Lab Work 2 (Part 1) - Processes and Threads

Process: Introduction

A process is essentially a computer program being executed (a.k.a. 'running') on a computer hardware. In this exercise, we will look at how a process can be created and controlled. The codes in this section are specific to Linux platform and cannot be used on a Windows platform. There are equivalent functions in the Win32 API that can be used to achieve the same goal but they will NOT be discussed here.

Notes

Executing a process is also known as 'running' a program code. A shell (terminal or console or command prompt on Win32) is also a process that is started by another process in the Operating System (OS). When we type a program name at the shell prompt (i.e. execute a process), the shell actually creates a child process, and put the program in the newly created process space and initiate its code execution. The first step we need to understand is how a process can create another process.

Process Creation

On Linux, the function that enables us to create a process is called `fork`. The prototype is

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

which is declared in `unistd.h`. It basically duplicates the current process and the return value will indicate which process the code belongs to.

Simple Fork

A simple example:

smith.c

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    pid_t smith = fork();
    printf("I am Agent Smith %d (%d)\n",getpid(),smith);
    return 0;
```

```
}
```

Using the program from movie 'The Matrix', the above code will duplicate itself, with both copies introducing themselves and exiting gracefully (of course, this is not the case in the movie 😬). You should note that `pid_t` is a data type defined to represent process ID (an integer value assigned by the operating system to all running processes). The introduction text includes 2 integers - the first is the process ID for the running process and the second is the value returned by `fork`.

If the example given above is executed, the output should be something like:

```
user@localhost:~$ smith
I am Agent Smith 5539 (5540)
I am Agent Smith 5540 (0)
```

The first line is 'printed' by the original process while the second line is 'printed' by the created process. This is indicated by the return value (assigned to local variable *smith*) - a value of 0 means that the process is the newly created process, while the original process will get the process ID for the newly created process. The fact that the two processes prints different value for the local variable *smith* also shows that the processes have separate address space.

To clarify this, let us modify the code further to check the address for the local variable *smith*:

smith.c

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    pid_t smith = fork();
    printf("I am Agent Smith %d @%p (%d)\n",getpid(),&smith,smith);
    return 0;
}
```

Now, the output is something like:

```
user@localhost:~$ smith
I am Agent Smith 2698 @0x7fff7e6f81dc (2699)
I am Agent Smith 2699 @0x7fff7e6f81dc (0)
```

Notice that even the local variable have the exact same address, the value is still different. This is because that address is actually a virtual address (will be covered in subsequent topic).

Parent Child Fork

All processes created this way are usually called the child processes of the original process. Naturally, the original process is known as the parent process of the newly created process. To make the code

reflect this scenario, let us rewrite into:

family.c

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    pid_t child = fork();
    if(child == 0)
    {
        printf("I am a child %d with parent %d\n",getpid(),getppid());
    }
    else
    {
        printf("I am a parent %d with child %d\n",getpid(),child);
    }
    return 0;
}
```

Now, the output should be something like:

```
user@localhost:~$ family
I am a parent 6666 with child 6667
I am a child 6667 with parent 6666
```

Multiple Forks

Let us try to breed more child processes based on user request in command line parameter:

breed.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
    pid_t child;
    int loop, count, check, status = 0;
    /* must have exactly ONE parameter */
    if(argc!=2)
    {
        printf("Usage: '%s' <count>\n",argv[0]);
        return -1;
    }
    /* get count from command line */
```

```
    count = atoi(argv[1]);
    if(!count)
    {
        printf("No child required? Abort!\n");
        return -2;
    }
    /* loop to create child */
    for(loop=0;loop<count;loop++)
    {
        child = fork();
        if(child == 0)
        {
            child = getpid();
            printf("I am a child %d\n",child);
            status = child;
            break;
        }
        else
        {
            waitpid(child,&check,0);
            if(WIFEXITED(check))
                printf("Child ended with %d\n",WEXITSTATUS(check));
                /** note: WEXITSTATUS grabs just the LSByte */
        }
    }
    /* parent text here! */
    if(!status)
    {
        printf("I am a parent - just created %d sub-
process(es)\n",count);
    }
    return status;
}
```

Please spend some time and try to understand the code. This can easily be the basis for your lab exercises in this topic. Now that we know how to create processes, let us try to find a way to control the created processes in order to make it more useful.

