Operating Systems & Systems Programming I

Introduction

Keith Bugeja / Joshua Ellul / Alessio Magro / Kevin Vella

February 19, 2018

Department of Computer Science
Faculty of ICT, University of Malta

Welcome

Systems Programming Component

- Every other lecture (generally)
- Assignment-based assessment
- Weekly tutorials

Textbooks

- Advanced Programming in the UNIX Environment
  – W. Richard Stevens
- The C Programming Language
  – B. Kernighan and D. Ritchie
- Linux System Programming
  – R. Love

Systems programming is the practice of writing computer systems software:

- software or software platforms that provide services to other software, are performance constrained, or both
- code that lives at a low level, talking directly to the kernel and core system libraries
- examples: shell, compiler, debugger, disk defragmenter, game engine, web server, etc.

Application Programming

Application programming is the practice of writing application software:

- software designed to perform any specific task or activity for the benefit of the user
- occasionally (if ever) operates at low level by **directly** talking to the kernel or core system libraries
- examples: office productivity suites, media players, web browsers, etc.

During application software development:

- frameworks, language runtimes and engines typically abstract (or simplify) hardware and operating system specifics
- examples: Java Virtual Machine, .NET Framework, JavaScript interpreter, Unity3D

During application software development:

- frameworks, language runtimes and engines typically abstract (or simplify) hardware and operating system specifics
- examples: Java Virtual Machine, .NET Framework, JavaScript interpreter, Unity3D

However, someone still has to write the Java VM, the JavaScript interpreter or the game engine!

During application software development:

- frameworks, language runtimes and engines typically abstract (or simplify) hardware and operating system specifics
- examples: Java Virtual Machine, .NET Framework, JavaScript interpreter, Unity3D

However, someone still has to write the Java VM, the JavaScript interpreter or the game engine!

Systems software lies at the heart of all software!

Programming Landscape

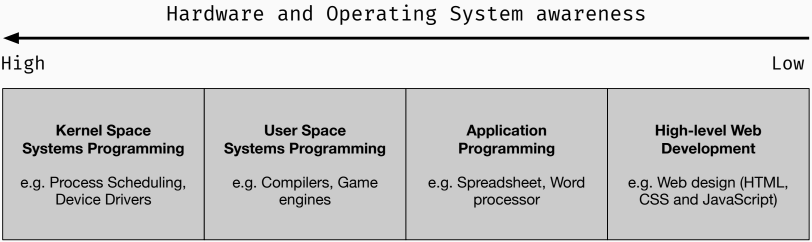


Figure 1: Operating system and hardware awareness spectrum

The unit focusses on Linux systems programming:

- user space low level programming (everything above the kernel)
- assume *some* familiarity with C programming and the Linux environment (e.g. shell, vim, gcc, gdb, make, etc.)

Note

Kernel and driver development are not addressed in this unit.

Linux

Two of the most famous products of Berkeley are LSD and Unix.
I don’t think that is a coincidence.

– The UNIX-HATERS Handbook

Unix Timeline

1969	Ken Thompson and Dennis Ritchie create **AT&T Unix**
1973	Unix Version 4 rewritten in **C** (developed by Ritchie)
1977	**BSD Unix** released
1982	AT&T first commercial UNIX system (System III)
1983	Richard Stallman launches **GNU** (GNU's not Unix)
1987	Andrew Tannenbaum creates **Minix**
1988	Stallman creates the **GNU GPL** (General Public Licence)
1991	Linux Torvalds creates i386 **Linux**
1992	Linux relicensed under the GPL
1994	Version 1.0 of Linux kernel is released
2001	**Mac OS X** released (based on BSD Unix)
2004	**Ubuntu** Linux distribution released
2008	**Android** mobile operating system released
2015	Version 4.0 of Linux kernel is released

The Open Group is an industry standards consortium that owns the **UNIX** trademark

- maintains the **SUS** (Single UNIX Specification) - a family of standards for operating systems
- systems fully compliant with and certified to SUS qualify as UNIX
- non-certified systems called Unix-like
- a UNIX vendor is required to pay substantial certification and annual trademark royalties

Unix Today

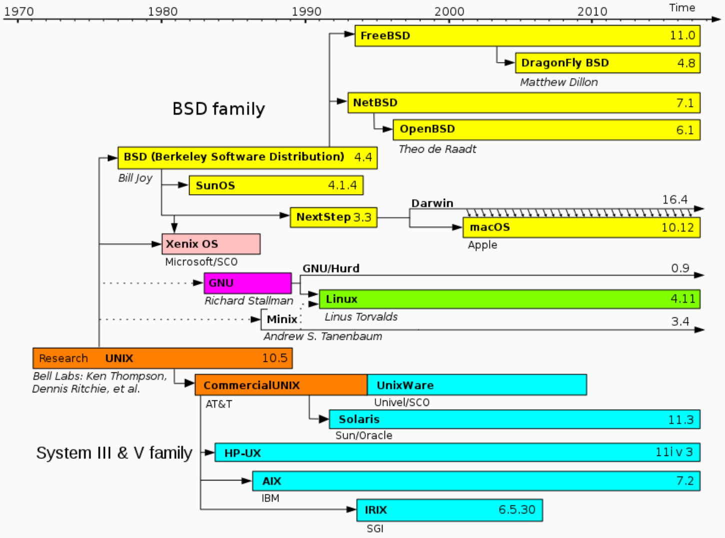
The Open Group is an industry standards consortium that owns the **UNIX** trademark

- maintains the **SUS** (Single UNIX Specification) - a family of standards for operating systems
- systems fully compliant with and certified to SUS qualify as UNIX
- non-certified systems called Unix-like
- a UNIX vendor is required to pay substantial certification and annual trademark royalties

Unix or UNIX?

