Machine-Level Programming V: Memory layout

Jinyang Li

Some slides adapted from Bryant and O'Hallaron

x86 Procedure Recap

- `call`
  - push return address on stack, jump to label
- `ret`
  - pop 8 bytes from stack into PC
- **Argument passing from caller to callee**
  - First 6 arguments passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Rest on stack
- **Return value passing from callee to caller**
  - %rax
- **Local variable**
  - either registers, or allocated on stack (deallocated before `ret`)
- **Caller vs. callee-save registers**
  - Caller-save: all "argument" registers, %rax, %r10, %11
  - Callee-save: %rbx, %r12, %r13, %r14, %rbp

Recap: Procedure call example

```
int add2(int a, int b)
{
  return a + b;
}
```

```
add2:
    leal    (%rdi,%rsi), %eax
    ret
```

a: %edi
b: %esi
c: %edx

```
int add3(int a, int b, int c)
{
  int r = add2(a, b);
  r = r + c;
  return r;
}
```

```
add3:
    pushq       %rbx
    movl        %edx, %ebx
    movl        $0, %eax
    call        add2
    addl        %ebx, %eax
    popq         %rbx
    ret
```

%edx (containing c) is needed after call, so save in %ebx

Registers
- First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %9
- Return value: %rax

Today

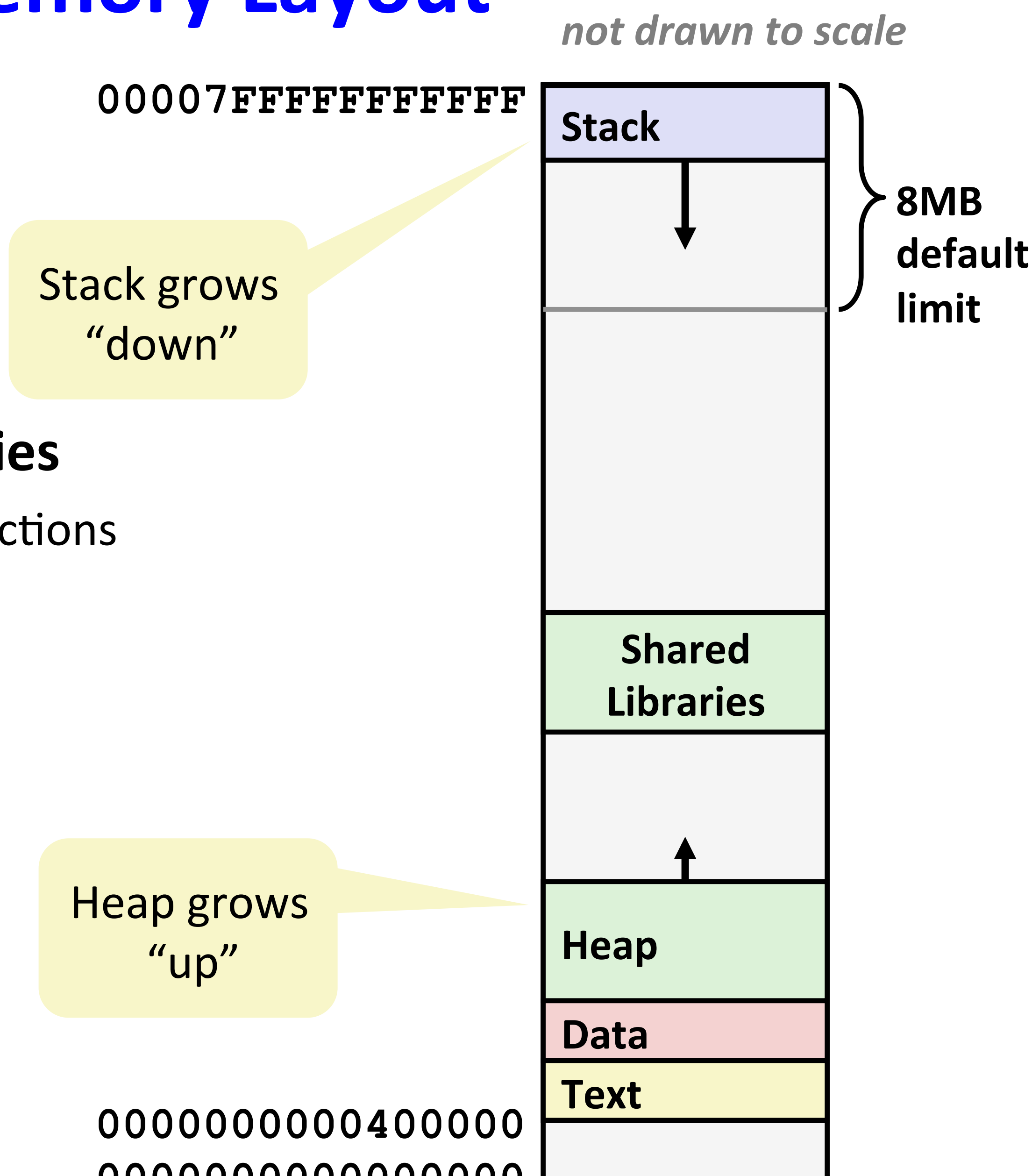
- Memory layout
- Demo: Using gdb for binary forensics

