

# CSE 127 Week 1

## Discussion

Fall 2026

# CSE 127 Fall 2026: Course staff

Professor Stefan Savage

Office hours: Fri 9 AM, CSE 3106, or by appointment

Trey Shneour (TA)

Office hours: Wednesday 2-4 PM, Friday 2-4PM

Location: CSE B215

Morty Zhang (TA)

Office hours: Tue, Thu 2-4 PM

Location: area outside CSE 3140

Sachin Ramanathan (Tutor)

Office hours: Mon 2-4 PM, Wed 11 AM–12 PM

Location: Mon CSE B275, Wed CSE B260A

Subject to Change!

Check the course calendar on the [website's home page](#) for the most up-to-date OH.

Double-check Piazza just to make sure though ;)

# Today's Agenda

- Setting up the Virtual Machine (VirtualBox)
- Intro to GDB
  - How-to-use
  - Essential commands
- Intro to x86 assembly
  - Registers/Useful instructions
  - Stack Layout
  - AT&T syntax quirks
  - Syscalls

# PA1: GDB + x86

- Due date: October 6, 2026 at 11:59 PM PDT
- 20 total points (GDB and x86)
- Group of maximum 2 people (or solo)
- Up to **three** late days allowed for PA submissions over the course of this class

Prep for buffer overflows assignment. More GDB! More x86! Yay!

# Getting the VM started

## VM Image

In order to match the environment in which your submission will be graded, all work for this assignment must be done on the VirtualBox VM we provide, named `palbox`. You can download the VM image [here](#). (For M1/M2 Macs, see the [instructions at the bottom of this page](#).)

The VM is configured with two users: `student` (note that the user name is `student` instead of your school user name), with password `hacktheplanet`; and `root`, with password `hackallthethings`. The VM is configured with SSH on port 2222. Please note that SSH is disabled for `root`, so you can only SSH in as the `student` user. You can still log in as `root` using `su` or by logging into the VM directly.

To SSH into the VM:

```
ssh -p 2222 student@127.0.0.1
```

To copy files from your computer to the VM:

```
scp -P 2222 -r /path/to/files/ student@127.0.0.1:/home/student
```

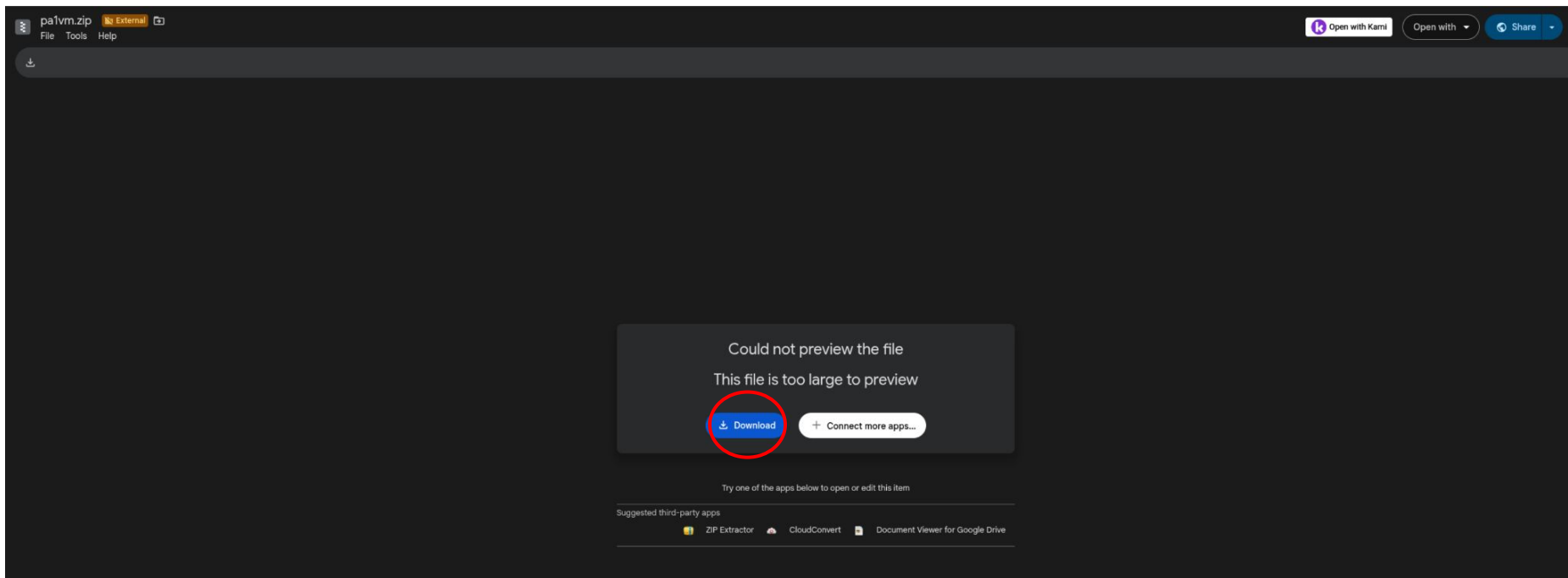
To copy files from the VM to your computer:

```
scp -P 2222 student@127.0.0.1:/path/to/files/ /destination/path
```

