Process and signals

Introduction

1. Processes and signals form a fundamental part of the UNIX operating environment. They control almost all activities performed by UNIX computer system.

2. An understanding of how UNIX manages processes will hold any systems programmer, applications programmer or system administrator in good stead.

3. We will look at how processes are handled in the Linux environment and how to find out what computer is doing at any given time. And how to start and stop process, how to make process cooperates with each other, how to avoid zombies(in which parents died before children).

Process

- What is process?

  1. An address space with one or more threads executing within that address space, and the required system resources for those threads. (the Single UNIX specification, version 2)

  2. **A process is a program in execution and can identified by its unique PID (process identification) number.**

  3. The kernel controls and manages processes. Multiple processes are running and monitored by the Linux kernel, allocating each of the processes a little slice of the CPU in a way that is unnoticeable to the user.

- Structure

A process consists of the executable program, its data and stack, variables(occupying system memory), open files(file descriptor) and an environment.

UNIX allows many users to access the system at the same time. Each user can run many programs, or even many instances of the same program, at the same time. The system itself runs other programs to manage system resources and control user access.

Typically, **UNIX system will share code and system libraries between processes**, so that there is only one copy of the code in memory at any one time.

For example: two users, Neil and rick both run the command "grep" on the shell at the same time to look for different strings in different files. This produces out two processes, each has it own process identifier, **PID**. And one copy of program code of "grep" is loaded into the memory as read only, two users share this code. And the system library is also shared(remind you the shared library).

- What process is running?

  1. **ps command**

     ps prints out the information about the active processes. without options it only prints out the information about process associated with the controlling terminal.

     ```
     $ ps
       PID TTY          TIME CMD
     25089 pts/1    00:00:00 tcsh
     25115 pts/1    00:00:00 ps

     $ps -au
     USER       PID %CPU %MEM   VSZ  RSS TTY
     lchang   25089  0.0  0.5  2860 1380 pts/1
     lchang   25116  0.0  0.3  2856  896 pts/1
     ```

     Explanation: USER: user' name; PID: process identifier; CPU: cputime/real time percentage; MEM: percentage of memory occupied; VSZ: virtual size of the process; RSS: resident pages and amount of shared pages; TTY: terminal identifier; STAT: process state, R means runable, S means sleeping. START: starting time. TIME: cumulative execution time; COMMAND : command on shell.

2. **pstree command pstree** is another way to see what process are running and what process are child process.

```
$ pstree
init-+-atd
     |-3*[automount]
     |-bdflush
     |-cannaserver
     |-crond
     |-dhcpcd
     |-esd
     |-geyes_applet
     |-gmc
```

```
|-gnome-name-serv
|-gnome-smproxy
|-gnome-terminal-+-gnome-pty-helpe
|                |-tcsh
|                `-tcsh---pstree
|-identd---identd---3*[identd]
|-jserver
|-keventd
|-khubd
|-klogd
|-kreclaimd
|-kswapd
|-kupdated
|-lockd
|-login---tcsh---startx---xinit-+-.xin
|                               `-X
|-magicdev
|-mdrecoveryd
|-5*[mingetty]
|-panel
|-portmap
|-rpc.statd
|-rpciod
```

```
|-sawfish
|-sshd
|-syslogd
|-tasklist_applet
|-xemacs
|-xfs
|-xinetd
|-xscreensaver
`-ypbind---ypbind---2*[ypbind]
```

Explanation: pstree command displays all process as a tree with its root being the first process that runs, called init. If the user name is specified, then that user's processes are at the root of the tree.If a process creates more than one process of the same name, pstree visually merges the identical branches by putting them in square brackets and prefixing them with the number of times the process is repeated.

- Parent and child

1. System call and Process From shell running a program is done with **system calls**, that call for service from Kernel. And this the only way that a process can access the hardware of the system. There are number of system calls allow a process to be created, executed and terminated.

2. Create a new process from within a program.

   ```
   #include <sys/types.h>
   #include <unistd.h>

   pid_t fork(void)
   ```

   A new process is created with the **fork** system call. It creates a duplicate of the calling process. The new process is called the **child** and the process that create it is called the **parent**. Both processes share CPU, the child process has a copy of the parents' environment, open files, user identifier, current working directory and signals.

