CSC2/452 Computer Organization
Pointers and Arrays

Sreepathi Pai

URCS

October 12, 2022

Outline

Outline

Assignments and Homeworks

- Assignment #2, Part II will be released soon (probably Monday)
- Reading for this week online.
  - AddressSanitizer

Outline

Previously: Functions in x86/x64 Assembly Language

- Called using `CALL` instruction
- Return to caller using `RET` instruction
- For each function instance, a stack frame is created
  - Contains arguments to functions (when arguments passed on stack)
  - Contains return address
  - Caller's saved registers (e.g. `%ebp`, `%rbp`)
  - Stores function's local variables
- Stack frame created by caller
  - Can be destroyed by caller or callee (depends on calling convention)

What happens when we run this code?

```
#include <stdio.h>

int sum(unsigned int n) {
  printf("%u\n", n);
  return sum(n - 1);
}

int main(void) {
  sum(100);
}
```

What happens when we run this code?

```
#include <stdio.h>

int sum(unsigned int n) {
  printf("%u\n", n);
  return sum(n - 1);
}

int main(void) {
  sum(100);
}
```

Output: (results may vary on your system)

```
 ...
4294705546
4294705545
Segmentation fault (core dumped)
```

- Each call of `sum` needs a stack frame
- Eventually runs out of memory on the stack: *stack overflow*
  - Manifests in C programs on Linux as a segmentation fault

Previously: Addresses

```
greeting:
    .string "hello world\n"

 ...
```

Which of the following instruction sequences will load the address of *greeting* into %rbx?

- `movq greeting, %rbx`
- `leaq greeting, %rbx`
- `movq (greeting), %rbx`

Loading data using indirect addresses

```
greeting:
    .string "hello world\n"

 ...

   leaq greeting, %rbx
```

Now, which instruction should follow `leaq` to load the first byte of greeting into `%al`?

- `movb (greeting, 0), %al`
- `movb 0(greeting), %al`
- `movb 0(%rbx), %al`
- `movl $0, %rsi` followed by `movb (%rbx, %rsi, 1), %al`

Re-summarizing Addresses

- Labels are addresses
  - Addresses are memory
  - All storage in memory has an address
  - All variables in memory have an address
- On x86-64, we can load an address into a register using the `leaq` instruction
- On x86-64, we can load data using the `mov` instruction
  - Must also indicate size of data

Outline

Local variables

```
int main(void) {
    int s = 0;

    ...
}
```

- We know (from last class) that `s` is on the stack
  - Its address is `-4(%rbp)`

The address-of operator (&)

- The unary C address-of operator & obtains the address of an *lvalue*.
- It is equivalent to the `lea` instruction
  - But only works on *lvalues* (not C labels)
- A C lvalue is anything that can appear on the left-hand side of an assignment

Lvalues Quiz

```
int sum(int x, int y) {
   int s = 0;

   s = x + y;

   return s;
}
```

Which of the following are lvalues?

- `0`
- `s`
- `x` and `y`
- `sum`

Using the address-of operator

```
#include <stdio.h>

int main(void) {
    int s = 0;

    printf("address-of s = %p\n", &s);

    return 0;
}
```

The output of this code is:

```
address-of s = 0x7ffc06d15e04
```

- The %p conversion specifier for `printf` indicates the value to be printed is a pointer
- Addresses on 64-bit x86 systems currently use only 48-bits
  - Note: addresses will change across machines/runs, etc.

What happened under the hood?

```
000000000000064a <main>:
 64a:   push   %rbp
 64b:   mov    %rsp,%rbp
 64e:   sub    $0x10,%rsp
 652:   movl   $0x0,-0x4(%rbp)      # s = 0
 659:   lea    -0x4(%rbp),%rax      # %rax = &s
 65d:   mov    %rax,%rsi            # %rsi = %rax, 2nd parameter
 660:   lea    0x9d(%rip),%rdi      # %rdi = &format string, 1st para
 667:   mov    $0x0,%eax            # number of vector registers
 66c:   callq  520 <printf@plt>
 671:   mov    $0x0,%eax
 676:   leaveq
 677:   retq
```

- Note that `0x9d(%rip)` is the address of the format string `"address-of s =..."`

Doing arithmetic on addresses

```
#include <stdio.h>

int main(void) {
    int s = 0;

    printf("address-of s = %p %p\n", &s, &s + 1);

    return 0;
}
```

Why does this result in the following output?

```
address-of s = 0x7ffdf7b9db5c 0x7ffdf7b9db60
```

- What is `0x7ffdf7b9db60 - 0x7ffdf7b9db5c`?

