Report of Operating System Curriculum Design

叶剑飞
Victor Jianfei Ye

September 7, 2012

Contents

1 Expense Comparison among different I/O Functions

There are three methods of file I/O operation in Linux operating system, namely ANSI C I/O functions, UNIX C I/O functions and memory map. We now make a file reversion operation to test the time consumed by the three kinds of functions. All the memories are allocated dynamically. By the way, my C source code includes "for" loop initial declarations. This is not allowed according to C89 standard, but C99 standard supports it. Consequently, "-std=c99" is needed while compiling my C source code if you are using "gcc".

1.1 File I/O Functions Supported by ANSI C

The file I/O functions supported by ANSI C are fopen, fclose, fread, fwrite, fprintf, fscanf, fgets, fputs, etc. What we need now is binary operation. Consequently, only fopen, fclose, fread, fwrite are used.

Firstly, fopen is used to open a file. On success, it returns a pointer to the file structure (FILE). Otherwise, NULL is returned. After that, we can read the test file by the fread function. We read the file n characters[1] every time. Then the array will be reversed. The reversed arrary should be written back into the original file by the fwrite function. Writing speed is still n characters every time. Finally the file should by closed by the fclose function.

1.2 File I/O Functions Supported by UNIX C

The file I/O functions supported by UNIX C are very similar in forms to those supported by ANSI C. However, there are only binary opertions. Firstly, the file should be open by the open function. On success, it returns an integer, which is the file descriptor. Otherwise, -1 is returned. Then the read function can be used to read the file data. The characters read each time is still from the the second command-line argument vector. The array should also be reversed and write back to the original file by the write function. Finally, the file descriptor should be closed by the close function.

1.3 Memory Map

The mmap function map files into memory. While we are operating the memory, we are actually modifying the file. We still need to open the file by the open function. The mmap function needs the file descriptor. It is used as below:

```
mmap( NULL, filesize , PROT_READ|PROT_WRITE,
            MAP_SHARED, fd , 0 );
```

After mapping, only thing we need is to reverse the array in the memory. But we are actually reversing the file. Finally unmap the file and close the file descriptor.

[1] n comes from the second command-line argument vector, i.e. argv[2].

According to the test, memory map is the most effective way to read and write the file.

2 Make a Simple Shell

2.1 Victor Shell

My English name is Victor. As a result, the shell made by me is called "Victor Shell" and an alternative name "victorshell", with the copyright information in the "version" part, which is "Copyright © Victor Jianfei Ye"[2].

A "shell" is nothing but a command interpreter. After pressing the "Enter" key, the command should be execute. The shell need to analyze the command string. I analyze the command string by the sscanf function in order to seperate the space and tabulate key. Then the shell should judge whether it is a internal command. If it is, then the shell should execute the command by the shell itself. Otherwise, the command should be regarded as external command and create a new process to execute the command. If it failed, it means that it isn't a external command and therefore the shell should prompt the error message "command not found". The command could be read from either keyboard (stdin) or a shell script file. It depends on the command-line argument to the shell. As a result, the command is read by fgets function. The internal command supported by my shell are "exit"[3], "cd"[4], "echo"[5], "pwd"[6], "clear"[7], "environ" and "help". Another thing I did is to launch a background application. When "&" is typed after the command. The program should be launched immediately but run in background. As a result, I need to judge whether the last character is "&". If it is, I just simply assign "false" the boolean variable $background$. The shell launch the application as usual, but the shell need to judge whether it need to block itself to wait for the child process according to the boolean variable $background$. If the value is "true", then wait for the child process by the function waitpid. Otherwise, the shell cannnot wait for the child process in the current thread, but it need to create a new thread in order to wait for the child process in order to prevent the defunct process. Consequently, my C source code should be linked to the POSIX thread library pthread while linking the object files, by typing the command-line linking argument "-lpthread". By the way, compiling my C source code of "victorshell" also needs the command-line argument "-std=gnu99".

[2]"Jianfei Ye" is my Chinese name (叶剑飞) in Pinyin, written in the English style, i.e. the given name is before the family name

[3]Exit the shell with the return value, default to zero (i.e. EXIT_SUCCESS).

[4]Change the current working directory. Support "cd - " and "cd ~ ", even bare "cd".

[5]Simply print the words after "echo", without "$" variable support.

[6]Simply print the current working directory.

