C and x86_64 toolchains

CS351: Systems Programming
Day 2: Aug. 25, 2022

Instructor:

Nik Sultana

Quick poll

- Who has accessed **the course webpage** so far?
- Who has accessed **Fourier** so far?
  - Who has tried but failed to access Fourier from **on-campus**?
  - Who has tried but failed to access Fourier from **off-campus**?
- Who has **compiled a C program** since the last lecture**?**
- Who has **dabbled in assembly** since the last lecture**?**

(If you're not sure how to do any of the above, ask your TA)

Overview

- Overview of the C language
- Tools for C programming
- Overview of x86_64
- Examples of x86_64 programs

Overview of the C language

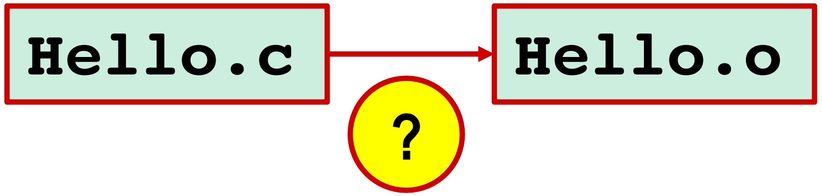
- **Extremely influential language!**
- **Used for both <u>systems</u> and <u>applications</u>.**
  Originally used to develop UNIX: the kernel, shell, and various utilities – including the C compiler toolchain.
- **What else is written in C?**
  OS kernels: Linux (and Android), Windows, parts of macOS.
  Games, applications, device drivers …
- **Original goal: portability and convenience.**
  More convenient that writing assembly by hand.
- **Powerful (expressive), allowing you to bend abstractions.**
  But beware:
  - Static types but permissive casting.
  - Manual memory management.

Tools for C programming

- **Compiler:** translates C source code to machine code.
- **Linter:** warns about possible language misuse – bugs!
- **Linker:** separately-compiled files are "linked" together.
- **Debugger:** inspects compiled and running programs.
- **Memory tracer:** detects potential memory bugs.
- **Profiler:** detects potential performance bugs.
- **Source control:** tracks changes/revisions to code.
- **Build automation**: compiles large code-bases (thousands of files)

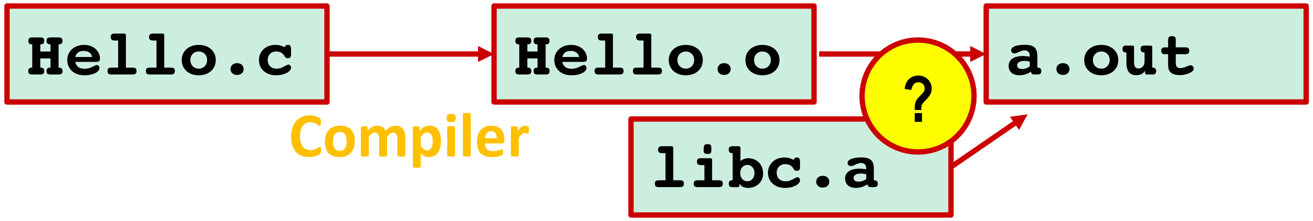
Tools for C programming

- **Compiler:** translates C source code to machine code.
- **Linter:** warns about possible language misuse – bugs!
- **Linker:** separately-compiled files are “linked” together.
- **Debugger:** inspects compiled and running programs.
- **Memory tracer:** detects potential memory bugs.
- **Profiler:** detects potential performance bugs.
- **Source control:** tracks changes/revisions to code.
- **Build automation**: compiles large code-bases (thousands of files)



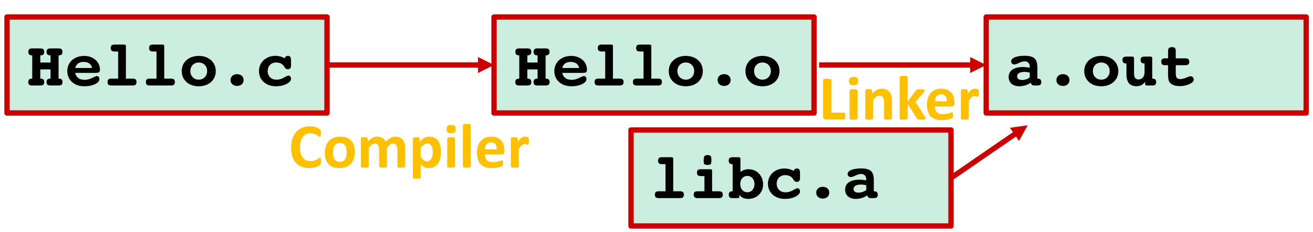
Tools for C programming

- **Compiler:** translates C source code to machine code.
- **Linter:** warns about possible language misuse – bugs!
- **Linker:** separately-compiled files are "linked" together.
- **Debugger:** inspects compiled and running programs.
- **Memory tracer:** detects potential memory bugs.
- **Profiler:** detects potential performance bugs.
- **Source control:** tracks changes/revisions to code.
- **Build automation**: compiles large code-bases (thousands of files)



