UiT The Arctic University of Norway

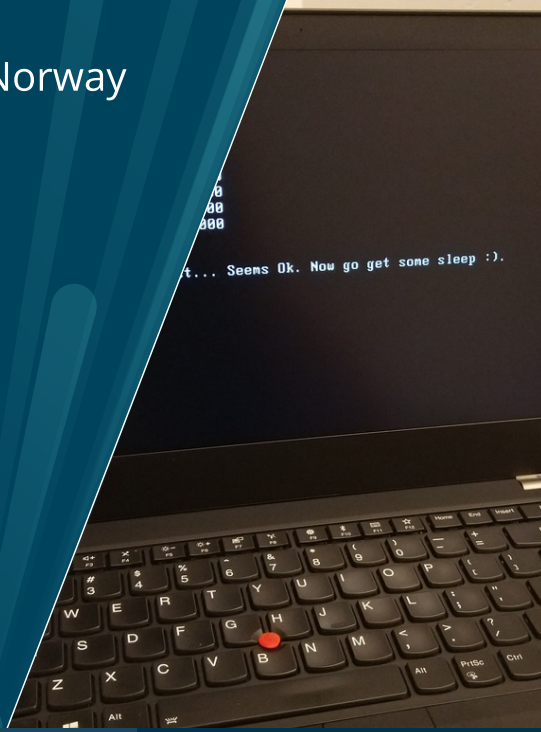
Project 1

Loading Programs

Mike Murphy

UiT INF-2203

Spring 2026



Hello, World!

Mike Murphy

- ~~Michael~~ **Mike**
- Not as scary as I look
- American (sorry)
- Æ kan norsk men engelsk e lettere

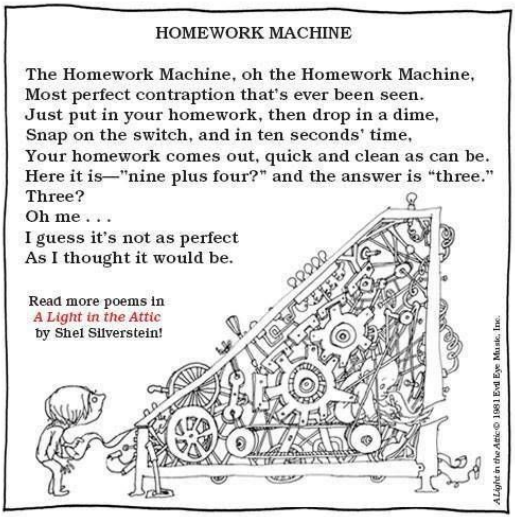
Aging into a Unix Greybeard

- Gen AI bad
- Open source good
- Linux rules!
- Windows? Haven't used it since 2006

No AI!

Linda Hamilton in *Terminator 2: Judgment Day*, judging you[1]

[1] Photo from *Terminator 2: Judgment Day* (1991, dir. James Cameron), meme text origin unknown



"The Homework Machine" by Shel Silverstein[2]

[2] Poem and illustration from *A Light in the Attic*, 1981

Projects Overview

Build an OS from Scratch(-ish)

Munix — Munin/Micro Unix

- P1: Running a program
- P2: Protection
- P3: Multitasking
- P4: Inter-Process Communication

Basic format

- We provide a basic skeleton (“precode”)
- You fill in the important bits
- Time ~3–4 weeks
- Continuation options: yours or ours

Work in Pairs or Alone

- Projects: Groups of 1–2
- Work *together*. Do not divide work.
- Oral exams: Assignment group

Deliverables

- Midpoint design review
- **Hand-in: Code, Report**
- Possible postmortem

Challenging Projects

Challenge

- Was challenging as a 20-ECTS course
- Will be challenging at 10-ECTS
- Tough but rewarding
- Tough but *fun*[3]

Get Your Hands Dirty

- The projects are to give you *experience*
- Do the work → easy exam

Recommended Prereqs[4]