The call to **fork** in the parent returns the PID of the new child process. At the same time, the call return to the new child a 0 to indicate it is a child.

```
pid_t new_pid;
new_pid = fork();

switch(new_pid){
case -1: /* erro */
break;
case 0: /*we are child*/
break;
default:      /*we are parent*/
break;
}
```

3. Exercise: A program calls fork()

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

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

int main()
{
  pid_t pid;
  char *message;
  int n;

  printf("fork program starting\n");
  pid = fork();

  switch(pid){
    case -1:
      fprintf(stderr, "fork failed");
      exit(1);
    case 0:
      n=3;
      message = "Child";
      break;
    default:
      n=4;
      message = "Parent";
```

```
        break;
      }
    for(; n>0 ; n--){
      printf("%s %d\n", message, n);
      sleep(1);
    }
    exit(0);
  }
```

4. Waiting for the children

   The parent is programmed to go to sleep while waiting for the child takes care of details, such as handling pipes, background processing, redirection.

   ```
   #include <sys/types.h>
   #include <sys/wait.h>
   pid_t wait(int *stat_loc);
   ```

   wait causes the parent process to pause until one of its child processes is terminated. **If the wait is successful, it returns the PID of the child process that stopped and the child's**

exit status. The status information is written to location that “stat loc” points to.

If the parent doesn’t wait and child still exits, the child is put in a zombie state, it will stay in the state until either the parent call wait or the parent dies.

If the parent dies before child, the init process adopts the orphaned zombie processes.

The wait call is not only used to put parent go to sleep, also to make sure that the process terminates properly.

The exit status are defined in $< sys/wait.h >$ as followed:

```
WIFEXITED(stat_val): Non-zero if the child
                     is terminated normally.
WEXITSTATUS(stat_val): If WIFEXITED is
     non-zero, this returns the exit code.
WIFSIGNALED(stat_val):Non-zero if the child
```

```
             is terminated on an uncaught signal.
WTERMSIG(stat_val):If WIFSIGNAL is non-zero
                  this returns the signal number.
WIFSTOPPED(stat_val):Non-zero if the child
                           has stopped.
WSTOPPED(stat_val): If WIFSTOPPED is
  non-zero, this returns the signal number.
```

5. Exercise: A program calls wait()

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

int main()
{
  pid_t pid;
  char *message;
  int n;
  int stat_val;

  printf("fork program starting\n");
  pid = fork();

  switch(pid){
    case -1:
      fprintf(stderr, "fork failed");
      exit(1);
    case 0:
      n=3;
      message = "Child";
      break;
    default:
      n=2;
      message = "Parent";
      break;
    }
  for(; n>0 ; n--){
    printf("%s %d\n", message, pid);
    sleep(1);
  }

if(pid !=0){
 pid_t child_pid;
 child_pid = wait(&stat_val);
 printf("Child %d has finished\n",
                        child_pid);
 if(WIFEXITED(stat_val))
  printf("Child exit with code %d\n",
                   WEXITSTATUS(stat_val));
 else
    printf("Child terminate abnormally\n");
 }
exit(WEXITSTATUS(stat_val));
}
```

Signals

- What is signal?

  A signal is an event generated by the UNIX system in response to some condition, upon which a process may in turn take some action.

  Signals are generated by some error conditions, such as memory segment violations, floating point processor errors or illegal instructions. They are generated by the shell and terminal handlers to cause interrupts.

  Signals can also be explicitly sent from one process to another as a way of passing information or modifying behavior.

  Generally, **Signals can be generated, caught and acted upon, or ignored.**

  Signal names are defined by including the header file $< signal.h >$ as followed:

SIGABORT: Process abort;
SIGALRM: Alarm clock;
SIGFPE: Floating point exception
SIGHUP: Hangup
SIGILL: illegal instruction
SIGINT: Terminal interrupt.
SIGKILL: kill (can't be caught or ignored)
SIGPIPE: write on a pipe with no reader
SIGQUIT: Termianl quit
SIGSEGV: Invalid memory segment access
SIGTERM: Termination
SIGUSR1: User-defined signal 1
SIGUSR2: User-defined signal 2

- Programming interface of signal handling

  1. signal()

     ```
     #include <signal.h>
     void (*signal(int sig, void(*func)(int)));
     ```

     It says that **signal** is a function that takes two parameters, **sig** and **func**.

