System Programming
Chapter 00
Introduction to System Programming

Adlane Habed

Télécom Physique Strasbourg
Université de Strasbourg

Content

Operating Systems

A computer hardware cannot function without software.
In particular, a computer system must possess an operating system.

Example of services provided by an operating system :

- provide a framework for executing programs,
- share system resources (CPU, memory, disk) among programs and users,
- allow communication with devices (monitor, keyboard, network, etc.) and other programs,
- open a file, read from a file,
- get the time of the day, etc.

History of Unix

Unix started at Bell Laboratories

- 1969: Multics $\longrightarrow$ Unix by Ken Thomson,
  The first version was a primitive single-user one, written in assembly language.
  But it was more efficient and faster than Multics.
- 1970s: B $\longrightarrow$ C by Dennis Ritchie.
- 1973: Unix rewritten in C by Ritchie and Thomson.
- 1974: Unix Time Sharing System, the first Unix paper was published by Ritchie and Thomson.

Versions of Unix

Two popular versions of Unix

Unix (Thomson & Ritchie) $\longrightarrow$

- System V : from Bell Laboratories (V.2. . . V.4.1).
- BSD Unix - Berkeley Standard Distribution (4.2. . . 4.3): introduction of sockets and networking programming.

$\longrightarrow$ Sun OS (Solaris)

Features & philosophy of Unix

Unix features

- Unix is available for all platforms: PCs, minis and mainframes.
- Unix is among few operating systems that allow more than one user to share a computer system at a time.
- Unix is written in C and its source code is freely distributed among the community of users.
- Unix is simple and elegant

The philosophy of Unix

Unix has a simple philosophy :

- a program (utility) should do one thing and do it well
- a complex problem should be solved by combining multiple existing utilities.

$\longrightarrow$ Unix achieves this goal using pipes.

Unix Layers

Unix employs several layers:

1. The first layer is the **kernel**. It runs on the actual machine hardware and manages all interaction with it.
2. The second layer includes all **applications and Unix commands** which interact with the kernel rather than the hardware itself.
3. The third layer is called the **shell**. It interprets commands and manages the interaction between the user, the applications, and Unix commands.
4. The fourth layer is the **windowing system**. It usually interacts with the shell, but can also interact directly with applications.

The user

The user interacts with the entire operating system through either the shell or a combination of the shell and the windowing system.

Loggin in

`login`

- The `login` (`/bin/login`) command is invoked by the system. It is used at the beginning of each terminal session to identify oneself to the system.
- `login` checks our login-name in `/etc/passwd` and matches our password with the one in the encrypted file `/etc/shadow`.

Login-name and password

Example of an entry in `/etc/passwd`: `oconnel:x:1003:10:David O'Connell, M.Sc. Student:/users/oconnel:/bin/tcsh`
where

- `oconnel` is the login-name(or user-name),
- `x` used to be the encrypted password in the old version of Unix,
- `1003` is the user ID,
- `10` is the group ID,
- `David O'Connel` is a comment,
- `/users/oconnel` is the home directory,
- `/bin/tcsh` is the shell program used by this user.

The file `/etc/passwd` can be read by all users.
Example of an entry in `/etc/shadow`

```
oconnel:oaIL4MQMGaJBQ:::::::
```

The file `/etc/shadow` can only be read by *super users*.

User Identification (UID)

User ID

Each user is assigned a user ID, a unique nonnegative integer called uid.The user ID is used by the kernel to check if the user has the appropriate permissions to perform certain tasks like accessing files, . . . etc.

The superuser (root)

The user `root` (*superuser*) has uid=0. This user has special privileges, like accessing any file on the system.

getuid()

A process can obtain its uid using the system call `getuid()`.

```
#include <stdio.h> int main(void){
printf(‘‘hello, my uid is %d\n’’, getuid());
}
```

Group Identification (GID)

Group ID

The Group ID (gid) is a positive integer allowing to group different users into different categories with different privileges.
A group ID is also used by Unix for permissions verifications.
Both uid and gid are assigned by the system administrator at the time of the creation of the user account.
There is a group file, `/etc/group`, that maps group names into numeric IDs.

Example of entries in the `/etc/group` file:

```
sysadmin::14:steve,walid,anas,maunzer,mcdade,oracle
www::20:steve,walid,maunzer,anas,ai04,moodle,www
cs212::1020:danwu,walid,ejelike,chikker,sood8,uddin2
cs140::1021:steve,daemon
```

Shells

A *shell* is a command-line interpreter that reads and executes user commands.
A shell reads user inputs either from a terminal or, from a file called *script file*.

Example

Some existing shells:

- the Bourne shell, /bin/sh
- the C shell, /bin/csh
- the Korn shell, /bin/ksh

Important

Unix is case sensitive.

Some useful shell commands

- `man`: find and display reference manual pages

  `man command_name`
- `who`: who is on the system
- `ps`: give details of user's processes
- `date`: write the date and time
- `lpr`: print out a file
- `lpq`: list the jobs queued for printing
- `passwd`: change your password
- `quota`: information on a user's disk space quota and usage
- `grep`: search file for a specified string or expression
- `more`: page through a text file
- `echo`: writes its arguments, separated by blanks and terminated by a newline, to the standard output

Pipes

What are pipes?

Pipes are a mechanism that allows the user to specify that the output of one program is to be used as the input of another program.
$\longrightarrow$ several programs can be connected in this fashion to make a pipeline of data flowing from the first process through the last one.

