Machine-Level Programming V: Unions and Memory layout

Slides adapted from Bryant and O'Hallaron

FAQ

- Call conventions recap
  - The first six integer or pointer argument are stored in RDI, RSI, RDX, RCX, R8, R9.
  - The return value is stored in RAX.
- Why there are no memory load/store instructions when accessing some variables?
  - Compiler code generation
  - Compiler optimizations
- RBP and RSP
  - RBP points to the current stack frame.
  - RSP points to the top of the stack.
  - Stack frame is not used when doing -O2 (and higher).

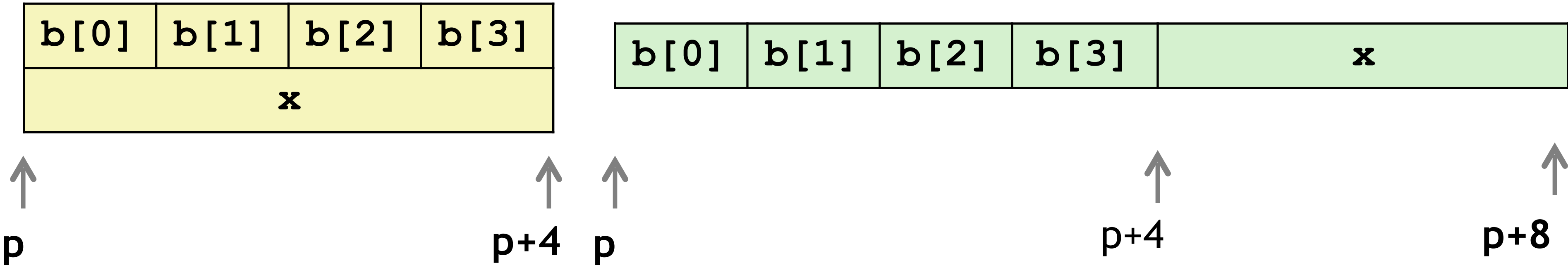
Today

- Structs vs. Unions
- Memory layout

C union vs. struct

```
union my_union {
  unsigned char b[4];
  int x;
}
union my_union U;
```

```
struct my_struct {
  unsigned char bytes[4];
  int x;
};
struct my_struct S;
```



- **All members in a union start at the same location**
  - i.e. only one member can contain a value at a given time

C union vs. struct

```
union my_union {
  unsigned char b[4];
  int x;
}
union my_union U;
```

```
struct my_struct {
  unsigned char b[4];
  int x;
};
struct my_struct S;
```

```
sz = sizeof(U); //sz??
U.b[3] = 0;
U.x = 0x01020304;
U.b[3] = ??
```

```
sz = sizeof(S); //sz??
S.b[3] = 0;
S.x = 0x01020304;
S.b[3] = ??
```

Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```

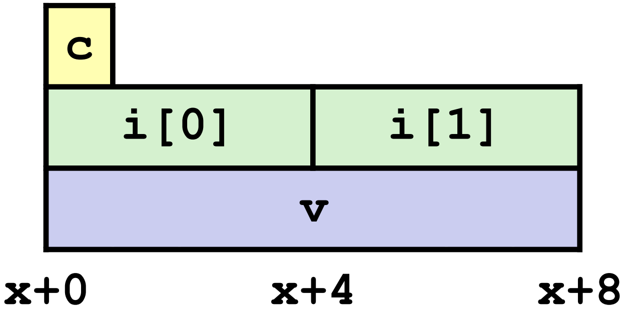
```
bit_float_t x;
x.f = 1.1;
//what is the value of x.u?
```

A. 1
B. 0x03f8cccd
C. 0xf3ebdddc
D. 0x03000000
E. 0x00000001

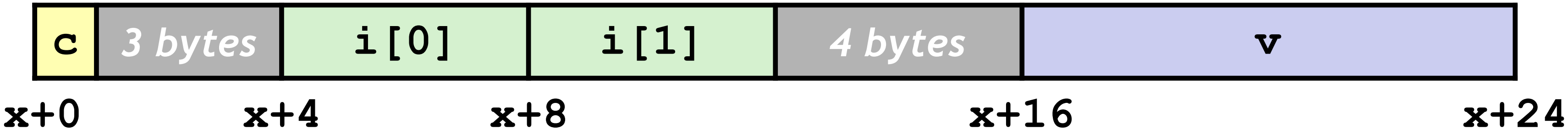
Union Allocation

- Allocate according to largest element

```
union U1 {
  char c;
  int i[2];
  double v;
} u;
```



```
struct S1 {
  char c;
  int i[2];
  double v;
} s;
```