Relevant assembly

```
652:       movl   $0x0,-0x4(%rbp)
659:       lea    -0x4(%rbp),%rax    # %rax = &s
65d:       add    $0x4,%rax          # %rax = %rax + 4
661:       lea    -0x4(%rbp),%rcx    # %rcx = &s
665:       mov    %rax,%rdx          # %rdx = %rax, third parameter
668:       mov    %rcx,%rsi          # %rsi = %rcx, second paramter
66b:       lea    0xa2(%rip),%rdi    # ...
672:       mov    $0x0,%eax
677:       callq  520 <printf@plt>
```

More on the address-of operator

- The address-of operator does *not* return a 64-bit integer
- It returns a C *pointer*, C's abstraction of a machine address
- Ordinarily, each pointer "knows" the size of data it points to
  - Recall, to load data from a address, you have to know the size of the data!
- Therefore, `&s` is a pointer value to an integer (since `s` is an integer)
- Arithmetic on pointer values increases them by size of data pointed to
  - So `&s + 1` actually increases `&s` by `1*4` (on systems with a 32-bit `int`)
  - This is very useful, as we shall see later

Dereferencing Pointers: On the RHS

```
#include <stdio.h>

int main(void) {
    int s = 1024;

    printf("s = %d, *(&s) = %d\n", s, *(&s));

    return 0;
}
```

Output:

```
s is 1024, *&s = 1024
```

- On the right-hand side, the * *unary* operator applied to an pointer reads the data at that address
  - Called a dereference operator or an indirection operator

Dereferencing Pointers: On the LHS

```
#include <stdio.h>

int main(void) {
    int s = 1024;

    *(&s) = 1234;

    printf("s = %d, *(&s) = %d\n", s, *(&s));

    return 0;
}
```

Output:

```
s is 1234, *&s = 1234
```

- On the left-hand side, the * *unary* operator applied to an pointer writes the data to the address

Storing Pointer Values

- A pointer value is stored in a *pointer variable*

```
#include <stdio.h>

int main(void) {
    int s = 0;
    int *ps;

    ps = &s;

    printf("address-of s = %p %p\n", &s, ps);

    return 0;
}
```

Output:

```
address-of s = 0x7ffde5cad8d4 0x7ffde5cad8d4
```

Pointer declarations

```
#include <stdio.h>

int main(void) {
    int s = 0;
    int *ps;

    ps = &s;

    printf("address-of s = %p %p\n", &s, ps);

    return 0;
}
```

- `int *x` is read as:
  - "`x` is an `int` pointer"
  - "`x` is a pointer to `int`"
- The name of the pointer is `ps`
- We are storing `&s`, the address of `int s` into it

Pointer variables behaviour

```
#include <stdio.h>

int main(void) {
    int s = 1024;
    int *ps;

    ps = &s;
    *ps = 1234;

    printf("s = %d, *ps = %d\n", s, *ps);

    return 0;
}
```

Output?

- `s = 1024, *ps = 1024`
- `s = 1234, *ps = 1234`
- `s = 1024, *ps = 1234`
- `s = 1234, *ps = 1024`

Diving into the disassembly

```
000000000000064a <main>:
 64a:       push   %rbp
 64b:       mov    %rsp,%rbp
 64e:       sub    $0x10,%rsp
 652:       movl   $0x400,-0xc(%rbp)    # s = 1024
 659:       lea    -0xc(%rbp),%rax      # %rax = &s
 65d:       mov    %rax,-0x8(%rbp)      # ps = %rax
 661:       mov    -0x8(%rbp),%rax      # %rax = ps
 665:       movl   $0x4d2,(%rax)        # *ps = 1234
 66b:       mov    -0x8(%rbp),%rax      # %rax = ps
 66f:       mov    (%rax),%edx          # %edx = *ps
 671:       mov    -0xc(%rbp),%eax      # %eax = s
 674:       mov    %eax,%esi            # ...
 676:       lea    0x97(%rip),%rdi
 67d:       mov    $0x0,%eax
 682:       callq  520 <printf@plt>
```

