Introduction to Computer Organization

DIS 1A – Week 4

Slides modified from Eric Kim

Agenda

- Matrix, Struct, Union, Alignment, C function pointers
- Security
- Floating Point
- GDB
- Lab 2
- Midterm

Matrices in x86 (nested arrays)

- int A[2][3]
  - A matrix with 2 rows, 3 columns
- C style: Row-Major order
  - Row-Major: In memory, matrix is laid out row-by-row
  - Column-Major: Matrix is laid out column-by-column

1	2	3
4	5	6

```
[ 1 2 3 4 5 6 ]
```

A in **Row-Major** format
(this is what C does!)

```
[ 1 4 2 5 3 6]
```

A in **Column-Major** format

Matrix: Row-Major

- How to access element at `int A[i][j]`?
    - Recall: A is a 2x3 matrix.
- In C, using pointer arithmetic:

```
int val = *(A + 3*i + j)
```

Matrix: Row-Major

- How to access element at `int A[i][j]`?
  - Recall: A is a 2x3 matrix.
- In x86 (`%eax = &A, %ebx = i, %ecx = j`)

```
leal (%ebx,%ebx,2), %ebx     # 3*i
imull $4, %ebx              # 4*3*i (int is 4 bytes)
addl %ebx, %eax              # A + 4*3*i
leal (%eax, %ecx, 4) %eax    # A + 4*3*i + 4*j
movl (%eax) %eax
```

Recall: C Pointers

- Why does x86 multiply by 4, but C code does not?
    - C pointers remember data type, ie how large each element is!

```
int *p;
*(p+1);  // This goes 4 bytes forward!
```

- To x86, bytes are bytes. Compiler must keep track of data sizes.

Practice Problem 3.37

Consider the following source code, where M and N are constants declared with `#define`:

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j) {
    return mat1[i][j] + mat2[j][i];
}
```

In compiling this program, GCC generates the following assembly code:

```
  i at %ebp+8, j at %ebp+12
    movl    8(%ebp), %ecx
    movl    12(%ebp), %edx
    leal    0(,%ecx,8), %eax
    subl    %ecx, %eax
    addl    %edx, %eax
    leal    (%edx,%edx,4), %edx
    addl    %ecx, %edx
    movl    mat1(,%eax,4), %eax
    addl    mat2(,%edx,4), %eax
```

Use your reverse engineering skills to determine the values of M and N based on this assembly code.

Solution to Problem 3.37 (page 236)

This problem requires you to work through the scaling operations to determine the address computations, and to apply Equation 3.1 for row-major indexing. The first step is to annotate the assembly code to determine how the address references are computed:

```
movl    8(%ebp), %ecx              Get i
movl    12(%ebp), %edx             Get j
leal    0(,%ecx,8), %eax           8*i
subl    %ecx, %eax                 8*i-i = 7*i
addl    %edx, %eax                 7*i+j
leal    (%edx,%edx,4), %edx        5*j
addl    %ecx, %edx                 5*j+i
movl    mat1(,%eax,4), %eax        mat1[7*i+j]
addl    mat2(,%edx,4), %eax        mat2[5*j+i]
```

We can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this, we can determine that `mat1` has 7 columns, while `mat2` has 5, giving $M = 5$ and $N = 7$.

Structs

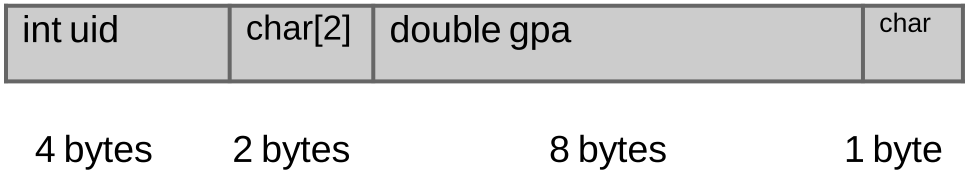
- How are structs laid out in memory?
- In x86, how to access struct fields?