Process Control

This is actually part of Inter-Process Communication (IPC) sub-topic but I prefer to cover this as process control mechanism - showing how the processes can send small amount of information (control signal) between one another. On Linux, there are a few methods that can be used to achieve this: signals, pipes, files, shared memory and sockets. We are going to use only two methods in this course - signals and files.

Using Signals

Using signals takes advantage of a system's interrupt facility. The easiest way to implement signals is to use the `signal` (duh!) function declared in `signal.h`. Example:

ctrlsig.c

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

int flag_SIGUSR1 = 0;
int flag_SIGUSR2 = 0;

void signal_handler(int signum)
{
    switch(signum)
    {
    case SIGUSR1:
        flag_SIGUSR1 = 1;
        flag_SIGUSR2 = 0;
        break;
    case SIGUSR2:
        flag_SIGUSR1 = 0;
        flag_SIGUSR2 = 1;
        break;
    /** other signals are ignored? */
    }
}

int main(int argc, char *argv[])
{
    pid_t child;
    if(signal(SIGUSR1,signal_handler)==SIG_ERR)
    {
        printf("Cannot set handler for SIGUSR1\n");
        exit(1);
    }
    if(signal(SIGUSR2,signal_handler)==SIG_ERR)
    {
        printf("Cannot set handler for SIGUSR2\n");
        exit(1);
    }
    if((child=fork())==0)
    {
        /** child stuff - wait for parent */
        printf("[Child-%d] Waiting for parent... ",getpid());
        while(!flag_SIGUSR1); /* wait for parent signal */
        printf("YES!
```

```
(flag1=%d)(flag2=%d)\n",flag_SIGUSR1,flag_SIGUSR2);
        /** child stuff - send ack to parent */
        printf("[Child-%d] Sending acknowledgement to
parent.\n",getpid());
        kill(getppid(),SIGUSR1);
        /** child stuff - wait for parent */
        printf("[Child-%d] Waiting for parent... ",getpid());
        while(!flag_SIGUSR2); /* wait for parent signal */
        printf("YES!
(flag1=%d)(flag2=%d)\n",flag_SIGUSR1,flag_SIGUSR2);
        /** child stuff - send ack to parent */
        printf("[Child-%d] Sending acknowledgement to
parent.\n",getpid());
        kill(getppid(),SIGUSR2);
    }
    else
    {
        /** parent stuff - send 'command' to child */
        printf("[Parent-%d] Sending 'command' to child.\n",getpid());
        kill(child,SIGUSR1);
        /** parent stuff - wait for child */
        printf("[Parent-%d] Waiting for child response... ",getpid());
        while(!flag_SIGUSR1); /* wait for child signal */
        printf("YES!
(flag1=%d)(flag2=%d)\n",flag_SIGUSR1,flag_SIGUSR2);
        /** parent stuff - send 'command' to child */
        printf("[Parent-%d] Sending 'command' to child.\n",getpid());
        kill(child,SIGUSR2);
        /** parent stuff - wait for child */
        printf("[Parent-%d] Waiting for child response... ",getpid());
        while(!flag_SIGUSR2); /* wait for child signal */
        printf("YES!
(flag1=%d)(flag2=%d)\n",flag_SIGUSR1,flag_SIGUSR2);
    }
    printf("[%d] Done\n",getpid());
    return 0;
}
```

However, the implementation of `signal` function (ANSI C specification) varies even across versions of Linux (and UNIX) and therefore not recommended for practical use. The recommended function for this purpose is `sigaction`, but this is left for you to explore.

Using Files

The above example can be rewritten to use file-based control:

ctrlfile.c

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

#define PARENT_FLAG1 "PFLAG1"
#define PARENT_FLAG2 "PFLAG2"
#define CHILD_FLAG1 "CFLAG1"
#define CHILD_FLAG2 "CFLAG2"

void wait_flag(char *pname)
{
    FILE *pfile = 0x0;
    while(!pfile) pfile = fopen(pname,"r");
    fclose(pfile);
}

void send_flag(char *pname)
{
    FILE *pfile = fopen(pname,"w");
    if(pfile) fclose(pfile);
}

void hide_flag(char *pname)
{
    remove(pname); /* actually deletes the flag file! */
}

int main(int argc, char *argv[])
{
    pid_t child;
    if((child=fork())==0)
    {
        /** child stuff - wait for parent */
        printf("[Child-%d] Waiting for parent... ",getpid());
        wait_flag(PARENT_FLAG1); /* wait for parent signal */
        printf("YES!\n");
        /** child stuff - send ack to parent */
        printf("[Child-%d] Sending acknowledgement to
parent.\n",getpid());
        send_flag(CHILD_FLAG1);
        /** child stuff - wait for parent */
        printf("[Child-%d] Waiting for parent... ",getpid());
        wait_flag(PARENT_FLAG2); /* wait for parent signal */
        printf("YES!\n");
        /** child stuff - send ack to parent */
        printf("[Child-%d] Sending acknowledgement to
parent.\n",getpid());
        send_flag(CHILD_FLAG2);
    }
    else
    {
        /** parent stuff - send 'command' to child */
```

```
        printf("[Parent-%d] Sending 'command' to child.\n",getpid());
        send_flag(PARENT_FLAG1);
        /** parent stuff - wait for child */
        printf("[Parent-%d] Waiting for child response... ",getpid());
        wait_flag(CHILD_FLAG1); /* wait for child signal */
        printf("YES!\n");
        /** parent stuff - cleanup! */
        hide_flag(PARENT_FLAG1);
        hide_flag(CHILD_FLAG1); /** should be done by child? */
        /** parent stuff - send 'command' to child */
        printf("[Parent-%d] Sending 'command' to child.\n",getpid());
        send_flag(PARENT_FLAG2);
        /** parent stuff - wait for child */
        printf("[Parent-%d] Waiting for child response... ",getpid());
        wait_flag(CHILD_FLAG2); /* wait for child signal */
        printf("YES!\n");
        /** parent stuff - cleanup! */
        hide_flag(PARENT_FLAG2);
        hide_flag(CHILD_FLAG2); /** should be done by child? */
    }
    printf("[%d] Done\n",getpid());
    return 0;
}
```

