number
        inc     rbx                     ; rbx is originally 1
```

```
push x     : rsp -= sizeof(x); [rsp] := x;
pop x      : x := [rsp]; rsp += sizeof(x);
jnz label  : if Z flag is set, jump to label
call label: push address of next inst; jump to label
ret        : Pop into the instruction pointer
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

Fibonacci Example

```
print:
        ; We need to call printf, but we are using rax, rbx, and rcx.  printf
        ; may destroy rax and rcx so we will save these before the call and
        ; restore them afterwards.

        push    rax                     ; caller-save register
        push    rcx                     ; caller-save register

        mov     rdi, format             ; set 1st parameter (format)
        mov     rsi, rax                ; set 2nd parameter (current_number)
        xor     rax, rax                ; because printf is varargs

        ; Stack is already aligned because we pushed three 8 byte registers
        call    printf                  ; printf(format, current_number)

        pop     rcx                     ; restore caller-save register
        pop     rax                     ; restore caller-save register

        mov     rdx, rax                ; save the current number
        mov     rax, rbx                ; next number is now current
        add     rbx, rdx                ; get the new next number
        dec     ecx                     ; count down
        jnz     print                   ; if not done counting, do some more

        pop     rbx                     ; restore rbx before returning
        ret
format:
        db  "%20ld", 10, 0
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

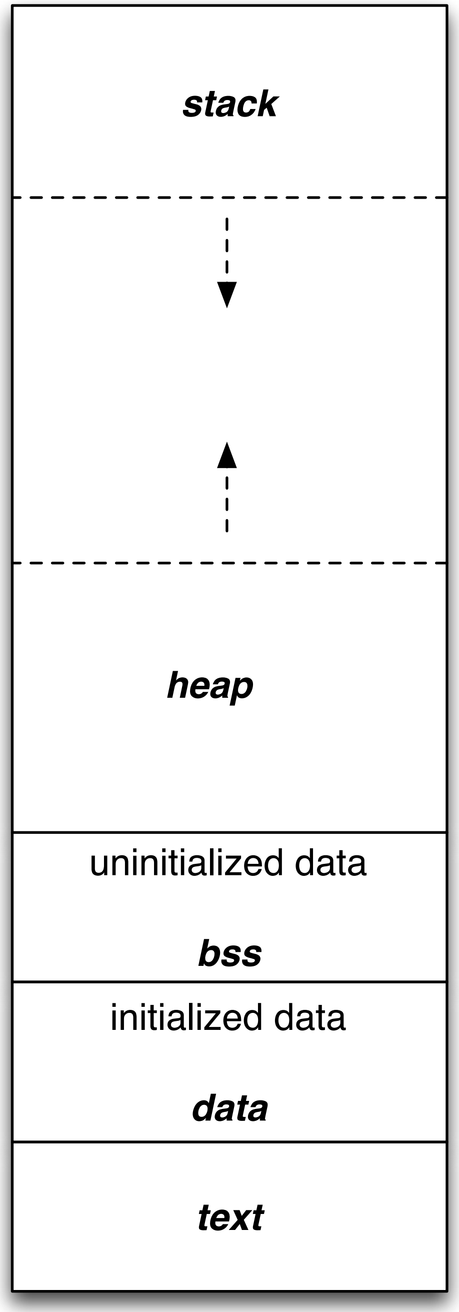
Recursion Example

```
; ----------------------------------------------------------------------------
; An implementation of the recursive function:
;
;   uint64_t factorial(uint64_t n) {
;       return (n <= 1) ? 1 : n * factorial(n-1);
;   }
; ----------------------------------------------------------------------------