Tools for C programming

- **Compiler:** translates C source code to machine code.
- **Linter:** warns about possible language misuse – bugs!
- **Linker:** separately-compiled files are "linked" together.
- **Debugger:** inspects compiled and running programs.
- **Memory tracer:** detects potential memory bugs.
- **Profiler:** detects potential performance bugs.
- **Source control:** tracks changes/revisions to code.
- **Build automation**: compiles large code-bases (thousands of files)



Tools for C programming

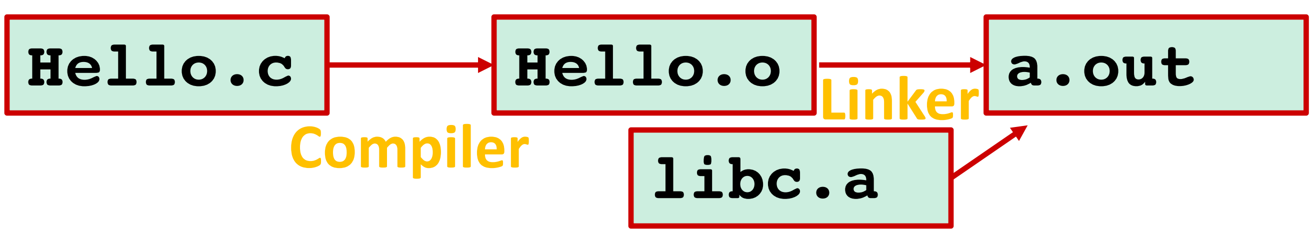
- **Compiler:** translates C source code to machine code.
- **Linter:** warns about possible language misuse – bugs!
- **Linker:** separately-compiled files are “linked” together.
- **Debugger:** inspects compi
- **Memory tracer:** detects p
- **Profiler:** detects potential
- **Source control:** tracks cha
- **Build automation**: compiles large code-bases (thousands of files)

File extension conventions in UNIX

.o “object file” (nothing to do with OOP)

.a static library

.so dynamic library



This involves **resolving** cross-object references.
Static vs Dynamic. We'll have a whole lecture on linking.

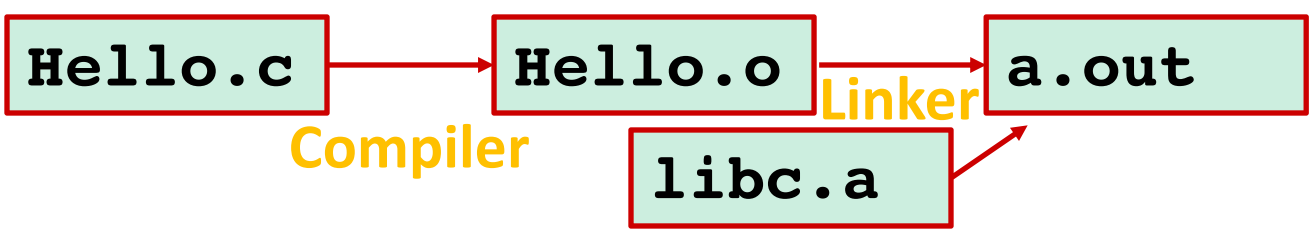
- **Com**[...]ates C source code to machine code.
- **Linte**[...]arns about possible language misuse – bugs!
- **Linker:** separately-compiled files are "linked" together.
- **Debugger:** inspects compi[...]
- **Memory tracer:** detects p[...]
- **Profiler:** detects potential[...]
- **Source control:** tracks cha[...]
- **Build automation**: compiles large code-bases (thousands of files)

File extension conventions in UNIX

.o "object file" (nothing to do with OOP)

.a static library

.so dynamic library



Tools for C programming

- **Compiler:** e.g., gcc, clang
- **Linter:** these days C compilers emit lint-like warnings**.**
- **Linker:** e.g., ld
- **Debugger:** e.g., gdb
- **Memory tracer:** e.g., valgrind
- **Profiler:** e.g., gprof
- **Source control:** e.g., git
- **Build automation**: e.g., make
- Other tools: editor, terminal multiplexer, test manager.