[7]Simply clear the screen by a series of control characters, that is "\033[H\033[2J".

3 Synchronization among Processes/Threads

3.1 Synchronization among Processes

Synchronization among processes in Linux operating system needs the System V semaphore. It contains three important APIs, namely, semget, semctl and semop. These three APIs cost me many days to comprehend.

The semget function is used to create a semaphore. The first parameter is the key. We can simply use IPC_PRIVATE here. The second parameter is the amount of the semaphores in the semaphore set. The third parameter is the flag to the semaphore, specifies both IPC_CREAT and IPC_EXCL and a semaphore set already exists for key. On success, it returns the semaphore set identifier, otherwise -1 is returned.

The semctl function performs the control operation to the semaphore. The first parameter is the semaphore set identifier, which is from the return value of the function semget. The second parameter is the subscript to the semaphore set. The third parameter is the exact operation command, e.g. SETVAL, IPC_RMID, etc. The fourth parameter is a union type variable. This function returns -1 on failure.

The semop function performs operations on selected semaphores in the set. The first parameter is the semaphore set identifier. The second parameter is a pointer to the structure type "struct sembuf", which contains the semaphore number, semaphore operation and operation flags. The third parameter refers to the amount of elements in the array.

Now we know the meaning of the three System V APIs. We can write P/V operations by these functions, as the code shown below:

```
bool P( int semid )
{
        struct sembuf spos;
        spos.sem_num = 0;
        spos.sem_op  = -1;
        spos.sem_flg = 0;
        if ( semop( semid, &spos, 1 ) == -1 )
                return false;
        return true;
}
```

```
bool V( int semid )
{
        struct sembuf spos;
        spos.sem_num = 0;
        spos.sem_op  = 1;
        spos.sem_flg = 0;
        if ( semop( semid, &spos, 1 ) == -1 )
                return false;
        return true;
}
```

The return value is boolean type, which stands for whether the operation succeed or not.

3.2 Synchronization among Threads

Thread synchronization is much easier than process synchronization. Only sem_wait function and sem_post function are needed. Before that, we need the function sem_init to initialize the semaphore. Be careful, it needs POSIX thread library, so link with the command-line argument "-lpthread".

All of my C source codes in process synchronization and thread synchronization have "for" loop initial declarations, which is only supported by C99 standard. Moreover, I also used "usleep" function in order to suspend execution for microsecond intervals, but it isn't an ANSI C function. Thus, the source codes should be compiled by the command-line argument "-std=gnu99".

Appendices — C Source Code

A Source Code of File I/O Operation

A.1 file1.c

```c
/*
 * file1.c
 *
 * Copyright Victor Jianfei Ye 2012
 *
 * Compile command:
 *                  gcc file1.c -o file1 -std=c99 -pedantic -Wall
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main ( int argc, char * argv[] )
{
        FILE * fp = NULL;
        char * pText = NULL;
        char temp;
        int bytes, offset, filesize;
        struct stat statbuf;
        if ( argc != 3 )
        {
                fprintf( stderr, ”Usage:␣%s␣<filename>␣<bytes>\n\n”, argv
                return EXIT_FAILURE;
        }
        if ( sscanf( argv[2], ”%d”, &bytes ) != 1 )
        {
                fprintf( stderr, ”Bytes␣number␣must␣be␣number!\n\n” );
                return EXIT_FAILURE;
        }
        if ( stat(argv[1], &statbuf ) != 0 )
        {
                fprintf( stderr, ”File␣Access␣Error\n\n” );
                return EXIT_FAILURE;
        }
        filesize = statbuf.st_size;
```

```
fp = fopen( argv[1], "rb" );
if ( fp == NULL )
{
        fprintf( stderr, "File Open Error\n\n" );
        return EXIT_FAILURE;
}
pText = (char *)malloc( filesize * sizeof(char) );
if ( pText == NULL )
{
        fclose(fp);
        fprintf( stderr, "No Enough Memory!!!\n\n" );
        return EXIT_FAILURE;
}
offset = 0;
while ( !feof(fp) )
{
        fread( pText+offset, bytes * sizeof(char), 1, fp );
        offset += bytes;
}
fclose(fp);
fp = NULL;
for ( int i = (filesize >> 1) ; i < filesize; i ++ )
{
        temp = pText[i];
        pText[i] = pText[filesize-i-1];
        pText[filesize-i-1] = temp;
}
fp = fopen( argv[1], "wb" );
if ( fp == NULL )
{
        fprintf( stderr, "File Open Error\n\n" );
        free( pText );
        return EXIT_FAILURE;
}
offset = 0;
while ( offset+bytes < filesize )
{
        fwrite( pText+offset, bytes * sizeof(char), 1, fp );
        offset += bytes;
}
fwrite( pText+offset, (filesize-offset)*sizeof(char), 1, fp );
fclose(fp);
free(pText);
return EXIT_FAILURE;
```

```
}
```