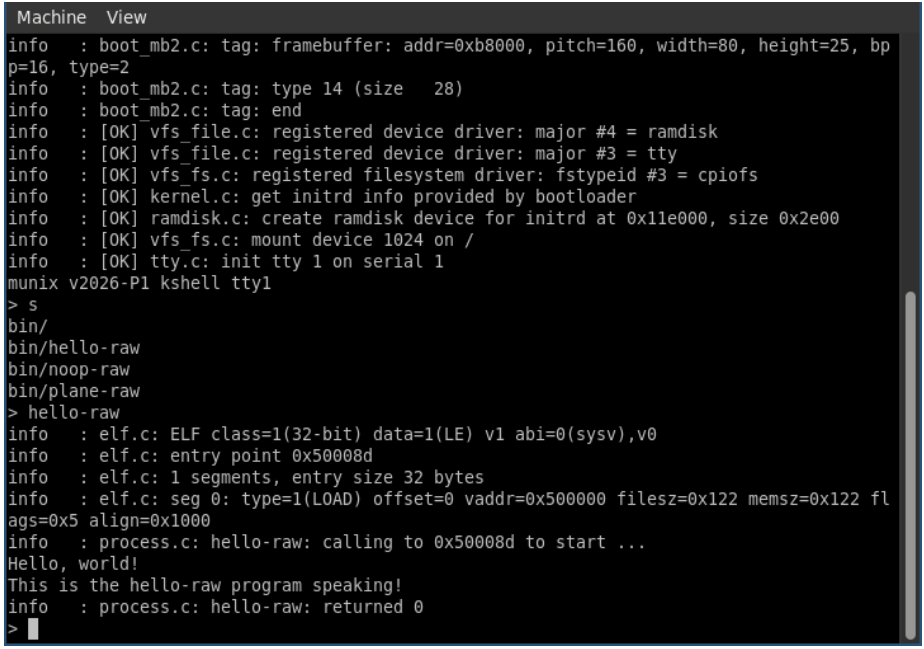
- INF-1101 Data Structures and Algorithms
- INF-2200 Architecture

Important Concepts

- C language
- Assembly language
- Unix command line
- Git

[3] *Fun* might be in the Dwarf Fortress sense

[4] See UiT course info: https://uit.no/utdanning/emner/emne/906652/inf-2203



Project 1 in QEMU (serial I/O): running a “hello world” program

Finish Boot, Load Programs

We Provide

- Build system
- Bootloader (GRUB)
- Skeleton kernel
- Basic I/O
- Test programs

Your Tasks

1. Finish reading boot info
2. Read and load executable files

Finish Boot, Load Programs

We Provide

- Build system
- Bootloader (GRUB)
- Skeleton kernel
- Basic I/O
- Test programs

Your Tasks

1. Finish reading boot info
2. Read and load executable files

- Less than 100 lines of code

Finish Boot, Load Programs

We Provide

- Build system
- Bootloader (GRUB)
- Skeleton kernel
- Basic I/O
- Test programs

Your Tasks

1. Finish reading boot info
2. Read and load executable files

- Less than 100 lines of code
- Hard part: getting ready to write it

Finish Boot, Load Programs

We Provide

- Build system
- Bootloader (GRUB)
- Skeleton kernel
- Basic I/O
- Test programs

Your Tasks

1. Finish reading boot info
2. Read and load executable files

- Less than 100 lines of code
- Hard part: getting ready to write it

Task Zero

- Set up build environment
- Understand precode organization
- Understand build system
- Understand kernel startup
- Understand how GRUB passes data
- Understand ELF format
- Brush up on weak spots

Finish Boot, Load Programs

We Provide

- Build system
- Bootloader (GRUB)
- Skeleton kernel
- Basic I/O
- Test programs

Your Tasks

1. Finish reading boot info
2. Read and load executable files

- Less than 100 lines of code
- Hard part: getting ready to write it

Task Zero

- Set up build environment
- Understand precode organization
- Understand build system
- Understand kernel startup
- Understand how GRUB passes data
- Understand ELF format
- Brush up on weak spots

Lots of Background

Let's get into it ...