```
struct student_record {
    int uid;                // 000000404
    char initials[2];       // EK
    double gpa;             // 0.42 (Ouch)
    char is_graduated;      // 0: No 1: Yes
}
```

Structs

- Fields are placed in memory contiguously
  - Always contiguous! C is not allowed to reorder struct fields.

```
struct student_record {
    int uid;
    char initials[2];
    double gpa;
    char is_graduated;
}
```



We'll talk about struct alignment in a bit

Unions

- Hacky way to save space in structs (IMO)

```
union node {
    struct {
        union node* left;
        union node* right;
    } internal;
    double data;
}
```

Suppose node represents a binary tree, with the following structure:
-If the node is a leaf node, then it stores a numerical data.
-Otherwise, it stores two pointers to its left/right children.

Note: A node can't have both data *and* pointers to children! In other words, only leaf nodes store data.

Unions

- Two equivalent-ish ways

```
union node {
    struct {
        union node* left;
        union node* right;
    } internal;
    double data;
}
```

```
struct node {
    struct {
        struct node* left;
        struct node* right;
    } internal;
    double data;
}
```

Difference: `union node` uses 8 bytes, but `struct node` uses 16 bytes!

Unions

- **Warning**: Consider the following code.

```
union node my_node = get_some_node();
```

Is `my_node` an internal node? Or is it a leaf node?

We don't know!

Need to use context to find out which "flavor" of node `my_node` is.

Unions

- **Warning**: Consider the following code.

```
union node my_node =
get_some_node();  printf("Value:
%f\n", my_node.data);
```

Here, `my_node` turns out to be the leaf-node variant of `union node`.

Unions

- **Warning**: Consider the following code.

```
union node my_node =
get_some_node();
print_tree(my_node.internal.left);
```

Here, `my_node` turns out to be the internal-node variant of `union node`.

Unions

- **Tip**: When reverse engineering x86 code for union code, you'll need to figure out the correct union "flavor" of each variable..

Review: Little-endian vs Big-endian

```
long long bit2ll(unsigned int word0, unsigned int word1) {
  union {
    long long d;
    unsigned u[2];
  } temp;
  temp.u[0] = word0;
  temp.u[1] = word1;
  return temp.d;
}
```

unsigned int x = 0x0ABCDEF0
unsigned int y = 0xFACEB00F
long long val = bit2ll(x,y);

What is val if:
(1) The machine is little-endian?
(2) The machine is big-endian?
For both cases, how is val laid out in memory?

Review: Little-endian vs Big-endian

```
long long bit2ll(unsigned int word0, unsigned int word1) {
  union {
    long long d;
    unsigned u[2];
  } temp;
  temp.u[0] = word0;
  temp.u[1] = word1;
  return temp.d;
}
```

unsigned int x = 0x0ABCDEF0
unsigned int y = 0xFACEB00F
long long val = bit2ll(x,y);

Answer:

Little-endian: `0xFACEB00F 0ABCDEF0`

Big-endian: `0x0ABCDEF0 FACEB00F`

Review: Little-endian vs Big-endian

```
long long bit2ll(unsigned int word0, unsigned int word1) {
  union {
    long long d;
    unsigned u[2];
  } temp;
  temp.u[0] = word0;
  temp.u[1] = word1;
  return temp.d;
}
```

unsigned int x = 0x0ABCDEF0
unsigned int y = 0xFACEB00F
long long val = bit2ll(x,y);

Addresses grow left->right

Little-endian:

0xf0	0xde	0xbc	0x0a	0x0f	0xb0	0xce	0xfa

How is val laid out in memory?

Big-endian:

0x0a	0xbc	0xde	0xf0	0xfa	0xce	0xb0	0x0f

Alignment

- x86 convention: total stack space used by a function must be a multiple of 16 bytes
- Arch-dependent rules on data-alignment
  - Linux: 2-byte data types (ie short) must have an addr that is a multiple of 2.
    - Larger data types (ie int, double) must have an addr that is a multiple of 4
  - Windows: ANY data type of K bytes must have an addr that is a multiple of K.

Struct alignment

- Might need to add padding in between fields to satisfy alignment
- For struct arrays, might need to add padding at *end* of each struct to satisfy alignment

C Pointers

- One warning: casting priority
  - Suppose p is a pointer to a char

What is the memory offset of the following expressions?

`(int*) p+7`	4*7 = 28 bytes (p is first cast as int*, then incremented)
`(int*) (p+7)`	7 bytes (p is still treated as a char* ptr)

C Function Pointers

```c
int (*f)(int,char)
```

Means: f is a pointer to a function that takes two arguments (int, char), and returns an int.