Unions in assembly

```
typedef union {
      char c;
      int i[2];
      double v;
 } u_t;

int
get_member(u_t *u,
           int first) {
   if (first) {
      return U->c;
   }else{
      return U->i[1];
   }
}
```

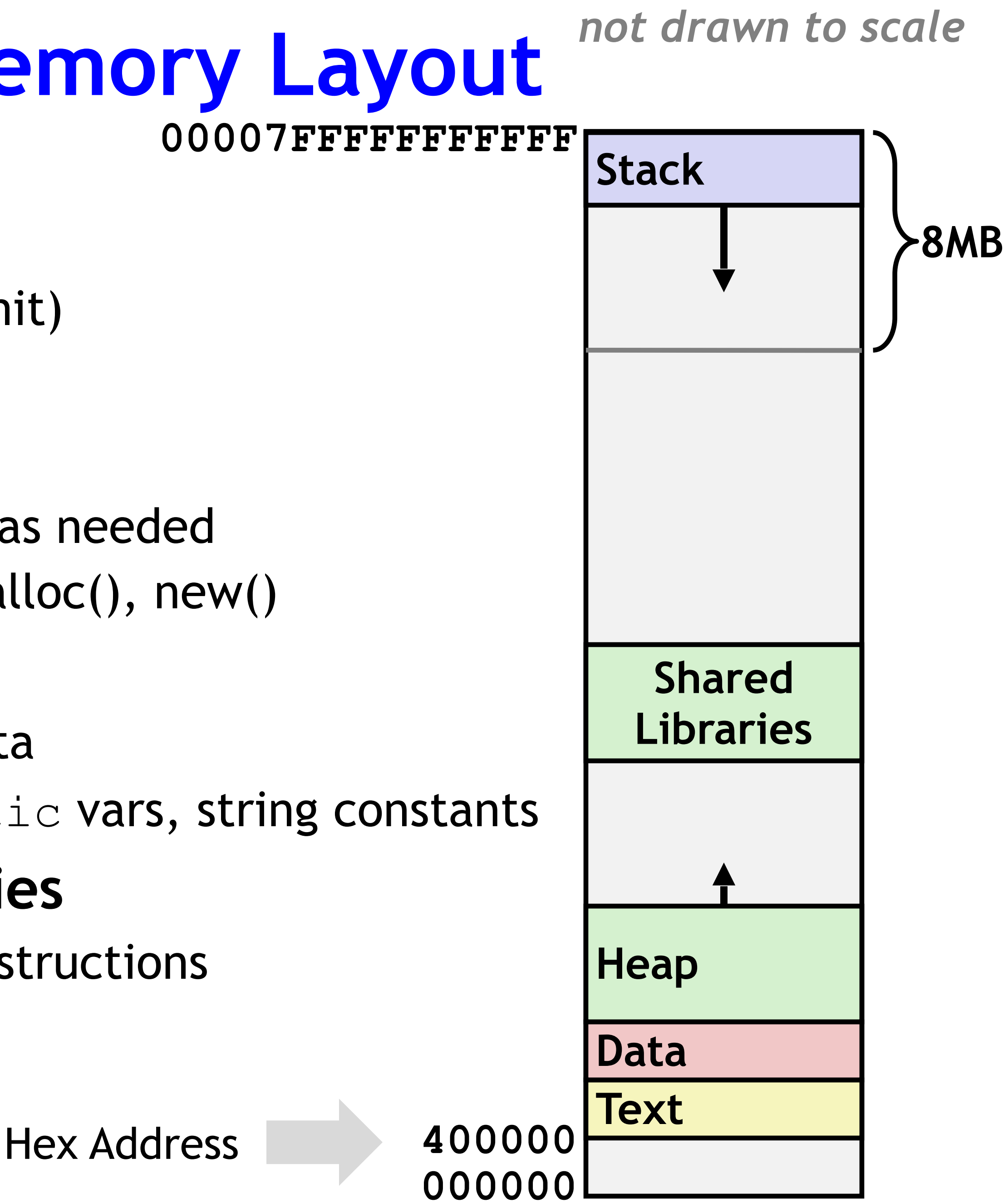
```
    testl %esi, %esi
    je .L2
    movsbl (%rdi), %eax
    ret
.L2:
    movl 4(%rdi), %eax
    ret
```

