1

Unix System Overview

★ Programs and processes

Program

- A program is an executable file residing on disk in a directory.
- A program is read into memory and is executed by the kernel as a result of one of the seven exec functions.

Process

- An executing instance of a program is called a process

Process Id

- The UNIX System guarantees that every process has a unique numeric identifier called the process ID.
- The process ID is always a non-negative integer.
- getpid() system call will return process id of the calling process.
  - It returns data in pid_t datatype, which can be easily casted to long for printf function.

Process Control

- There are three primary functions for process control: fork, exec, and waitpid.

fork

- We call fork to create a new process, which is a copy of the caller.
- We say that the caller is the parent and that the newly created process is the child.
- Then fork returns the non-negative process ID of the new child process to the parent, and returns 0 to the child.
- Because fork creates a new process, we say that it is called once—by the parent — but returns twice—in the parent and in the child.

• Threads and thread Id

Thread

- All threads within a process share the same address space, file descriptors, stacks, and process-related attributes.
- Each thread executes on its own stack, although any thread can access the stacks of other threads in the same process.

- Because they can access the same memory, the threads need to synchronize access to shared data among themselves to avoid inconsistencies.
- Like processes, threads are identified by ID (Thread ID)
- Thread IDs, however, are local to a process.
  - A thread ID from one process has no meaning in another process.
  - We use thread IDs to refer to specific threads as we manipulate the threads within a process.

Error Handling

⇒ 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 tells why.

⇒ The file <errno.h> defines the symbol errno and constants for each value that errno can assume.

⇒ There are two rules to be aware of with respect to errno.

- 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.
- 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:

1.
```
#include <string.h>

char *strerror(int errnum);
```
maps errnum, into an error message string and returns a pointer to the string

2.
```
#include <stdio.h>

void perror(const char *msg);
```
produces an error message on the standard error, based on the current value of errno

Error Recovery

⇒ The errors defined in <errno.h> can be divided into two categories: fatal and nonfatal.

- A fatal error has no recovery action.

  The best we can do is print an error message on the user ’s screen or to a log file, and then exit.
- Nonfatal errors, on the other hand, can sometimes be dealt with more robustly.

★ User Identification

User ID

- It is a numeric value that identifies user to the system.
- The user ID is normally assigned to be unique for every user.
- We call the user whose user ID is 0 either root or the superuser.
- password file maintains the mapping between login names and user IDs.

Group ID

- Groups are normally used to collect users together into projects or departments.
- This allows the sharing of resources, such as files, among members of the same group.
- There is also a group file that maps group names into numeric group IDs (/etc/group).
- With every file on disk, the file system stores both the user ID and the group ID of a file's owner.
- group file provides the mapping between group names and group IDs.
- We call the functions getuid and getgid to return the user ID and the group ID.

Supplementary Group Id

- 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 other groups.

★ 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 the signal.
- Let the default action occur.
- Provide a function that is called when the signal occurs (this is called ''catching'' the signal).

⇒ Many conditions generate signals.

- Two terminal keys:
  1. interrupt key: Control + C
  2. quit key: Control + \
  are used to interrupt the currently running process.

⇒ Another way to generate a signal is by calling the kill function.

- We can call this function from a process to send a signal to another process.

★ Time Values

⇒ Historically, UNIX systems have maintained two different time values:

1. Calendar time.
   - → This value counts the number of seconds since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC).
   - → These time values are used to record the time when a file was last modified, for example.
   - → The primitive system data type time_t holds these time values.

2. Process time.
   - → This is also called CPU time and measures the central processor resources used by a process.
   - → Process time is measured in clock ticks.
   - → The primitive system data type clock_t holds these time values.

⇒ UNIX System maintains three values for a process:

1. Clock time
   - → The clock time, sometimes called wall clock time, is the amount of time the process takes to run, and its value depends on the number of other processes being run on the system.

2. User CPU time
   - → The user CPU time is the CPU time attributed to user instructions.

3. System CPU time
   - → The system CPU time is the CPU time attributed to the kernel when it executes on behalf of the process.

⇒ The sum of user CPU time and system CPU time is often called the CPU time.

★ System calls and Library functions

System Calls

- → All implementations of the UNIX System provide a well-defined, limited number of entry points directly into the kernel called system calls.
- → Its definition is in the C language, no matter which implementation technique is actually used on any given system to invoke a system call.
- → The technique used on UNIX systems is for each system call to have a function of the same name in the standard C library.
- → The user process calls this function, using the standard C calling sequence.

↳ This function then invokes the appropriate kernel service, using whatever technique is required on the system.

Library functions

↳ Aren't entry points into the kernel, although they may invoke one or more of the kernel's system calls.