Both forms are common, used interchangeably and grounded in ancient usage; we use **Unix** in deference to Dennis Ritchie's wishes.

Unix and Unix-like Systems



The structure of Unix

Characteristically, a Unix system is comprised of the following components:

Kernel The core of the operating system. It is responsible for the management of system resources and the abstraction of underlying hardware.

Shell The interface between the user and the kernel. It is a utility that processes user input and launches other utilities and applications.

Utilities Programs to help *analyse*, *configure*, *optimise* or *maintain* the system: compilers (`cc`), file management (`cp`), process management (`ps`), text processing (`vi`) et cetera (IEEE Std 1003.1-2008).

Applications Business applications, database management, media development, video games, etc.

The structure of Unix

In the context of systems software:

- ignore the *Applications* component
- adopt a layered view of the system
- top layer is users, bottom layer is hardware
- kernel and utility programs sandwiched in-between

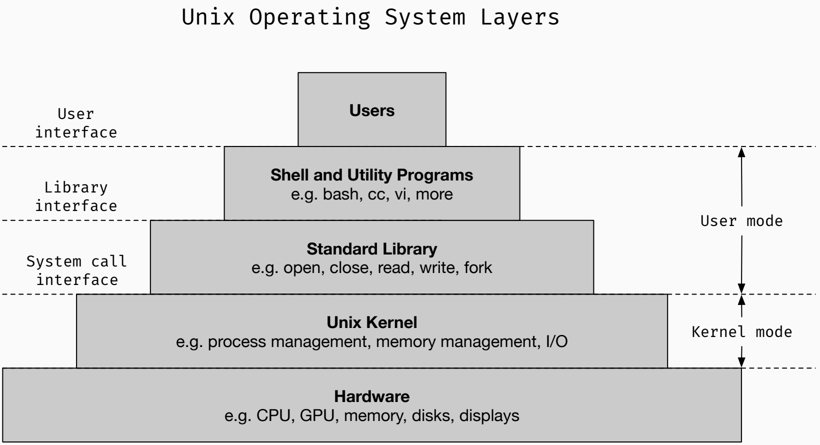


Figure 2: Typical layering of the Unix and Unix-like operating systems

Where does Linux fit in?

GNU Project started in 1984 with the goal of providing a **free** portable Unix-like operating system:

- **GNU software** replacement for Unix shells and utilities
  - GNU Compiler Collection (GCC)
  - GNU C library (glibc)
  - GNU Core Utilities (coreutils)
  - GNU Debugger (GDB)
  - GNU Binary Utilities (binutils)
  - GNU Bash shell
- **GNU Hurd**, a Unix-like microkernel designed as a replacement for the Unix kernel

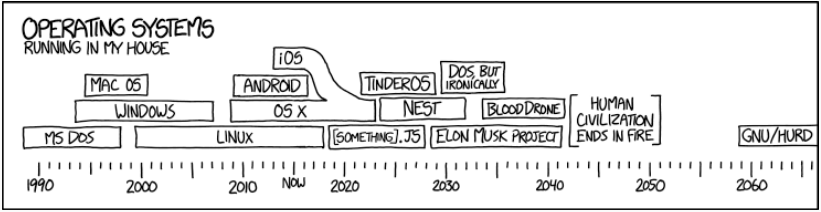


Figure 3: XKCD 1508 - Operating Systems

Where does Linux fit in?

GNU Project started in 1984 with the goal of providing a **free** portable Unix-like operating system:

- **GNU software** replacement for Unix shells and utilities
- **GNU Hurd**, a Unix-like microkernel designed as a replacement for the Unix kernel

Where does Linux fit in?

GNU Project started in 1984 with the goal of providing a **free** portable Unix-like operating system:

- **GNU software** replacement for Unix shells and utilities
- **GNU Hurd**, a Unix-like microkernel designed as a replacement for the Unix kernel
  - Not yet production-ready

Where does Linux fit in?

GNU Project started in 1984 with the goal of providing a **free** portable Unix-like operating system:

- **GNU software** replacement for Unix shells and utilities
- **GNU Hurd**, a Unix-like microkernel designed as a replacement for the Unix kernel
    - Not yet production-ready

GNU/Linux

The combination of GNU software and the Linux kernel, adopted in lieu of the GNU Hurd.

Linux Concepts

In Unix everything is either a file [descriptor] or a process:

- employs **everything-is-a-file** philosophy (though not as extensively as operating systems such as Plan9)
- a wide range of resources are exposed through the filesystem name space, even when the resource is not (what one would typically consider) a file
  - documents, directories, hard-drives, keyboards, printers, etc.
- resources that do not appear as part of the file system are still associated with a file descriptor that can be operated upon using a common set of primitives

The filesystem name space is arranged in a hierarchical structure:

```
/
├── home
│   ├── john
│   │   └── docs
│   │       ├── thesis.tex
│   │       └── thesis.pdf
│   └── mary
│       ├── code
│       │   └── main.cpp
│       └── docs
│           ├── image.jpg
│           └── thesis.pdf
├── bin
├── usr
└── …
```

A path is a position in the directory tree, which can be expressed either *relatively* or *absolutely*.

- A **relative path** depends on the current working directory:
    - e.g. `docs/thesis.pdf` may refer to either thesis file in John or Mary's home directory, or be ill-formed and point to no valid file
- An **absolute path** is unique and does not depend on the current working directory:
    - e.g. `/home/john/docs/thesis.pdf` unambiguously points to the thesis file in John's home directory

Linux Concepts

There are a number of special files worth mentioning:

- `.` the current directory
- `..` the parent directory
- `~` the current user's home directory

Note

Avoid using any of the following characters in file names: `@ # & ( ) ' ` " ; < > | * $? [ ]`

Unix commands can be issued through a terminal to be interpreted by the shell.