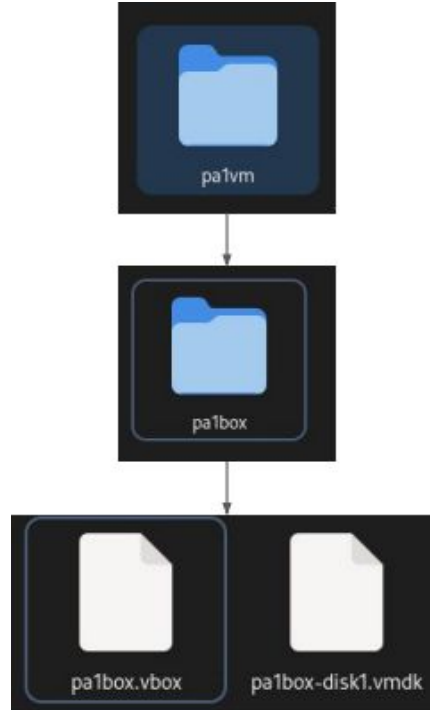
Course Homepage > Assignments > Programming Assignment 1

[Current assignment page](#)

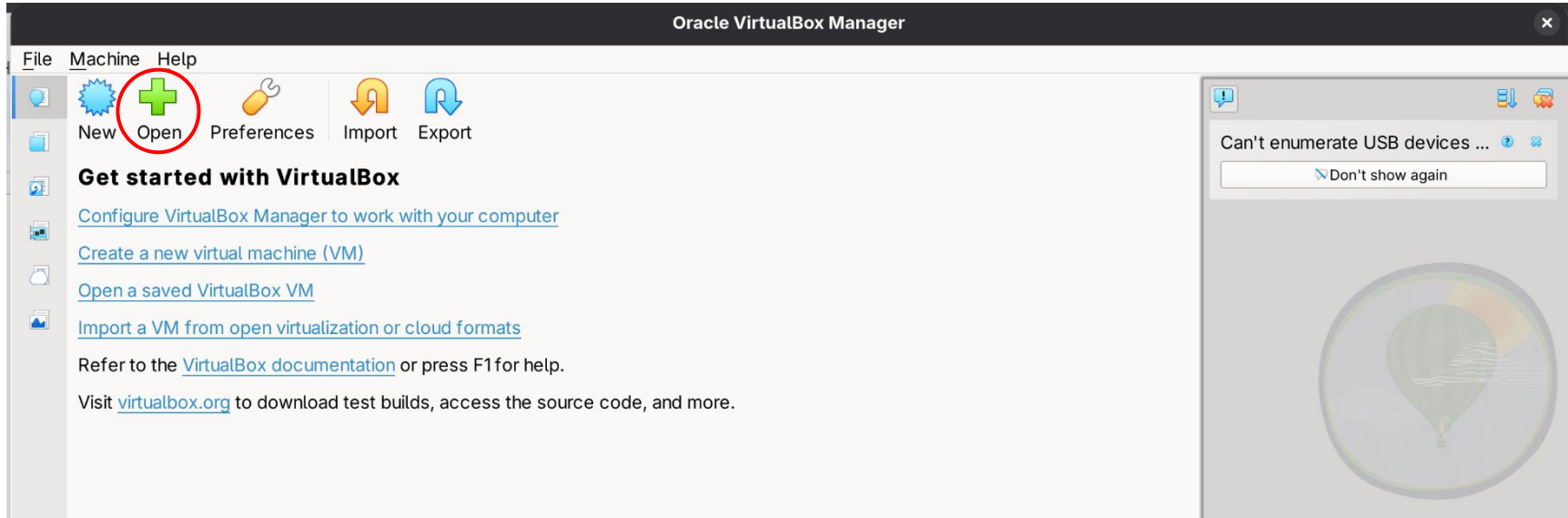
# Getting the VM started (cont.)