        global  factorial

        section .text
factorial:
        cmp     rdi, 1                  ; n <= 1?
        jnbe    L1                      ; if not, go do a recursive call
        mov     rax, 1                  ; otherwise return 1
        ret
L1:
        push    rdi                     ; save n on stack (also aligns %rsp!)
        dec     rdi                     ; n-1
        call    factorial               ; factorial(n-1), result goes in %rax
        pop     rdi                     ; restore n
        imul    rax, rdi                ; n * factorial(n-1), stored in %rax
        ret
```

ref: http://cs.lmu.edu/~ray/notes/nasmtutorial/

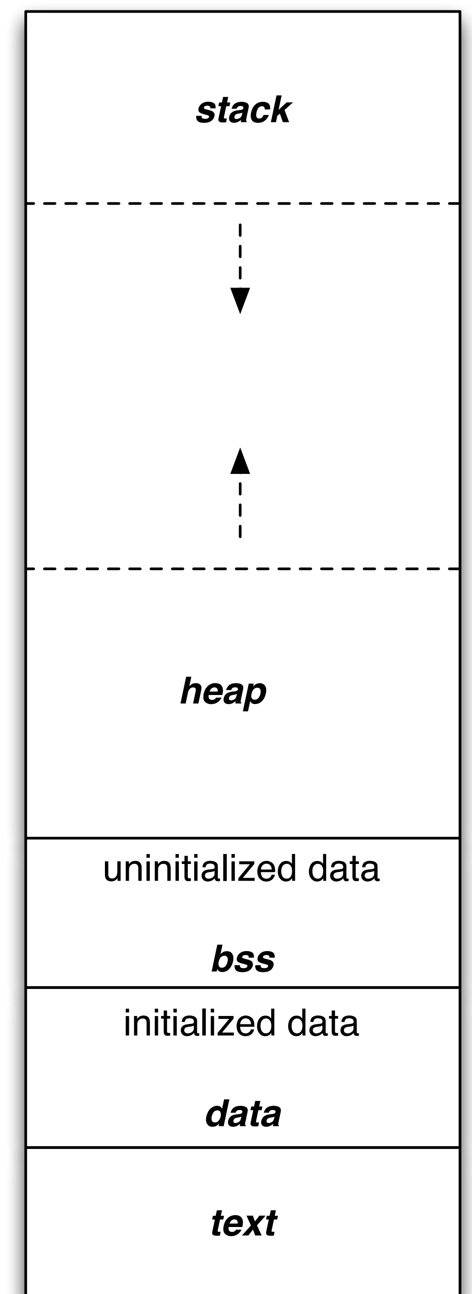
Heap?

```
; nasm -f elf64 -F dwarf -g sbrk.asm && ld -o sbrk sbrk.o && gdb sbrk

section .data
initial_break: dq 0
current_break: dq 0

section .text
global _start
_start:

; get current break address
;-----------------------------------------------------------------------
    mov    rax, 12               ;system call brk
    mov    rdi, 0                ;invalid address
    syscall
    mov    [current_break], rax
    mov    [initial_break], rax
```



Heap?

```
; allocate another 104857600 bytes of memory on the heap
;-------------------------------------------------------------------------
    mov    rax, 12                ;system call brk
    mov    rdi, [current_break]
    add    rdi, 104857600         ;allocate 104857600 bytes
    syscall

.b0: ;Break the program here in GDB. Also watch the memory used by
     ;using "top" command. You'll noticed the memory used by this
     ;program dropped from 100M to 160k.
    mov    [current_break], rax

    sub rax        , 8
    mov qword [rax], 123          ;write a qword to the end of heap

; free all allocated memory on the heap
;-------------------------------------------------------------------------
    mov    rax, 12                ;system call brk
    mov    rdi, [initial_break] ;reset break address to its initial addr
    syscall

.b1: ;Break the program here in GDB, to see the memory drop.

.exit:
    mov    rax, 60 ;system call exit
    mov    rdi, 0  ;return value := 0
    syscall
```



Learn NASM by Disassembly

- c2nasm.bash: https://gist.github.com/abcdabcd987/acb76b101094edac57537ab54ef1c4ef

- `./c2nasm.bash program.c`
- `cat program.asm`

Summary

- Linux System Call
- C Library
- NASM Format
- x86-64 Registers, Memory Addressing, Instruction Set
- x86-64 SystemV Calling Convention