- When a terminal is launched, the user is presented with the command prompt:

```
keith@nilfgaard:~$
```

- The general form of a command is:

```
$ command [-option(s)] [argument(s)]
```

Unix commands can be issued through a terminal to be interpreted by the user's shell.

```
$ ls
Desktop      Documents  Music     Public     Tools
Development  Downloads  Pictures  Templates  Videos
$
```

Unix commands can be issued through a terminal to be interpreted by the user's shell.

```
$ ls
Desktop      Documents  Music     Public     Tools
Development  Downloads  Pictures  Templates  Videos
$
```

```
$ ls -la Tools
total 420072
drwxrwxr-x  4 keith keith      4096 Set 28 11:54 .
drwxr-xr-x 25 keith keith      4096 Jan 31 16:05 ..
drwxrwxr-x  8 keith keith      4096 Set  5 16:52 clion-2017.2.2
-rw-rw-r--  1 keith keith 317678557 Set  5 16:52 CLion-2017.2.2.tar.gz
-r--r--r--  1 keith keith  56375699 Set 28 11:54 VMwareTools-10.1.15-6627299.tar.gz
-r--r--r--  1 keith keith  56072885 Set  4 19:29 VMwareTools-10.1.6-5214329.tar.gz
drwxr-xr-x  9 keith keith      4096 Set 14 11:13 vmware-tools-distrib
$
```

The man pages...

```
$ man ls
```

```
LS(1)                              User Commands                              LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information about the FILEs (the current directory by default).  Sort entries alpha
       betically if none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..
 Manual page ls(1) line 1 (press h for help or q to quit)
```

Redirection operators in the shell:

- **>** output redirection
  - redirects standard output to a file or device

```
$ echo "Hello!" > myfile.txt
```

- **>>** append
  - append standard output to a file

```
$ echo "Another greeting!" >> myfile.txt
```

- **<** input redirection
  - redirects standard output to a file or device

```
$ sort < myfile.txt
```

- **|** pipe
  - pipe the output of first command to the input of second

```
$ cat myfile.txt | sort
```

/	**bin**	Essential command binaries the need to be available in single-user mode
	boot	Boot loader files, e.g. kernels, `initrd`
	dev	Essential device files, e.g., `/dev/null`
	etc	Host-specific system-wide configurations
	home	User home directories, containing saved files, personal settings, etc.
	lib	Libraries essential for binaries in `/bin` and `/sbin`
	media	Mount point for removable media such as CD-ROMs
	mnt	Temporarily mounted filesystems
	opt	Add-on/optional application software packages
	proc	Virtual filesystem documenting kernel and process status as files
	root	Home directory for the root user
	run	Runtime variable data: information about running system since last boot
	sbin	Essential system binaries
	srv	Data for services provided by this system
	tmp	Temporary files; often not preserved between reboots
	usr	(Multi-)user utilities and applications
	var	Variable files - files whose content is expected to change continually, e.g. logs

Figure 4: Linux Filesystem Hierarchy Standard

Fundamentals of Systems Programming

Three fundamental components to systems programming:

- C language compiler
- system calls
- C standard library

C language compiler

C Compiler

The standard C Compiler on Linux is provided by the GNU Compiler Collection (`gcc`):

- originally `gcc` was GNU's version of `cc`, the C Compiler
- processes input files through one or more of four stages, from source to executable

Note on C++

Although the lingua franca of systems programming on Unix is C, C++ can be used as a "better C". The GNU C++ Compiler is `g++`.

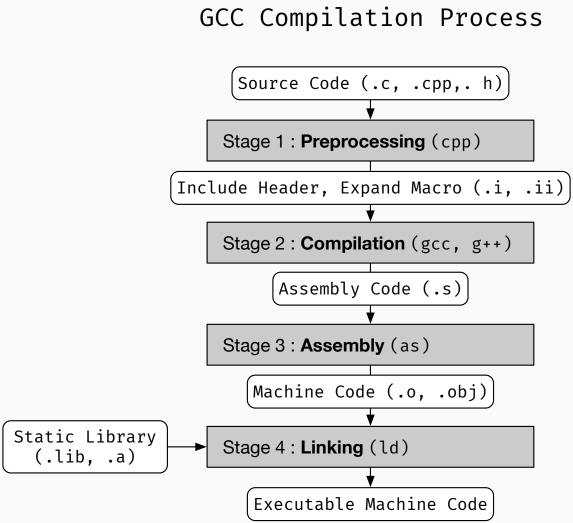


Figure 5: Four stages of processing, from source to executable

Let’s run a short C program through the four stages:

```c
#include <stdio.h>
#include <stdlib.h>

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

The first stage of compilation invokes the C preprocessor:

- substitutes include directives (`#include`) with the specified files into your program
- handles macro expansion (e.g. `#define`), replacing abbreviations for arbitrary fragments of C code with their definitions, throughout the program
- performs conditional compilation (`#if`, `#ifdef`, `#else`, etc.), where parts of the program can be included or excluded according to various conditions
- text processing and line control

GCC : Preprocessor

Let's run the example program through the C preprocessor:

```
$ # type in the code; save and quit using :x
$ vi hello.c
$ # execute the c preprocessor and redirect output to hello.i
$ cpp hello.c > hello.i
$ # compare the original hello.c with the expanded hello.i in
$ # terms of the number of lines of code
$ wc -l hello.c hello.i
    6 hello.c
 1870 hello.i
 1876 total
$ # optionally view the expanded source code
$ more hello.i
```

Another example (`pre.h` on next slide):