## Getting the VM started (cont.)

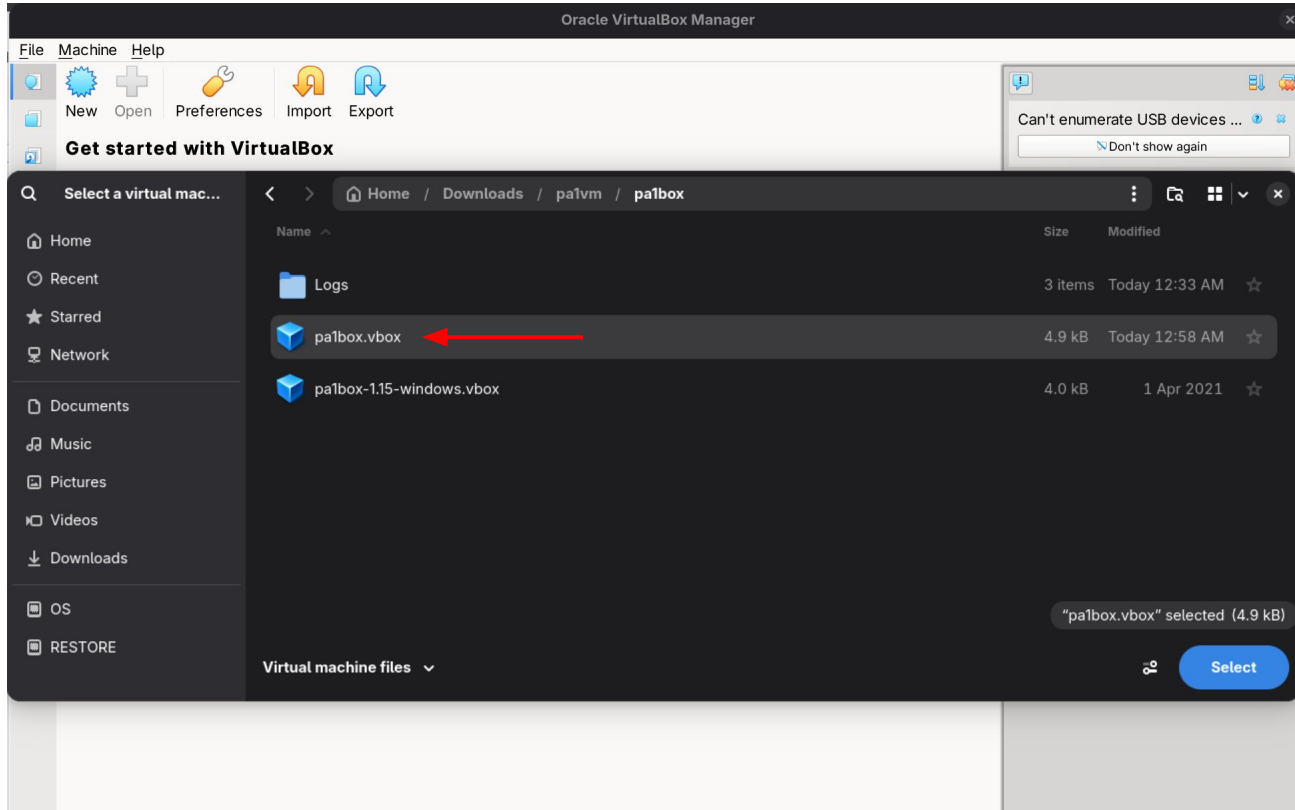


# If you are a PC (Windows/Linux) user:

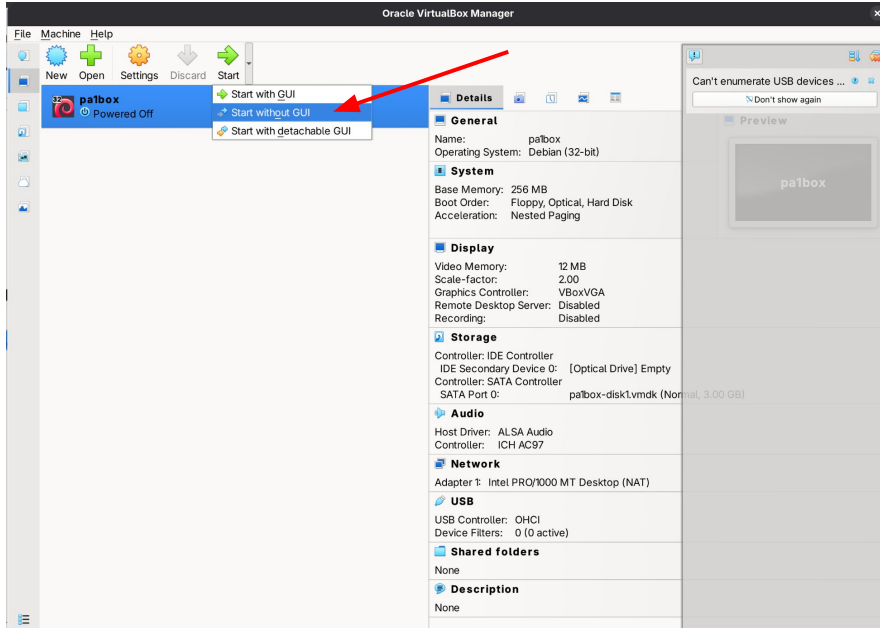


<https://www.virtualbox.org/wiki/Downloads>

# If you are a PC (Windows/Linux) user (cont.)



# If you are a PC (Windows/Linux) user (cont.)



```
tshneour@fedora:~/Downloads$ ssh -p 2222 student@127.0.0.1
student@127.0.0.1's password:
Linux CSE127 4.9.0-8-686-pae #1 SMP Debian 4.9.130-2 (2018-10-27) i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

No mail.
Last login: Fri Oct  2 00:56:36 2026 from 10.0.2.2
student@CSE127:~$
```

```
ssh -p 2222 student@127.0.0.1
```

## If you are a PC (Windows/Linux) user (cont.)

```
(base) tshneour@fedora:~/Downloads$ ssh -p 2222 student@127.0.0.1
student@127.0.0.1's password:
Linux CSE127 4.9.0-8-686-pae #1 SMP Debian 4.9.130-2 (2018-10-27) i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
No mail.
Last login: Fri Oct  2 00:56:15 2026 from 10.0.2.2
student@CSE127:~$ ls
gdb  x86
student@CSE127:~$ cd ~/gdb
student@CSE127:~/gdb$ ls
fib.c  hw1.txt  Makefile
student@CSE127:~/gdb$ cd ~/x86
student@CSE127:~/x86$ ls
echo.o  echo.s  Makefile
student@CSE127:~/x86$
```