Messaging with Files

The advantage of having file-based communication is we can send more than just 'signals' - we can now send sizable data. Given a set of source files to form a project for file-based communication - starting with the main source:

fmsg.c

```
/*------------------------------------------------------------------------
-------*/
#include <unistd.h>
#include <stdio.h>
/*------------------------------------------------------------------------
-------*/
#include "fmsglib.h"
/*------------------------------------------------------------------------
-------*/
#define DATAFILE "message.txt"
#define WFLAG "WRITEDONE"
#define RFLAG "READDONE"
/*------------------------------------------------------------------------
-------*/
int main(int argc, char *argv[])
{
```

```
    if(fork() == 0)
    {
        /** child stuff */
        char child_msg[MSG_SIZE_MAX];
        while(check_flag(WFLAG));
        read_message(DATAFILE,child_msg,MSG_SIZE_MAX);
        child_msg[MSG_SIZE_MAX-1]=0x0;
        printf("[Child-%d] MsgRead: %s\n",getpid(),child_msg);
        setup_flag(RFLAG);
        while(!check_flag(WFLAG));
        clear_flag(RFLAG);
    }
    else
    {
        /** parent stuff */
        char parent_msg[] = "I AM LEGEND!";
        write_message(DATAFILE,parent_msg);
        printf("[Parent-%d] MsgWrite: %s\n",getpid(),parent_msg);
        setup_flag(WFLAG);
        while(check_flag(RFLAG));
        clear_flag(WFLAG);
        clear_flag(DATAFILE);
    }
    printf("[%d] Done\n",getpid());
    return 0;
}
/*-------------------------------------------------------------------------
-------*/
```