Tools for C programming

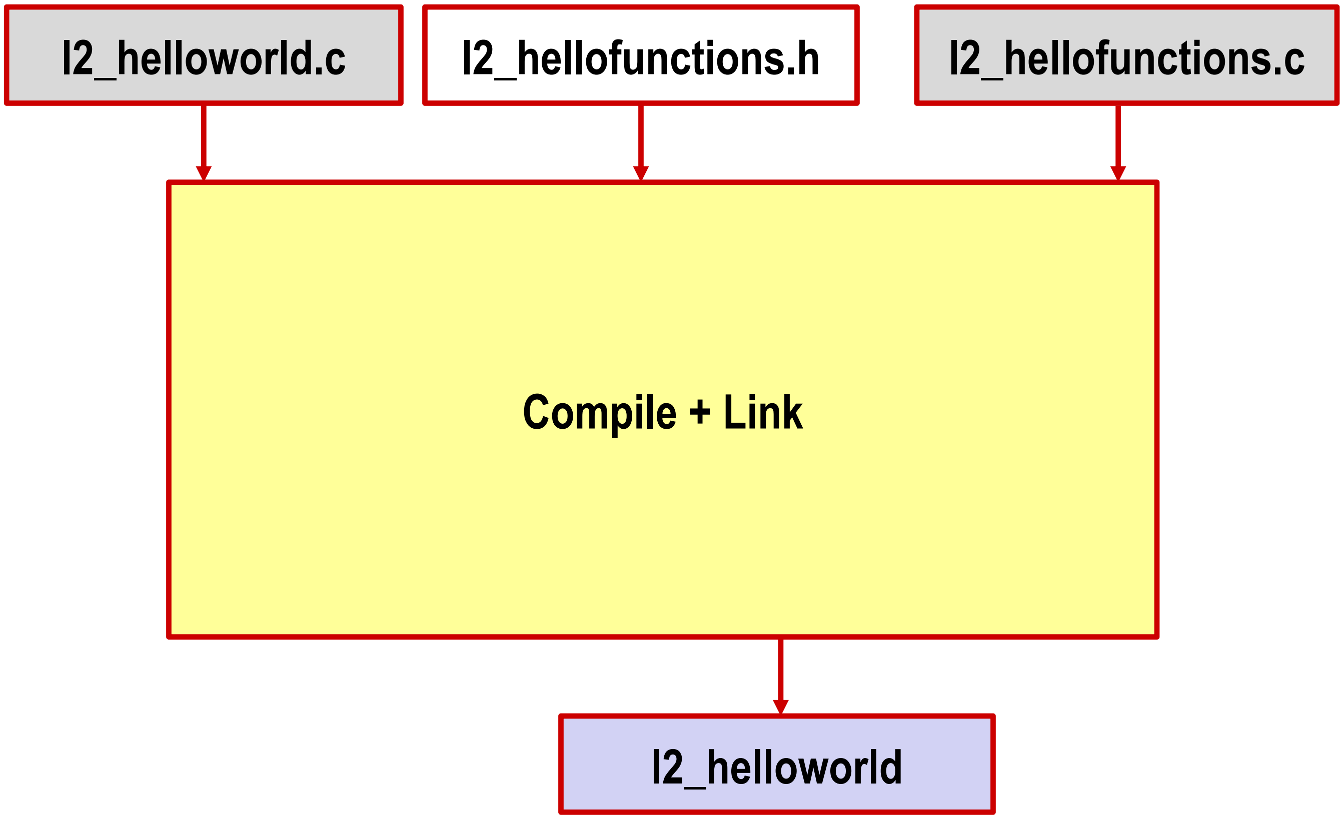
- **Compiler:** e.g., gcc, clang
- **Linter:** these days C compilers emit lint-like warnings.
- **Linker:** e.g., ld
- **Debugger:** e.g., gdb
- **Memory tracer:** e.g., valgrind
- **Profiler:** e.g., gprof
- **Source control:** e.g., git
- **Build automation**: e.g., make
- Other tools: editor, terminal multiplexer, test manager.

Let's look at an example workflow!