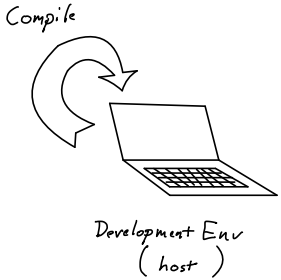
A Brief History of x86

Year	CPU	Regs	Addresses	Hot Feature
1978	Intel 8086	16-bit	20-bit (1 MiB)	16-bit!
1982	Intel 286	16-bit	24-bit (16 MiB)	Protection
1985	Intel 386	32-bit	32-bit (4 GiB)	Virtual Memory
...				
2003	AMD Opteron	64-bit	52-bit (4 PiB)	64-bit!

- All backwards compatible
- Modern CPU boots up in 8086 16-bit mode
- Same basic instruction set, just extended

- We are targeting the 386 (aka generic 32-bit x86)
- Emulator: QEMU (`qemu-system-i386`)

Cross Compiler

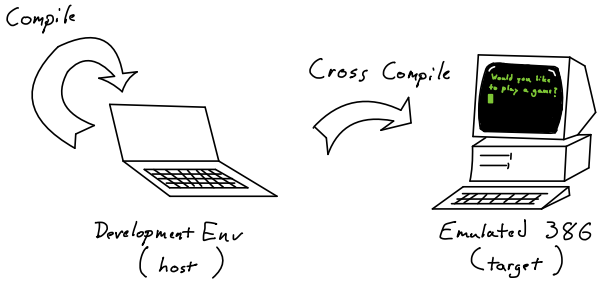


Normal Compiler

- Runs on *host*
- Compiles for host
- Invoke as normal:

```
gcc hello.c
```

Cross Compiler



Normal Compiler

- Runs on *host*
- Compiles for host
- Invoke as normal:
  `gcc hello.c`

Cross Compiler

- Runs on *host*
- Compiles for *target*
- Invoke with prefix:
  `i386-elf-gcc hello.c`

Installing a Cross Compiler

- Do I really need a cross compiler?
  - Yes
  - Even on x86_64, you have a different OS
  - We need to compile for generic ELF-based system, not Linux specifically
- Installing a cross compiler:
  1. Start with Linux/Unix environment
  2. Install regular compiler
  3. Download GNU source code:
     - **binutils**: assembler, readelf, objdump, etc.
     - **gcc**: GNU Compiler Collection
  4. Configure with `--target=i386-elf`
  5. Build from source
- Instruction docs in precode repository

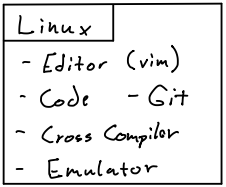
Linux/Unix is the Base

- This code is developed under Linux
  - Build uses common command-line tools: `make`, shell scripts, etc.
- We will assume you have a Linux/Unix development environment:
  - Ubuntu 22.04 is the reference
  - Other Linux distros should work
  - Mac should also work (MacOS is Unix under the hood)
  - Windows + WSL2 should work
- You are responsible for your development environment
- Know thy tools

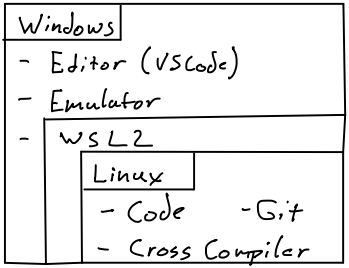
"It's a Unix System. I know this."[5]

[5] Ariana Richards in *Jurassic Park* (1993, dir: Steven Spielberg)

About Windows



Linux Environment



Windows Environment (I think)

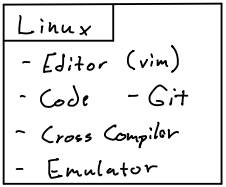
If You Use WSL

- Be aware of the layers
  - What is inside WSL
  - What is outside
- Use Git *inside WSL* to get precode
- Windows's Git will mangle the files/permissions

