Lecture - VIII

Unix Process Environment

Tevfik Koşar

Louisiana State University
September 25th, 2008

In Today’s Class

• Unix Process Environment
  - Process Concept
  - ps -- get process info
  - Shell & its implementation
  - Exec() & Fork()
  - Creation & Termination of Processes

Process Concept

• An operating system executes a variety of programs:
  - Batch system - jobs
  - Time-shared systems - user programs or tasks
• Process - a program in execution; process execution must progress in sequential fashion
• A process includes:
  - program counter
  - stack: temporary data

Process Control Block (PCB)

Information associated with each process
• Process state
  - (running, waiting..)
• Program counter
• CPU registers
• CPU scheduling information
  - (i.e. process priority)
• Memory-management information
  - (i.e. page & segment tables)
• Accounting information
• I/O status information

Process State

• As a process executes, it changes state
  - new: The process is being created
  - ready: The process is waiting to be assigned to a process
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - terminated: The process has finished execution

$ ps
PID TTY TIME CMD
18684 pts/4 00:00:00 bash
18705 pts/4 00:00:00 ps
Shell

- A tool for process and program control
- Three main functions
  - Shells run programs
  - Shells manage I/O
  - Shells can be programmed
- Main Loop of a Shell

  ```
  while (!end_of_input){
    get command
    execute command
    wait for command to finish
  }
  ```

How does a Program run another Program?

- Program calls `execvp`

  ```
  int execvp(const char *file, char *const argv[]);
  ```

- Kernel loads program from disk into the process
- Kernel copies arglist into the process
- Kernel calls main(argc, argv)

![Image of a computer console displaying a `ps` command output]
Exec Family

```c
int exec(const char *path, const char *arg, ...);
int execl(const char *file, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
            char * const envp[]);
int execv(const char *path, const char *arg, ...,
            char * const argv[]);
int execvp(const char *file, const char *argv[]);
```

Running “ls -l”

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

int main()
{
    char *arglist[3];
    arglist[0] = "ls";
    arglist[1] = "-l";
    arglist[2] = 0;
    printf("** About to exec ls -l\n");
    execvp("ls", arglist);
    printf("** ls is done. bye\n");
}
```

execvp is like a Brain Transplant

- execvp loads the new program into the current process, replacing the code and data of that process!

Writing a Shell v1.0

```c
int main()
{
    char *arglist[MAXARGS+1];
    /\ an array of ptrs /
    int numargs;
    /\ index into array /
    char argv[MAXLEN];
    /\ read stuff here /
    char *makestring();
    /\ malloc etc /

    numargs = 0;
    while ( numargs < MAXARGS )
    {
        printf("Arg[%d]? ", numargs);
        if ( fgets(argv, ARGLEN, stdin) && *argv != '\n' )
        {
            arglist[numargs++] = makestring(argv);
        }
        else
        {
            if ( numargs > 0 )
            {
                /* any args? */
                arglist[numargs-1] = NULL;
                /* close list */
            }
        }
    }
    return 0;
}
```

Writing a Shell v1.0 (cont.)

```c
int execute( char *arglist[] )
{
    execvp(arglist[0], arglist); /* do it */
    perror("execvp failed");
    exit(1);
}
```

How to Create a New Process?

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process’ memory space with a new program

How fork works?

```c
pid_t fork(void);
```

- Allocates a new chunk of memory and data structures
- Copies the original process into the new process
- Adds the new process to the set of running processes
- Returns control back to both processes

Fork Implementation

```c
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
    }
}
```

Fork Example 1

```c
#include <stdio.h>
main()
{
    int ret_from_fork, mypid;
    mypid = getpid();
    /* who am i? */
    printf("Before: my pid is %d\n", mypid);
    /* fork then exec */
    ret_from_fork = fork();
    sleep(1);
    printf("After: my fork returns pid : %d, said %d\n",
        ret_from_fork, getpid());
}
```

Fork Example 2

```c
#include <stdio.h>
main()
{
    fork();
    fork();
    fork();
    printf("my pid is %d\n", getpid());
}
```

How many lines of output will this produce?

Writing a Shell v2.0

```c
execute( char *arglist[] )
{
    int pid,exitstatus; /* of child*/
    pid = fork(); /* make new process */
    switch( pid ){
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            execvp(arglist[0], arglist); /* do it */
            perror("execvp failed");
            exit(1);
        default:
            while( wait(&exitstatus) != pid )
            {
                printf("child exited with status %d, %d\n",
                        exitstatus>>8, exitstatus&0377);
            }
    }
}
```
Exercise

Improve the Shell v2.0 by:

- Allow the user to type all the arguments on one line
- Allow the user to quit by typing exit

Process Termination

- Five ways to terminate:
  - Normal Termination
    - Return from main()
    - Call exit()
    - Call _exit()
  - Abnormal termination
    - Call abort()
    - Be terminated by a signal

Process Start and Termination

Summary

- Unix Process Environment
  - Process Concept
  - ps -- get process info
  - Shell & its implementation
  - Exec() & Fork()
  - Creation & Termination of Processes

- Next Class: Process Control
  - Try “fork” and “shell” examples

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), and B. Knicki (WPI).