Lab Work 2 (Part 2) - Processes and Threads

Thread: Introduction

Threads are parts of a running program that can be made to execute as if they are done in parallel. In this exercise, we will look at how a thread 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

The main difference between a thread and a process is the allocated address space. Unlike a process, a newly created thread shares the same address space as its parent (main thread). This is about as far as the difference goes because other than that, they are actually created to achieve the same thing - multitasking. In fact, in some operating systems like Linux, they are even treated/handled the same way.

The threads facility on Linux is implemented by an external POSIX threads (pthread) library and needs to be linked. So, remember to include the `-pthread` switch when linking codes using the threads facility.

Thread Creation

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

```
int pthread_create(pthread_t *restrict thread, const pthread_attr_t
*restrict attr,
  void *(*start_routine)(void*), void *restrict arg);
```

which is declared in `pthread.h`. It will return a value of 0 upon success (non-zero error number otherwise). You may want to read the man page (`man pthread_create`) to get information on the arguments.

In short, this function creates a thread (a task on Linux) with attributes specified in `attr` (default attributes will be used if this is NULL). If everything goes well, it will store the ID of the thread in `thread` and starts execution at the function pointed by `start_routine` with a single argument `arg`. The argument `arg` is actually very useful when you need to pass data structure to the newly created thread.

Simple Thread

A simple example that is equivalent to the code in Simple Fork:

smith_thread.c

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

static int check = 0;

void* go_smith(void* arg)
{
    if(arg)
    {
        pthread_t *psmith = (pthread_t*) arg;
        printf("I am Agent Smith %x @%p (%d)\n",*psmith,&check,check);
        check = 2;
        pthread_exit(0);
    }
    else
    {
        printf("I am Agent Smith 0 @%p (%d)\n",&check,check);
    }
    return 0x0;
}

int main(int argc, char *argv[])
{
    pthread_t smith;
    check = 1;
    pthread_create(&smith,0x0,&go_smith,&smith);
    pthread_join(smith,0x0);
    go_smith(0x0);
    return 0;
}
```

The output will be something like:

```
user@localhost:~$ smith_thread
I am Agent Smith 58956700 @0x600c80 (1)
I am Agent Smith 0 @0x600c80 (2)
```

Notice that when the global variable check is changed in the thread (`check = 2;`), this value is what gets printed out by the main thread. This shows that threads share the same address space as the main thread (notice that no parent-child concept here). You should note that `pthread_join` is equivalent to `waitpid` used earlier.

Thread vs Process

smith_thread.c

```c
#include <stdio.h>
#include <pthread.h>
/* do not need stdlib.h */
/* do not need wait.h */

/* declare a global variable */
static int check = 0;

/* function to be executed by created thread */
void* go_smith(void* arg)
{
    pthread_t *psmith = (pthread_t*) arg;
    check++;
    printf("I am Agent Smith %u @%p
(%d)\n",(unsigned)*psmith,&check,check);
    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    pthread_t smith;
    check++;
    pthread_create(&smith,0x0,&go_smith,&smith);
    pthread_join(smith,0x0); /* wait for create thread to finish */
    printf("I am Agent Smith 0 @%p (%d)\n",&check,check);
    return 0;
}
```

smith_process.c

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

/* declare a global variable */
static int check = 0;

/* function to be executed by created process */
void* go_smith(void* arg)
{
    pid_t *psmith = (pid_t*) arg;
    check++;
    printf("I am Agent Smith %u @%p
(%d)\n",(unsigned)*psmith,&check,check);
```

```
    exit(0);
}

int main(int argc, char *argv[])
{
    pid_t smith;
    check++;
    if(!fork()) { smith = getpid(); go_smith(&smith); }
    waitpid(smith,0x0,0); /* wait for create process to finish */
    printf("I am Agent Smith 0 @%p (%d)\n",&check,check);
    return 0;
}
```

Thread Control

This is actually much simpler due to the fact that threads share the same address space. But this also introduces race issues - what is the correct sequence when two threads are trying to access the same address? Therefore, a special flag is required to ensure only one thread can access a common address memory at one time, which will validate proper sequence of code. The flag is know as a MUTEX (MUTual EXclusive) and in the `pthread` library, it can be declared and initialized as follows:

```
pthread_mutex_t shared_region = PTHREAD_MUTEX_INITIALIZER;
```

`PTHREAD_MUTEX_INITIALIZER` is actually a macro that initializes a statically allocated mutex with default attributes.