```c
int (*f)(int, char) = &my_fn;
```

What gets printed out?

```c
int add_two(x) {
  return x + 2;
}
int add_three(x) {
  return x + 3;
}
int compose(int val, int (*f)(int), int(*g)(int)) {
  return f(g(val));
}
int main(int argc, char** argv) {
  int (*fnptr1)(int) = &add_two;
  int (*fnptr2)(int) = &add_three;
  int s = compose(42, fnptr1,fnptr2);
  printf("s is: %d\n", s);
  return 1;
}
```

Warning: Function Ptr vs Prototype

```
(int*) f(char*,int)
```

This is a **function prototype**, declaring a function f that takes 2 args (char*,int), and returns an int*.

```
int* (*f)(char*,int)
```

f is a **pointer** to a function that takes two args (char*,int), and returns an int*.

Bounds Checking

Scenario: A function declares a local char buffer with a **fixed** size, and allows user to input characters from the keyboard into the buffer.

BUT! The function doesn't check to see if the user typed past the end of the buffer.

Bounds Checking

"Best" case: Program crashes

What scenario could result in a crash?

Worst case: Attacker gains control of your machine!

Stack Smashing

When a function writes past the end of a buffer (ie array), this is called a **buffer overflow**.

In the Computer Security community, this is also known as **Stack Smashing**, especially when a buffer overflow is used for malicious purposes.

Stack Smashing (Reading)

If you're curious, Google "Stack Smashing for Fun and Profit"

Purely optional, ie if you're bored and somehow have free time :P

Stack Smashing

How to exploit a buffer overflow?

Recall: Goals of attacker are typically:

1. Read sensitive data (passwords, etc)
2. Disrupt service (ie DDoS)
3. Execute code on machine

Stack Smashing

Trick 1: Overwrite caller's saved eip on stack, and write the address of code that *we* want to execute!

Super Neat Trick: Write our malicious code into the array we are overflowing, then set caller's saved eip to start of our code!

Stack Smash Defenses

- Only allow OS to execute code from read-only section of memory
  - Called "Data Execution Prevention" (DEP)
  - Known workarounds
    - store code on heap
    - Call syscalls to disable DEP

Address Space Layout Randomization (ASLR)

- Several exploits require knowing the precise address of locations on the stack (ie the address of the caller's saved eip).
- Defense: randomize the stack
  - Start stack at some random offset
  - Defeats attacks that assume a specific memory layout

NOP-Sleds

- Scenario: we are injecting malicious code into a buffer.
  - Goal: Need to put the address of the first malicious instruction into the caller's saved eip
  - With ASLR, this is much more difficult. We could do a brute-force search, but search space is large.

...	...	0x08	0x02	0xf3	0xff	0x33	0x74	0x11	0x00

Start of my malicious code.