← GDB Part

← x86 Part

# If you are a Mac user (M1/M2 chipset)

- VirtualBox *will not work*
- Use QEMU OR UTM and the provided UTM image
- Refer to course webpage for details:
- <https://cseweb.ucsd.edu/classes/fa26/cse127-001/pa1/pa1.html>

## M1/M2 machines

VirtualBox doesn't work on M1/M2 machines. To use the vm image on M1/M2 machines, you will have to install [qemu](#) and run the vm with it.

```
brew install qemu

qemu-system-x86_64 \
-cpu qemu64 \
-m 4096 \
-device e1000,netdev=net0,mac=08:00:27:D3:23:52 \
-netdev user,id=net0,hostfwd=tcp::2222-:22 \
-drive file=/Path/to/palbox/palbox-disk1.vmdk
```

User and ssh settings are the same as using VirtualBox.

Alternatively, on M1/M2 Macs, install [UTM](#) and download [this VM image](#); after extracting it, you should be able to open it directly in UTM and start the VM.

For SSH access, configure TCP port forwarding from port 2222 on your Mac to port 22 in the VM, then use `ssh -p 2222 student@127.0.0.1`; this [VirtualBox port-forwarding example](#) illustrates the idea, though UTM's settings are different.

# GDB

- The GNU Debugger
  - Very powerful - allows you to see inside your program!
  - Setting breakpoints to pause execution at specific instructions
  - Examining register values, contents of memory, etc
  - Shows you the stack frame!
- Not very easy to use... (But 100% worth using!)
  - Command-line driven
  - A lotttt of commands
  - You can examine every bit of memory in the program!

# GDB: Control Flow

Go to the gdb directory and start GDB first with *gdb fib*

You should see: 

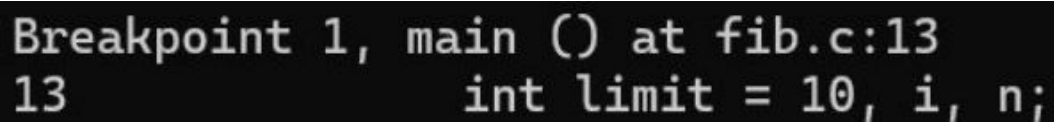
Then, where do I start? **Breakpoints!!**

- Adding breakpoints:
  - *b [function name]* to add breakpoint at some function, eg. *b main*
  - *b [line number]* to add breakpoint at a line of code in your c program, eg. *b 15*
- Viewing your breakpoints:
  - *i b*, which shows you all your breakpoints and additional details
- Deleting your breakpoints:
  - *del [some breakpoint's number ID]*, eg. *del 1*

# Cont.

What does a breakpoint do exactly?

It pauses your program at a point before some designated line of code is executed

Eg. The screenshot shows a debugger window with a dark background. It displays 'Breakpoint 1, main () at fib.c:13' on the first line and '13 int limit = 10, i, n;' on the second line.

This is a breakpoint at line 13 of fib.c, meaning that the next line of code to be executed is line 13: *int limit = 10, i, n;*

## Cont.

After setting all your breakpoints, start your program via the command `r`

```
(gdb) r
```

Then you should encounter your first breakpoint!

For better viewing experience, enable the Text User Interface via the command. You should see a more user friendly UI that shows you the source code and the line of code you are breaking at.

```
(gdb) tui enable
```

You may navigate the program with the following 3 commands:

- `c`, or `continue`, which continues with execution until the next breakpoint or the end of the program
- `n`, or `next`, which execute the next line of code
- `s`, or `step`, which also executes the next line of code, but would step into any function
- If you use `ni/si` instead of `n/s`, you will execute the next **instruction** instead

# GDB: Accessing Values

For starter, you can access values stored in registers via:

```
(gdb) i r
```

Then you should a table of registers and their values:

Special registers:

- eip - stores instruction pointer
- ebp - base (stack frame) pointer
- esp - stack pointer

|        |            |                    |
|--------|------------|--------------------|
| eax    | 0xb7fc8dbc | -1208185412        |
| ecx    | 0xbffff6c0 | -1073744192        |
| edx    | 0xbffff6e4 | -1073744156        |
| ebx    | 0x402000   | 4202496            |
| esp    | 0xbffff690 | 0xbffff690         |
| ebp    | 0xbffff6a8 | 0xbffff6a8         |
| esi    | 0x1        | 1                  |
| edi    | 0xb7fc7000 | -1208193024        |
| eip    | 0x40060c   | 0x40060c <main+29> |
| eflags | 0x216      | [ PF AF IF ]       |
| cs     | 0x73       | 115                |
| ss     | 0x7b       | 123                |
| ds     | 0x7b       | 123                |
| es     | 0x7b       | 123                |
| fs     | 0x0        | 0                  |
| gs     | 0x33       | 51                 |

# Cont.

You may access values of a variable at any point via: *print [name of variable]*

e.g. 

```
(gdb) print limit
$3 = 10
```

Use command *where* to see the current call stack

e.g. 

```
(gdb) where
#0  f (x=0) at fib.c:4
#1  0x0040063c in main () at fib.c:17
```

Command *disas [function name]* to disassemble a function into assembly code

e.g. 

```
(gdb) disas main
Dump of assembler code for function main:
0x004005ef <+0>:    lea    0x4(%esp),%ecx
0x004005f3 <+4>:    and    $0xffffffff0,%esp
0x004005f6 <+7>:    pushl  -0x4(%ecx)
0x004005f9 <+10>:   push   %ebp
0x004005fa <+11>:   mov    %esp,%ebp
```

Use *list [function name]* or *list line\_start, line\_end* to see C code

e.g. 

```
(gdb) list main
7      else {
8          return f(x-1) + f(x-2);
9      }
10     }
11
12     int main(void) {
13         int limit = 10, i, n;
14
15         printf("Here are the first %d numbers in the Fibonacci sequence:\n", limit);
16         for(i = 0; i < limit; i++) {
```

## Cont.

You can inspect the address of symbols (functions/variables) via:

*i add [symbol name]*

e.g. 

```
(gdb) i add main
Symbol "main" is a function at address 0x4005ef.
```

You can inspect the contents of memory via:

*x /[Format] [Address/Register]*

e.g. 

```
(gdb) x /5i $eip
=> 0x40060c <main+29>: movl    $0xa,-0x10(%ebp)
    0x400613 <main+36>: sub     $0x8,%esp
    0x400616 <main+39>: pushl   -0x10(%ebp)
    0x400619 <main+42>: lea     -0x1900(%ebx),%eax
    0x40061f <main+48>: push    %eax
```

Some formats:

i = instruction

x = hexadecimal

w = word

b = byte

... etc

# GDB Command References

<https://visualgdb.com/gdbreference/commands/>

^ (Incomplete) reference of useful gdb commands by category

<https://sourceware.org/gdb/current/onlinedocs/gdb.pdf>

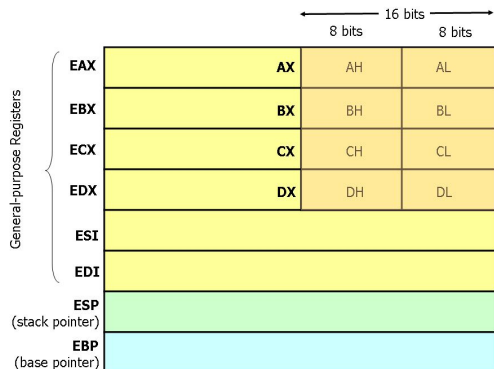
^ also official GDB manual but personally too much information for what you need