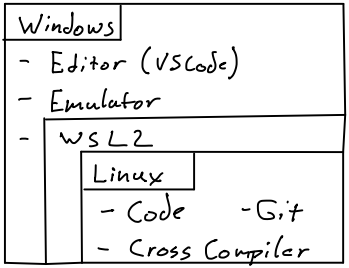
Beware of Microsoft

- WSL + VSCode

About Windows



Linux Environment



Windows Environment (I think)

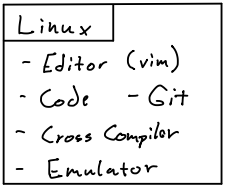
If You Use WSL

- Be aware of the layers
  - What is inside WSL
  - What is outside
- Use Git *inside WSL* to get precode
- Windows's Git will mangle the files/permissions

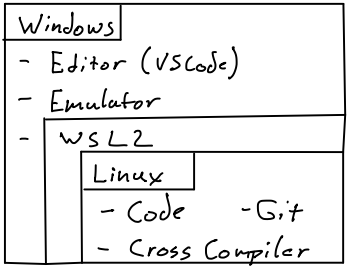
Beware of Microsoft

- WSL + VSCode + GitHub

About Windows



Linux Environment



Windows Environment (I think)

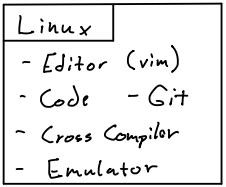
If You Use WSL

- Be aware of the layers
  - What is inside WSL
  - What is outside
- Use Git *inside WSL* to get precode
- Windows's Git will mangle the files/permissions

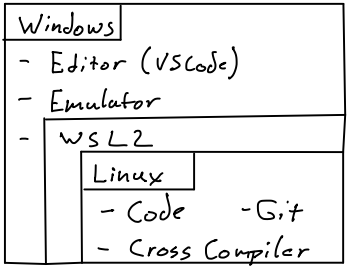
Beware of Microsoft

- WSL + VSCode + GitHub
- Suddenly the de-facto dev suite

About Windows



Linux Environment



Windows Environment (I think)

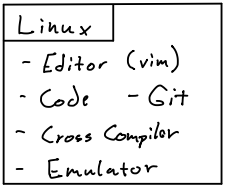
If You Use WSL

- Be aware of the layers
  - What is inside WSL
  - What is outside
- Use Git *inside WSL* to get precode
- Windows's Git will mangle the files/permissions

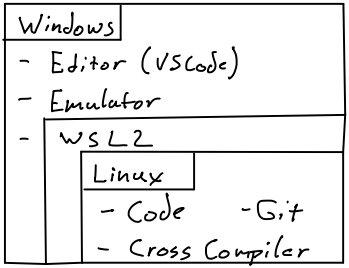
Beware of Microsoft

- WSL + VSCode + GitHub
- Suddenly the de-facto dev suite
- True goal: profit

About Windows



Linux Environment



Windows Environment (I think)

If You Use WSL

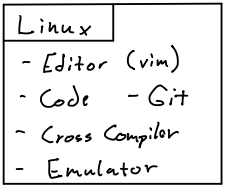
- Be aware of the layers
  - What is inside WSL
  - What is outside
- Use Git *inside WSL* to get precode
- Windows's Git will mangle the files/permissions

Beware of Microsoft

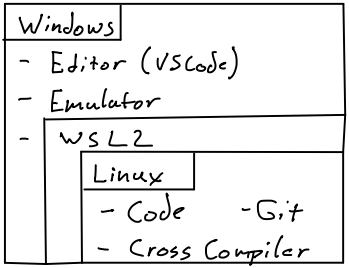
- WSL + VSCode + GitHub
- Suddenly the de-facto dev suite
- True goal: profit

"Embrace and Extend"

About Windows



Linux Environment



Windows Environment (I think)

If You Use WSL

- Be aware of the layers
  - What is inside WSL
  - What is outside
- Use Git *inside WSL* to get precode
- Windows's Git will mangle the files/permissions

Beware of Microsoft

- WSL + VSCode + GitHub
- Suddenly the de-facto dev suite
- True goal: profit

"Embrace and ~~Extend~~" Extinguish

Classic C Hello World

C source (hello.c)

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, world!\n");
    return 0;
}
```

Example commands

```
gcc hello.c -o hello.s -S \
    -fno-asynchronous-unwind-tables

gcc hello.c -o hello
./hello
objdump -d hello | less
```

Compiled x86-64 assembly (hello.s)

```
    .section    .rodata
.LC0:
    .string "Hello, world!"

