The Design Of The
Unix Operating system

Apurba Sarkar

Department of Computer Science and Tech.
Bengal Engg. and Sc. University Shibpur.

July 24, 2012

Program vs Process

Definition

A program is an executable file. It is usually created by a link editor and reside on a disk file. The only way a program can be executed by the Unix system is by issuing `exec()` system call.

Definition

A process is an instance of a program that is being executed by the operating system. the only way a new process can be created by the Unix system is by issuing `fork()` system call.

- Also called task instead of a process.
- Multitasking operating system can execute more than one task(process) at a time.
- Multiple instance of the same program can be running at the same time.

System calls

Definition

A Unix kernel provide a limited number(typically between 60 and 200) of direct entry points through which an active process can obtain services from the kernel. These are named system calls.

- The C programmer, however, does not need to worry about the actual steps required to invoke each system call.
- This makes the actual system call appear as normal C functions to the programmer.
- Most system calls return -1 if an error occurs, or a value greater than or equal to zero if all is OK.
- Some system calls return a structure of information in addition to an integer value. eg `stat()` and `fstat()`.

- A C program normally starts execution with a function called `main`.

```
int main()
 {
      printf("hello world\n");
 }
```

- Most C compilers arrange for a special start-up function to be called when a C program is executed.
- This start-up function handles any initialization that is required and then calls the function `main`.

Argument List

- Whenever a program is executed, a variable length argument list is passed to the process.
- The list is an array of pointers to character strings.
- The upper bound on the size of the arg. list is typically 5120 or 10240 bytes.
  if we enter `echo hello world` to a Unix shell, the program `echo` is executed and is passed three argument string `echo`, `hello` and `world`.
- process is then free to do whatever it wants with these argument once it starts execution.

Environment list

- Whenever a program is executed,it is also passed a variable-length list of environment variables.
- the list is an array of pointers to character strings.
- there is no count of the number of elements in this array.
- it is terminated by a`NULL` pointer.
- the environment string are usually of the form *variable=string*.

Environment list contd

- the following C program prints the values of all environment strings

```
int main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{
int i;
for(i = 0; envp[i] != (char*) 0; i++)
printf("%s\n", envp[i]);
exit(0);
}
```

Environment list contd

output

```
HOME=/usr/apurba
SHELL=/bin/bash
COLORTERM=gnome-terminal
USER=apurba
PATH=/usr/lib/lightdm/lightdm:/usr/local/sbin:
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/games
PWD=/home/apurba
```

Environment list contd

- an equivalent program to the one above is

```
int main(argc, argv)
int argc;
char *argv[];
{
int i;
extern char **environ;
for(i = 0; environ[i] != (char*) 0; i++)
printf("%s\n", environ[i]);
exit(0);
}
```

why is this extra copy of the pointer to the environment list is provided????
what is the use of the environment list???

Environment list contd

- The following program prints the value of the environment variable HOME.

```
int main()
{
char *ptr,*getenv();
if( (ptr = getenv("HOME")) == (char *)0)
printf("HOME is not defined\n");
else
printf("HOME = %s\n", ptr);
exit(0)
}
```

Actually you can obtain the value of any variable using the function getenv() as above.

Process again

A process typically has the following things

- **text portion:** contains the actual machine instruction that are executed by the hardware. On some OS this portion is read-only. **why??**
- **data portion:** contains the program's data. It is possible for this to be divided into three pieces:
  - **Initialized read-only data:** contains data elements that are initialized by the program and are read-only while the process is executing. This area can be used for items such as literal strings that the programmer can initialize, but not change.
  - **Initialized read-write data** contains data elements that are initialized by the program and may have their values modified during execution of the process.
  - **Uninitialized data** data elements that are not initialized by the program but are set to zero before the process starts execution.

Process again continued

- the **heap** is used while a process is running to allocate more data space dynamically to the process.
- The **stack** is used dynamically while the process is running to contain the stack frames that are used by many programming languages. These stack frames contain the return address linkage for each function call and also data elements required by a function.

Process id

- Every process has a unique *process ID* or *PID*.
- The *PID* is an integer.
- It typically ranges from 0 through 30000.
- The kernel assigns the *PID* when a new process is created and a process can obtain its *PID* using the `getpid` system call.
  `int getpid();`
- The process with process ID 1 is a special process called the `init` process.
- Process ID 0 is also a special kernel process termed either the "swaper" or the "scheduler".