OS loads a program to memory

- **OS loads different parts of a program into different memory regions**
- **Parts of a running program:**
  - Stack
    - e.g. local variables
  - Heap
    - e.g. malloc(), new
  - (statically allocated) Data
    - global variables, string constants
  - Executable instructions
- **Why different regions?**
  - need to grow independently
  - need different permissions

x86-64 Linux Memory Layout

- **Stack**
- **Heap**
- **Data**
- **Text / Shared Libraries**
  - aka executable instructions

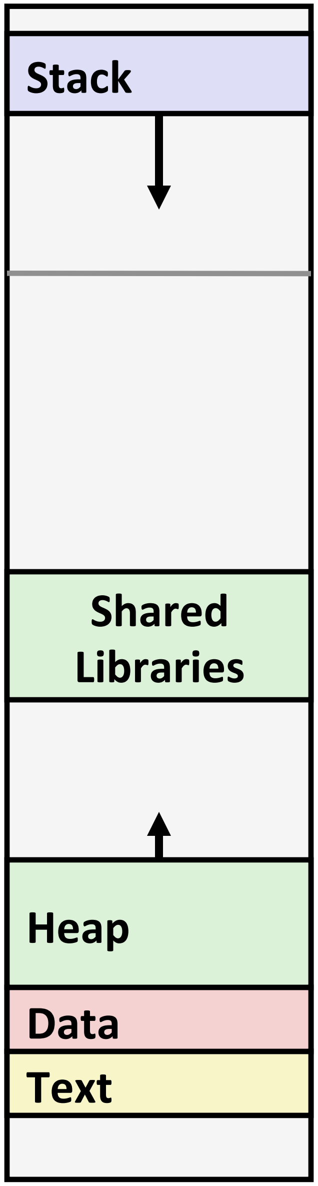


Memory Allocation Example

```
char big_array[1<<24];  /* 16 MB */
char huge_array[1<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1 << 28); /* 256 MB */
    p2 = malloc(1 << 8);  /* 256  B */
    p3 = malloc(1 << 32); /*   4 GB */
    p4 = malloc(1 << 8);  /* 256  B */
    …
}
```

Where does everything go?