The * operator maps to indirect addressing

- The deference or indirect operator maps to the indirect addressing mode
- First load the pointer into a register
  - `mov -0x8(%rbp),%rax` (this is loading `ps` into `%rax`)
- Then, access the data for reading:
  - `mov (%rax),%edx`, (this is the equivalent of `... = *ps`)
- Or for writing:
  - `movl $0x4d2,(%rax)`, (this is the equivalent of `*ps = ...`)

Summary of C pointers

- C pointers hold addresses to particular data types
  - Declared using a `*` on variable name: `int *p`
- Addresses are obtained using the `&` operator on lvalues
  - `ps = &v`
  - If reading or writing `ps`, the indirection operator `*` must NOT be used
  - `ps` is said to *alias* `v`
- To read and write data, you must use the indirection operator
  - `*ps = 1`, sets the value of `v` to 1
  - `s = *ps`, sets the value of `s` to the value of `v`
  - The indirection operator does not change `ps`

Swap

```
#include <stdio.h>

void swap(int x, int y) {
  int tmp = x;

  x = y;
  y = tmp;

  printf("x=%d y=%d\n", x, y);
}

int main(void) {
  int a = 1;
  int b = 2;

  swap(a, b);

  printf("a=%d b=%d\n", a, b);
}
```

Output:

```
x=2 y=1
a=1 b=2
```

Swap stack frame

Just before returning to main

```
    swap                                          main
    tmp    ebp    ret    x      y                 b      a
    [1   ][    ][    ][2   ][1   ] .... [2   ][1   ]...
```

What does `main` do immediately after call returns?

Stack frame for swap destroyed by main after return

After `main` has destroyed stack frame:

```
                    main
                    b      a
.... [2    ][1    ]...
```

- C passes arguments by value, by copying them
  - Note, when passing arguments in registers, the copies are in registers, not on the stack frame
  - But in both cases, copies are made
- But we want to modify original `a` and `b`
  - We can use pointers for this!

Swap reimplemented using pointers

```
#include <stdio.h>

void swap(int *x, int *y) {
  int tmp = *x;

  *x = *y;
  *y = tmp;

  printf("x=%d y=%d\n", *x, *y);
}

int main(void) {
  int a = 1;
  int b = 2;

  swap(&a, &b);

  printf("a=%d b=%d\n", a, b);
}
```

Stack frame for swap just before returning

```
swap                                        main
tmp    ebp    ret    x      y               b      a
[1    ][     ][     ][&a   ][&b   ] ... [1    ][2    ]...
```

Note:

- `[&v ]` indicates the stack location contains the address of `v`
- Values `a` and `b` have been modified indirectly through pointers

Pointer arithmetic

```
#include <stdio.h>

int main(void) {
    int s = 1024;
    int *ps;

    ps = &s + 1;
    *ps = 1234;

    printf("s = %d, *ps = %d\n", s, *ps);

    return 0;
}
```

What will the output be?

Result

```
Segmentation fault (core dumped)
```

Trying to reason what happened

```
main
s      ebp    ret
[1024][    ][    ]
```

- `s` is a scalar variable, it can only hold one value
  - That value is stored at address `&s`
- `(&s + 1)` is an address "outside" of `s`
  - `&s + 1` *may* be valid machine-level address
  - But we have no idea what it means in the context of our C program, so dereferencing it (i.e., `*(&s + 1)`) is undefined
- In this case, maybe the previous value of %ebp was at that location and was overwritten
  - There are ways to verify this, using a machine-level debugger
- However, in C, reading/writing to an address beyond the extent of a variable is *undefined*
  - Anything can happen, including a segfault

Segmentation Faults

- A segmentation fault (or a general protection fault on x86) is almost always an attempt to dereference an invalid address
- The processor and OS work together to track all reads/writes and raise a fault if your program attempts to read/write an invalid address
  - Or if you don't have permission to write or read an address
- For many reasons (which we will learn about later in the course), a fault may not be raised in all invalid situations
- Can make tracking down these bugs difficult

Pointer Arithmetic: What is it good for?