Parent Process ID

- Every process has a parent process, and a corresponding *parent process ID*.
- The kernel assigns it when a new process is created and a process can obtain its value using the `getppid()` system call.
  `int getppid();`

The following C program prints the PID and parent process ID of a process.

```
int main()
{
printf("pid = %d, ppid = %d\n",getpid(), getppid());
exit(0);
}
```

the output of the program could be

```
pid = 6731, ppid = 110
```

Real User ID

- Each user is assigned a positive integer *user ID*
- A process can obtain the real user ID of the user executing the process by calling by calling the `getuid()` system call.
- The file `/etc/passwd` maintains the mapping between login names and numeric user IDs.
- This user ID is used in the file system to record the owner of a file.
- There is a unique user ID assigned to each user.

Real User ID

- Each user is assigned a positive integer *user ID*
- A process can obtain the real user ID of the user executing the process by calling by calling the `getuid()` system call.
- The file `/etc/passwd` maintains the mapping between login names and numeric user IDs.
- This user ID is used in the file system to record the owner of a file.
- There is a unique user ID assigned to each user.

Real Group ID

- Along with numeric user ID, each user is also assigned a positive integer *group ID*.
- A process can obtain its real group ID by calling `unsigned short getgid();` system call.
- group ID is typically used to aggregate the users of a Unix system.
- Unlike user IDs, there are generally many users with the same group ID.
- The file `/etc/group` maintains the mapping between group names and and numeric group IDs.
- Each file also has group id of the owner of the file associated with it.
- user ID and group ID of a file is used by the system to grant access to the file.

Effective User Id

- Each process also has an *effective user ID*.
- A process can obtain it by `geteuid()` system call.
- Normally this value is the same as real user ID.
- It is however possible for a program to have a special flag set that says *"when the program is executed, change the effective user id of the process to be the user ID of the owner of this file"*. A program with this special flag set is called *set-user-ID* program
- When a program file has its *set-user-ID* bit set and the file's owner id is zero, we call this a "set-user-ID root" program.

Effective Group Id

- Each process also has an *effective group ID*.
- A process can obtain it by `getegid()` system call.
- Normally this value is the same as real group ID.
- A program can, however, have a special flag set that says *"when the program is executed, change the effective group id of the process to be the group ID of the owner of this file"*. A program with this special flag set is called *set-group-ID* program
- Like the *set-user-ID* feature, this provides additional permission to users while the *set-group-ID* program is being executed.

superuser

- User id zero is special - it defines the *superuser*.
- The login name for the superuser is usually *root*.
- The superuser is allowed unrestricted access to files and additional permission over the process.
  e.g it can terminate any other process in the system.
- A process with an effective ID of zero is termed a superuser process.

Password File

- Each line in the `/etc/passwd` file has the following format:

  `login-name:encrypted-password:user-ID:group-ID:miscellany:login-directory:shell`

- The superuser is allowed unrestricted access to files and additional permission over the process.
  e.g it can terminate any other process in the system.
- A process with an effective ID of zero is termed a superuser process.

Password File

- The standard C library provides two functions to search the `/etc/passwd` file, looking for a matching user ID or login name.

```
#include <pwd.h>
struct passwd *getpwuid(int uid);
struct passwd *getpwnam(char *name);
```

Password File

- The header file pwd.h defines a structure with the following elements:

```
struct passwd {
    char *pwname;          /* login name */
    char *pw_passwd;       /* encrypted password */
    int pw_uid;            /* user-ID */
    int pw_gid;            /* group-ID */
    char *pw_age;          /* System V only;password age */
    char *pw_comment;      /* not used */
    char *pw_gecos;        /* miscellany */
    char *pw_dir;          /* login directroy */
    char *pw_shell;        /* shell */
 };
```

Group File

- Each line in the /etc/group file has the following format

  `login-name:encrypted-password:group-ID:user-list`
- The group name is the name of the group.
- *encrypted password* is used by *newgrp* command.
- the *group-ID* is the numeric group id.
- the *user-list* is the comma separated list of the login names allowed in this group.