A.2 file2.c

```
/*
 * file2.c
 *
 * Copyright Victor Jianfei Ye 2012
 *
 * Compile command:
 *                      gcc file2.c -o file2 -std=c99 -pedantic -Wall
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main ( int argc, char * argv[] )
{
        int fd = 0;
        char * pText = NULL;
        char temp;
        int bytes, offset, filesize;
        struct stat statbuf;
        if ( argc != 3 )
        {
                fprintf( stderr, "Usage: %s <filename> <bytes>\n\n", argv
                return EXIT_FAILURE;
        }
        if ( sscanf( argv[2], "%d", &bytes ) != 1 )
        {
                fprintf( stderr, "Bytes number must be number!\n\n" );
                return EXIT_FAILURE;
        }
        fd = open( argv[1], O_RDWR );
        if ( fd == -1 )
        {
                fprintf( stderr, "File Open Error\n\n" );
                return EXIT_FAILURE;
        }
        if ( fstat(fd, &statbuf) != 0 )
        {
                fprintf( stderr, "Error␣Getting␣File␣Information\n\n" );
                close(fd);
                return EXIT_FAILURE;
        }
        filesize = statbuf.st_size;
        pText = (char *)malloc( filesize * sizeof(char) );
        if ( pText == NULL )
        {
                close(fd);
                fprintf( stderr, "No␣Enough␣Memory!!!\n\n" );
                return EXIT_FAILURE;
        }
        offset = 0;
        while ( offset < filesize )
                offset += read( fd, pText+offset, bytes * sizeof(char) );
        if ( lseek( fd, 0, SEEK_SET ) == -1 )
        {
                fprintf( stderr, "lseek␣error\n\n" );
                free( pText );
                close(fd);
                return EXIT_FAILURE;
        }
        for ( int i = (filesize >> 1) ; i < filesize; i ++ )
        {
                temp = pText[i];
                pText[i] = pText[filesize-i-1];
                pText[filesize-i-1] = temp;
        }
        offset = 0;
        while ( offset < filesize )
                offset += write( fd, pText+offset, bytes*sizeof(char) );
        close(fd);
        free(pText);
        return EXIT_FAILURE;
}
```

A.3 file3.c

```c
/*
 * file3.c
 *
 * Copyright Victor Jianfei Ye 2012
 *
 * Compile command:
 *                  gcc file3.c -o file3 -std=c99 -pedantic -Wall
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main ( int argc, char * argv[] )
{
        int fd = 0;
        char * pText = NULL;
        char temp;
        int filesize;
        struct stat statbuf;
        if ( argc != 2 )
        {
                fprintf( stderr, "Usage: %s <filename>\n\n", argv[0] );
                return EXIT_FAILURE;
        }
        fd = open( argv[1], O_RDWR );
        if ( fd == -1 )
        {
                fprintf( stderr, "File Open Error\n\n" );
                return EXIT_FAILURE;
        }
        if ( fstat(fd, &statbuf) != 0 )
        {
                fprintf( stderr, "Error Getting File Information\n\n" );
                close(fd);
                return EXIT_FAILURE;
        }
        filesize = statbuf.st_size;
        pText = mmap( NULL, filesize,
                        PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0 );
        for ( int i = (filesize >> 1) ; i < filesize; i ++ )
        {
                temp = pText[i];
                pText[i] = pText[filesize-i-1];
                pText[filesize-i-1] = temp;
        }
        munmap( pText, filesize );
        close(fd);
        return EXIT_FAILURE;
}
```