    .text
    .globl  main
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    movl    $0, %eax
    leave
    ret
```

C Standard Library: Hosted vs Freestanding

Provided by libc

- printf
- strcpy
- memcpy
- open
- …

- Basics like printf are provided by C Standard Library, aka libc

C Standard Library: Hosted vs Freestanding

Provided by libc

- printf
- strcpy
- memcpy
- open
- …

Provided by OS

- System services
- Hardware access
- Actual I/O
- …

- Basics like printf are provided by C Standard Library, aka libc
- C library is an *interface* to OS services

C Standard Library: Hosted vs Freestanding

Provided by Compiler

- Basic operations
- Number type/limits
- size_t
- stdarg
- …

Provided by libc

- printf
- strcpy
- memcpy
- open
- …

Provided by OS

- System services
- Hardware access
- Actual I/O
- …

- Basics like printf are provided by C Standard Library, aka libc
- C library is an *interface* to OS services
- Core parts of C library come from compiler itself

C Standard Library: Hosted vs Freestanding

Provided by Compiler

- Basic operations
- Number type/limits
- size_t
- stdarg
- …

Provided by libc

- printf
- strcpy
- memcpy
- open
- …

Provided by OS

- System services
- Hardware access
- Actual I/O
- …

- Basics like printf are provided by C Standard Library, aka libc
- C library is an *interface* to OS services
- Core parts of C library come from compiler itself
- We will start with only that core *freestanding* C environment

Freestanding C

- Program starts at `_start`, not `main`
- Only basic headers available, such as:

header	purpose	example
<stddef.h>	Important defs	`NULL`, `size_t`, `ptrdiff_t`
<stdint.h>	Fixed-width ints	`uint8_t`, `uint32_t`
<limits.h>	Integer limits	`INT_MIN`, `INT_MAX`
<float.h>	Float limits	`FLT_MIN`, `FLT_MAX`
<stdarg.h>	Argument lists	`va_list`, `va_start`

- No printf. No I/O.

“Raw” Test Programs Included in Precode

process/noop-raw.c

```
int _start(void) { return 0; }
```

process/hello-raw.c

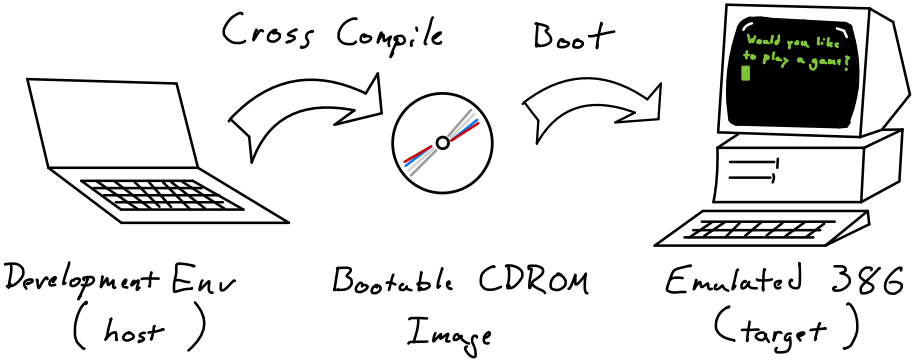
- A freestanding “hello world”
- Writes to serial port
- Includes minimal serial port code

process/plane-raw.c

- Fly a plane across the screen
- Writes to screen
- Includes minimal screen driver code