How to guess this address?

NOP-Sleds

- Workaround:

...	...	0x08	0x02	0xf3	0xff	0x33	0x74	0x11	0x00

Instead of having to guess this address
(hard)

NOP-Sleds

- Workaround:

"Payload" Code

0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x08	0x02	0xf3	0xff

Instead, guess *any* of these addresses!

If we hit any of the NO-OP instructions, then the processor will "slide" from left to right until it reaches our malicious code.

Canaries

- Idea: Instead of trying to **stop** buffer overflows, instead try to **detect** them.
  - If detect, then halt the program.

Canaries

- Compiler adds special value (canary) to stack at the end of a local buffer.
- When function is returning, check canary value.
  - If the canary value changed, then a buffer overflow must have happened.
    - Issue a "Stack Overflow Exception"
  - Else, return to caller as normal

Canaries

- By halting before returning, we prevent the eip being set to an address of the attacker's choosing.

Computer Security

- Studying ways to attack vulnerable systems (and defend against malicious attackers)
  - Web security is ***hugely*** important these days
    - Banks, customer data, SSN's, etc.
- Very active field of research
  - Web security, mobile security, network security, ...

Computer Security (cont.)

- Cryptography
  - Using math to design robust, secure cryptosystems
  - Ie "one-way" functions: functions that are simple to evaluate in one direction, but computationally infeasible to invert.
  - Involves a crazy amount of number theory
    - Ie properties of prime numbers

Floating Point

- So far, have worked with integer data types
  - signed: two's complement
  - unsigned
- Integers: 0, 1, 42, -101, 9001

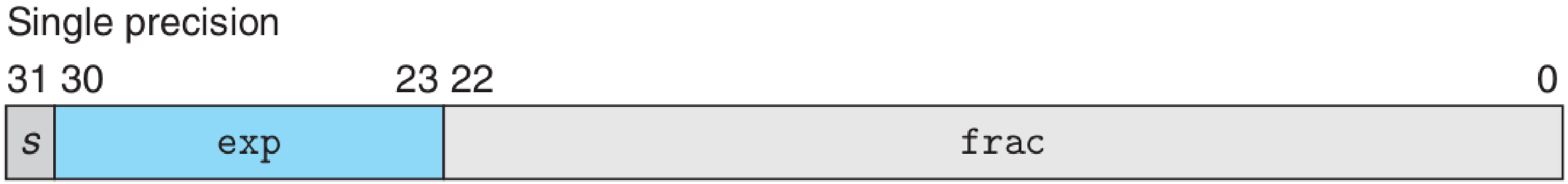
How to represent a non-integer, like 0.5?

Answer: Floating Point Representation!

Floating Point (IEEE 754)

- Goal: Represent **rational** numbers with a **fixed** number of bits
  - float: 32 bits "single" precision
  - double: 64 bits "double" precision

Floating Point:



Sign bit — "Exponent" Field — "Fraction" Field

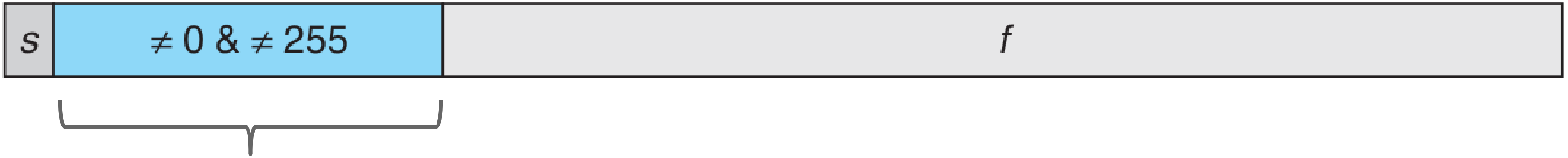
1 bit — 8 bits — 23 bits

Two Types of FP

- There are two different "types" of a floating point number
- Case 1: "Normalized"
  - Common case. Represent large and moderately-small values.
- Case 2: "Denormalized"
  - Represent very small values (close to 0).

Case 1: Normalized

1. Normalized



Exp is not all 0's or all 1's

$$V = (-1)^s \times M \times 2^E$$

s = sign bit
M = 1 + f
E = e - Bias

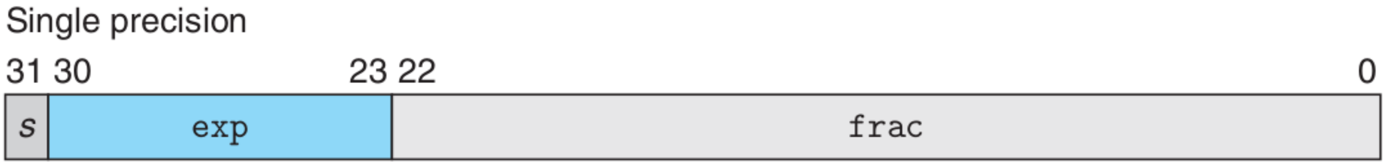
Bias = $2^{k-1} - 1 = \begin{cases} \text{127 for single} \\ \text{1023 for double} \end{cases}$

k is # bits in exp

Case 1: Normalized

Bias = $2^{k-1} - 1 = \begin{cases} \text{127 for single} \\ \text{1023 for double} \end{cases}$

k is # bits in exp