B Source Code of a Simple Shell

B.1 victorshell.c

```c
/*
 * victorshell.c
 *
 * Copyright Victor Jianfei Ye 2012
 *
 * Compile command:
 *         gcc victorshell.c -o victorshell   \
 *           -std=gnu99 -D_REENTRANT -lpthread -Wall
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdbool.h>
#include <stdarg.h>
#include <sys/types.h>
#include <pwd.h>
#include <sys/wait.h>
#include <pthread.h>

#define MAX_LENGTH 256

typedef int COUNT;

typedef struct
{
        pid_t pid;
        char cmdline[MAX_LENGTH];
} PROCESS_STRUCTURE;

void version(void)
{
        puts( "Victor␣Shell,␣version␣1.0.0-beta" );
        puts( "Copyright␣(C)␣2012␣Victor␣Jianfei␣Ye" );
        puts("");
}
```

```
void arghelp( const char * currentFilename )
{
        version();
        printf ( "Usage:%s␣[option]␣...\n", currentFilename );
        puts ( "Options:" );
        puts( "\t--version");
        puts( "\t--help");
        puts( "\t-v" );
}

void help(const char * content)
{
        if ( content == NULL )
        {
                version();
                puts ( "exit" );
                puts ( "cd" );
                puts ( "clear" );
                puts ( "pwd" );
                puts ( "echo" );
                puts ( "" );
        }
        else
        {
                if ( !strcmp( content, "exit" ) )
                {
                        puts("exit:␣exit␣[n]");
                        puts("\tExit␣the␣shell.");
                        puts("");
                        puts("\tExits␣the␣shell␣with␣a␣status␣of␣N.␣␣");
                        puts("If␣N␣is␣omitted,␣the␣exit␣status");
                        puts("\tis␣that␣of␣the␣last␣command␣executed.");
                }
                else if ( !strcmp( content, "cd" ) )
                {
                        puts( "cd:␣cd␣<directory␣name>" );
                        puts( "\tchange␣to␣the␣directory" );
                        puts( "" );
                }
                else if ( !strcmp( content, "clear" ) )
                {
                        puts( "clear:␣clear" );
                        puts( "\tclear␣the␣terminal␣screen" );
                }
                else if ( !strcmp( content, "pwd" ) )
                {
                        puts( "pwd:␣pwd" );
                        puts( "Print␣the␣name␣of␣the␣current␣working␣dire
                }
                else if ( !strcmp( content, "echo" ) )
                {
                        puts( "echo:␣echo␣[arg␣...]" );
                        puts( "\tWrite␣arguments␣to␣the␣standard␣output."
                        puts( "\tDisplay␣the␣ARGs␣on␣the␣standard␣");
                        puts( "output␣followed␣by␣a␣newline." );
                        puts( "" );
                }
        }
}

bool exeInternalCmd( const char *cmd )
{
        int argc;
        char argv[MAX_LENGTH][MAX_LENGTH];
        memset( argv, 0, sizeof(argv) );
        argc = sscanf(cmd,
         "%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
          argv[0], argv[1], argv[2], argv[3], argv[4], argv[5], argv[6],
          argv[7], argv[8], argv[9], argv[10], argv[11], argv[12],
          argv[13], argv[14], argv[15], argv[16], argv[17], argv[18],
          argv[19], argv[20], argv[21], argv[22], argv[23], argv[24],
          argv[25], argv[26], argv[27], argv[28], argv[29], argv[30],
          argv[31], argv[32], argv[33], argv[34] );
        if ( argc <= 0 )
                return true;
        else if ( argc >= 1 )
        {
                if ( argv[0][0] == '#' )
                        return true;
                if ( !strcmp( argv[0], "exit" ) )
                {
                        puts("exit");
                        if ( argc == 1 )
                                exit(EXIT_SUCCESS);
                        else if ( argc == 2 )
                        {