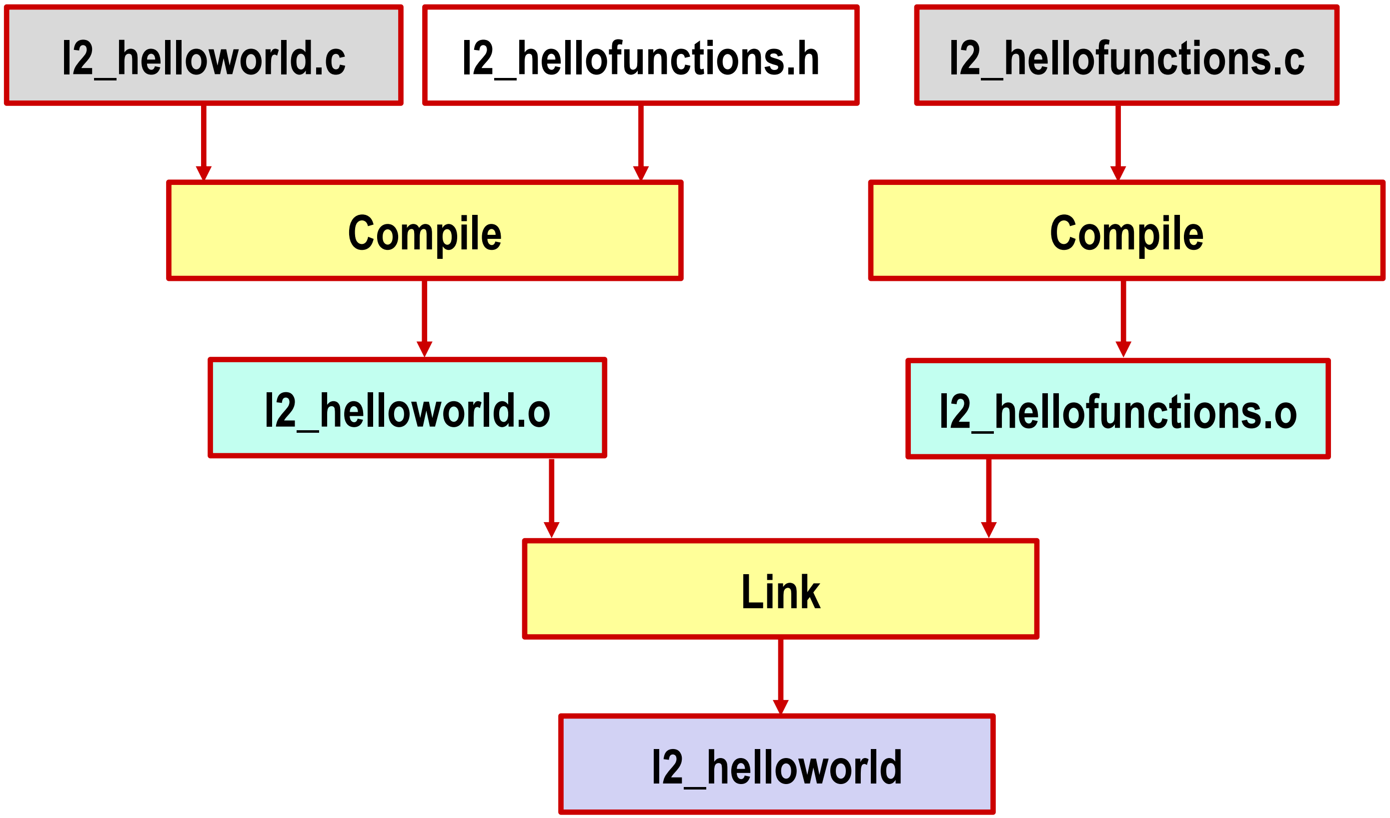
The classic starter program in C

- Print **"Hello world!"** to the terminal**.**
- The first lab assignment is a variation on this theme.
- We'll see the use of **language features**:
  - Types and variables
  - Functions
  - Control flow
  - IO
- We'll see the use of **tools**:
  - Compiler (**gcc**)
  - Memory tracer (**valgrind**)
  - Build tool (**make**)

Compiler driver hides intermediate steps



A different compilation flow



Bonus tools

- **man**: Display “manual page” for a function/program/command.
- Examples:
  - man man
  - man ldd
  - man printf
  - man syscalls
- Other bonus tools: **nm, ldd, objdump**

The classic starter program in C

- Print **“Hello world!”** to the terminal.
- The first lab assignment is
- We’ll see the use of **language features**:
  - Types and variables
  - Functions
  - Control flow
  - IO
- We’ll see the use of **tools**:
  - Compiler (**gcc**)
  - Linker (**ld**)
  - Debugger (**gdb**)
  - Memory tracer (**valgrind**)
  - Build tool (**make**)

That went by quickly but don’t worry!
Retry this in your first lab assignment.