```
# Simplified cross-compiler commands
i386-elf-gcc -ffreestanding  -c -o hello-raw.o hello-raw.c
i386-elf-gcc -static -nostartfiles -nolibc hello-raw.o -o hello-raw
```

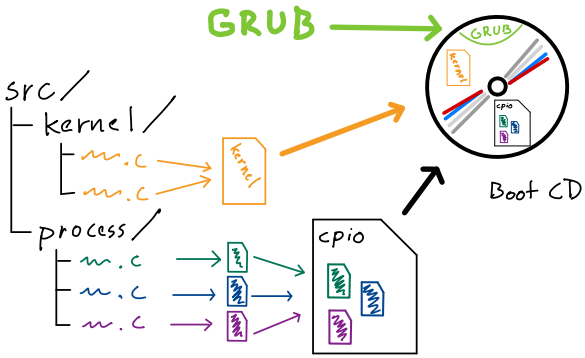
Bootable CD Image



Cross compiling to a bootable CD-ROM

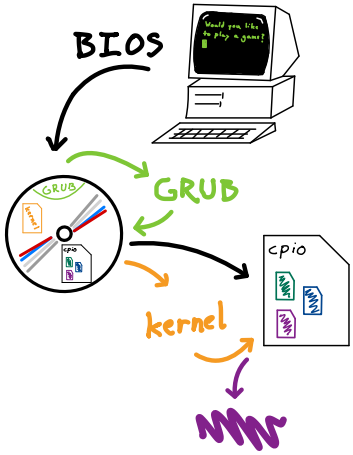
Building the Boot Image

1. Kernel
   - Sources in `src/kernel/*.c`
   - Combine to make kernel
2. Programs
   - Sources in `src/process/*.c`
   - Each file is one program
   - "Zip" into `cpio` archive *(initrd: initial ramdisk)*
3. Boot filesystem
   - Kernel
   - `cpio` archive
4. Boot image
   - GRUB adds boot info and bootloader



Building the boot image

Booting the Boot Image



Booting the boot image

1. BIOS (firmware) loads and runs GRUB
2. GRUB loads kernel and initrd
3. GRUB runs kernel
4. Kernel reads archive and loads programs
   - 4.1 **Gets archive location from GRUB**
   - 4.2 Treats archive as filesystem
   - 4.3 Reads program file from inside archive
   - 4.4 **Loads and runs program**

Build Files/Process

```
munix/
|-- Makefile
|-- src/
|   |-- configure
|   `-- Makefile.template
|
`-- out/
    `-- i386-elf/
        `-- Makefile
```

Top-level Makefile:
`make image`

1. Creates `out/i386-elf/` output directory
2. Runs configure script
   - Generates `out/i386-elf/Makefile` with i386 variables
3. Uses `out/i386-elf/Makefile`: `make image`
   - Includes `src/Makfile.template` for common rules
   - Builds image

Source

```
munix/
`-- src/
    |-- kernel/
    |   |-- kernel.[ch]
    |   |-- kshell.[ch]
    |   `-- process.[ch]
    |-- lib/
    `-- process/
        |-- noop-raw.c
        |-- hello-raw.c
        `-- plane-raw.c
```

Kernel

- Built from `src/kernel/*.c` plus libraries
- `kernel.[ch]` = kernel main
- `kshell.[ch]` = kernel shell
- `process.[ch]` = loading processes

Programs

- Each C file in `process/*.c` is one program
- “-raw” ending = raw code. No libraries

Libraries

```
munix/
`-- src/
    `-- lib/
        |-- core/
        |-- arch/i386/
        |-- drivers/
        `-- oss/
```

libcore

- Essential utilities
- No I/O: no `fprintf`, only `sprintf`

libarch

- Architecture-specific code
- How to use CPU: I/O ports, halt, startup

libdrivers

- Code to talk to hardware devices
- e.g. serial port, screen, etc.

oss/

- Open-source headers (Multiboot, ELF)

“Everything is a File”

- Core tenet of the Unix Philosophy
- Files are sequences of bytes
  - Everything is just a sequence of bytes, really
  - So let’s read and write to devices as if they were files
  - Serial port? File
  - Terminal interface? File
  - Disk parition? File!
  - File? Oh you better believe that’s a file
- Virtual Filesystem
  - Not `C:\` or `D:\`
  - Everything is *mounted* into one unified hierarchy
  - Hierarchy starts with *root*, represented by a slash `/`
  - By convention
    - `/dev/` = Devices
    - `/bin/` = Binaries (programs)
    - …

Drivers

