Lecture 1. Introduction

The operating system provides services for the applications running on it:
- executing a program;
- opening a file;
- allocating memory;
- etc.

This course describes the services provided by the UNIX operating system.

UNIX Architecture

The OS is the software controlling the hardware resources of the computer, providing a programming interface to applications. This software is layered into:
- the **kernel**, the part providing the interface to the hardware;
- **system calls**, providing the interface to the kernel for the applications;
- the **shell**, an interface to running other applications;
- the applications themselves, running under the OS.

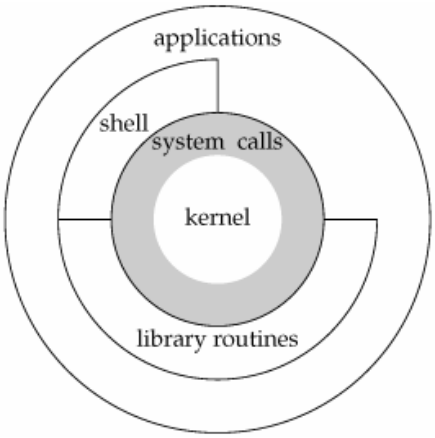

Figure 1.2. Common shells used on UNIX systems

Name	Path	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
Bourne shell	`/bin/sh`	•	link to `bash`	link to `bash`	•
Bourne-again shell	`/bin/bash`	optional	•	•	•
C shell	`/bin/csh`	link to `tcsh`	link to `tcsh`	link to `tcsh`	•
Korn shell	`/bin/ksh`				•
TENEX C shell	`/bin/tcsh`	•	•	•	•

Basics, Reviewed

The **login name** is stored in the `/etc/passwd` file. The **password** is stored, encrypted, in a different file, the `/etc/shadow` file.

The **shell** is a command-line interpreter that reads user input and executes commands. Below is a list of the common shells on UNIX systems:

- the Bourne shell, developed by Steve Bourne at Bell Labs;
- the C shell, developed by Bill Joy at Berkeley (available on BSD and System V);
- the Korn shell, developed by David Korn at Bell Labs (available on System V Release 4, or SVR4);
- the Bourne-again (bash) shell, developed for the Linux systems;

All the shells are standardized in the POSIX 1003.2 standard (where POSIX stands for Portable Operating System Interface, with the X added for pronounceability).

Files and Directories

The **filesystem** is a hierarchical organization of files and directories, where all starts at root, whose name is the forward slash character, `/`. Directories are files containing directory entries. These are filenames and information describing the attributes of a file:

- type (regular file, directory);
- size;
- owner;
- permissions;
- date of last modification.

The functions `stat()` and `fstat()` return information on all attributes of a file.

Filenames should be restricted, according to POSIX.1, to letters (a-z, A-Z), numbers (0-9), period (.), dash (-), and underscore (_).

Pathnames, or filenames preceded by the containing directories, can be absolute (starting from `/`), or relative (to a directory).

The following example is a simple implementation of the `ls` command on a directory, whose path is provided as an argument at the command-line:

```
$ ./myls /Users/awise/Stevens
```

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

int main(int argc, char *argv[])
{
     DIR *dp;
     struct dirent *dirp;
     if (argc!=2)
          printf("usage: myls directory_name");

     dp=opendir(argv[1]);
     if (dp==NULL)
          printf("can't open %s", argv[1]);

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

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

The command-line argument `argv[1]` is the path to the directory to be listed. The system function `opendir()` returns a pointer to a `DIR` structure, which is passed to the `readdir()` function. This is called in a loop, to read each directory entry and print its d_name struct member at the command-line.

Input and Output

File descriptors are small non-negative integers used by the kernel to identify the files accessed by a process. Upon opening an existing file or creating a new file, the kernel returns a file descriptor.

Three descriptors are opened whenever a new program is run: standard input, standard output, and standard error. The shell also provides a way to redirect standard output into a file. Using the previous example, we can issue the command:

```
$ ./myls ~awise >file.list
```

This will provide the directory name `~awise` to the program as the first command line argument, and will result in the names of all the files in the directory being printed at the command line.

Unbuffered I/O is provided by the functions `open()`, `read()`, `write()`, `lseek()`, and `close()`. The following program copies a file on a UNIX system:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <signal.h>
#include <unistd.h>

#define BUFFSIZE 4096

int main(int argc, char *argv[])
{
     int n;
     char buf[BUFFSIZE];

     while ((n=read(STDIN_FILENO, buf, BUFFSIZE))>0)
     {
          if (write(STDOUT_FILENO, buf, n)!=n)
               printf("write error\n");
     }
     if (n<0)
          printf("read error\n");
     exit(0);
}
```

The `read()` function takes as arguments a file descriptor, a pointer to a buffer, and the number of bytes to read. The `write()` function takes as arguments a file descriptor, a pointer to a buffer, and the number of bytes to write.

The constants `STDIN_FILENO` and `STDOUT_FILENO`, defined in `<unistd.h>`, are file descriptors for standard input and standard output. Their values are 0 and 1. The `read()` function returns the number of bytes read. When it reaches end of file, `read()` returns 0 and the loop condition becomes false. If a read error occurs, `read()` (like most system functions) returns -1.

To run this program:

```
$ ./mycp < foo > bar
```

This copies file foo (which is standard input for the `./mycp` command) into file bar (which is standard output for the `./mycp` command), creating bar if it doesn't exist.

Processes

A program is an executable file residing on disk in a directory. It gets loaded into memory and executed by the kernel, as a result of one of seven `exec()` functions.

An instance of a program is called a **process**. The UNIX system attaches to each process a unique numeric identifier, called the **process ID**, always a non-negative integer.

The following program calls `getpid()` to print its own process ID:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <signal.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
     printf("this is process ID %ld\n", (long) getpid());
     exit(0);
}
```

Process control is achieved primarily with one of three functions: `fork()`, `exec()`, and `waitpid()`.

