# Advanced Programming in the UNIX Environment — *Using UNIX*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

# Table of Contents



# UNIX Architecture I

- In a strict sense, an **operating system** can be defined as the software that controls the hardware resources of the computer and provides an environment under which programs can run. Generally, we call this software the **kernel**. Figure 1 shows a diagram of the UNIX System architecture.

# UNIX Architecture II

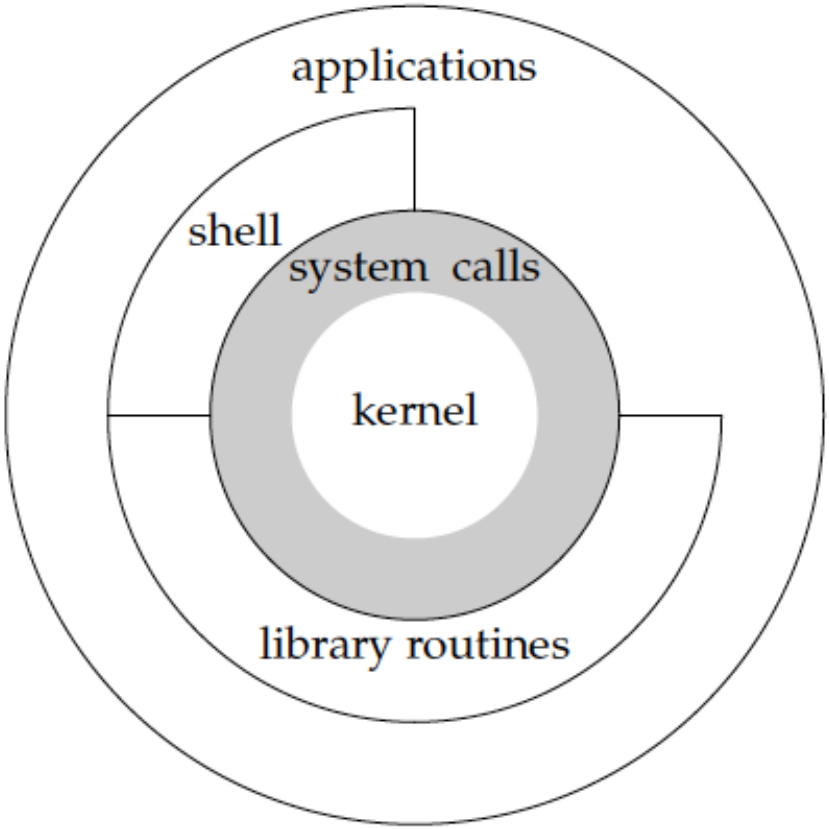


Figure: Architecture of the UNIX operating system

# UNIX Architecture III

- The interface to the kernel is a layer of software called the **system calls** (the shaded portion in Figure). **Libraries** of common functions are built on top of the system call interface, but **applications** are free to use both. The **shell** is a special application that provides an interface for running other applications.
- In a broad sense, an operating system consists of the kernel and all the other software that makes a computer useful and gives the computer its personality: system utilities, applications, shells, libraries of common functions, and so on.

# Login

- Before login into a UNIX system, you should have a valid account in that host.
- Install an Ubuntu/FreeBSD system on your own PC is a nice choice.
- For detail about using a UNIX (Linux) operating system please refer another lecture slides at room 2-301 13:30-15:20 each Friday in this semester.
- If you choose to installing a UNIX os on your own PC, you can operate under the console.
- If you choose to using the account that someone else provided to you, you must login the host through ethernet by an `ssh` client with the login account and the password provided by the owner of the host.

# Shell

- A *shell* is a command interpreter and a big UNIX program too that reads user input and executes commands.
- There are several popular shells like `bsh`, `csh`, `ksh`, etc.
- The last field of file `/etc/passwd` determine which kind of shell should be executed after login.
- The most frequently used shell under Linux and MacOS is `bash`. FreeBSD and Solaris often use `csh`.
- The shell was standardized in the POSIX 1003.2 standard. The specification was based on features from the Korn shell and Bourne shell.
- All examples in this book were tested under the Bourne shell, the Korn shell, and the Bourne-again shell environments.