                                int exitcode;
                                if ( sscanf( argv[1], "%d", &exitcode ) =
            exit( exitcode );
        else
        {
            printf("victorshell:␣"
            printf("exit:␣%s:␣numeric␣argumen
                argv[1] );
            exit(255);
        }
    }
    else
    {
        puts("victorshell:␣exit:␣too␣many␣argumen
        exit(EXIT_FAILURE);
    }
}
else if ( !strcmp(argv[0], "cd" ) )
{
    if( argc == 1 || (argc == 2 && !strcmp( argv[1],
    {
        if ( chdir(getenv("HOME")) != 0 )
            printf( "victorshell:␣cd:␣%s:␣no␣
    }
    else if ( argc == 2 && !strcmp( argv[1] , "-" ) )
    {
        char * oldPwd = getenv("OLDPWD");
        char curPwd[MAX_LENGTH];
        if ( getcwd( curPwd, sizeof(curPwd) ) ==
            strncpy( curPwd, "", 3 ) ;
        if ( oldPwd == NULL )
            puts( "victorshell:␣cd:␣OLDPWD␣no
        else
        {
            if ( chdir(oldPwd) != 0 )
                printf( "victorshell:␣cd:
            else
            {
                puts( oldPwd );
                if ( strcmp( curPwd, "" )
                    setenv( "OLDPWD",
                else
                    unsetenv( "OLDPWD
            }
        }
    }
```

```
        else
        {
            char curPwd[MAX_LENGTH];
            if ( getcwd( curPwd, sizeof(curPwd) ) ==
                strncpy( curPwd, "", 3 ) ;
            if( chdir(argv[1]) != 0 )
                printf( "victorshell:␣cd:␣%s:␣no␣
            else
            {
                if ( strcmp( curPwd, "" ) )
                    setenv( "OLDPWD", curPwd,
                else
                    unsetenv( "OLDPWD" );
            }
        }
        return true;
    }
    else if ( !strcmp(argv[0], "echo" ) )
    {
        if ( argc > 1 )
        {
            printf( "%s", argv[1] );
            for ( int i = 2; i < argc; i ++ )
                printf ( "␣%s", argv[i] );
        }
        puts("");
        return true;
    }
    else if ( !strcmp( argv[0], "pwd" ) )
    {
        char path[MAX_LENGTH];
        if ( getcwd(path, sizeof(path)) == NULL )
            puts("invalid␣path");
        else
            puts(path);
        return true;
    }
    else if ( !strcmp( argv[0], "clear" ) )
    {
        printf("\033[H\033[2J");
        return true;
    }
    else if ( !strcmp( argv[0], "environ" ) )
    {
```

```
                        char path[MAX_LENGTH];
                        getcwd( path, sizeof(path) );
                        printf("USER=%s\n",getpwuid(getuid())->pw_name);
                        printf( "PWD=%s\n", path);
                        printf( "HOME=%s\n", getenv("HOME") );
                        printf( "PATH=%s\n", getenv("PATH") );
                        return true;
                }
                else if ( !strcmp(argv[0], "help" ) )
                {
                        if ( argc == 1 )
                                help(NULL);
                        else
                                help(argv[1]);
                        return true;
                }
        }
        return false;
}

void * waitBackgroundProcess ( void * args )
{
        int status;
        PROCESS_STRUCTURE processStructure = *(PROCESS_STRUCTURE *)args;
        pid_t pid = processStructure.pid;
        waitpid( pid, &status, 0 );
        if ( WEXITSTATUS(status) == EXIT_FAILURE )
        {
                fprintf( stderr, "victorshell:␣%s:␣command␣not␣found...\n
        }
        else
        {
                printf ( "Done\t\t\t%s\n", processStructure.cmdline );
        }
        return NULL;
}

bool exeExternalCmd( const char * cmd )
{
        int argc;
        COUNT i;
        pid_t pid;
        char * argv[MAX_LENGTH];
        int status;
```

```
bool background;
for ( i = 0 ; i < MAX_LENGTH ; i ++ )
{