The following program emulates a shell, reading commands from standard input and executing them:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAXLINE 4096

int main(int argc, char *argv[])
{
     char buf[MAXLINE];
     pid_t pid;
     int status;
```

```
     printf("%% "); /* print prompt */
     while (fgets(buf, MAXLINE, stdin)!=NULL)
     {
          if (buf[strlen(buf)-1]=='\n')
               buf[strlen(buf)-1]=0; /* replace newline with
null */

          if ((pid=fork())<0)
          {
               printf("fork error\n");
          }
          else if (pid==0) /* if child process */
          {
               execlp(buf, buf, (char *) 0);
               printf("couldn't execute: %s", buf);
               exit(127);
          }
          if ((pid==waitpid(pid, &status, 0))<0) /* if parent
process */
          {
               printf("waitpid error\n");
          }
          printf("%% ");
     }
     exit(0);
}
```

Function `fgets()` reads one line at a time from standard input (note its three arguments). When it reads the end-of-file character (`Ctrl-D`), `fgets()` returns a null pointer, and the while loop stops. Unless the user specifically types an end-of-file character, we don't want the loop to terminate. But on each line of input `fgets()` returns an end-of-line character, which we don't want passed on to `execlp()`. To get rid of it, the end-of-line character is set to a null byte in `buf[strlen(buf)-1]=0;`.

Function `fork()` creates a new process, a copy of the caller. The caller is called the **parent**, and the copy is the **child**. Function `fork()` returns the process ID of the child to the parent, and 0 to the child.

In the child process, function `execlp()` executes the new program (the command read from standard input). This replaces the child with the new program. The combination of `fork()` and `exec()` is called **spawning** a new process.

Function `waitpid()` is used with the child process ID as argument, so the parent process can wait for the child process to terminate. It also returns the `status` variable, which is the termination status of the child.

User Identification

The **user ID** is a number in the passwd file that uniquely identifies each user to the kernel. It is used by the kernel to determine permissions of that user to perform certain operations.

The **superuser**, or **root**, is the user with administration privileges on the system. Most file permission checks by the kernel are bypassed.

The **group ID** is also present in the passwd file, and is used to group a number of users into projects, departments etc. This allows the sharing of resources (files) among the members of the same group. The file that maps group names into numeric group IDs is `/etc/group`. Historically, the combined user and group IDs were mapped onto two-byte integers each, for a total of four bytes, or a word.

The following program prints the current user's user and group ID:

```
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for exit() */
#include <unistd.h> /* for getuid(), getgid() */

int main(void)
{
     printf("UID=%d, GID=%d\n", getuid(), getgid());
     exit(0);
}
```

Signals

Signals are integers that are passed on to the function `signal()` along with a function that should be called when the signal occurs. They are a mechanism to allow the program to terminate gracefully when an error condition is encountered. The process has three ways of dealing with a signal:

1. Ignore it (not recommended, as the signal may alert of conditions for which the outcomes are undetermined, such as dividing by 0, or referencing memory outside of the address space for the process);
2. Let the default action occur;

3. Provide a function which will be called when the signal occurs. This is called **catching the signal**.

The following is an improved version of the earlier program `myshell.c`, which catches the signal caused by the user pressing the interrupt key (Ctrl-C) and calls the function `sig_int()`:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAXLINE 4096

static void sig_int(int); /* the signal catching function */

int main(int argc, char *argv[])
{
     char buf[MAXLINE];
     pid_t pid;
     int status;

     if (signal(SIGINT, sig_int)==SIG_ERR)
          printf("signal error\n");

     printf("%% "); /* print prompt */
     while (fgets(buf, MAXLINE, stdin)!=NULL)
     {
          if (buf[strlen(buf)-1]=='\n')
               buf[strlen(buf)-1]=0; /* replace newline with
null */

          if ((pid=fork())<0)
          {
               printf("fork error\n");
          }
          else if (pid==0) /* if child process */
          {
               execlp(buf, buf, (char *) 0);
               printf("couldn't execute: %s", buf);
               exit(127);
          }
```

```
          if ((pid==waitpid(pid, &status, 0))<0) /* if parent
process */
          {
               printf("waitpid error\n");
          }
          printf("%% ");
     }
     exit(0);
}

void sig_int(int signo)
{
     printf("interrupt\n%% ");
}
```

System Calls and Library Functions

System calls are functions through which the programmer can access the kernel directly. There are of the order of 380 (Linux 3.2.0) to 450 (FreeBSD 8.0) system calls, depending on the operating system.

On UNIX systems system calls have the same name in the standard C library. The user process calls this function, using the standard C calling sequence. This C-like function invokes the appropriate kernel service.

Although they all look like C functions, there is a fundamental difference between system calls and library functions. For example, `printf()` uses the `write()` system call to output a string, but `strcpy()` and `atoi()` don't invoke the kernel at all. Library functions can be rewritten, whereas the system calls cannot.

Homework (due Tuesday, Feb-2-2016):

1. Write a two-page history of UNIX, from any resources you may find (the Internet, Richard Stevens' book etc.), which should include an enumeration and short description of your favorite flavor of UNIX.
2. Write a small system program that creates a file and writes your name in it, taking your name as input from the command line. It should use the system functions `creat()` and `write()`. For the synopsis of these functions, consult section 2 of the `man` pages on your system. Then, adapt the ./mycp program from earlier, to copy your newly created file into another file, without the redirect operators. To to that, you need to pass the source and the target files as command line arguments to your program.