```
0100 0010 0010 1000 0000 0000 0000 0000
```

What is V?

exp = 100 0010 0 = 0x84 = 8*16 + 4 = 132

f = 010 1000 0000 0000 0000 0000

= 0*2^(-1) + 1*2^(-2) + 0*2^(-3) + 1*2^(-4) = 0.3125

V = (-1)^0 * (1 + 0.3125) * 2^(132 - 127) = **42.0**

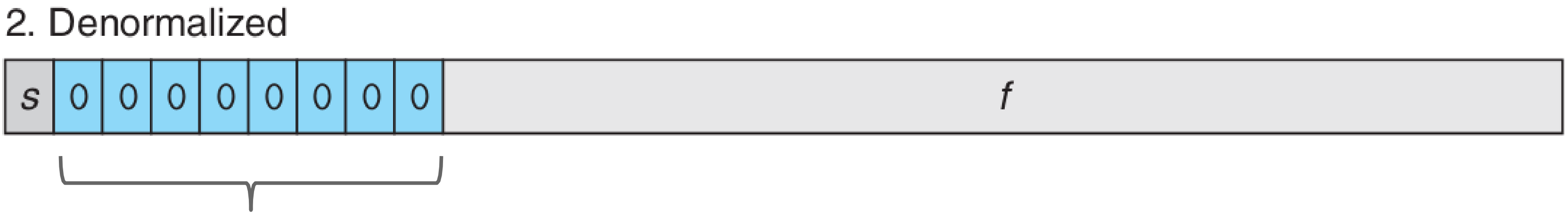
s = sign bit

M = 1 + f

E = e - Bias

$$V = (-1)^s \times M \times 2^E$$

Case 2: Denormalized



Exp is all 0's or all 1's

$$V = (-1)^s \times M \times 2^E$$

s = sign bit
M = f
E = 1 - Bias

Bias = $2^{k-1} - 1$ = $\begin{cases} \text{127 for single} \\ \text{1023 for double} \end{cases}$

k is # bits in exp

Case 2: Denormalized

Bias = $2^{k-1} - 1$ = $\begin{cases} \text{127 for single} \\ \text{1023 for double} \end{cases}$

k is # bits in exp

`1000 0000 0010 1100 0000 0000 0000 0000`

exp = 000 0000 0 = 0

f = 010 1100 0000 0000 0000 0000

What is V?

= 1*2^(-2) + 1*2^(-4) + 1*2^(-5) = 0.34375

V = (-1)^1 * (0.34375) * 2^(1-127)

= **-4.040761830951613e-39**

$$V = (-1)^s \times M \times 2^E$$

s = sign bit

M = f

E = 1 - Bias

Case 3: Special Values

- Represent infinity, NaN via certain bit patterns

3a. Infinity

s	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3b. NaN

s	1	1	1	1	1	1	1	1	$\neq 0$

Note: +Inf and -Inf are different (sign bit).

Case 3: Special Values

- Can represent 0.0 in two ways!
  - All bits 0, and sign bit is 1: -0.0
  - All bits 0, and sign bit is 0: +0.0

Example: Largest/Smallest

- Suppose we use an 8-bit floating-point format. There are 4 exponent bits, and 3 fraction bits.

What is the bias?

```
2^(4-1)-1 = 7
```

What is the smallest/largest positive value?

```
Smallest: 0 0000 001
Largest: 0 1110 111
```

How to represent 1.0?