- For scalar variables, it makes no sense to do pointer arithmetic
- It is provided in C for *arrays*

Outline

Arrays in C

```
#include <stdio.h>

int main(void) {
  int a[10];
  int i;

  for(i = 0; i < 10; i++) {
    a[i] = i;
  }
}
```

- The array a is allocated on the stack (it is a local)
  - Cannot be very big
- Each element of a is placed next to each other in memory
- C does *not* record the size of the array

Arrays == Pointers in C

- An array is a pointer in C
  - It is "syntactic sugar" over pointers
- The array name `a` is a pointer to the first element of the array
  - `&(a[0])` is just `a`
  - `&(a[4])` is just `a + 4`
  - `&(a[i])` is just `a + i`
- The `[]` operator is translated internally to pointer notation
  - `a[i] = i` is translated to `*(a + i) = i`
  - `x = a[i]` is translated to `x = *(a + i)`

Equivalence

```
#include <stdio.h>

int main(void) {
  int a[10];
  int *b;
  int i;

  b = a;

  for(i = 0; i < 10; i++) {
    b[i] = i;
  }
}
```

- a is notionally an array, b is notionally a pointer
- However, b is made to point to a
- And a is updated indirectly through b

C pointers and arrays

- C cannot distinguish between pointers and arrays
- It assumes every pointer is an array
  - Scalar variables can be viewed as arrays of length 1
- It does not record size of arrays either
  - Unlike nearly every other language that came after
- Therefore, it cannot tell you if you are accessing an out-of-bounds address
  - Out-of-bounds: beyond the bounds of an array, e.g. `a[10]`

Outline

The NULL pointer value

```
int *p = NULL;
```

- The `NULL` value is a special value assigned to a pointer
- It indicates (by convention) that it is not pointing anywhere
- Convention also requires that the machine trap (i.e. fault) when you try to dereference a NULL pointer
  - `*p = 1` will always cause a segmentation fault

The `void` pointer type

```
int v = 10;
void *x = &v;
```

- The `void *` type can only store addresses
  - Can be assigned from any other pointer type
- Since there is no associated size information with `void`, it cannot:
  - do pointer arithmetic
  - dereference addresses
- To do this, you must cast the pointer to a non-`void` type.

```
printf("%d\n", *x); // fails to compile

printf("%d\n", *((int *) x)); // succeeds, and prints value of v
```

Type Punning

```
float pi = 3.14;
uint32_t *px;

float *ppi = &pi;
void *tmp = ppi;

// attempt to read the single-precision float value (32-bits) as a
// unsigned 32-bit integer, by casting a void pointer

px = (uint32_t *) tmp;
printf("%u\n", *px);
```

- DO NOT DO THIS, IT IS UNDEFINED BEHAVIOUR
  - Even if your textbook says its okay (they're wrong)
- Specifically, C forbids you from having pointers of different types pointing to the same address
  - C assumes strict aliasing
- Your program will break at high levels of optimization

Integers are NOT pointers

```
int x = 20;
int *px = &x;

uint64_t ip;

ip = px;
```

- Assigning pointers to integers (and vice versa) is *implementation-defined*
  - I.e., results are not guaranteed to be the same everywhere
- If you must really do this, use C99's `uintptr_t`
  - Most useful when loading addresses directly, instead of using `&`
  - Almost always in very low-level code (e.g. in OS or driver)
- Read: CERT Secure C coding standards for more details

Dangling Pointers

```
int *mul2(int a) {
   int r = a * 2;

   return &r;
}
```

- The `mul` function returns a pointer
- Unfortunately, it returns a pointer to a local variable
  - That local variable no longer exists after `mul` returns!
- This is called a dangling pointer
  - Points to data that longer exists
  - May lead to segfaults when dereferenced