Notice that no error checking has been done for the flag and message management functions. This is left as an exercise for you. Next, let us take a look at the 'library' source:

fmsglib.c

```
/*-------------------------------------------------------------------------
-------*/
#include "fmsglib.h"
/*-------------------------------------------------------------------------
-------*/
#include <stdio.h>
/*-------------------------------------------------------------------------
-------*/
int check_flag(char *pname)
{
    int status = FLAG_EXISTS;
    FILE *pfile = fopen(pname,"r");
    if(pfile) fclose(pfile);
    else status = FLAG_MISSING;
    return status;
}
```

```
/*-------------------------------------------------------------------------
-------*/
int clear_flag(char *pname)
{
    int status = remove(pname);
    if(status<0)
        status = FLAG_UNTOUCHED;
    return status;
}
/*-------------------------------------------------------------------------
-------*/
int setup_flag(char *pname)
{
    int status = check_flag(pname);
    if(status==FLAG_MISSING)
    {
        FILE *pfile = fopen(pname,"wt");
        if(pfile) fclose(pfile);
        else status = FLAG_SETFAILED;
    }
    else
    {
        status = FLAG_SETEXISTS;
    }
    return status;
}
/*-------------------------------------------------------------------------
-------*/
int write_message(char *pname, char *message)
{
    int status = MSG_WRITE_SUCCESS;
    FILE *pfile = fopen(pname,"wt");
    if(pfile)
    {
        fprintf(pfile,message);
        fclose(pfile);
    }
    else status = MSG_WRITE_FAILED;
    return status;
}
/*-------------------------------------------------------------------------
-------*/
int read_message(char *pname, char *message, int size)
{
    int status = MSG_READ_SUCCESS;
    FILE *pfile = fopen(pname,"rt");
    if(pfile)
    {
        int loop = 0, test;
        while((test=fgetc(pfile))!=EOF)
        {
```

```
            message[loop++] = (char) test;
            if(loop>=size-1)
            {
                status = MSG_READ_OVERFLOW;
                break;
            }
        }
        fclose(pfile);
        message[loop] = 0x0;
    }
    else status = MSG_READ_FAILED;
    return status;
}
/*-------------------------------------------------------------------------
-------*/
```

The functions are pretty straight forward, using file access functions available in `stdio.h`. Notice that it uses a few constants that is defined in the header file:

fmsglib.h

```
/*-------------------------------------------------------------------------
-------*/
#ifndef __CMSGLIBH__
#define __CMSGLIBH__
/*-------------------------------------------------------------------------
-------*/
/** <function>_flag return values */
#define FLAG_EXISTS 0
#define FLAG_MISSING -1
#define FLAG_CLEARED 0
#define FLAG_UNTOUCHED -2
#define FLAG_SETUP 0
#define FLAG_SETEXISTS -3
#define FLAG_SETFAILED -4
/*-------------------------------------------------------------------------
-------*/
/** message size limit */
#define MSG_SIZE_MAX 80
/** message write status */
#define MSG_WRITE_SUCCESS 0
#define MSG_WRITE_FAILED -1
/** message read status */
#define MSG_READ_SUCCESS 0
#define MSG_READ_FAILED -1
#define MSG_READ_OVERFLOW -2
/*-------------------------------------------------------------------------
-------*/
int check_flag(char *pname);
int clear_flag(char *pname);
```

```
int setup_flag(char *pname);
int write_message(char *pname, char *message);
int read_message(char *pname, char *message, int size);
/*-------------------------------------------------------------------------
-------*/
#endif
/*-------------------------------------------------------------------------
-------*/
```

Finally, a makefile to help manage build:

makefile

```
# makefile to compile a c program
PROJECT = fmsg
OBJECTS = fmsglib.o $(PROJECT).o
CC = gcc
CFLAGS =
LDFLAGS =
$(PROJECT): $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $+ $(LDFLAGS)
%.o: %.c %.h
    $(CC) $(CFLAGS) -c $<
%.o: %.c
    $(CC) $(CFLAGS) -c $<
clean:
    rm -rf $(PROJECT) *.o
```

As with the previous 'long' code, spend some time to understand them. Try them out and modify them to test your knowledge.

Process: Things to Tinker

Thing 1 Write a program that create 5 separate child processes that executes in parallel (the example above creates a child process and wait for it to finish before creating another child). Each child waits for a random number of seconds (between 10 to 20) before exiting. The program must be able to detect all its child processes have finished before exiting.

Thing 2 Write a program that manages the listing, creation, deletion of child processes. Thus, when a child process is created, it should simply hang around until being told to quit (delete process). The listing of child processes should include information of its process ID and, optionally, its running time.

From:
https://azman.my1matrix.cc/ - **Azman @MY1**

Permanent link:
https://azman.my1matrix.cc/doku.php?id=archive:pgt200lab01a

Last update: **2026/02/08 03:46**