The actual functions used to initialize and destroy a mutex are:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,const
pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Whenever a thread requires access to the common address memory, it needs to request a lock on the mutex object using:

```
pthread_mutex_lock(&shared_region);
```

If the mutex is already locked by another thread, this function will block execution until it can get an exclusive lock. Of course, this means that all threads should be responsible enough to unlock the mutex once it is done:

```
pthread_mutex_unlock(&shared_region);
```

Multiple Threads

An example to show the use of mutexes:

parallel.c

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

/* statically allocated mutex initialization */
pthread_mutex_t shared_region = PTHREAD_MUTEX_INITIALIZER;
static int active = 0;

typedef struct __thread_data
{
    pthread_t thread_id;
    int thread_index;
    int thread_pause;
    int thread_exec;
    int thread_wait;
    int thread_run;
}
thread_data;

#include <unistd.h> /* STDIN_FILENO */
#include <termios.h> /* struct termios */
#define MASK_LFLAG (ICANON|ECHO|ECHOE|ISIG)
int getch(void)
{
    struct termios oldt, newt;
    int ch;
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~MASK_LFLAG;
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    ch = getchar();
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    return ch;
}

void* thread_exec(void* arg)
{
    if(arg)
    {
        thread_data *pdata = (thread_data*) arg;
        pdata->thread_run = 1;
        while(pdata->thread_exec)
        {
            if(pdata->thread_pause)
```

```
            {
                if(pdata->thread_run)
                {
                    printf("[Thread-%d] Thread
Paused!\n",pdata->thread_index);
                    pdata->thread_run = 0;
                }
                continue;
            }
            if(!pdata->thread_run)
            {
                printf("[Thread-%d] Thread
Resumes\n",pdata->thread_index);
                pdata->thread_run = 1;
            }
            printf("[Thread-%d] Sleeping %d
seconds\n",pdata->thread_index,
                pdata->thread_wait);
            sleep(pdata->thread_wait);
            /* something happened while you were sleeping */
            if(!pdata->thread_exec) break;
            pthread_mutex_lock(&shared_region);
            printf("[Thread-%d] Changing active from %d to %d!\n",
                pdata->thread_index,active,pdata->thread_index);
            active = pdata->thread_index;
            pthread_mutex_unlock(&shared_region);
        }
        pthread_exit(0);
    }
    return 0x0;
}

int main(int argc, char *argv[])
{
    thread_data exec1, exec2;
    int not_done = 1, test;
    /* start random number generator */
    srand(time(0x0));
    /** initialize exec1 */
    exec1.thread_index = 1;
    exec1.thread_pause = 1;
    exec1.thread_exec = 1;
    exec1.thread_wait = (rand()%10)+1;
    exec1.thread_run = 0;
    pthread_create(&exec1.thread_id,0x0,&thread_exec,(void*)&exec1);
    /** initialize exec2 */
    exec2.thread_index = 2;
    exec2.thread_pause = 1;
    exec2.thread_exec = 1;
    exec2.thread_wait = (rand()%10)+1;
    exec2.thread_run = 0;
```

```
    pthread_create(&exec2.thread_id,0x0,&thread_exec,(void*)&exec2);
    /** main loop for main thread */
    sleep(1);
    printf("[Main] Starting threads execution...\n");
    exec1.thread_pause = 0;
    exec2.thread_pause = 0;
    while(not_done)
    {
        printf("[Main] <ESC> Exit, <SPACE> Pause/Resume, <R>andomize
Sleep\n");
        switch(test=getch())
        {
            case 0x1B: /** ASCII ESC */
                not_done = 0;
                break;
            case 0x20: /** ASCII SPACE */
                exec1.thread_pause = !exec1.thread_pause;
                exec2.thread_pause = !exec2.thread_pause;
                break;
            case (int)'R':
                printf("[Main] Pause threads execution...\n");
                exec1.thread_pause = 1;
                exec2.thread_pause = 1;
                printf("[Main] Waiting for threads to pause...\n");
                while(exec1.thread_run||exec2.thread_run);
                printf("[Main] Randomizing Sleep Params...\n");
                exec1.thread_wait = (rand()%10)+1;
                exec2.thread_wait = (rand()%10)+1;
                printf("[Main] Wait1=%d, Wait2=%d\n",exec1.thread_wait,
                    exec2.thread_wait);
                printf("[Main] Resume threads execution...\n");
                exec1.thread_pause = 0;
                exec2.thread_pause = 0;
                break;
            default:
                printf("[CHECK]=>'%02X'\n",test);
        }
    }
    /* stop threads */
    printf("[Main] Stopping threads execution...\n");
    exec1.thread_exec = 0;
    exec2.thread_exec = 0;
    /* wait threads to finish */
    printf("[Main] Waiting for threads to finish...\n");
    pthread_join(exec1.thread_id,0x0);
    pthread_join(exec2.thread_id,0x0);
    printf("[Main] Done!\n");
    return 0;
}
```

Another long one... you know what to do!

Thread: Things to Tinker

Thing1 Modify `parallel.c` so that the main thread can send a text message for the worker threads to print (or anything that is more interesting 😛). *Hint:* You should modify the structure so that it contains a buffer for message passing.

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