School of Computing and Information Systems

COMP30023: Computer Systems

Practical Week 3

1 Introduction

In this practical, we will be exploring threads, processes and interprocess communication.
NOTE: You can do this lab without following the order.

2 Creating a Thread

The `main()` function of a C program runs on its own thread (commonly called the 'main' thread)
We can create additional, independent threads by using the `pthread_create` function provided by `pthread.h`.
`thread1.c` on the LMS creates such a thread. The thread runs the function `say_hello()` upon creation.

1. Compile and run `thread1.c`. Note the use of the **`-lpthread`** option to explicitly link the pthread library
   Command: `$ gcc thread1.c -o thread1 -lpthread && ./thread1`

2. Notice how the second thread said hello before the first thread? This is because the `pthread_join` function will wait for the thread specified in the function call to finish executing before proceeding with the current thread.

   In this scenario the main thread 'waited' for the other thread to 'join' it before proceeding.

3. Can you guess what might happen if we did not call the `pthread_join` function? Comment that line in the code, compile and rerun to observe the output.
   Discuss with your classmates (or demonstrator) if you are unable to understand the behaviour you observe.

4. Do you think these threads are user or kernel threads?

3 Threads and Race Conditions

Race conditions, where the final result of a computation depends on the order in which threads happened to run, may occur when several threads access a shared resource.

1. The code `thread2.c` given on the LMS has two threads accessing the common global variable `count`. This code has a race condition.

   Run the code several times and observe that the output changes each time.

2. We can solve race conditions such as these by defining a section of code that can only be executed by one thread at a time (called a '**critical section**' or '**critical region**').

   We can use a **mutex** to define a critical section. The methods `pthread_mutex_lock(&lock)` and `pthread_mutex_unlock(&lock)` can be used to define a critical section, where `lock` is a global variable that is of type `pthread_mutex_t`.

3. The definition, initialisation, and destroying of the mutex have been written for you in `thread2.c`. Determine the critical section that would prevent the race condition and use the function calls to lock and unlock the mutex to fix the race condition.

   ```
   pthread_mutex_lock(&lock)
   /* Code in Critical Section */
   pthread_mutex_unlock(&lock)
   ```

4. Challenge task: try to introduce a deadlock into your program.

4 OS Processes

4.1 fork

`fork()` creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
The child process and the parent process run in separate memory spaces. At the time of `fork()` both memory spaces have the same content.

First, compile the demo fork program (see Appendix):

```
$ gcc -o fork fork.c
```

Run the program in the background.

```
$ ./fork &
```

While the program runs in background, run `top` in tree mode to watch how child processes are spawned from the parent process.

```
$ top
```

While `top` is running, push `Shift + V` to enable forrest view[1]. Find the `fork` program and watch how child processes get spawned.

4.2 exec

Taken from the manpages verbatim:
The exec family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the new process image file. There shall be no return from a successful exec, because the calling process image is overlaid by the new process image.
First, compile the demo exec program:

```
$ gcc -o exec exec.c
```

Run the program.

```
$ ./exec
```

What program does it actually exec into? Discuss with your classmate and demonstrator.

4.3 pipe

The pipe() function shall create a pipe and place two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the pipe() call.
First, compile the demo pipe program:

```
$ gcc -o pipe pipe.c
```

Run the program.

```
$ ./pipe
```

You now have a brief idea how `exec`, `fork` and `pipe` works. You are encouraged to author a simple C program that can utilise all 3 libc functions. An idea would be to author a program that forks a new process and waits for input from the parent process (via `stdin`) to print from the child process. For example, try simulating execution of `$ ls *.c | wc -l`.

[1]This view shows parent-child relationships between processes

A thread1.c

```
/************************************
Demo for pthread commands
compile: gcc threadX.c -o threadX -lpthread
*************************************/

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

void* say_hello(void* param); /* the work_function */

int main(int args, char** argv) {
        pthread_t tid; /* thread identifier */

        /* create the thread */
        pthread_create(&tid, NULL, say_hello, NULL);

        /* wait for thread to exit */
        pthread_join(tid, NULL);

        printf("Hello from first thread\n");
        return 0;
}

void* say_hello(void* param) {
        printf("Hello from second thread\n");
        return NULL;
}
```

B thread2.c

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

#define ITERATIONS 1000000

void* runner(void* param); /* thread doing the work */

int count = 0;
pthread_mutex_t lock;

int main(int argc, char** argv) {
        pthread_t tid1, tid2;
        int value;

        if (pthread_mutex_init(&lock, NULL) != 0) {
                printf("mutex init failed\n");
                exit(1);
        }

        if (pthread_create(&tid1, NULL, runner, NULL)) {
                printf("Error creating thread 1\n");
                exit(1);
        }
        if (pthread_create(&tid2, NULL, runner, NULL)) {
                printf("Error creating thread 2\n");
                exit(1);
        }

        /* wait for the threads to finish */
        if (pthread_join(tid1, NULL)) {
                printf("Error joining thread\n");
                exit(1);
        }
        if (pthread_join(tid2, NULL)) {
                printf("Error joining thread\n");
                exit(1);
        }

        if (count != 2 * ITERATIONS)
        printf("** ERROR ** count is [%d], should be %d\n", count, 2 * ITERATIONS);
        else
        printf("OK! count is [%d]\n", count);

        pthread_exit(NULL);
        pthread_mutex_destroy(&lock);
}

/* thread doing the work */
void* runner(void* param) {
        int i, temp;
        for (i = 0; i < ITERATIONS; i++) {
                temp = count;    /* copy the global count locally */
                temp = temp + 1; /* increment the local copy */
                count = temp;    /* store the local value into the global count */
        }
}
```

C fork.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char** argv) {
        pid_t root = getpid();
        // when forking, program does not start again since memory/register values are exactly the same
        // i.e. instruction pointer is at the same line too. So fork() wont execute again.
        pid_t pid = fork();
        printf("from %d forking into %d\n", root, pid);
        sleep(20);

        // watch 2 different PIDs spawn 2 more child processes.
        pid_t mypid = getpid();
        pid = fork();
        printf("from %d forking into %d\n", mypid, pid);
        sleep(20);

        if (getpid() == root) {
                sleep(20);
                printf("root exiting\n");
        } else {
                printf("Child -- PID %d exiting\n", getpid());
        }
        return 0;
}
```

D exec.c

```c
#include<unistd.h>

int main(int argc, char **argv) {
    return execv("/usr/bin/ls", argv);
}
```

E pipe.c

```c
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
        int fd[2], nbytes;
        pid_t childpid;
        char string[] = "Hello, world!\n";
        char readbuffer[80];

        pipe(fd);

        if ((childpid = fork()) == -1) {
                perror("fork");
                exit(1);
        }

        if (childpid == 0) {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string) + 1));
                exit(0);
        } else {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return (0);
}
```