        argv[i] = (char *)malloc( MAX_LENGTH * sizeof(char) );
}
argc = sscanf(cmd,
 "%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%
  argv[0], argv[1], argv[2], argv[3], argv[4], argv[5], argv[6],
  argv[7], argv[8], argv[9], argv[10], argv[11], argv[12],
  argv[13], argv[14], argv[15], argv[16], argv[17], argv[18],
  argv[19], argv[20], argv[21], argv[22], argv[23], argv[24],
  argv[25], argv[26], argv[27], argv[28], argv[29], argv[30],
  argv[31], argv[32], argv[33], argv[34] );
for ( i = argc ; i < MAX_LENGTH ; i ++ )
{
        free( argv[i] );
        argv[i] = NULL;
}
if ( !strcmp( argv[argc-1], "&") )
{
        free( argv[argc-1] );
        argv[argc-1] = NULL;
        background = true;
}
else if ( argv[argc-1][strlen(argv[argc-1])-1] == '&' )
{
        argv[argc-1][strlen(argv[argc-1])-1] = '\0';
        background = true;
}
else
{
        background = false;
}

if ( (pid = fork() ) < 0 )
{
        puts ( "fork() failed." );
        return true;
}
else if ( pid == 0 )
{
        if ( execvp( argv[0], argv ) < 0 )
        {
                exit( EXIT_FAILURE );
                }
        }
        if ( background )
        {
                pthread_t backgroundWaitingThread;
                PROCESS_STRUCTURE processStructure;
                processStructure.pid = pid;
                strncpy( processStructure.cmdline, argv[0] , MAX_LENGTH )
                pthread_create( &backgroundWaitingThread,
                        NULL, waitBackgroundProcess,
                        (void *)&processStructure );
                return true;
        }
        else
        {
                waitpid( pid, &status, 0 );
                if ( WEXITSTATUS(status) == EXIT_FAILURE )
                        return false;
                else
                        return true;
        }
}

int main (int argc, char * argv[])
{
        FILE * fp = NULL;
        bool debug = false;
        char cmd[MAX_LENGTH];
        memset( cmd, 0, sizeof(cmd) );
        if ( argc >= 2 )
        {
                if ( !strcmp(”−v”, argv[1] ) )
                        debug = true;
                else if ( !strcmp(”−−help”, argv[1] ) )
                {
                        arghelp(argv[0]);
                        return EXIT_SUCCESS;
                }
                else if ( !strcmp(”−−version”, argv[1] ) )
                {
                        version();
                        return EXIT_SUCCESS;
                }
                else
        {
                if ( ( fp = fopen(argv[1] , "r" ) ) == NULL )
                {
                        printf( "victorshell:␣%s:␣invalid␣option\
                        arghelp( argv[0] );
                        return EXIT_FAILURE;
                }
        }
}
if ( fp == NULL )
        fp = stdin;
while (true)
{
        char path[MAX_LENGTH];
        char hostname[MAX_LENGTH];
        if ( fp == stdin )
        {
                gethostname( hostname, sizeof(hostname) );
                getcwd( path, sizeof(path) );
                printf ( "[%s@%s␣%s]", getpwuid(getuid())->pw_nam
                if ( getuid() == 0 )
                        printf ( "#␣" );
                else
                        printf ( "$␣" );
                fflush( stdout );
        }
        if ( fgets(cmd, sizeof(cmd)-1, fp ) == NULL )
        {
                if ( fp != stdin )
                        fclose( fp );
                return EXIT_SUCCESS;
        }
        cmd[strlen(cmd)-1] = '\0';
        if (debug)
                puts(cmd);
        if ( !exeInternalCmd(cmd) )
        {
                if( !exeExternalCmd(cmd) )
                {
                        char argfirst[MAX_LENGTH];
                        sscanf( cmd, "%s", argfirst );
                        printf( "victorshell:␣%s:␣command␣not␣fou
                }
        }
        }
        return EXIT_SUCCESS;
}
```