```
0 0111 000
```

Rounding

- Floating point still can't represent every rational number exactly
    - Why? Finite number of bits (32, 64)
- IEEE standard defines several **rounding modes**

Four Rounding Modes

Mode	\$1.40	\$1.60	\$1.50	\$2.50	\$–1.50
Round-to-even	\$1	\$2	\$2	\$2	\$–2
Round-toward-zero	\$1	\$1	\$1	\$2	\$–1
Round-down	\$1	\$1	\$1	\$2	\$–2
Round-up	\$2	\$2	\$2	\$3	\$–1

Figure 2.36 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

FP Operations

- After every operation on two floating point values, a round is performed.
  - Ex: (f+g) => round(f+g)

FP: Addition

- Addition commutes correctly
  - `(f+g) = (g+f)`
- Addition is generally *not* associative
  - `(3.14 + 1e10)-1e10 = 0.0`
  - `3.14 + (1e10-1e10) = 3.14`

FP: Multiplication

- Generally not associative
  - `(1e20*1e20)*1e-20 = +Inf`
  - `1e20*(1e20*1e-20) = 1e20`
- Does not distribute over addition

Exercise

Practice Problem 2.53

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0:

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

Exercise

We assume that the value `1e400` overflows to infinity.

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

C FP: Casting Rules

- In C, exists float and double data types
  - Can cast to/from float types to integer types.
  - Can lose information due to rounding/truncation!

```
#include <stdio.h>
int main() {
    float v = 42.675;
    int foo = (int) v;
    printf("foo is: %d\n", foo);
    return 0;
}
```

```
$ gcc -o code code.c
$ ./code
foo is: 42
```

Casting Rules Quiz

	Exact conversion?	Can overflow/underflow?
int -> float		
int -> double		
float -> double		
double -> float		
double -> int		
float -> int		

Casting Rules Quiz

	Exact conversion?	**Can overflow?**
int -> float	No! Float can't repr very large ints.	No
int -> double	Yes	No
float -> double	Yes	No
double -> float	No	Yes
double -> int	No: rounded toward zero (1.99 -> 1, -1.99 -> -1)	Yes
float -> int	No: rounded toward zero	Yes

Debugging Process

Reproduce the bug
Simplify program input
Use a debugger to track down the origin of the problem
Fix the problem

GDB – GNU Debugger

Debugger for several languages

- C, C++, Java, Objective-C… more

Allows you to inspect what the program is doing at a certain point during execution

Logical errors and segmentation faults are easier to find with the help of gdb

Using GDB

1. Compile Program

Normally: `$ gcc [flags] <source files> -o <output file>`

Debugging: `$ gcc [other flags]` **`-g`** `<source files> -o <output file>`

enables built-in debugging support

2. Specify Program to Debug

```
$ gdb <executable>
```

or

```
$ gdb
(gdb) file <executable>
```

Using GDB

3. Run Program

`(gdb) run` or

`(gdb) run [arguments]`

4. In GDB Interactive Shell

Tab to Autocomplete, up-down arrows to recall history

`help [command]` to get more info about a command

5. Exit the gdb Debugger

`(gdb) quit`

Setting Breakpoints

Breakpoints

used to stop the running program at a specific point

If the program reaches that location when running, it will pause and prompt you for another command

Example:

`(gdb) break file1.c:6`

Program will pause when it reaches line 6 of file1.c

`(gdb) break my_function`

Program will pause at the first line of `my_function` every time it is called

(gdb) break [*position*] if *expression*

Program will pause at specified position only when the expression evaluates to true

Breakpoints

Setting a breakpoint and running the program will stop program where you tell it to

You can set as many breakpoints as you want

`(gdb) info breakpoints|break|br|b` shows a list of all breakpoints

Deleting, Disabling and Ignoring BPs

`(gdb) delete [bp_number | range]`

Deletes the specified breakpoint or range of breakpoints

`(gdb) disable [ bp_number | range]`

Temporarily deactivates a breakpoint or a range of breakpoints

`(gdb) enable [ bp_number | range]`

Restores disabled breakpoints

If no arguments are provided to the above commands, all breakpoints are affected!!

`(gdb) ignore bp_number iterations`

Instructs GDB to pass over a breakpoint without stopping a certain number of times.

bp_number: the number of a breakpoint

Iterations: the number of times you want it to be passed over

Displaying Data

Why would we want to interrupt execution?

to see data of interest at run-time:

`(gdb) print [/format] expression`

Prints the value of the specified expression in the specified format

Formats:

d: Decimal notation (default format for integers)

x: Hexadecimal notation

o: Octal notation

t: Binary notation

Resuming Execution After a Break

When a program stops at a breakpoint

- 4 possible kinds of gdb operations:
  - **c or continue**: debugger will continue executing until next breakpoint
  - **s or step**: debugger will continue to next source line
  - **n or next**: debugger will continue to next source line in the current (innermost) stack frame
  - **f or finish**: debugger will resume execution until the current function returns. Execution stops immediately after the program flow returns to the function's caller
    - the function's return value and the line containing the next statement are displayed

Watchpoints

Watch/observe changes to variables

