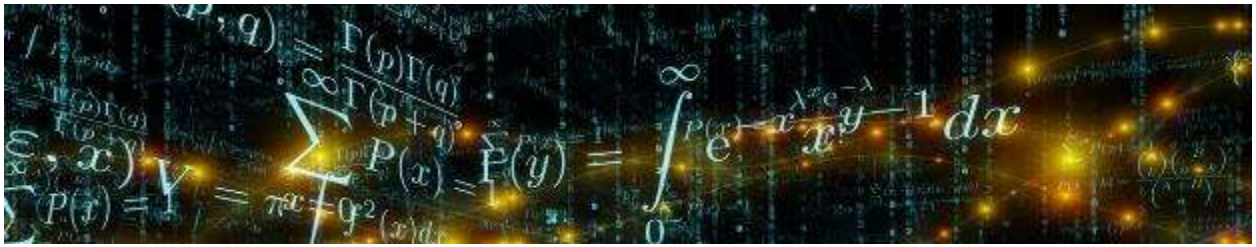
RUTGERS
UNIVERSITY | NEWARK

School of Arts & Sciences-Newark



Mathematics & Computer Science

RUTGERS
UNIVERSITY | NEWARK

School of Arts & Sciences-Newark

Intensive Programming in Linux
21:198:288 (3 credits)

Chapter 02 Types, Operators, and Expressions

C The Programming Language

Book By Brian W. Kernighan • Dennis M. Ritchie

Linux System Programming

Book By Robert Love

Computer Organization and Design THE HARDWARE / SOFTWARE INTERFACE

Book By David A. Patterson • John L. Hennessy

L1 Linux System Programming Tools –

Traditionally, all Unix programming was system-level programming. Unix systems historically did not include many higher-level abstractions, thus exposed in full view to the core Unix system API.

Where system programs interface primarily with the kernel and system libraries, application programs also interface with high-level libraries.

These libraries *abstract* away the details of the underlying hardware and operating system. Such abstraction goals:

- portability with different systems,
- compatibility with different versions of those systems, and
- the construction of higher-level toolkits that are easier to use, more powerful, or both.

L1 Linux System Programming Tools –

A trend in application programming away from

System-level programming and toward very high-level development.

There still exists the interpreter and the VM, which are themselves system programming.

The majority of Unix and Linux code is still written at the system level. Much of it is C and C++ and subsists primarily on interfaces provided by the C library and the kernel.

What is the system-level interface, and how do I write system-level applications in Linux? What exactly do the kernel and the C library provide?

There are three cornerstones of system programming in Linux: system calls, the C library, and the C compiler.

L1 Linux System Calls

System programming starts and ends with *system calls*. System calls (often shortened to *syscalls*) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) in order to request some service or resource from the operating system. System calls range from the familiar, such as read() and write(), to the exotic, such as get_thread_area() and set_tid_address().

In the Linux kernel, each machine architecture (such as Alpha, x86-64, or PowerPC) can augment the standard system calls with its own.

L1 Linux System Calls

Invoking system calls: It is not possible to directly link user-space applications with kernel space. For reasons of security and reliability, user-space applications must not be allowed to directly execute kernel code or manipulate kernel data. Instead, the kernel must provide a mechanism by which a user-space application can “signal” the kernel that it wishes to invoke a system call. The application can then *trap* into the kernel through this well-defined mechanism and execute only code that the kernel allows it to execute. The exact mechanism varies from architecture to architecture.

The application tells the kernel which system call to execute and with what parameters via *machine registers*.

Parameter passing is handled in a similar manner.

L1 Linux The C Library

The C library (*libc*) is at the heart of Unix applications. Even when you're programming in another language, the C library is most likely in play, wrapped by the higher-level libraries, providing core services, and facilitating system call invocation. On modern Linux systems, the C library is provided by *GNU libc*, abbreviated *glibc*, and pronounced *gee-lib-see* or, less commonly, *glib-see*.

The GNU C library provides more than its name suggests. In addition to implementing the standard C library, *glibc* provides wrappers for system calls, threading support, and basic application facilities.

L1 Linux The C Compiler

In Linux, the standard C compiler is provided by the *GNU Compiler Collection* (*gcc*). Originally, *gcc* was GNU's version of *cc*, the *C Compiler*. Thus, *gcc* stood for *GNU C Compiler*. Over time, support was added for more and more languages. Consequently, nowadays *gcc* is used as the generic name for the family of GNU compilers. However, *gcc* is also the binary used to invoke the C compiler.

The compiler used in a Unix system—Linux included—is highly relevant to system programming, as the compiler helps implement the C standard and the system ABI.

L1 Linux APIs

An API defines the interfaces by which one piece of software communicates with another at the source level.

It provides abstraction by providing a standard set of interfaces——usually functions—that one piece of software (typically, although not necessarily, a higher-level piece) can invoke from another piece of software (usually a lower-level piece).

The API merely defines the interface; the piece of software that actually provides the API is known as the *implementation* of the API.

L1 Linux ABIs

Whereas an API defines a source interface, an ABI defines the binary interface between two or more pieces of software on a particular architecture.

It defines how an application interacts with itself, how an application interacts with the kernel, and how an application interacts with libraries.

Whereas an API ensures source compatibility, an ABI ensures *binary compatibility*, guaranteeing that a piece of object code will function on any system with the same ABI, without requiring recompilation.

L1 Linux ABIs

ABIs are concerned with issues such as calling conventions, byte ordering, register use, system call invocation, linking, library behavior, and the binary object format.

The calling convention, for example, defines how functions are invoked, how arguments are passed to functions, which registers are preserved, and which are mangled, and how the caller retrieves the return value.

The ABI is defined and implemented by the kernel and the toolchain.