not drawn to scale

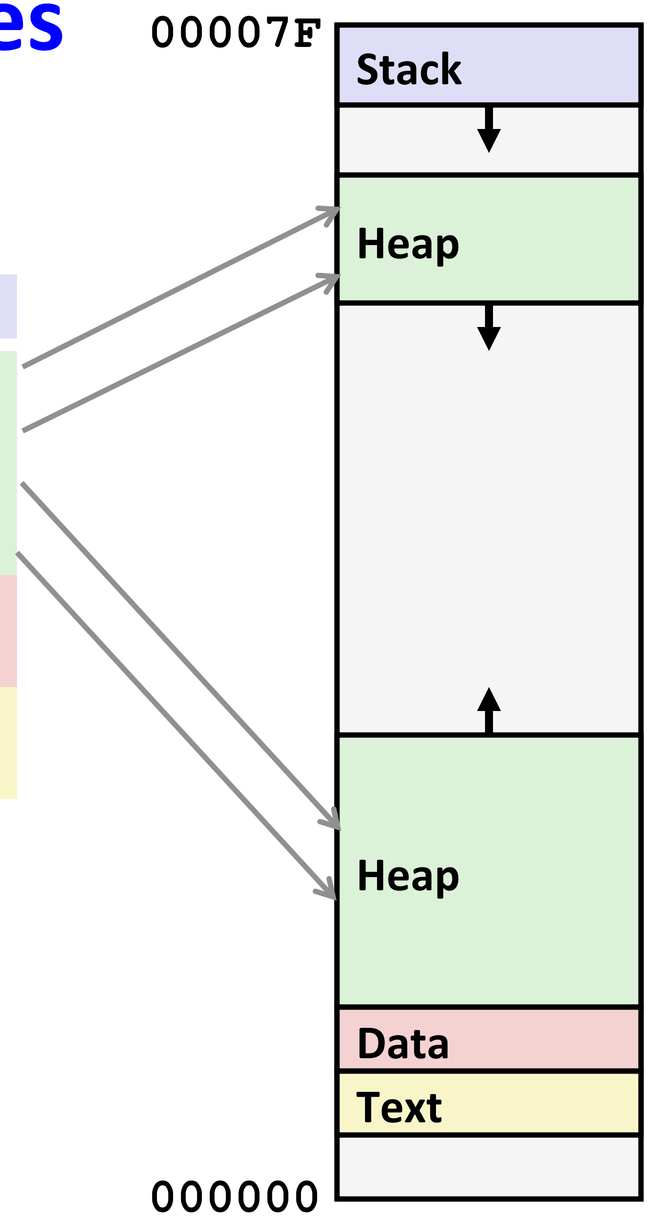


x86-64 Example Addresses

address range ~2^{47}

not drawn to scale

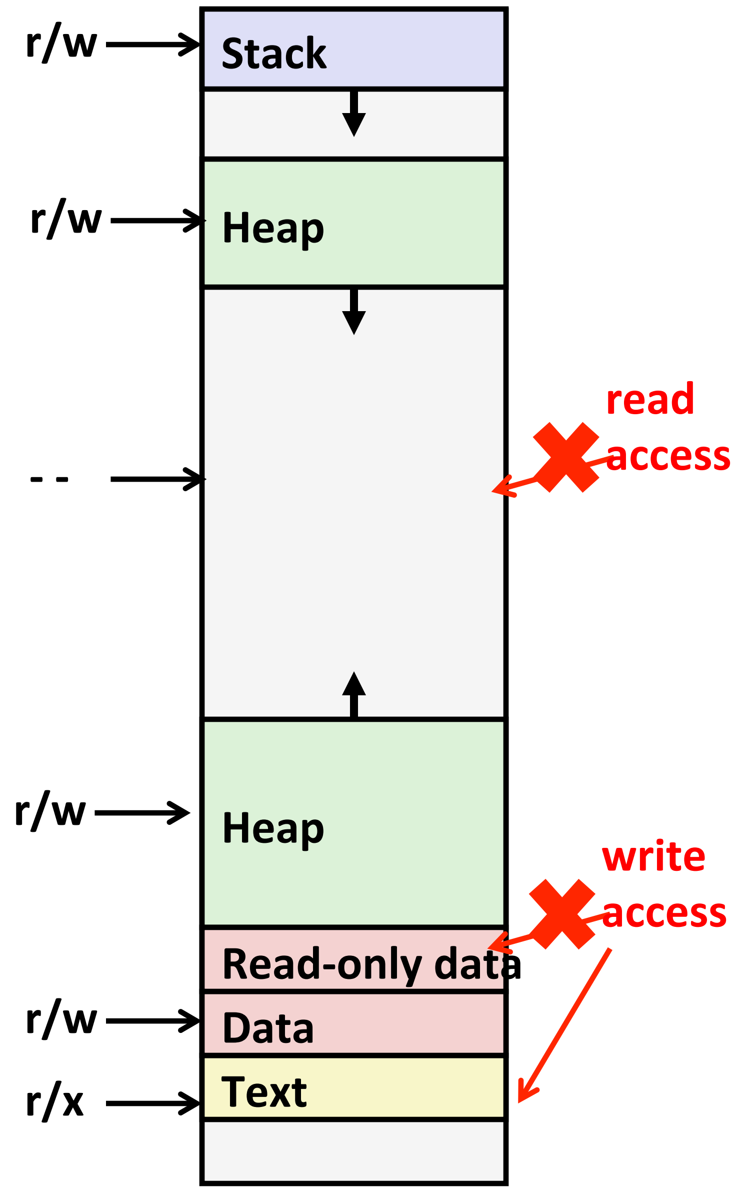
`local`	`0x00007ffe4d3be87c`
`p1`	`0x00007f7262a1e010`
`p3`	`0x00007f7162a1d010`
`p4`	`0x000000008359d120`
`p2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`



Segmentation Fault

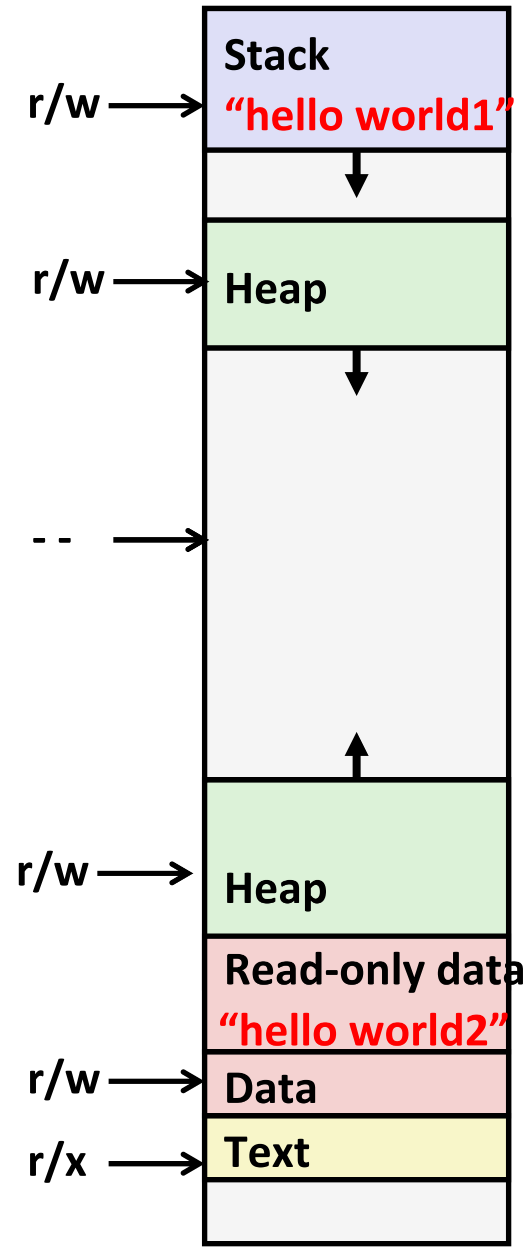
- Each memory segment can be readable (r), executable (x), writable(w), or none at all (-)