```
/* Download paste: https://pastebin.com/raw/awBWKQaL */
#include "pre.h"

#define TEST "This is a test!"

int main(int argc, char** argv) {
  /* A comment - the preprocessor will cull this line */

  /* Conditional compilation based on whether INVERT is defined */
#ifdef INVERT
  int num_1 = TWO,
      num_2 = ONE;
#else
  int num_1 = ONE,
      num_2 = TWO;
#endif

  int num_3 = MIN(num_1, num_2);
  int num_4 = MAX(num_1, num_2);

  int add_2 = add_two(num_3, num_4);
  int sub_2 = sub_two(num_3, num_4);

  /* While we're at it, let's define a string */
  char *str_1 = TEST;

  return 0;
}
```

```c
/* Download paste: https://pastebin.com/raw/3FJu73z2 */
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

#define ONE 1
#define TWO 2

int add_two(int a, int b) {
  return a + b;
}

int sub_two(int a, int b) {
  return a - b;
}
```

GNU Compiler Collection

```c
/* Download paste: https://pastebin.com/raw/3FJu73z2 */
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

#define ONE 1
#define TWO 2

int add_two(int a, int b) {
  return a + b;
}

int sub_two(int a, int b) {
  return a - b;
}
```

How is `pre.c` transformed by the C preprocessor?

- Try running `pre.c` through `cpp`.

GNU Compiler Collection

```
/* Download paste: https://pastebin.com/raw/3FJu73z2 */
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

#define ONE 1
#define TWO 2

int add_two(int a, int b) {
  return a + b;
}

int sub_two(int a, int b) {
  return a - b;
}
```

How is `pre.c` transformed by the C preprocessor?

- Try running `pre.c` through `cpp`.

Source code

Note that the first line of some listings contains a link to the source code on `Pastebin.com`.

GCC : Compiler

The second stage of compilation invokes the C compiler:

- transforms preprocessed C source code to the target language (e.g. assembly source or object code)
- many intermediate steps hidden within compilation: e.g. lexical analysis, parsing, semantic analysis, code optimisation and code generation (more in CPS2000)
- `cpp` and `gcc` constitute GCC's C language front-end

The second stage of compilation invokes the C compiler:

- transforms preprocessed C source code to the target language (e.g. assembly source or object code)
- many intermediate steps hidden within compilation: e.g. lexical analysis, parsing, semantic analysis, code optimisation and code generation (more in CPS2000)
- `cpp` and `gcc` constitute GCC's C language front-end

C preprocessor invocation

Note that the C preprocessor need not be directly invoked during compilation; `gcc` will invoke it automatically for files ending in `.c` and `.h`.

Let's run the preprocessed source through the C compiler:

```
$ # compile the preprocessed source and generate assembly
$ # language source code
$ gcc -S hello.i
$ # optionally view the assembly language source
$ more hello.s
```

Let's run the preprocessed source through the C compiler:

```
$ # compile the preprocessed source and generate assembly
$ # language source code
$ gcc -S hello.i
$ # optionally view the assembly language source
$ more hello.s
```

Note

The generated assembly language uses the AT&T assembler syntax. To use the Intel syntax, compile with the option `-masm=intel`.

GCC : Compiler

The assembly language translation of the “Hello, world!” program:

```
	.file	"hello.c"
	.section	.rodata
.LC0:
	.string "Hello, world!"
	.text
	.globl	main
	.type	main, @function
main:
.LFB5:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$16, %rsp
	movl	%edi, -4(%rbp)
	movq	%rsi, -16(%rbp)
	leaq	.LC0(%rip), %rdi
	call	puts@PLT
	movl	$0, %edi
	call	exit@PLT
	.cfi_endproc
.LFE5:
	.size	main, .-main
	.ident	"GCC: (Ubuntu 7.2.0-8ubuntu3) 7.2.0"
	.section	.note.GNU-stack,"",@progbits
```

The third stage of compilation invokes the assembler:

- creates object code by translating assembly mnemonics and syntax for operations and addressing modes into their numerical equivalents
- the GNU assembler (`as`) is the GCC's default back-end
- used to assemble GNU software and the Linux kernel

The third stage of compilation invokes the assembler:

- creates object code by translating assembly mnemonics and syntax for operations and addressing modes into their numerical equivalents
- the GNU assembler (`as`) is the GCC's default back-end
- used to assemble GNU software and the Linux kernel

Assembler invocation

Similarly to the C preprocessor, the assembler need not be directly invoked during compilation.

Let's run the assembly language source through the GNU assembler:

```
$ # assemble the code and generate an object file
$ as -o hello.o hello.s
```

Let’s run the assembly language source through the GNU assembler:

```
$ # assemble the code and generate an object file
$ as -o hello.o hello.s
```

And observe how the assembly language source has been transformed by the assembler:

```
$ # examine the object file (output on next slide)
$ objdump -s -d hello.o
```

GCC : Assembler

```
hello.o:     file format elf64-x86-64

Contents of section .text:
 0000 554889e5 4883ec10 897dfc48 8975f048  UH..H....}.H.u.H
 0010 8d3d0000 0000e800 000000bf 00000000  .=..............
 0020 e8000000 00                          .....
Contents of section .rodata:
 0000 48656c6c 6f2c2077 6f726c64 2100      Hello, world!.
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520372e  .GCC: (Ubuntu 7.
 0010 322e302d 38756275 6e747533 2920372e  2.0-8ubuntu3) 7.
 0020 322e3000                             2.0.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 25000000 00410e10 8602430d  ....%....A....C.
 0030 06000000 00000000                    ........