C Source Code of Synchronization

C.1 process_synchronization.c

```c
/*
 * process_synchronization.c
 *
 * Copyright Victor Jianfei Ye 2012
 *
 * Compile command:
 *       gcc process_synchronization.c -o  \
 *           process_synchronization -std=gnu99 -Wall
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>

typedef int COUNT;

union semun {
        int              val;    /* Value for SETVAL */
        struct semid_ds * buf;    /* Buffer for IPC_STAT, IPC_SET */
        unsigned short  * array;  /* Array for GETALL, SETALL */
        struct seminfo  * __buf;  /* Buffer for IPC_INFO
                                     (Linux-specific) */
};

int createSemaphore ( void )
{
        int semid;
        union semun arg;
        semid = semget( IPC_PRIVATE, 1, 0666|IPC_CREAT );
        if ( semid == -1 )
                return -1;
        arg.val = 1;
        if ( semctl( semid, 0, SETVAL, arg ) == -1 )
```

```
                return -1;
        return semid;
}

bool destroySemaphore ( int semid )
{
        union semun arg;
        if ( semctl( semid, 0, IPC_RMID, arg ) == -1 )
                return false;
        else
                return true;
}

bool P( int semid )
{
        struct sembuf spos;
        spos.sem_num = 0;
        spos.sem_op  = -1;
        spos.sem_flg = 0;
        if ( semop( semid, &spos, 1 ) == -1 )
                return false;
        return true;
}

bool V( int semid )
{
        struct sembuf spos;
        spos.sem_num = 0;
        spos.sem_op  = 1;
        spos.sem_flg = 0;
        if ( semop( semid, &spos, 1 ) == -1 )
                return false;
        return true;
}

int main (void)
{
        const int n = 10;
        int maxLoop[7];
        COUNT i;
        pid_t pid;

        int p12 = createSemaphore();
        int p13 = createSemaphore();
        int p24 = createSemaphore();
        int p25 = createSemaphore();
        int p35 = createSemaphore();
        int p46 = createSemaphore();
        int p56 = createSemaphore();
        int p6  = createSemaphore();

        srand( time(NULL) );
        for ( i = 1; i < 7 ; i ++ )
                maxLoop[i] = (rand() % n) + 1;
        P(p12);
        P(p13);
        P(p24);
        P(p25);
        P(p35);
        P(p46);
        P(p56);
        P(p6);

        for ( i = 1; i <= 6 ; i ++ )
        {
                if ( (pid = fork() ) < 0 )
                {
                        fprintf( stderr, "fork()␣error!\n\n" );
                        return EXIT_FAILURE;
                }
                else if ( pid == 0 )
                        break;
        }

        switch ( i )
        {
                case 1:
                        for ( COUNT j = 0 ; j < maxLoop[1] ; j ++ )
                        {
                                puts( "I␣am␣process␣one." );
                                usleep( 500000 );
                        }
                        V(p12);
                        V(p13);
                        return EXIT_SUCCESS;
                case 2:
                        P(p12);
                        for ( COUNT j = 0 ; j < maxLoop[2] ; j ++ )
                        {
                                puts( "I␣am␣process␣two." );
                                usleep( 500000 );
                        }
                        V(p24);
                        V(p25);
                        return EXIT_SUCCESS;
                case 3:
                        P(p13);
                        for ( COUNT j = 0 ; j < maxLoop[3] ; j ++ )
                        {
                                puts( "I␣am␣process␣three." );
                                usleep( 500000 );
                        }
                        V(p35);
                        return EXIT_SUCCESS;
                case 4:
                        P(p24);
                        for ( COUNT j = 0 ; j < maxLoop[4] ; j ++ )
                        {
                                puts( "I␣am␣process␣four." );
                                usleep( 500000 );
                        }
                        V(p46);
                        return EXIT_SUCCESS;
                case 5:
                        P(p25);
                        P(p35);
                        for ( COUNT j = 0 ; j < maxLoop[5] ; j ++ )
                        {
                                puts( "I␣am␣process␣five." );
                                usleep( 500000 );
                        }
                        V(p56);
                        return EXIT_SUCCESS;
                case 6:
                        P(p46);
                        P(p56);
                        for ( COUNT j = 0 ; j < maxLoop[6] ; j ++ )
                        {
                                puts( "I␣am␣process␣six." );
                                usleep( 500000 );
                        }
                        V(p6);
```

```
                        return EXIT_SUCCESS;
                default:
                        P(p6);
                        destroySemaphore( p12 );
                        destroySemaphore( p13 );
                        destroySemaphore( p24 );
                        destroySemaphore( p25 );
                        destroySemaphore( p35 );
                        destroySemaphore( p46 );
                        destroySemaphore( p56 );
                        destroySemaphore( p6 );
                        puts("All␣Processes␣have␣finished.");
                        return EXIT_SUCCESS;
        }


        return EXIT_SUCCESS;
}
```