Ask your TA if you’re stuck.

- And bonus **tools**:
  - Documentation (**man**)
  - Symbols (**nm**)
  - Dynamic dependencies (**ldd**)
  - Disassembler (**objdump**)
  - We saw **strace** last time.

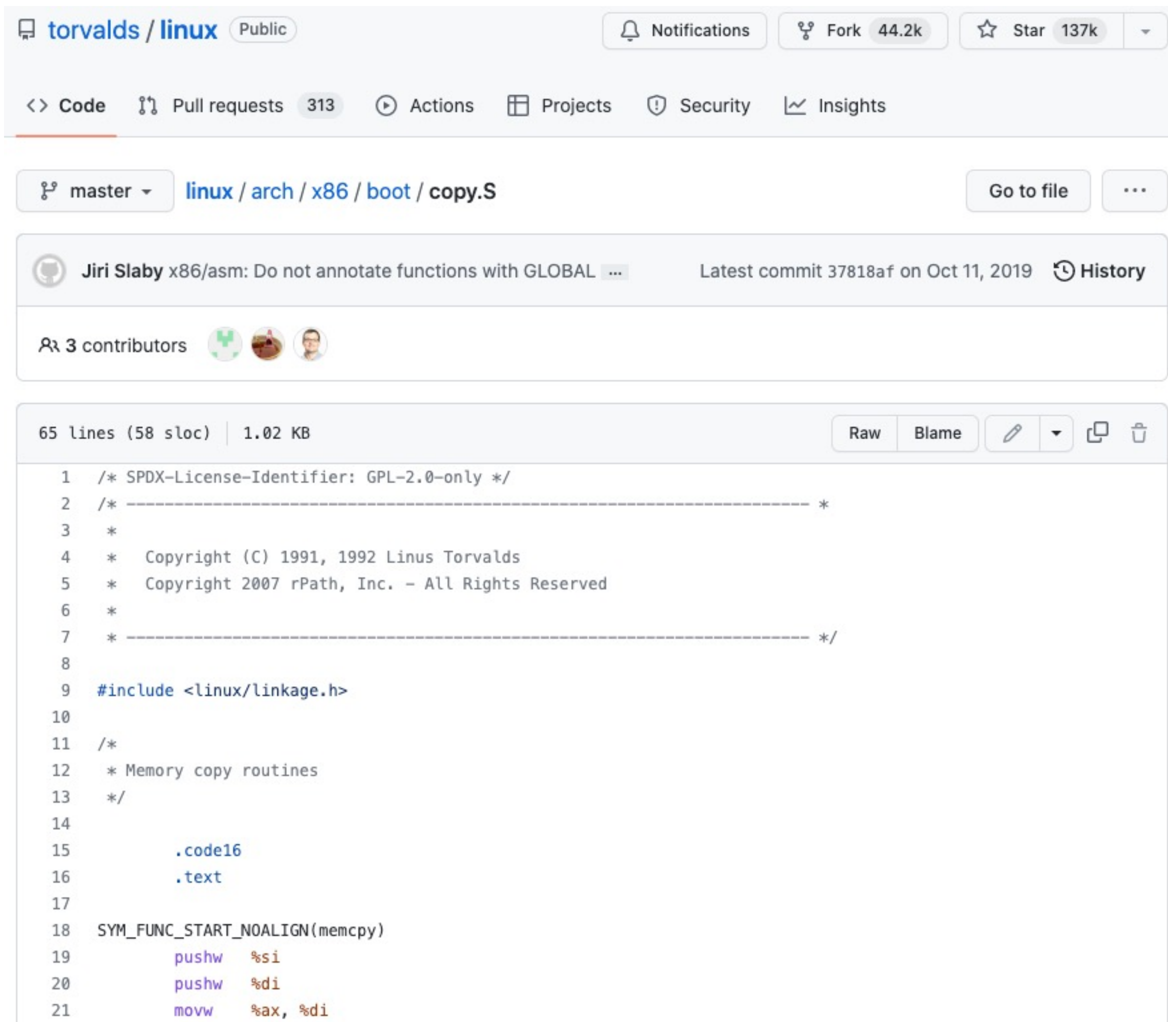
How to learn C?