# Files and Directories I

- There must be only one logical filesystem in an UNIX OS. The UNIX filesystem is a hierarchical arrangement of files.
- Files are placed in various directories. A directory is a special file that contains directory entries.
- All directories are arranged like a upset tree. The top most directory named **root**.
- The entries' name in a directory are called filenames. The only two characters that cannot appear in a filename are the slash character(/) and the null character.
- Two filenames are automatically created whenever a new directory is created: . and ..
- Dot refers to the current directory and dot-dot refers to the parent directory.

# Files and Directories II

- A sequence of zero or more filenames, separated by slashes, and optionally starting with a slash, forms a **pathname**. A pathname leading by a slash is named an **absolute pathname**, otherwise it's called a **relative pathname**.
- Example (Figure 1.3, `intro/ls1.c`).
- **Working Directory** is the directory from which all relative pathnames are interpreted.
- The second last field of file `/etc/passwd` denotes the **Home Directory** of each user. When we login successfully, Home Directory is our Working Directory.

# Input and Output

- UNIX kernel uses small nonnegative integers to identify the files being accessed by a particular process, named **file descriptors**.
- By default, all shells open three descriptors wherever a new program is run: `standard input`, `standard output` and `standard error`.
- All these three file are connected to your terminal by default.
- Example (Figure 1.4, `intro/mycat.c`).
- Example (Figure 1.5, `intro/getcputc.c`).

# Programs and Processes I

- A **program** is an executable file residing in a disk file. A program is read into memory and executed by the kernel as a result of one of the six `exec` functions.
- An executing instance of a program is called a **process**.
- Every UNIX process is guaranteed to have a unique nonnegative integer as numeric identifier called **process ID**.
- Example (Figure 1.6, `intro/hello.c`).
- Process ID 0 is usually scheduler process and is often know as the `swapper`. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure.
- There are three primary functions used for process control: `fork`, `exec` and `waitpid`.
- Example (Figure 1.7, `intro/shell1.c`).

# Error Handling I

- When an error occurs in one of the UNIX System functions, a negative value is often returned, and the integer `errno` is usually set to a value that gives additional information.
- Most functions that return a pointer to an object return a null pointer to indicate an error.
- The file `errno.h` defines the symbol `errno` and constants for each value that `errno` can assume.
- The first page of Section 2 of the UNIX system manuals, named `intro`, usually lists all these error constants. On Linux, the error constants are listed in the `errno` manual page.
- The traditional definition of `errno` is:

```
1 extern int errno;
```

# Error Handling II

Modern UNIX supports multithreaded access to errno by defining it as

```
1 extern int *_ _errno_location(void);
2 #define errno  (*_ _errno_location())
```

- There are two rules to be aware of with respect to `errno`.
  1. First, its value is never cleared by a routine if an error does not occur. Therefore, we should examine its value only when the return value from a function indicates that an error occurred.
  2. Second, the value of `errno` is never set to 0 by any of the functions, and none of the constants defined in `errno.h` has a value of 0.
- Two functions are defined by the C standard to help with printing error messages.

# Error Handling III

```
1 #include <string.h>
2 char *strerror(int errnum);
3 #include <stdio.h>
4 void perror(const char *msg);
```

- Figure 1.8 shows the usage of these two error functions.
- Instead of calling either `strerror` or `perror` directly, all the examples in this text use the error functions shown in Appendix B of *APUE, 3rd Edition*.
- Example (Figure 1.8, `intro/testerror.c`)

# User Identification I

- The **user ID** from our entry in the password file is a numeric value that identifies us to the system.
- This user ID is assigned by the system administrator when our login name is assigned, and we cannot change it.
- We call the user whose user ID is 0 either **root** or the **superuser**. The **superuser** has free rein over the system.
- Our entry in the password file also specifies our numeric **group ID**.
- Groups are normally used to collect users together into projects or departments. This allows the sharing of resources among members of the same group.
- There is also a group file that maps group names into numeric group IDs, usually `/etc/group`.