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
   f:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 16 <main+0x16>
  16:   e8 00 00 00 00          callq  1b <main+0x1b>
  1b:   bf 00 00 00 00          mov    $0x0,%edi
  20:   e8 00 00 00 00          callq  25 <main+0x25>
```

GCC : Linker

The fourth and final stage of compilation invokes the linker:

- takes one or more object files generated by a compiler and combines them into a single executable file, a library file or another object file
- relocates data and ties up symbol references
- `ld` is GNU's static linker for GCC

GCC : Linker

Let's link the example program object code into an executable file using the GNU linker:

```
$ # linker command calls take the following form:
$ # ld -o output inputs [additional options and libraries]
$ ld -o hello hello.o -e main -lc --dynamic-linker
  ↪  /lib64/ld-linux-x86-64.so.2
$ # run generated executable
$ ./hello
Hello, world!
```

GCC : Linker

Let's link the example program object code into an executable file using the GNU linker:

```
$ # linker command calls take the following form:
$ # ld -o output inputs [additional options and libraries]
$ ld -o hello hello.o -e main -lc --dynamic-linker
  ↪  /lib64/ld-linux-x86-64.so.2
$ # run generated executable
$ ./hello
Hello, world!
```

However, it may be safer to let `gcc` manage platform specific linker options, for maximum portability:

```
# Let gcc decide upon the platform specific inputs to the linker
$ gcc -o hello hello.o
```

Let's link the example program object code into an executable file using the GNU linker:

```
$ # linker command calls take the following form:
$ # ld -o output inputs [additional options and libraries]
$ ld -o hello hello.o -e main -lc --dynamic-linker
  ↪  /lib64/ld-linux-x86-64.so.2
$ # run generated executable
$ ./hello
Hello, world!
```

However, it may be safer to let `gcc` manage platform specific linker options, for maximum portability:

```
# Let gcc decide upon the platform specific inputs to the linker
$ gcc -o hello hello.o
```

Note

You can view the `gcc` linker invocation by specifying the verbose (`-v`) option: `gcc -v -o hello hello.o`

GNU Debugger : gdb

A very important tool in the systems programmer's toolbox is the debugger; we will be using the GNU debugger, `gdb`.

GNU Debugger : gdb

A very important tool in the systems programmer's toolbox is the debugger; we will be using the GNU debugger, `gdb`.

To facilitate the debugging process, instruct `gcc` to emit debugging information when compiling a program:

```
# Compile the program with debugging information in the operating system's native format (e.g. stabs, COFF, XCOFF, or DWARF)
$ gcc -g -o hello hello.c
# Alternatively, generate debugging information specifically for use with gdb
$ gcc -ggdb -o hello hello.c
```

A very important tool in the systems programmer's toolbox is the debugger; we will be using the GNU debugger, `gdb`.

To facilitate the debugging process, instruct `gcc` to emit debugging information when compiling a program:

```
# Compile the program with debugging information in the operating system's native format (e.g. stabs, COFF, XCOFF, or DWARF)
$ gcc -g -o hello hello.c
# Alternatively, generate debugging information specifically for use with gdb
$ gcc -ggdb -o hello hello.c
```

Running `gdb` will greet you with the `(gdb)` prompt:

```
# Run the GNU debugger
$ gdb hello
...
(gdb)
```

The source code of the *inferior* (program currently under debugging) can be shown using `list`:

```
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       int main(int argc, char **argv)
5       {
6               printf("Hello, world!\n");
7               exit(0);
8       }
(gdb)
```

GNU Debugger : gdb

The source code of the *inferior* (program currently under debugging) can be shown using `list`:

```
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       int main(int argc, char **argv)
5       {
6               printf("Hello, world!\n");
7               exit(0);
8       }
(gdb)
```

The command is very flexible and allows, amongst others, to list a range of line numbers:

```
(gdb) list 4,8
4       int main(int argc, char **argv)
5       {
6               printf("Hello, world!\n");
7               exit(0);
8       }
(gdb)
```

GNU Debugger : gdb

`gdb` may also provide an assembly language view of the inferior, by disassembling the program in memory:

```
(gdb) disassemble /s main
Dump of assembler code for function main:
hello.c:
5       {
   0x000055555555468a <+0>:     push   %rbp
   0x000055555555468b <+1>:     mov    %rsp,%rbp
   0x000055555555468e <+4>:     sub    $0x10,%rsp
   0x0000555555554692 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555554695 <+11>:    mov    %rsi,-0x10(%rbp)

6               printf("Hello, world!\n");
   0x0000555555554699 <+15>:    lea    0x94(%rip),%rdi        # 0x555555554734
   0x00005555555546a0 <+22>:    callq  0x555555554550 <puts@plt>

7               exit(0);
   0x00005555555546a5 <+27>:    mov    $0x0,%edi
   0x00005555555546aa <+32>:    callq  0x555555554560 <exit@plt>
End of assembler dump.
```

The `/s` option forces `disassemble` to include the C language source, if available.

GNU Debugger : gdb

Breakpoints for fine-grained debugging can be added through the `break` command; for example, to add a breakpoint to a function:

```
(gdb) break main
Breakpoint 1 at 0x699: file hello.c, line 6.
```

Alternatively, to break on a particular line:

```
(gdb) break 6
Breakpoint 1 at 0x699: file hello.c, line 6.
```

The `break` command also allows conditional breakpoints to be set up, where a breakpoint only triggers if the respective condition is true.

GNU Debugger : gdb

Compile and load the following program in `gdb`:

```c
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int sum = 0;
6
7      for (int index=0; index < 1000; ++index) {
8          sum += index;
9      }
10
11     printf("Sum is: %d\n", sum);
12 }
```

Set a breakpoint on line 8 that triggers if `index == 20`:

```
(gdb) break 8 if index == 20
Breakpoint 1 at 0x669: file break.c, line 8.
(gdb) run
Starting program: /home/keith/code/examples/break

Breakpoint 1, main (argc=1, argv=0x7fffffffdff8) at break.c:8
8                       sum += index;
(gdb) print index
$1 = 20
(gdb)
```

The `run` command executes a program until it either terminates, crashes or encounters a breakpoint:

```
(gdb) run
Starting program: /home/keith/code/examples/hello

Breakpoint 1, main (argc=1, argv=0x7fffffffdff8) at hello.c:6
6               printf("Hello, world!\n");
(gdb)
```

The `run` command will always start a program from the beginning, even if it is currently executing. When a breakpoint is triggered, execution can be resumed via the `continue` command:

```
(gdb) continue
Continuing.
[Inferior 1 (process 72047) exited normally]
(gdb)
```

Through the `step` and `next` commands, `gdb` allows the user to step through a program, returning control to the debugger after executing a specified number of statements:

```
(gdb) step
Hello, world!
7               exit(0);
(gdb)
```

While `next` treats a call to a subroutine as a unit, `step` considers the statements in the subroutine individually.

Through the `step` and `next` commands, `gdb` allows the user to step through a program, returning control to the debugger after executing a specified number of statements:

```
(gdb) step
Hello, world!
7               exit(0);
(gdb)
```

While `next` treats a call to a subroutine as a unit, `step` considers the statements in the subroutine individually.

Hint

Think of the respective behaviours of `step` and `next` as *step into* and *step over*.

A running program stores information about its execution in a data structure known as the **call stack**.

A running program stores information about its execution in a data structure known as the **call stack**.

- Each time a method is called, the program pushes a new **stack frame** on top of the call stack; it contains:
    - arguments passed to the method (if any)
    - local variables of the method (if any)
    - address to return after the method call finishes

A running program stores information about its execution in a data structure known as the **call stack**.

- Each time a method is called, the program pushes a new **stack frame** on top of the call stack; it contains:
    - arguments passed to the method (if any)
    - local variables of the method (if any)
    - address to return after the method call finishes

A **backtrace** is a list of the currently active function calls, starting from the current frame (*frame zero*), and so on up the call stack.

A running program stores information about its execution in a data structure known as the **call stack**.

- Each time a method is called, the program pushes a new **stack frame** on top of the call stack; it contains:
    - arguments passed to the method (if any)
    - local variables of the method (if any)
    - address to return after the method call finishes

A **backtrace** is a list of the currently active function calls, starting from the current frame (*frame zero*), and so on up the call stack.

`gdb` provides the `backtrace` function to help reason more clearly about the chain of events that caused a program to be in its current state.

GNU Debugger : gdb

Compile and load the following program in gdb:

```c
1  #include <stdio.h>
2
3  int fib(int n) {
4      const int f[] = {0, 1, 1};
5      return (n < 3) ? f[n] : fib(n-1) + fib(n-2);
6  }
7
8  int main(int argc, char **argv) {
9      int f40 = fib(40);
10     printf("F(40) = %d\n", f40);
11     return 0;
12 }
```

Set a breakpoint on line 5 that triggers if `n == 35`:

```
(gdb) break 5 if n == 35
Breakpoint 1 at 0x6da: file back_00.c, line 5.
(gdb) run
Starting program: /home/keith/examples/code/back_00

Breakpoint 1, fib (n=35) at back_00.c:5
5               return (n < 3) ? f[n] : fib(n-1) + fib(n-2);
(gdb)
```

Use the `backtrace` command to show all the stack frames

- passing `full` as an argument will also print the values of the local variables

```
(gdb) backtrace full
#0  fib (n=35) at back_00.c:5
        f = {0, 1, 1}
#1  0x00005555555546f8 in fib (n=36) at back_00.c:5
        f = {0, 1, 1}
#2  0x00005555555546f8 in fib (n=37) at back_00.c:5
        f = {0, 1, 1}
#3  0x00005555555546f8 in fib (n=38) at back_00.c:5
        f = {0, 1, 1}
#4  0x00005555555546f8 in fib (n=39) at back_00.c:5
        f = {0, 1, 1}
#5  0x00005555555546f8 in fib (n=40) at back_00.c:5
        f = {0, 1, 1}
#6  0x000055555555473d in main (argc=1, argv=0x7fffffffdff8) at back_00.c:9
        f40 = 0
(gdb)
```

`gdb` provides commands for examining the contents of stack frames

`info frame [N]` : displays information about the N^{th} stack frame; omitting N defaults to the current stack frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd60:
 rip = 0x5555555546da in fib (back_00.c:5); saved rip = 0x5555555546f8
 called by frame at 0x7fffffffddb0
 source language c.
 Arglist at 0x7fffffffdd50, args: n=35
 Locals at 0x7fffffffdd50, Previous frame's sp is 0x7fffffffdd60
 Saved registers:
  rbx at 0x7fffffffdd48, rbp at 0x7fffffffdd50, rip at 0x7fffffffdd58
(gdb)
```

gdb provides commands for examining the contents of stack frames

`info frame [N]` : displays information about the N^{th} stack frame; omitting N defaults to the current stack frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd60:
 rip = 0x5555555546da in fib (back_00.c:5); saved rip = 0x5555555546f8
 called by frame at 0x7fffffffddb0
 source language c.
 Arglist at 0x7fffffffdd50, args: n=35
 Locals at 0x7fffffffdd50, Previous frame's sp is 0x7fffffffdd60
 Saved registers:
  rbx at 0x7fffffffdd48, rbp at 0x7fffffffdd50, rip at 0x7fffffffdd58
(gdb)
```

`info locals` : displays the list of local variables and their values