- **There's only one way: by writing programs.**
  If you know Java, some of the syntax will be familiar.
- **Work through the K&R book.**
  (Copies in the library)
- **Attend labs and engage your TA.**
- **Do the exercises in the CS:APP3e book.**
  (Copies in the library)
- **We'll see and understand C source code in this course.**
  This'll show you the language "in action",
  but won't replace the need for you to practice writing C.

x86_64

Assembly Usage

- **Linux** https://github.com/torvalds/linux/blob/master/arch/x86/boot/copy.S




```
/* SPDX-License-Identifier: GPL-2.0-only */
/* ----------------------------------------------------------------------- *
 *
 *   Copyright (C) 1991, 1992 Linus Torvalds
 *   Copyright 2007 rPath, Inc. - All Rights Reserved
 *
 * ----------------------------------------------------------------------- */

#include <linux/linkage.h>

/*
 * Memory copy routines
 */

        .code16
        .text

SYM_FUNC_START_NOALIGN(memcpy)
        pushw   %si
        pushw   %di
        movw    %ax, %di
```

Assembly Usage

- **Quake** https://github.com/id-Software/Quake/blob/master/QW/server/math.s

id-Software / Quake Public

Notifications | Fork 771 | Star 3.7k

Code | Pull requests 3 | Actions | Security | Insights

master | Quake / QW / server / math.s | Go to file | ...

Travis Bradshaw The Quake sources as originally release und... ... Latest commit 0023db3 on Jan 31, 2012 History

0 contributors

331 lines (295 sloc) | 8.14 KB | Raw | Blame

```
//
// math.s
// x86 assembly-language math routines.

#include "asm_i386.h"
#include "quakeasm.h"


#if id386

        .data

        .align  4
Ljmptab:        .long   Lcase0, Lcase1, Lcase2, Lcase3
                        .long   Lcase4, Lcase5, Lcase6, Lcase7

        .text


#define EMINS   4+4
#define EMAXS   4+8
```

Overview of x86_64

- **“x86” refers to a CPU architecture designed by Intel.**
  It’s also used to refer to the architecture’s **instruction set.**
  It supports word sizes of 32/16/8 bits.
- **“x86_64” is a backwards-compatible extension by AMD.**
  It supports 64-bit words.
  “x86_64” is also referred to as **“amd64”.**
- Many Internet servers are currently based on x86_64 CPUs.
  (And these days fewer laptops.)

- Ok, so what is the **x86_64 instruction set?**

WARNING

Programming in assembly can be too much fun!

From Day 1

Abstractions?

- **It means many things!**
- **For an example, let's take "Hello, world!"**
- Three versions of the program: **Python vs C vs Assembly**
- They all give the same output! **~500** **~30** **~5**
- How do they differ in their abstractions?
- How do they differ in the resources required?

How did that difference come about?

What else is your C program doing?

```
[nsultana@fourier l1]$ strace ./l1_helloworld_c >\dev\null
execve("./l1_helloworld_c", ["./l1_helloworld_c"], 0x7ffe4c6f2350 /* 25 vars */) = 0
brk(NULL)                               = 0x2302000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd518cc9000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=47878, ...}) = 0
mmap(NULL, 47878, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd518cbd000
close(3)                                = 0
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`&\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2156664, ...}) = 0
mmap(NULL, 3985920, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fd5186db000
mprotect(0x7fd51889f000, 2093056, PROT_NONE) = 0
mmap(0x7fd518a9e000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c3000) …
mmap(0x7fd518aa4000, 16896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) …
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd518cbc000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd518cba000
arch_prctl(ARCH_SET_FS, 0x7fd518cba740) = 0
access("/etc/sysconfig/strcasecmp-nonascii", F_OK) = -1 ENOENT (No such file or directory)
access("/etc/sysconfig/strcasecmp-nonascii", F_OK) = -1 ENOENT (No such file or directory)
mprotect(0x7fd518a9e000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7fd518cca000, 4096, PROT_READ) = 0
munmap(0x7fd518cbd000, 47878)           = 0
fstat(1, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd518cc8000
write(1, "Hello, world!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

What else is your ASM program doing?