Today

- Structs vs. Unions
- Memory layout

x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only



Memory Allocation Example

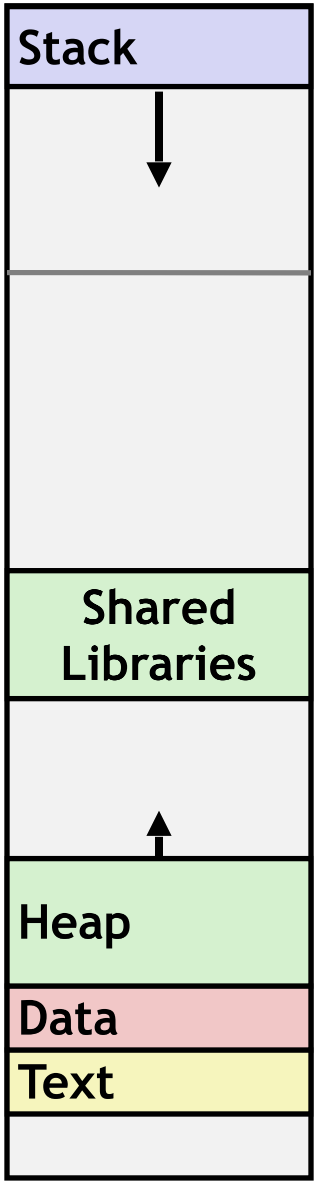
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

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

not drawn to scale

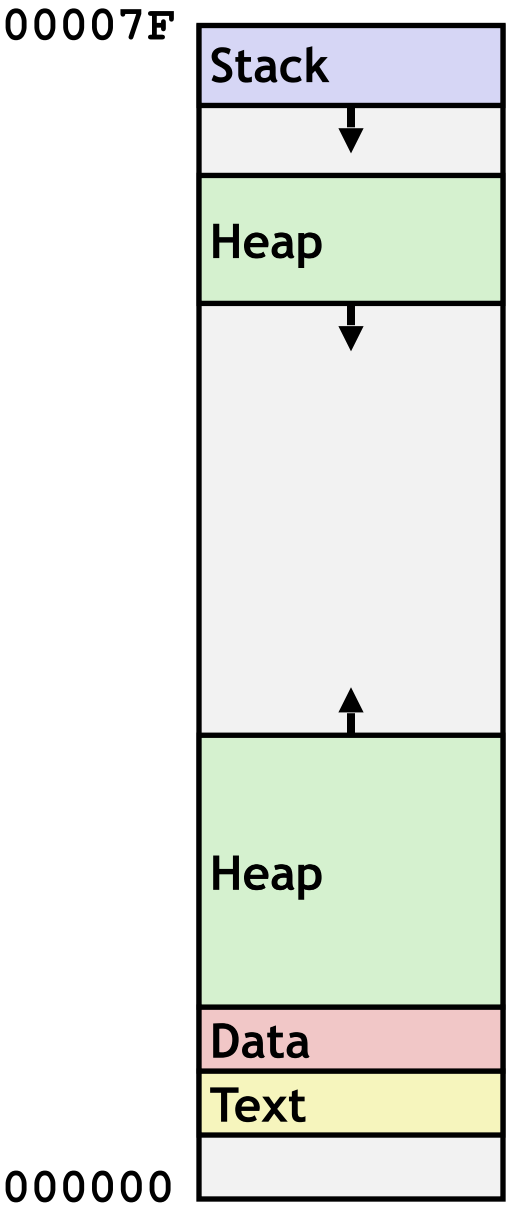


Where does everything go?