# User Identification II

- In addition to the group ID specified in the password file for a login name, most versions of the UNIX System allow a user to belong to additional groups.
- These **supplementary group IDs** are obtained at login time by reading the file `/etc/group` and finding the entries that list the user as a member.
- Example (Figure 1.9, `intro/uidgid.c`).

# Signals

- Signals are a technique used to notify a process that some condition has occurred.
- The process has three choices for dealing with the signal: ignore, default or custom.
- Many conditions generate signals: some special key stroke, `kill`(1) and `killall`(1) commands, `kill`(2), etc.
- Example (Figure 1.10, `intro/shell2.c`).

# Time Values

- Historically, UNIX systems have maintained two different time values:
  1. Calendar time, the number of seconds since the Epoch.
  2. Process time, the CPU resource used by process, measured in clock ticks.
- Command `time`(1) will give three time values of a running process: clock time, user CPU time, system CPU time.

# System Calls and Library Functions I

- All operating systems provide service points through which programs request services from the kernel.
- All implementations of the UNIX System provide a well-defined, limited number of entry points directly into the kernel called **system calls**.
- The technique used on UNIX systems is for each system call to have a function of the same name in the standard C library.
- For our purposes, we can consider the system calls as being C functions.
- We call the general-purpose functions available to programmers **library function**.
- Normally, we can replace the library functions, if desired, whereas the system calls usually cannot be replaced.

# System Calls and Library Functions II

- An application can call either a system call or a library routine. Also realize that many library routines invoke a system call.
- System calls usually provide a minimal interface, whereas library functions often provide more elaborate functionality.
- In this text, we'll use the term **function** to refer to both system calls and library functions, except when the distinction is necessary.

# System Calls and Library Functions III

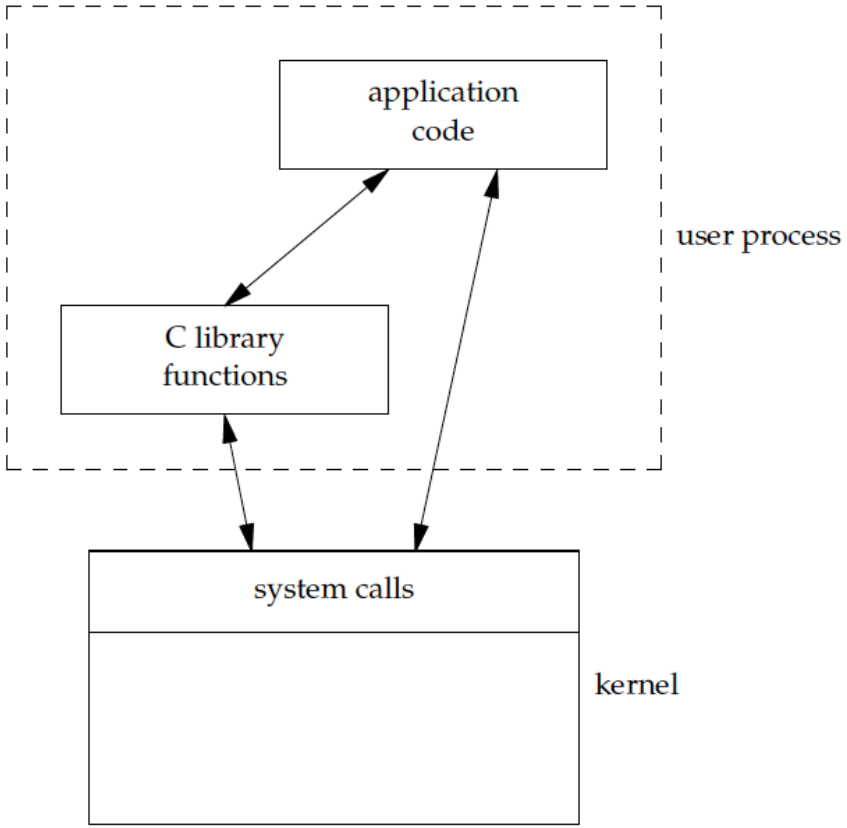


Figure: Difference between C library functions and system calls

# ISO C Features

- All the examples in these course are written in ISO C.

# The End

The End of Chapter 1.