```
(gdb) watch my_var
```

sets a watchpoint on my_var

the debugger will stop the program when the value of *my_var* changes

old and new values will be printed

```
(gdb) rwatch expression
```

The debugger stops the program whenever the program reads the value of any object involved in the evaluation of *expression*

Stack Info

A program is made up of one or more functions which interact by calling each other

Every time a function is called, an area of memory is set aside for it. This area of memory is called a **stack frame** and holds the following crucial info:

- storage space for all the local variables
- the memory address to return to when the called function returns
- the arguments, or parameters, of the called function

Each function call gets its own stack frame. Collectively, all the stack frames make up the **call stack**

Stack Frames and the Stack

```
1    #include <stdio.h>
2    void first_function(void);
3    void second_function(int);
4
5    int main(void)
6    {
7       printf("hello world\n");
8       first_function();
9       printf("goodbye goodbye\n");
10
11      return 0;
12   }
13
14
15   void first_function(void)
16   {
17      int imidate = 3;
18      char broiled = 'c';
19      void *where_prohibited = NULL;
20
21      second_function(imidate);
22      imidate = 10;
23   }
24
25
26   void second_function(int a)
27   {
28      int b = a;
29   }
```

Frame for `main()`

Frame for `first_function()`
- Return to `main()`, line 9
- Storage space for an int
- Storage space for a char
- Storage space for a void *

Frame for `second_function()`:
- Return to `first_function()`, line 22
- Storage space for an int
- Storage for the int parameter named `a`

[illegible]

execution within second_function()

Analyzing the Stack in GDB

`(gdb) backtrace|bt`

Shows the call trace (the call stack)

Without function calls:

#0 main () at program.c:10

one frame on the stack, numbered 0, and it belongs to main()

After call to function display()

#0 display (z=5, zptr=0xbffffb34) at program.c:15

#1 0x08048455 in main () at program.c:10

Two stack frames: frame 1 belonging to main() and frame 0 belonging to display().

Each frame listing gives

- the arguments to that function
- the line number that's currently being executed within that frame

Analyzing the Stack

(gdb) info frame

Displays information about the current stack frame, including its return address and saved register values

(gdb) info locals

Lists the local variables of the function corresponding to the stack frame, with their current values

(gdb) info args

List the argument values of the corresponding function call

Other Useful Commands

(gdb) info functions

Lists all functions in the program

(gdb) list

Lists source code lines around the current line

gdb - Debugger

For Lab 2, you may find these lines useful:

```
(gdb) break <function_name>
(gdb) run
(gdb) stepi
(gdb) stepn
(gdb) info registers
(gdb) disassemble
```

`All variants of “print”`

Midterm

- Study/prepare early
- Take advantage of resources
  - Piazza, UPE tutoring - Monday, 2/6 from 6:15PM - 8:00PM in Carnesale Hermosa AB
  - Can even e-mail me/DJ to go over things
  - Read textbook! ***Very*** helpful.
  - Doing the practice exercises helps consolidate things
  - Go through gdbnotes1.pdf under Week 4
- You can do it!