```
[nsultana@fourier l1]$ strace ./l1_helloworld_asm >\dev\null
execve("./l1_helloworld_asm", ["./l1_helloworld_asm"], 0x7fffb3b4ec50 /* 25 vars */) = 0
write(1, "Hello, world!\n", 14)         = 14
exit(0)                                 = ?
+++ exited with 0 +++
```

From Day 1

Do you see the abstractions?

```
%define NEWLINE 10

section .data
  message: db "Hello, world!", NEWLINE
  message_len: equ $-message

section .text
global _start

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, message_len
  syscall

  mov rax, 60
  mov rdi, 0
  syscall
```

Next time: You'll learn how to understand this.

Let's do that now!

From Day 1

Do you see the abstractions?

```
%define NEWLINE 10

section .data
  message: db "Hello, world!", NEWLINE
  message_len: equ $-message

section .text
global _start

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, message_len
  syscall

  mov rax, 60
  mov rdi, 0
  syscall
```

Next time: You'll learn how to understand this.

We'll use strace output to decipher what's happening

System calls

- **Invocation of OS-provided services.**
- **“man man”**
  we see: **“2   System calls (functions provided by the kernel)”**
- **“man 2 write”**
- **“man 2 exit”**

System calls

https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

master linux / arch / x86 / entry / syscalls / syscall_64.tbl Go to file ···

kvaneesh mm/mempolicy: wire up syscall set_mempolicy_home_node ... Latest commit 21b084f on Jan 14 History

22 contributors +10

419 lines (418 sloc) | 14.5 KB Raw Blame

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
⋮
57      common  fork                    sys_fork
58      common  vfork                   sys_vfork
59      64      execve                  sys_execve
60      common  exit                    sys_exit
61      common  wait4                   sys_wait4
62      common  kill                    sys_kill
```

What else is your ASM program doing?

```
[nsultana@fourier l1]$ strace ./l1_helloworld_asm >\dev\null
execve("./l1_helloworld_asm", ["./l1_helloworld_asm"], 0x7fffb3b4ec50 /* 25 vars */) = 0
write(1, "Hello, world!\n", 14)          = 14
exit(0)                                  = ?
+++ exited with 0 +++
```

From Day 1

Do you see the abstractions?

```
%define NEWLINE 10

section .data
  message: db "Hello, world!", NEWLINE
  message_len: equ $-message

section .text
global _start

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, message_len
  syscall

  mov rax, 60
  mov rdi, 0
  syscall
```

Next time: You'll learn how to understand this.

From Day 1

Do you see the abstractions?

```
%define NEWLINE 10

section .data
  message: db "Hello, world!", NEWLINE
  message_len: equ $-message

section .text
global _start

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, message_len
  syscall

  mov rax, 60
  mov rdi, 0
  syscall
```

What does the rest of it mean?
- How do we know to store 1 in “rax”
- What’s “rdi”, and what’s 1?

Next time: You’ll learn how to understand this.

System calls

- "System V Application Binary Interface: AMD64 Architecture Processor Supplement" pg 123 https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf Edited by Matz et al., 2012.

A.2.1 Calling Conventions

The Linux AMD64 kernel uses internally the same calling conventions as user-level applications (see section 3.2.3 for details). User-level applications that like to call system calls should use the functions from the C library. The interface between the C library and the Linux kernel is the same as for the user-level applications with the following differences:

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. The kernel interface uses `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.
2. A system-call is done via the `syscall` instruction. The kernel destroys registers `%rcx` and `%r11`.
3. The number of the syscall has to be passed in register `%rax`.