Group File contd

- The handling of groups is different between System V and 4.3BSD.
- With 4.3BSD you can be a member of up to 16 group in addition to the group specified in your password file entry.
- When you login, the `/etc/group` file is scanned and you become a member of each group which contains your login-name in the *user-list*.
- We call this list of group IDs that you belong to the *group access list*.
- This list is used only for determining resource accessibility.

Group File contd

- 4.3BSD provides the following following function to initialize the group access list.
  `int initgroup(char *name, int basegid);`
- This function scans the `/etc/group` file and adds all the group which list the name to the group access list.
- The *basegid* is also included in the group access list. It is usually the group ID value found in the `/etc/passwd` file entry for the user.

Group File contd

- System V restricts you to belong to a single group at a time. The command
  `newgrp group-name`; is provided you to allow you to change your *real group ID* to the value associated with the *group-name* int the `/etc/group` file.
- If the *encrypted password* field is not blank and if your login name is not in the *user-list*, you are prompted for password.
- if `newgrp` command is executed without arguments, your identification returns to the group specified in the password file.
- System V manual discourages the use of passwords in the `/etc/group` file. Instead users who should have the privileges of the group should be named in the *user-list*.

Group File contd

- The Standard C library provides two functions to search the /etc/group file, looking for a matching group ID or group name.

```
#include<grp.h>
struct group *getgrgid(int gid);
struct group *getgrnam(char* name);
```

- the include file <grp.h> defines a structure with the following elements:

```
struct group {
char *gr_name;          /* group name */
char *gr_passwd;        /* encrypted password */
int gr_gid;             /* group-ID */
char **gr_mem;          /* array of ptrs to user-l
};
```

Shells

- A Unix *Shell* is a program that sits between an interactive user and the Kernel.
- Typical shells are command line interpreters that read commands from the user at a terminal and execute the commands.
- The Unix shells are more than command line interpreters - they are programming languages.
- One or more of the three following shell programs are typically found on a Unix system
  - the Bourne shell: `/bin/sh`
  - the KornShell: `/bin/ksh`
  - the C shell: `/bin/csh`

Filename

- Every Unix file, directory, or special file has a *filename*.
- Some Unix systems limit the filename to 14 characters, but 4.3 allows it up to 255 characters.
- ASCII character '$\backslash 0$' and '/' is not allowed.
- convention is not to use other special characters.

Pathname

- Pathname is a null terminated character string that is built from one or more filenames.
- The filenames in a pathnames are separated from one another with a slash('/').
- A pathname can optionally begin with a slash, indicating that the path begins at the root directory.(also called *absolute pathname*).
- Pathname that does not start with with a slash is called textitrelative pathname and the path begins at the current directory.
- The pathname consisting of a '/' by itself refers to the root directory.
- The string '`junk.c`' and '`doc/book/chapter1`' are relative pathnames and the string '`/usr/lib`' is an absolute pathname.

File Descriptor

- A *file descriptor* is a small integer used to identify a file that has been opened for I/O.
- The allowable values are from zero up to some maximum, depending on the system
- Older Unix system had a limit of 20 open files per process, which allowed file descriptor between 0 and 19.
- Many Unix programs, including the shells, associate file descriptor 0, 1, 2 with standard input, standard output, and standard error, respectively, of a process.
- File descriptor are assigned by the Kernel when the following system calls are successful:
  `open()`, `create()`, `dup()`, `pipe()`, and `fnctl()`.

File Access Permissions

- As described earlier, every process has four IDs associated with it
  - real user ID
  - real group ID
  - effective user ID
  - effective group ID
- Additionally, every process has the following attributes
  - owner's user ID (a 16-b)
  - owner's group ID (a 16-bit integer)
  - user-read permission(a 1 bit flag)
  - user-write permission(a 1 bit flag)
  - user-execute permission(a 1 bit flag)

File Access Permissions contd.

- Additionally, every process has the following attributes
  - group-read permission(a 1 bit flag)
  - group-write permission(a 1 bit flag)
  - group-execute permission(a 1 bit flag)
  - others-read permission(a 1 bit flag)
  - others-write permission(a 1 bit flag)
  - others-execute permission(a 1 bit flag)
  - set-user-ID (a 1 bit flag)
  - set-group-ID (a 1 bit flag)