Chapter 1. Introduction

System Programming
http://www.cs.ccu.edu.tw/~pahsiung/courses/sp

熊博安
國立中正大學資訊工程學系
pahsiung@cs.ccu.edu.tw  Class: EA-104
(05)2720411 ext. 33119  Office: EA-512
Introduction

- Whirlwind tour of UNIX from a programmer's perspective
- Brief descriptions and examples
- Services provided by UNIX
UNIX Architecture

- Operating system is the software that
  - controls hardware resources
  - Provides program execution environment
- Architecture
  - Kernel
  - System Calls
  - Shell
  - Applications
Architecture of UNIX OS
Logging In

- Login name
- Password
- Colon-separated fields in each entry of /etc/passwd
  - name,
  - encrypted password or “x” (password is in /etc/shadow),
  - numeric user ID,
  - numeric group ID,
  - real name,
  - home directory,
  - shell program

sar:x:205:105:Stephen Rago:/home/sar:/bin/ksh
Logging In

- Shells: a command line interpreter that reads user input and executes commands
  - Bourne shell: /bin/sh (used by root)
  - C shell: /bin/csh (used by users)
  - Korn shell: /bin/ksh (successor of Bourne shell)
  - Bourne-again shell: /bin/bash (all Linux systems)
  - TENEX C shell: /bin/tcsh (replacement of C shell)
Files and Directories

- Filesystem: hierarchical arrangement of directories and files
- Root directory: /
- File attributes: type, size, owner, permissions, last modification time, ...
- `stat()`, `fstat()`: return file attribute struct
Files and Directories

- Filename
  -Chars not allowed: (/) and (NULL)

- Two filenames automatically created whenever a new dir is created:
  - . → current directory
  - .. → parent directory

- What is .. in root directory (/)?
Files and Directories

- Pathname
  - A sequence of zero or more filenames, separated by slashes (/), and optionally starting with a slash

- Absolute pathname

- Relative pathname
Program 1.3:
(bare bones implementation of ls command)

```c
#include <sys/types.h>
#include <dirent.h>
#include "apue.h"

int
main(int argc, char *argv[])
{
    DIR *dp;
    struct dirent *dirp;

    if (argc != 2)
        err_quit("a single argument (the directory name) is required");

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

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

    closedir(dp);
    exit(0);
}
```
Program 1.3

- Edit and save in myls.c

  `cc myls.c`  (output: a.out)

  `./a.out /dev` (output: .....)

  `./a.out /var/spool/mqueue`

  can’t open /var/spool/mqueue: Permission denied

  `./a.out /dev/tty`

  can’t open /dev/tty: Not a directory
Program 1.3 (details)

- apue.h (in Appendix B)
- ANSI C function declaration
- argv[1]: is an argument to our prog
- What is argv[0]?
- opendir(), readdir(), closedir()
- DIR: directory struct
- err_sys(), err_quit(): error routines
Program 1.3 (details)

- `exit(0)`: program done!
- Return value:
  - 0 → OK!
  - 1 ~ 255 → ERROR!
- Current working directory (CWD)
- `chdir()`: change working directory
- Eg: `doc/my/file`: relative to CWD
- Home directory: in password file
Input and Output

- File Descriptors
  - small non-negative integers that kernel uses to identify files being accessed by a process
- Standard Input
- Standard Output
- Standard Error
Input and Output

- `ls`
  - `stdin, stdout, stderr: → terminal`
- `ls > myfile.abc`
  - `stdout: myfile.abc`
- **How to redirect **`stderr`** to a file???**
- **How to redirect **`stdin`** from a file?**
- **Unbuffered I/O**
  - `open(), read(), write(), lseek(), close()`
#include "apue.h"

#define BUFFSIZE 8192

int
main(void)
{
    int n;
    char buf[BUFFSIZE];

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
    {
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

        if (n < 0)
            err_sys("read error");

    exit(0);
}
Program 1.4 (details)

- unistd.h: constants are defined:
  - STDIN_FILENO = 0
  - STDOUT_FILENO = 1
- BUFFSIZE constant:
  affects program efficiency
- read() returns:
  - #bytes read
  - 0 if EOF
  - -1 if error
Standard I/O

- A buffered interface
- No need to worry about BUFFSIZE
- Deal with “lines of input”
  - fgets() reads an entire line
  - read() reads a specified # of bytes
- printf() (#include <stdio.h>)
Program 1.5:
stdin $\rightarrow$ stdout using standard I/O

#include "apue.h"

int main(void)
{
    int c;

    while ( (c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
Program 1.5 (details)

- `getc()` reads 1 char at a time
- `putc()` writes 1 char at a time
- `#include <stdio.h>`
  - `stdin`: standard input
  - `stdout`: standard output
Programs and Processes

- Program: an **executable file on disk**
- Process: an **executing instance of a program**
- Process also called “task” by some OS
- Unique non-negative integer identifier for each process (pid)
Program 1.6: process ID

```c
#include "apue.h"

int main(void)
{
    printf("hello world from process ID \%d\n", getpid());
    exit(0);
}
```
Process Control