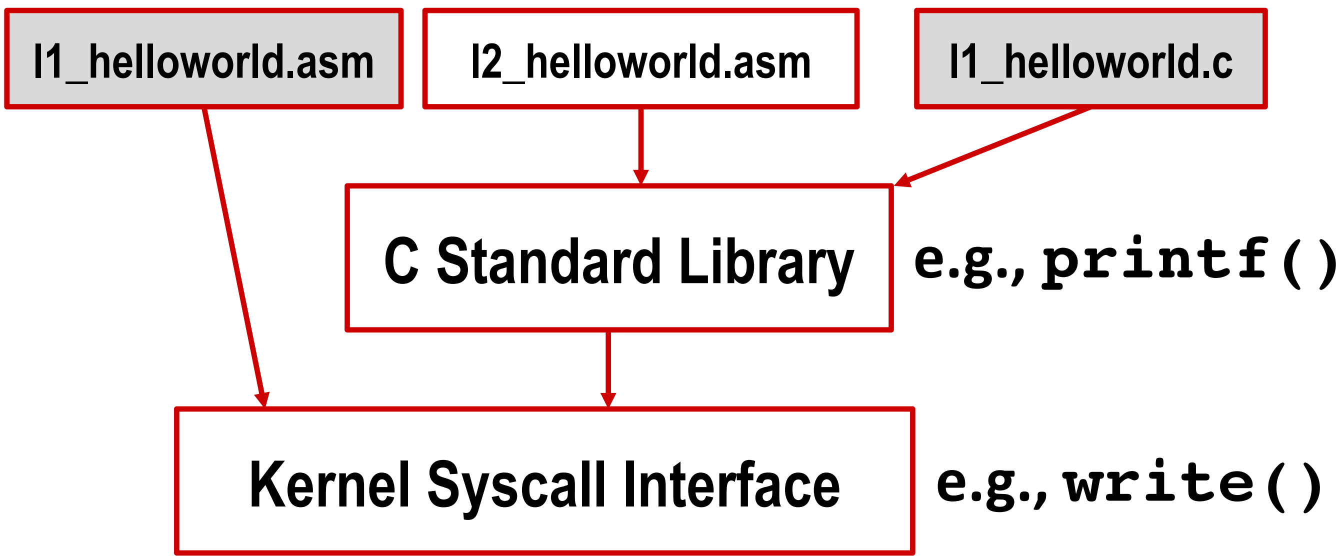
```
%define NEWLINE 10

section .data
  message: db "Hello, world!", NEWLINE
  message_len: equ $-message

section .text
global _start

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, message_len
  syscall

  mov rax, 60
  mov rdi, 0
  syscall
```



Above and beyond:
write & compile "Hello world" in C without using libc.

System calls vs Standard library

- **Functions made available by a programming language.**
- **Usually they wrap one/more syscalls.**
- **“man man”**
  we see: “**3   Library calls (functions within program libraries)**”
- **“man 3 printf”**

l2_helloworld.asm

A.2.1 Calling Conventions

The Linux AMD64 kernel uses internally the same calling conventions as user-level applications (see section 3.2.3 for details). User-level applications that like to call system calls should use the functions from the C library. The interface between the C library and the Linux kernel is the same as for the user-level applications with the following differences:

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. The kernel interface uses `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.
2. A system-call is done via the `syscall` instruction. The kernel destroys registers `%rcx` and `%r11`.
3. The number of the syscall has to be passed in register `%rax`.

l2_helloworld.asm

```
%define NEWLINE 10 ; '\n'

section .data
  message: db "Hello, world!", NEWLINE, 0

section .text

global main
extern printf

main:
  mov rdi, message
  sub rsp, 8
  call printf
  add rsp, 8
  ret
```

Ideas for "above and beyond"

(If you're up for a challenge)

- **Port the lab assignments** from C to another systems language, such as **Go** or **Rust**,
  or even to **x86_64** or **Aarch64**.
  Adapt the instructions for testing and debugging.

- **Port the Makefiles** to another build system,
  such as **Ninja** or **CMake**.
  Adapt the instructions for testing and debugging.

- There is no quantifiable academic credit for any of the above, but there's <u>non-zero good karma and learning</u>.

Your first CS351 Lab!

- **Make an effort to learn C and x86_64.**
  It will help you beyond this course.

Calendar

Monday	Tuesday
Aug 22	Aug 23 **LEC 1:** Introduction **Preparation:** Read CS:APP Chapter 1
Aug 29 **LAB**	Aug 30 **LEC 3:** Bits, Bytes, and Ints: Part 1 **Preparation:** Read CS:APP 2.1 **Assigned:** Lab 1: Preliminaries
Sep 05 Labor Day	Sep 06 **LEC 5:** Floating Point **Preparation:** Read CS:APP 2.4
Sep 12 **LAB** **DUE:** Lab 1 (Preliminaries)	Sep 13 **LEC 7:** Machine Prog: Control **Preparation:** Read CS:APP 3.6 **Assigned:** Lab 2: Datalab and Data Representations

Next steps

- Make sure that you can access **Fourier**.
- Once on Fourier, try out the C and assembly examples from the lectures.

(If you're not sure how to do any of the above, ask your TA)

Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- **The form is anonymous.** (It checks that you're at Illinois Tech to filter abuse, but I don't see who submitted any of the forms.)
- https://forms.gle/qoeEbBuTYXo5FiU1A
- I'll remind about this at each lecture.

Questions?