## Part 1 Submission

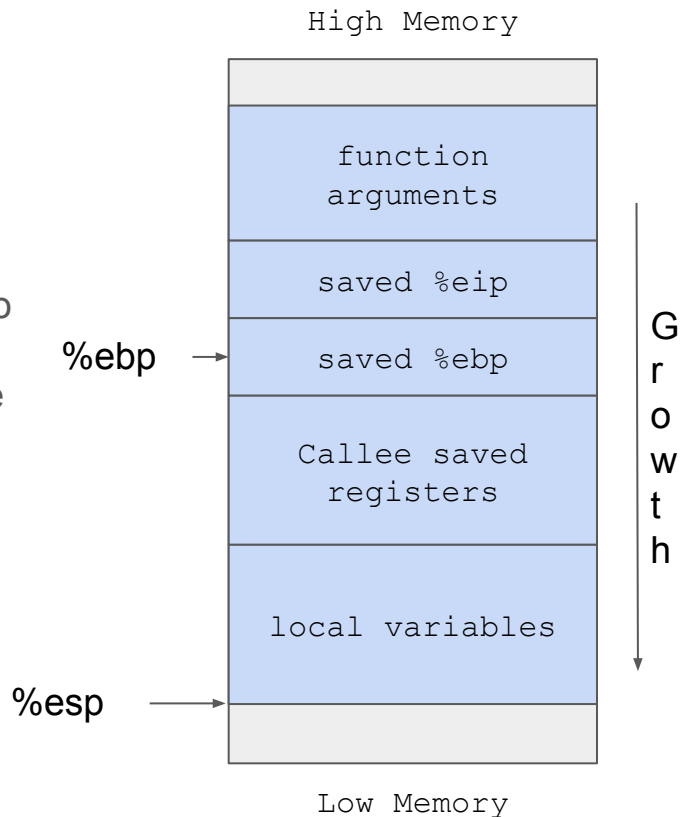
- Submit hw1.txt to "Assignment 1 - GDB" on Gradescope.
- Follow the directions in the template, **do not delete the square brackets**
  - Let's say your answer is 0xffffffff, on your final submission you should have `Your Answer: [0xffffffff]` instead of `Your Answer: 0xffffffff`
- If you remove the square brackets, the autograder won't recognize your answers.

# x86 Registers

- **%esp, or the Stack Pointer**
  - Designates the top of the stack
  - Grows from high to low memory addresses
- **%ebp, or the Frame/Base Pointer**
  - Reference point in the current stack frame
  - Memory address of variables are expressed as an offset to ebp typically
  - Doesn't move as function calls or local variables are made



Reference: <https://www.cs.yale.edu/flint/cs421/papers/x86-asm/asm.html>



# x86 Registers

- `%eip`, or the Instruction Pointer
  - Holds the address of the next instruction to be executed

Low Memory

`%eip` →

```
pushl    %ebp
movl     %esp, %ebp
subl     $40, %esp
movl     16(%ebp), %eax
movl     %eax, -28(%ebp)
movl     %gs:20, %eax
movl     %eax, -12(%ebp)
xorl     %eax, %eax
```

High Memory

# Stack Layout

Did you know? The actual entry point for a C program is `_start`, **not** `main`

- Once a process is created, the kernel populates stack with command-line arguments (among other things)
- Kernel passes control to `_start`
  1. Various bookkeeping tasks
  2. Prepares call to `main`