- Segmentation fault occurs when program tries to access "illegal" memory
  - Read from segment with no permission
  - Write to read-only segments



Segmentation fault example

```
int main() {
   char s1[100] = ”hello world1”;
   char *s2 = ”hello world2”;
   printf(”str1 %p str2 %p\n”, s1, s2);
   s1[0] = ’H’;
   s2[0] = ’H’;
   …
}
```



Not all Bad Memory Access lead to immediate segmentation

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

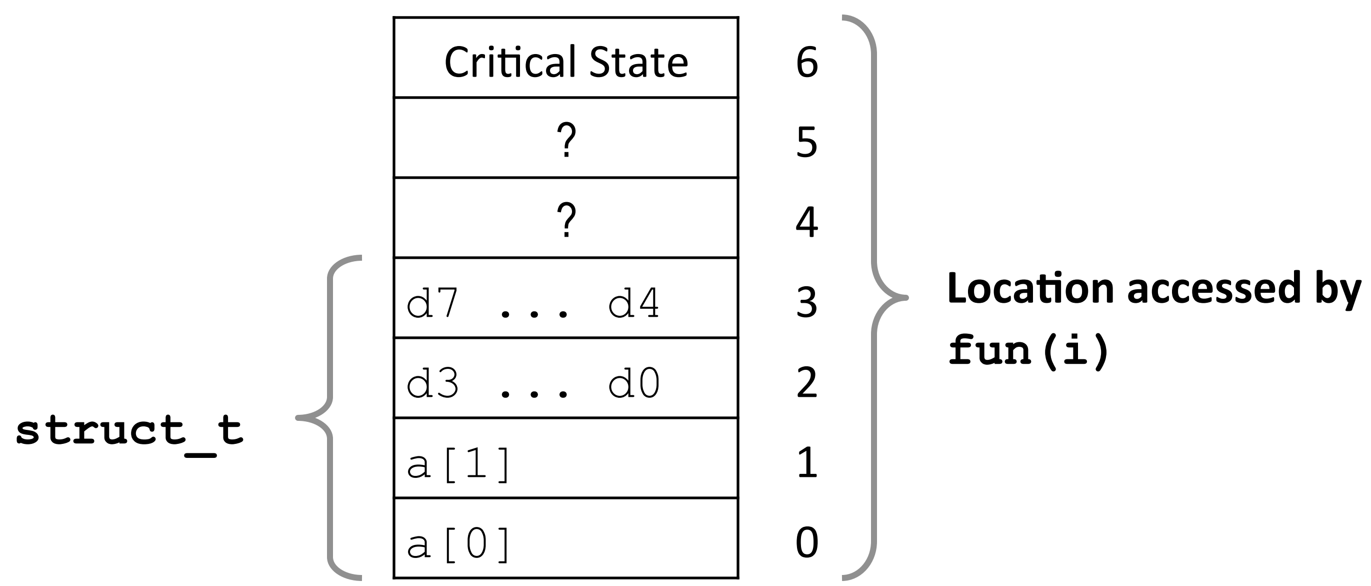
```
fun(0)  →   3.14
fun(1)  →   3.14
fun(2)  →   3.1399998664856
fun(3)  →   2.00000061035156
fun(4)  →   3.14
fun(6)  →   Segmentation fault
```

- Result is system specific

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault
```



Such problems are a BIG deal

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

Today

- Memory layout
- **Demo: Using gdb for binary forensics**

gdb cheat sheet

- info registers
- info proc mappings
- b <function>
- nexti
- continue
- bt: print backtrace
- disass <function>
- x/4xb <address> : print 4 bytes starting at address in hex
- x/4i <address>: print 4 instructions starting at address
- p/x $rax

(gdb) help x