C.2 thread_synchronization.c

```
/*
 * thread_synchronization.c
 *
 * Copyright Victor Jianfei Ye 2012
 *
 * Compile command:
 *       gcc thread_synchronization.c -o  \
 *               thread_synchronization -std=gnu99 \
 *               -D_REENTRANT -lpthread -Wall
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <time.h>

typedef int COUNT;

sem_t p12, p13, p24, p25, p35, p46, p56;
int maxLoop[7];

void * thread1 ( void * args )
{
        for ( COUNT i = 0 ; i < maxLoop[1] ; i ++ )
        {
                puts( "I am thread one." );
                usleep(500000);
        }
        sem_post( &p12 );
        sem_post( &p13 );
        return NULL;
}

void * thread2 ( void * args )
{
        sem_wait( &p12 );
        for ( COUNT i = 0 ; i < maxLoop[2] ; i ++ )
        {
                puts( "I am thread two." );
                usleep(500000);
        }
        sem_post( &p24 );
        sem_post( &p25 );
        return NULL;
}

void * thread3 ( void * args )
{
        sem_wait( &p13 );
        for ( COUNT i = 0 ; i < maxLoop[3] ; i ++ )
        {
                puts( "I am thread three." );
                usleep(500000);
        }
        sem_post( &p35 );
        return NULL;
}

void * thread4 ( void * args )
{
        sem_wait( &p24 );
        for ( COUNT i = 0 ; i < maxLoop[4] ; i ++ )
        {
                puts( "I am thread four." );
                usleep(500000);
        }
        sem_post( &p46 );
        return NULL;
}

void * thread5 ( void * args )
{
        sem_wait( &p25 );
        sem_wait( &p35 );
        for ( COUNT i = 0 ; i < maxLoop[5] ; i ++ )
        {
                puts( "I am thread five." );
                usleep(500000);
        }
        sem_post( &p56 );
        return NULL;
```

```
}

void * thread6 ( void * args )
{
        sem_wait( &p46 );
        sem_wait( &p56 );
        for ( COUNT i = 0 ; i < maxLoop[6] ; i ++ )
        {
                puts( "I␣am␣thread␣six." );
                usleep(500000);
        }
        return NULL;
}

int main (void)
{
        pthread_t threadid[7];
        const int n = 10;

        srand( time(NULL) );

        for ( COUNT i = 1 ; i < 7 ; i ++ )
                maxLoop[i] = (rand() % n) + 1;

        if ( sem_init( &p12, 0, 0 ) == -1 )
        {
                fprintf( stderr, "Error␣Initializing␣Semaphore\n\n");
                return EXIT_FAILURE;
        }
        if ( sem_init( &p13, 0, 0 ) == -1 )
        {
                fprintf( stderr, "Error␣Initializing␣Semaphore\n\n");
                return EXIT_FAILURE;
        }
        if ( sem_init( &p24, 0, 0 ) == -1 )
        {
                fprintf( stderr, "Error␣Initializing␣Semaphore\n\n");
                return EXIT_FAILURE;
        }
        if ( sem_init( &p25, 0, 0 ) == -1 )
        {
                fprintf( stderr, "Error␣Initializing␣Semaphore\n\n");
                return EXIT_FAILURE;
        }
```

```
if ( sem_init( &p35, 0, 0 ) == -1 )
{
        fprintf( stderr, "Error Initializing Semaphore\n\n");
        return EXIT_FAILURE;
}
if ( sem_init( &p46, 0, 0 ) == -1 )
{
        fprintf( stderr, "Error Initializing Semaphore\n\n");
        return EXIT_FAILURE;
}
if ( sem_init( &p56, 0, 0 ) == -1 )
{
        fprintf( stderr, "Error Initializing Semaphore\n\n");
        return EXIT_FAILURE;
}

if ( pthread_create( &threadid[1], NULL, thread1, NULL ) )
{
        fprintf( stderr, "Error Creating Thread.\n\n" );
        return EXIT_FAILURE;
}
if ( pthread_create( &threadid[2], NULL, thread2, NULL ) )
{
        fprintf( stderr, "Error Creating Thread.\n\n" );
        return EXIT_FAILURE;
}
if ( pthread_create( &threadid[3], NULL, thread3, NULL ) )
{
        fprintf( stderr, "Error Creating Thread.\n\n" );
        return EXIT_FAILURE;
}
if ( pthread_create( &threadid[4], NULL, thread4, NULL ) )
{
        fprintf( stderr, "Error Creating Thread.\n\n" );
        return EXIT_FAILURE;
}
if ( pthread_create( &threadid[5], NULL, thread5, NULL ) )
{
        fprintf( stderr, "Error Creating Thread.\n\n" );
        return EXIT_FAILURE;
}
if ( pthread_create( &threadid[6], NULL, thread6, NULL ) )
{
        fprintf( stderr, "Error Creating Thread.\n\n" );
```

```
                return EXIT_FAILURE;
        }

        for ( COUNT i = 1 ; i < 7 ; i ++ )
        {
                if ( pthread_join( threadid[i] , NULL ) )
                {
                        fprintf( stderr, "Error Joining Thread.\n\n" );
                        return EXIT_FAILURE;
                }
        }

        return EXIT_SUCCESS;
}
```