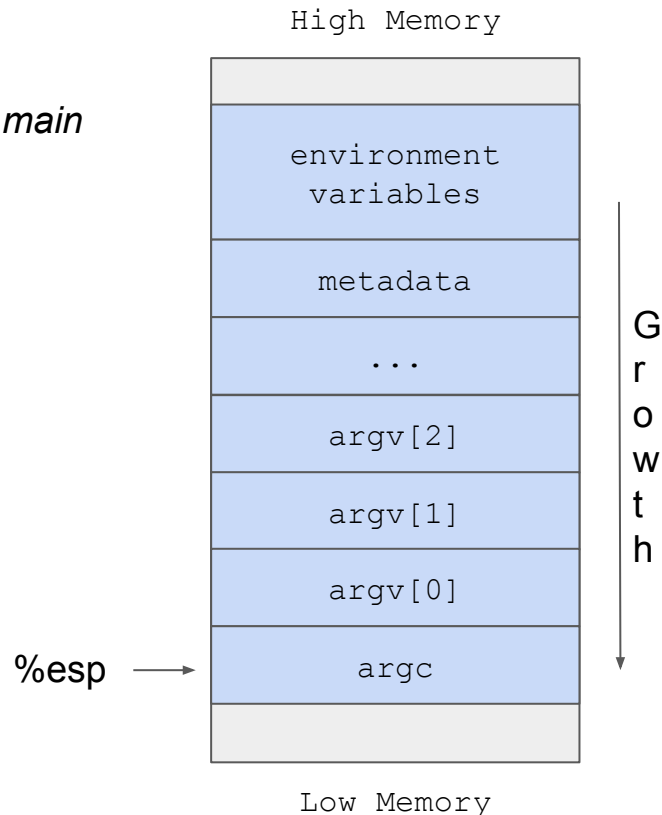
Compile and link assembly -> no C runtime library -> no `_start`\*  
-> no `main` call

Fortunately, `argc` and `argv` are already on the stack! Thanks kernel!  
Except some things are weird... (what's `ebp`?)

This will be relevant for Part 2 of the assignment...

\*unless you supply it yourself

## Stack before `_start`



# x86 Key Instructions

## Arithmetic

- `add src, dst`
- `sub src, dst`
- `inc dst`                      increment
- `dec dst`                      decrement

## Bitwise Logic

- `and src, dst`
- `or src, dst`
- `not src, dst`
- `shl count, dst`              shift left
- `shr count, dst`              shift right
- `sar count, dst`              shift right, preserve sign

## Data

- `mov src, dst`                move value
- `lea src, dst`                compute address
- `push src`                    write to stack, decrease esp
- `pop dst`                    read from stack, increase esp

## Branching/Control Flow

- `cmp src, dst`                computes `dst - src`
- `je label`                    jump if `diff == 0`
- `jne label`                   jump if `diff != 0`
- `jnl label`                   jump if `diff < 0`
- `jle label`                   jump if `diff <= 0`
- `jg label`                    jump if `diff > 0`
- `jge label`                   jump if `diff >= 0`
- `jmp label`                   unconditional jump

# AT&T Syntax - Memory Addressing

We use parentheses instead of brackets for addressing

- `inc %eax` → `eax++`
- `inc (%eax)` → `(*eax)++`
- `inc 4(%eax)` → `*(eax + 4)++`
- `inc 4(%eax, %ebx, 2)` → `*(eax + 2*ebx + 4)++`

# AT&T Syntax - Instruction Size Suffixes

- Byte - (b)
  - 8 bits
- Word - (w)
  - 16 bits = 2 bytes
- Long - (l)
  - 32 bits = 4 bytes
- Quad - (q)
  - 64 bits = 8 bytes

mov**l**

mov**q**

Typically you will use l as most data, including pointers, are 4 bytes

# AT&T Syntax - Other

## Register/Immediate Prefixes:

- Register prefixed by % e.g. %esp, 2(%esp)
- Immediate (constant) prefixed by \$ e.g. \$5, \$0x12345678, 0b10101010

## Operand Order:

- All 2-argument (both registers) instructions use src -> dst order

e.g.      add 4(%esp), %ecx  
          ecx <- ecx + \*(esp + 4)

# Syscalls

Typical syscall in assembly (only first 3 arguments for simplicity):

```
eax <- syscall code
ebx <- argument 1
ecx <- argument 2
edx <- argument 3
int $0x80
// Return value now in eax
```

- int (interrupt) calls an interrupt handler using an assigned **code**
- 0x80 is syscall handler, 0x0E is page fault handler, etc.

[Syscall reference table](#)

## Part 2 Submission

- Submit echo.s to "Assignment 2 - x86" on Gradescope.
- Make sure to thoroughly test your code before submission!

(if time)

Any questions about the assignment, GDB, x86, etc?