```
(gdb) info locals
f = {0, 1, 1}
(gdb)
```

gdb provides commands for examining the contents of stack frames

`info frame [N]` : displays information about the N^{th} stack frame; omitting N defaults to the current stack frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdd60:
 rip = 0x5555555546da in fib (back_00.c:5); saved rip = 0x5555555546f8
 called by frame at 0x7fffffffddb0
 source language c.
 Arglist at 0x7fffffffdd50, args: n=35
 Locals at 0x7fffffffdd50, Previous frame's sp is 0x7fffffffdd60
 Saved registers:
  rbx at 0x7fffffffdd48, rbp at 0x7fffffffdd50, rip at 0x7fffffffdd58
(gdb)
```

`info locals` : displays the list of local variables and their values

```
(gdb) info locals
f = {0, 1, 1}
(gdb)
```

`info args` : displays the list of arguments and their values

```
(gdb) info args
n = 35
(gdb)
```

Change the previous program slightly by moving the definition of `f` outside the function `fib`:

```
...
const int f[] = {0, 1, 1};
int fib(int n) {
    return (n < 3) ? f[n] : fib(n-1) + fib(n-2);
}
...
```

Repeating the previous steps and printing a backtrace now gives:

```
(gdb) backtrace full
#0  fib (n=35) at back_01.c:5
No locals.
#1  0x0000555555554682 in fib (n=36) at back_01.c:5
No locals.
#2  0x0000555555554682 in fib (n=37) at back_01.c:5
No locals.
#3  0x0000555555554682 in fib (n=38) at back_01.c:5
No locals.
#4  0x0000555555554682 in fib (n=39) at back_01.c:5
No locals.
#5  0x0000555555554682 in fib (n=40) at back_01.c:5
No locals.
#6  0x00005555555546b3 in main (argc=1, argv=0x7fffffffdff8) at back_01.c:9
        f40 = 0
(gdb)
```

GCC Summary

Compiling a C program:

```
$ gcc -o myprog myprog.c
```

GCC Summary

Compiling a C program:

```
$ gcc -o myprog myprog.c
```

Compiling a C program with debug information:

```
$ gcc -g -o myprog myprog.c
```

GCC Summary

Compiling a C program:

```
$ gcc -o myprog myprog.c
```

Compiling a C program with debug information:

```
$ gcc -g -o myprog myprog.c
```

Launching the debugger:

```
$ gdb myprog
```

GCC Summary

Compiling a C program:

```
$ gcc -o myprog myprog.c
```

Compiling a C program with debug information:

```
$ gcc -g -o myprog myprog.c
```

Launching the debugger:

```
$ gdb myprog
```

Note

Most `gdb` commands can be abbreviated: `step` → `s`, `next` → `n`, `run` → `r`, `continue` → `c`, `backtrace` → `bt`, etc.

System Calls

System Call (syscall)

System calls are a mechanism through which a program requests some service from the operating system kernel

- A program running in user-space cannot execute kernel code or manipulate kernel data directly.
- Kernel provides a mechanism by which user-space programs can *signal* they wish to invoke a system call.
- The user-space program can *trap* into the kernel and execute code the kernel allows it to execute.

System Call (syscall)

On the i386 architecture:

- system call is signalled through a software interrupt (trap) with a value of `0x80`;
- desired system call number is stored in register `eax`;
- software interrupt handler is system call handler.

System Call (syscall)

A concrete example:

- `sys_write` is a Linux system call that writes a number of characters to a file descriptor

System Call (syscall)

A concrete example:

- `sys_write` is a Linux system call that writes a number of characters to a file descriptor

Syntax

```
ssize_t sys_write(unsigned int fd, const char *buf, size_t count)
```

System Call (syscall)

A concrete example:

- `sys_write` is a Linux system call that writes a number of characters to a file descriptor

Syntax

```
ssize_t sys_write(unsigned int fd, const char *buf, size_t count)
```

The system call number for `sys_write` is 4 (as of writing - could change in the future)

- takes output file descriptor (`fd`), character buffer (`buf`) and number of characters to write (`count`) as arguments
- arguments are expected in registers `ebx`, `ecx` and `edx` respectively

System Call (syscall)

```c
int main(int argc, char **argv)
{
  char *message = "This is a system call test!\n";
  int i; for (i = 0; message[i] != '\n'; i++);

  __asm__ volatile ("int $0x80"
    :
    // eax = 4 (syscall number)
    // ebx = 1 (fd = stdout)
    // ecx = message (pointer to string "This is a ...")
    // edx = length of string
    : "a" (4), "b" (1), "c" (message), "d" (i));

  return 0;
}
```

Let's type in, compile and execute the code:

```
$ # type in the code; save and quit using :x
$ vi syscall_00.c
$ # compile syscall_00.c; output to syscall
$ gcc -o syscall syscall_00.c
$ # execute syscall
$ ./syscall
This is a system call test!
$
```

Let's type in, compile and execute the code:

```
$ # type in the code; save and quit using :x
$ vi syscall_00.c
$ # compile syscall_00.c; output to syscall
$ gcc -o syscall syscall_00.c
$ # execute syscall
$ ./syscall
This is a system call test!
$
```

Hint

If you try this example and get no output in the terminal, try compiling with `gcc -static -o syscall syscall_00.c`.

Exercise: Change the system call example program to declare and initialise `message` and `i` as follows:

```c
    char message[] = "This is a system call test!\n";
    int i = sizeof(message);
```

Compile and run the program on 64-bit Linux

- Did you get the program to work?
- Did it require any additional changes?