```
munix/src/lib/drivers/
|-- chrdev/
|   |-- ramdisk.c
|   |-- serial.c
|   `-- tty.c
|-- fileformat/
|   |-- ascii.h
|   |-- elf.c
|   `-- elf.h
|-- fs/
|   `-- cpiofs.c
|-- devices.h
|-- log.c
|-- log.h
|-- vfs_file.c
|-- vfs_fs.c
`-- vfs.h
```

Interfaces

- `vfs.h` = Common file and filesystem interfaces
- `devices.h` = Header for devices
- `log.h` = Macros to write to a file as a kernel log

chrdev/ = Character Devices

- Devices as files
- Serial port
- TTY: wraps another file to add terminal features
- Ramdisk: treat bytes in memory like a file/device

fs/ = Filesystems

- cpiofs: Treat a `cpio` archive as a filesystem
- Ramdisk + cpiofs $\rightarrow$ Read files from initrd

Project 1

Task Zero

- Get your environment set up
- Explore and learn the code
- Suggestion: run kernel in debugger and step through startup
- Brush up on your weak spots

Main tasks

1. Finish reading boot info and find ramdisk location
2. Read, load, and run a program file

- Not a lot of code
- But a lot to understand first

Design Reviews

- TAs will schedule *design reviews* about halfway through project time
- Meet with TA
- Present your plan for the assignment

Hand-in

- Code and report
- Zip (or tarball) and upload to Canvas

Report

- Tell us what you did and why
- Four pages is a good length

More on i386

Scratch Area

x86 Registers

Eight General Purpose Registers: AX, BX, CX, DX, SI, DI, SP, BP

- *General Purpose:* can load/store, do math, etc.
- But may have special talent (like Stack Pointer)
- Named for talent/conventional use

Data

- **AX**: Accumulator
- **DX**: Data
- **BX**: Base

String operations

- **SI**: Source Index
- **DI**: Destination Index
- **CX**: Counter

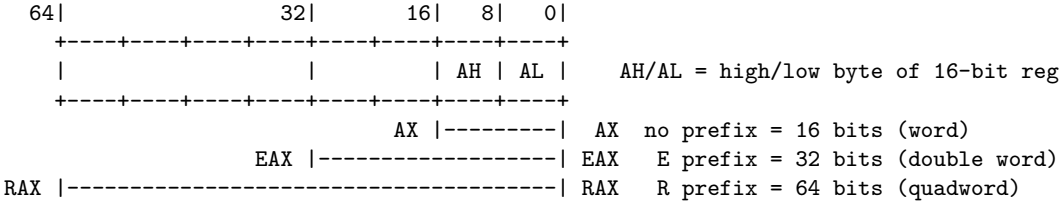
Stack

- **SP**: Stack Pointer
- **BP**: Base Pointer

Control Registers

- **IP**: Instruction Pointer
- **FLAGS**: Condition bits
- (and more)

Intel Registers: Prefixes



To Prepare for the Projects

1. Set up a Linux environment
   - Project code is developed on Linux
   - Reference distro: Ubuntu 22.04.5 LTS (Jammy Jellyfish)
   - Ideally: Use Linux directly

   Windows?

   - WSL2 *should* work
   - Beware of issues with Windows tools operating on WSL files

   Mac?

   - Murky territory. You will have do more troubleshooting
   - Cross compiler and emulator *should* work natively
   - May need to set up a container or virtual machine to be sure
2. Brush up on command line
3. Brush up on low-level development: C and Assembly

Columns Template