x86-64 Example Addresses

address range ~2^{47}

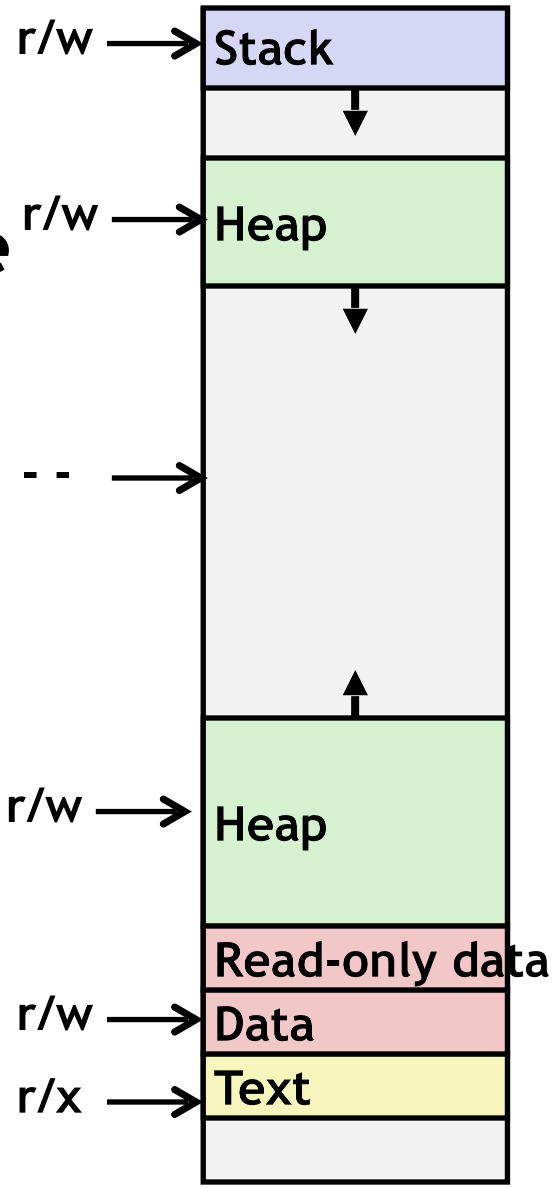
`local`	`0x00007ffe4d3be87c`
`p1`	`0x00007f7262a1e010`
`p3`	`0x00007f7162a1d010`
`p4`	`0x000000008359d120`
`p2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`



not drawn to scale

Segmentation Fault

- Each memory segment can be readable, executable, writable (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

```
char str1[100] = ”hello world1”;

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

Not all Bad Memory Access lead to immediate segmentation

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

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

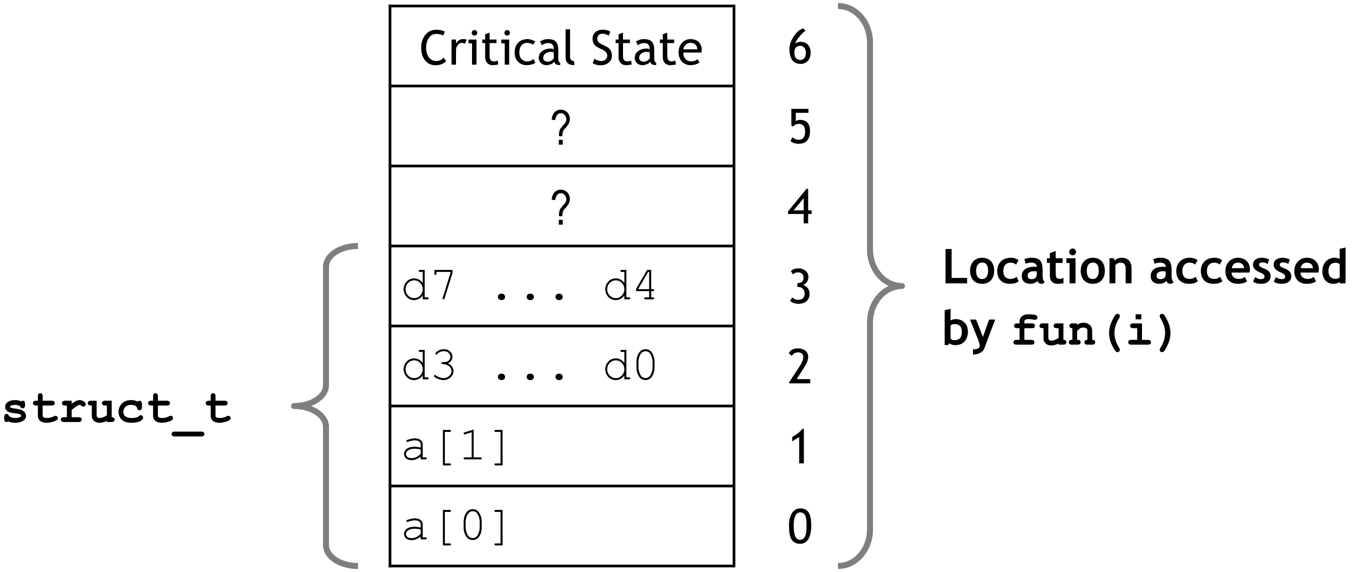
```
fun(0)  ➙    3.14
fun(1)  ➙    3.14
fun(2)  ➙    3.1399998664856
fun(3)  ➙    2.00000061035156
fun(4)  ➙    3.14
fun(6)  ➙    Segmentation fault
```

- 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
```



	Critical State	6
	?	5
	?	4
	d7 ... d4	3
	d3 ... d0	2
struct_t	a[1]	1
	a[0]	0

Location accessed by `fun(i)`

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