The signal is to be caught or ignored is given as argument **sig**. The function to be called when the specified signal is received is given as **func**. This function must take a single **int** argument(the signal received) and is of type **void**.
This function can be one of these two special values:

```
SIG_IGN: Ignore the signal
SIG_DFL : Restore the default behavoir
```

2. A program CtrlC

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

void ouch(int sig)
{
printf("Ouch! -I got the signal %d\n",sig);
(void) signal(SIGINT, SIG_DFL);
}
```

```
int main()
{
  (void) signal(SIGINT, ouch);
  while(1){
    printf("Hello World!\n");
    sleep(1);
  }
}
```

Explanation: the function ouch reacts to the signal which is passed in the parameter sig. And this function is called when a signal is occurred. It prints out a message, and then handle the signal SIGINT(by default generated by pressing Ctrl-C), reset it back to the default behavior.
The main function has an infinite loop of printing a message, unless a signal is received and handled by call for signal().
Execute this program to see what is happened.

3. sigaction()

```
#include <signal.h>
int sigaction(int sig, const struct
                 sigaction *act,
                 sigaction *oact);
```

The structure **sigaction** is used to define the actions to be taken on receipt of the signal specified by int **sig**. This structure include three fields:

```
void sa_handler/*function,SIG_DFL,SIG_IGN*/
sigset_t sa_mask /*signals to be blocked */
int sa_flags /*signal action modifier*/
```

4. A program contains a question

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

void question()
{
  printf("continue or quit?\n");
  sleep(5);
}

int main()
{
  struct sigaction act;

  act.sa_handler = question;
  sigemptyset(&act.sa_mask);
  act.sa_flags=0;

  sigaction(SIGINT, &act,0);

  while(1){
    printf("current time is:\n");
    system("date");
    sleep(1);
  }
}
```

Explanation: This program gives user choice to stop printing local time by call for “sigaction” to handle the signal SIGINT(Ctrl-C). The sigaction contains a structure “act”, whose handler is the function "question", where it asks user willing to continue or not, and waits 5 seconds for user's answer. Type "Ctrl- " to Quit this program.

- Sending Signals

  1. kill()

     ```
     #include <sys/types.h>
     #include <signal.h>
     int kill(pid_t pid, int sig);
     ```

     The **kill** function sends the specified signal, **sig**, to the process whose identifier is given by **pid**. It returns 0 on success. It can fail if the program doesn't have permission to send the signal, commonly because the target process is owned by another user, which means that both process should have the same user ID, although the superuser may send signals to any process.

2. alarm()

   ```
   #include <unistd.h>
   unsigned int alarm(unsigned int seconds);
   ```

   The alarm call schedules the delivery of a SIGALRM signal in seconds time. In fact, the alarm will be delivered shortly after that, due to processing delays and scheduling uncertainties. A value of 0 will cancel any outstanding alarm request.

   Calling alarm before the signal is received will cause the alarm to be rescheduled. Each process can have only one outstanding alarm. It returns number of seconds left before any outstanding alarm call would be sent, or -1 if fails.

3. A program implement alarm

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

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

static int alarm_fired=0;

void ding(int sig)
{
  alarm_fired =1;
}
int main()
{
  pid_t pid;
  pid =fork();
  if(pid <0){
    printf("unable to fork.\n");
    exit(1);
  }else{
    if(pid>0){
      signal(SIGALRM, ding);
      pause();
      if(alarm_fired)
printf("DingDong! alarm clock.\n");
        printf("current time is: \n");
system("date");
exit(0);
    }else{
      sleep(1);
      kill(getppid(),SIGALRM);
      exit(0);
    }
  }
}
```

Explanation: This program creates a child process to send alarm to parent process, parent process catch the alarm signal and produce out the current time.
Note: the child process gets the parent ID by function “getppid()”, and it sends the SIGALRM to parent process to this PPID.

THE END