Example

`ps -e | grep netscape | more`
where the three commands mean :
`ps -e`: report status on every process now running
`grep`: search a file for a pattern
`more`: page through a text file

Files and Directories

Unix File System

The Unix file system is a hierarchical arrangement of files and directories.
The `root` directory is represented by the slash character (`/`).
A directory is a file that contains entries for files and directories.
When a new directory is created, two filenames are automatically created:

- `.` (called *dot*) that refers to the current directory
- `..` (called *dot-dot*) that refers to the parent directory.

`pathnames`

A `pathname` is made of filenames separated by slashes. If a `pathname` starts with a `/` then it is an absolute pathname, otherwise, it is a relative one.

Example

`/export/home/users/zebra/set.txt` (absolute)
`images/city/park.jpg` (relative)

Some useful related shell commands

- `cd`: change the current directory
- `pwd`: return working directory name
- `ls`: list contents of directory
- `mkdir`: create a directory
- `rmdir`: remove a directory
- `cat`: concatenate and display files
- `cp`: copy files
- `rm`: remove files (and directory entries)
- `mv`: move files
- `find`: search for files in a named directory and all its subdirectories
- `sort`: sort a text file
- `mailx`: for sending and receiving mail messages

A bare bones implementation of the `ls` command.

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

int main(int argc, char *argv[]){
  DIR *dp;
  struct dirent *dirp;

  if(argc==1)
    dp = opendir("./");
  else
    dp = opendir(argv[1]);

  while ( (dirp=readdir(dp)) != NULL)
    printf("%s\n", dirp->d_name);

  closedir(dp);
  exit(0);
}
```

Standard Input, Standard Output and Standard Error

stdin, stdout AND stderr

Whenever a new program is run, three file descriptors are opened :

- the standard input `stdin`, by default the keyboard,
- the standard output `stdout`, by default the monitor,
- the standard error `stderr`, by default the monitor.

Redirection

Shells provide means to redirect standard input and standard output. For example:

- `ls > outputFile.txt`, the outputs are stored in the newly created file `outputFile.txt` instead of the monitor,.
- `mailx someone@uwindsor.ca < myFile.txt`, the inputs for `mailx` come from the file `myFile.txt` instead of the keyboard.

Standard Input, Standard Output and Standard Error

```
#include <stdio.h>

int main(int argc, char *argv[]){
  FILE *fd;
  char c;

  if(argc==1)
    fd=stdin;
  else
    if((fd = fopen(argv[1], "r"))==NULL){
      fprintf(stderr, "Error opening %s, exiting\n", argv[1]);
      exit(0);
    }
  while( (c=getc(fd)) != EOF)
    putc(c, stdout);

  exit(0);
}
```

Programs and Processes

- A *program* is an executable file residing on a disk.
- A *process* is an executing (running) program, usually with a limited life-time.
  Note that sometimes a process is called *task*.
- A *process ID (PID)* is a unique nonnegative integer assigned by Unix, used to identify a process.

Example

Example of the `ps` command output:

```
  PID  TTY      TIME CMD
10258  pts/6    0:01 gs
 7478  pts/6    0:06 emacs-20
 5598  pts/6    0:01 csh
10184  pts/6    0:01 netscape
```

`ps` reports process status.

getpid() and getppid()

A process can obtain its PID by the `getpid()` system call.
A process can also obtain its parent ID PPID by the `getppid()` system call.

```
#include <stdio.h>

int main(void){

  printf("Hello, my PID is %d\n", getpid());
  printf("Hello, my PPID is %d\n", getppid());
  exit(0);
}
```

```
Shell-Prompt> a.out Hello, my PID is 11723
Hello, my PPID is 5598
```

Process Control

There are three primary functions (system calls) for process control :

- `fork`: allows an existing process to create a new process which is a copy of the caller
- `exec`: allows an existing process to be replaced with a new one.
- `wait`: allows a process to wait for one of its child processes to finish and also to get the termination status value.

Note

The mechanism of spawning new processes is possible with the use of `fork` and `exec`.

Signals

Signals are used to notify a process of the occurrence of some condition.

Example

For example, the following generate signals :

- A division by zero: the signal `SIGFPE` is sent to the responsible process that has three choices. Ignore the signal, terminate the process or, call a function to handle the situation.
- The `Control-C` key: generates a signal that causes the process receiving it to interrupt.
- The function `kill`: a process can send a signal to another process causing its death. Unix checks our permissions before allowing the signal to be sent.

System calls

A user process can invoke either a system call or a library function.

System calls

Operating systems provide entry points for programs to request services from the kernel.
Entry points are called system calls in Unix.
In particular :

- Each system call in Unix has an interface function, in the C standard library, with the same name that the user process invokes.
- The interface function then invokes the appropriate kernel service, using whatever technique is required on the system.

Library functions

- An interface function for a system call cannot be replaced, however, a library function, such as `strcpy`, can be rewritten by the user.
- For our purpose, a system call will be viewed as a regular C function. A library function might invoke a system call.

Summary

- Operating system: Software to manage computer resources, in particular,
  - it runs a program for a user
  - it allows communication with devices and processes
- A program is a file containing instructions
- A process is a program being executed
- Unix is a multi-user operating system
- Most of Unix is written in the C language
- The Unix philosophy is simple: a program should do one thing and do it well.
- Entry points in Unix are called system calls. They allow the user to get services from the kernel.