- Three functions
  - fork()
  - exec(): 6 variants
  - waitpid()
Program 1.7: exec stdin cmds

```c
#include <sys/types.h>
#include <sys/wait.h>
#include "apue.h"

int
main(void)
{
    char buf[MAXLINE];
    pid_t pid;
    int status;

    printf("%% "); // print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0; /* replace newline with null */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid == 0) { /* child */
            execlp(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }
        else if (pid == 0) { /* child */
            execlp(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }
        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```
Program 1.7 (details)

- `fgets()` reads a newline-terminated line
- replace newline with NULL using `strlen()`
- `fork()` creates a new process
- `fork()` returns pid of new child process to parent
- `fork()` returns 0 to child
- `execlp()` in child executes stdin cmd
Program 1.7 (details)

- parent waits for child to finish
- `waitpid()`
- current version:
  can’t pass arguments to the stdin cmd
Threads

- A process can have one or more threads
  - Can exploit parallelism on multiprocessor systems
  - Same address space, file descriptors, stacks, process-related attributes
  - Need to synchronize access to shared data
  - Thread IDs: local to process
Error Handling

- Negative return value when error occurs
- `#include <errno.h>`
- `errno` variable
  - never cleared if error does not occur
  - never set to 0 by any function
Error Handling (contd)

- 2 functions for printing error messages:

```c
#include <string.h>
char *strerror(int errnum);

#include <stdio.h>
void perror(const char *msg);
```

- `strerror()` returns a string
- `perror()` outputs “msg: <error_msg>”
Program 1.8: use of error func

```c
#include <errno.h>
#include "apue.h"

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```
Program 1.8: results

$ a.out
EACCES: Permission denied
a.out: No such file or directory

- prog1 < inputfile | prog2 | prog3 > outputfile
- (argv[0] passed as arg to perror())
Error Recover

- Fatal error: no recovery action
- Nonfatal error: delay and try again
  - Improves robustness by avoiding an abnormal exit
- Examples
  - EAGAIN, ENFILE, ENOBUFS, ENOLCK, ENOSPC, ENOSR, EWOULDBLOCK, ENOMEM
User Identification

- User ID: numeric identifier of a user
- Group ID: numeric identifier of a group

```c
#include "apue.h"

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

- A technique to notify a process that some condition has occurred
- E.g.: divide by zero → SIGFPE

Process response to a signal
- Ignore the signal, OR
- Let the default action occur, OR
- Provide a function to handle the signal.
#include "apue.h"
#include <sys/wait.h>

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

int
main(void)
{
    char buf[MAXLINE];
    pid_t pid;
    int status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal error");

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0; /* replace newline with null */
        if ( (pid = fork()) < 0)
Program 1.10

```c
err_sys("fork error");

else if (pid == 0) /* child */
    execvp(buf, buf, (char *) 0);
    err_ret("couldn't execute: %s", buf);
    exit(127);

/* parent */
if ( (pid = waitpid(pid, &status, 0)) < 0)
    err_sys("waitpid error");
    printf("%s ");
exit(0);
}

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

- **Two different time values**
- **Calendar time**: #seconds since the Epoch, which is 00:00:00 Jan 1, 1970, Coordinated Universal Time (UTC).
- **Process time**: measures CPU resources used by a process, in clock ticks, which is 50, 60, or 100 ticks per second.
UNIX Time Values (contd)

- Execution time of a process has 3 values:
  - **clock time**: total amount of time from process start to finish
  - **user CPU time**: CPU time due to user instructions in a process
  - **system CPU time**: CPU time due to kernel activities on behalf of the process
UNIX Time Values (contd)

- To measure process execution time, use the “time” command as follows:

$ time ls > /dev/null

real 0m19.81s
user 0m0.43s
sys 0m4.53s
System Calls & Library Functions

- **System Calls:**
  - Entry points into an OS kernel
  - Cannot be changed by user
  - A function of the same name in the standard C library
  - User just calls those C functions whenever system calls are needed
Library Functions: not entry points into kernel, just functions, but they may invoke one or more system calls

- E.g.: `printf()` invokes `write()` system call
- E.g.: `strcpy()`, `atoi()`: do not invoke any system call

Implementor view: fundamental diff
Programmer view: no critical difference
System Calls & Library Functions

- `malloc()`: Memory allocation
- Many ways to do memory allocation and garbage collection (best fit, first fit)
- `sbrk()`: UNIX system call, increases or decreases address space of process by a specified number of bytes
- Can implement own mem alloc function using `sbrk()`
System Calls & Library Functions

application
code

memory allocation
function malloc()

sbrk()
system call

user process

kernel

Slides©2006 Pao-Ann Hsiung, National Chung Cheng University, Taiwan
System Calls & Library Functions

- Application code
- C library functions
- System calls
- Kernel
- User process

Slides © 2006 Pao-Ann Hsiung, National Chung Cheng University, Taiwan
System Calls & Library Functions

- **System call**: time in seconds since Jan 1, 1970
- **C function**: human-readable time and date using local time zone
- Both are called “functions” in the textbook