System Call (syscall)

Let's look more closely at the executable we generated:

```
$ # get the size (disk usage) of the file in bytes
$ du -b syscall
8560    syscall
```

System Call (syscall)

Let's look more closely at the executable we generated:

```
$ # get the size (disk usage) of the file in bytes
$ du -b syscall
8560    syscall
```

An 8 kilobyte program to output a message that is no longer than 30 characters!

Let's look more closely at the executable we generated:

```
$ # get the size (disk usage) of the file in bytes
$ du -b syscall
8560    syscall
```

An 8 kilobyte program to output a message that is no longer than 30 characters!

What is our program linking against?

```
# list the shared object (libraries) dependencies
$ldd syscall
      linux-vdso.so.1 =>  (0x00007fff66348000)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6d0ffde000)
      /lib64/ld-linux-x86-64.so.2 (0x0000563182e67000)
```

What are these shared objects?

`linux-vdso.so.*` - virtual dynamic shared object

- a small shared library the kernel maps automatically into the address space of all user-space applications
- exists to reduce overhead of system calls that are called very frequently
- `man vdso`

`ld-linux.so.*` - the dynamic linker

- find and load the shared objects needed by a program, prepare the program to run and then run it
- `man ld.so`

`libc.so.*` - the C standard library

- a library of standard functions that can be used by all C programs
- `man libc`

But we're not using any C standard library functionality (AFAWK)!

- Can we force `gcc` to remove vestiges of the C standard library?

System Call (syscall)

But we're not using any C standard library functionality (AFAWK)!

- Can we force `gcc` to remove vestiges of the C standard library?

```
$ # do not use the standard libraries when linking (-nostdlib)
$ gcc -nostdlib -o syscall_nolibc syscall_00.c
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 000000000040010c
$ # it's the linker from before; what is this _start symbol it's complaining about?
$ # let's ignore it for now; check file size
$ du -b syscall_nolibc
1744    syscall_nolibc
$ # program is much smaller now; list shared object dependencies
$ldd syscall_nolibc
        not a dynamic executable
$ # no shared object dependencies; execute!
$ ./syscall_nolibc
This is a system call test!
Segmentation fault
$
```

System Call (syscall)

But we're not using any C standard library functionality (AFAWK)!

- Can we force `gcc` to remove vestiges of the C standard library?

```
$ # do not use the standard libraries when linking (-nostdlib)
$ gcc -nostdlib -o syscall_nolibc syscall_00.c
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 000000000040010c
$ # it's the linker from before; what is this _start symbol it's complaining about?
$ # let's ignore it for now; check file size
$ du -b syscall_nolibc
1744    syscall_nolibc
$ # program is much smaller now; list shared object dependencies
$ldd syscall_nolibc
        not a dynamic executable
$ # no shared object dependencies; execute!
$ ./syscall_nolibc
This is a system call test!
Segmentation fault
$
```

That didn't go too well

- Maybe, just maybe, we shouldn't have ignored the message...

Let's go back to the linker warning:

```
warning: cannot find entry symbol _start; defaulting to
 ↪  000000000040010c
```

Let's go back to the linker warning:

```
warning: cannot find entry symbol _start; defaulting to
 ↪  000000000040010c
```

The message is telling us three things:

1. it is looking for a method called `_start`, which it didn't find
2. `_start` is an entry symbol
3. whatever code is at address `0x000000000040010c` will be called instead

System Call (syscall)

Displaying the symbol table for the compiled program:

```
$ # Show syscall_nolibc's symbol table for .text section
$ objdump -t -j .text syscall_nolibc

syscall_nolibc:     file format elf64-x86-64

SYMBOL TABLE:
000000000040010c l    d  .text  0000000000000000 .text
000000000040010c g     F .text  0000000000000058 main
```

The symbol (a function) at the given address is the `main` function

- the linker wired the program entry point to the function
- the warning can be removed by explicitly setting the `main` function as entry point with the option `-e main`

System Call (syscall)

When the standard system start-up files and libraries are not used

- forfeit functionality including graceful exit to the operating system

System Call (syscall)

When the standard system start-up files and libraries are not used

- forfeit functionality including graceful exit to the operating system

Therefore, we should explicitly invoke the `exit` system call when the program terminates

- perform clean-up
- return control to operating system

```c
int main(int argc, char **argv)
{
    char *message = "This is a system call test!\n";
    int i; for (i = 0; message[i] != '\n'; i++);

    __asm__ volatile ("int $0x80"
        :
        : "a" (4), "b" (1), "c" (message), "d" (i));

    /* exit system call [eax = 1, ebx = exit code] */
    __asm__ volatile ("int $0x80"
        :
        : "a" (1), "b" (0));
}
```

System Call (syscall) - TAKE 2

Compiling and running the program produces no warnings and more importantly, no segmentation faults!

```
$ # do not use the standard libraries when linking (-nostdlib)
$ # set main as the program entry point (-e main)
$ gcc -nostdlib -e main -o syscall_nolibc syscall_01.c
$ # check file size
$ du -b syscall_nolibc
1528    syscall_nolibc
$ # execute
$ ./syscall_nolibc
This is a system call test!
$
```

C Standard Library

C Standard Library (libc)

The C standard library provides macros, type definitions and functions for:

- string handling
- mathematical compuations
- input/output processing
- memory management
- operating system services

It lies at the heart of all Unix and Unix-like operating systems

- generally cannot function if the C library is erased
- provides core services to a host of other languages, e.g. C++, D, Perl, Ruby, Python and Rust

Fortunately, using the C library we can perform system calls using the proper function:

```c
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char message [] = "This is a system call test!\n";
    write(STDOUT_FILENO, (const void*)message, sizeof(message) - 1);
    exit(0);
}
```

Questions?