CS2106 Operating Systems

Assignment 2 – Processes and Threads

1. Introduction

In the previous assignment you learnt how to create processes using fork and wait, and how to use the exec family of functions to run programs. In the first part of this assignment we will continue on with using pipes to communicate between processes.

In the second part of this assignment we will look at threads. It's useful to think of threads as being "mini-processes", in that a single process can have many threads. You will find that threads are more convenient than processes because you can share global variables

SUBMISSION

To submit, please ZIP UP your programs (assg2p1.c to assg2p4.c) and the answer book CS2106Assg2AnsBk.docx together in a single ZIP file. Name the file using the matric number of your team leader (e.g. A012345X.zip), and upload to IVLE by **5 pm on Friday, 31 March 2016**.

PART I

2. Communication between Child and Parent

Unix provides a mechanism called a "pipe" to allow a parent and child to communicate. To create a pipe, first declare a two-element integer array, then pass in the array to the function "pipe", as shown here:

```
int fd[2];
…
pipe(fd);
```

If this call fails it will return a -1. However we will assume that it succeeds, and when it does, fd[0] will be used for reading from the pipe, and fd[1] will be used for writing to the pipe. Both fd[0] and fd[1] are called "file descriptors" when used in conjunction with pipes.

Important: The process that is READING from the pipe should close fd[1], and the process that is WRITING to the pipe should close fd[0], in order for the pipe to work correctly. Use the "close" function, NOT "fclose", to close ends of a pipe. So to close the input end, use:

```
close(fd[0]);
```

To write to the pipe, use the C "write" function:

```
      write(int file_desc, void *buf, size_t size);
```

To read from the pipe, use the C "read" function:

```
      read(int file_desc, void *buf, size_t size);
```

In both cases, file_desc is the file descriptor to write to/read from, buf is a pointer to a buffer (not necessarily of type void *), and size is the number of bytes to write/read.

Type out the following program and call it “assg2p1.c.” Then compile and execute your program and answer the questions that follow:

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

int main()
{
    int fd[2];
    char string[]="Hello child!";
    char buffer[80];
    int pnum=128, cnum;

    pipe(fd);

    if(fork())
    {
        close(fd[0]);
        write(fd[1], &pnum, sizeof(pnum));
        write(fd[1], string, strlen(string)+1);
    }
    else
    {
        close(fd[1]);
        read(fd[0], &cnum, sizeof(cnum));
        read(fd[0], buffer, sizeof(buffer));
        printf("Parent sent message: %s and %d\n", buffer, cnum);
    }
}
```

Question 1 (2 marks)

What output do you see on the screen? Describe in a single statement what this program does.

Question 2 (2 marks)

What do you think the sizeof() function returns?

3. Process Creation Challenge

We will now create a program that generates 16384 random integers, and return the number of prime numbers amongst the integers. Our strategy would be to split the list into two, with the parent counting prime numbers in the 0 to 8191 sublist, and the child counting in the 8192 to 16383 sublist. The parent finally prints out the number of prime numbers in the file.

Use the skeleton given below to start with. Call your program assg2p2.c. Note you must compile your program with "gcc lassg2p2.c –lm –o assg2p2". The "-lm" is important as we need to bring in the math library where the sqrt function resides.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define NUMELTS 16384

// IMPORTANT: Compile using "gcc assg2p2.c .lm -o assg2p2".
// The "-lm" is important as it brings in the Math library.

// Implements the naive primality test.
// Returns TRUE if n is a prime number
int prime(int n)
{
    int ret=1, i;

    for(i=2; i<=(int) sqrt(n) && ret; i++)
        ret=n % i;

    return ret;
}

int main()
{
    int data[NUMELTS];

    // Declare other variables here.
    // Create the random number list.
    srand(time(NULL));

    for(i=0; i<NUMELTS; i++)
        data[i]=(int) (((double) rand() / (double) RAND_MAX) * 10000);

    // Now create a parent and child process.
            //PARENT:
                // Check the 0 to 8191 sub-list
                // Then wait for the prime number count from the child.
                // Parent should then print out the number of primes
    // found by it, number of primes found by the child,
                // And the total number of primes found.
           // CHILD:
                // Check the 8192 to 16383 sub-list.
                // Send # of primes found to the parent.
}
```

Question 3 (3 marks)

Cut and paste your completed program into the answer book, and submit the source code in your ZIP file.

PART 2

4. Introduction to POSIX Threads

POSIX threads (henceforth called "pthreads") is an API specification for threads on Unix based systems. Pthreads packages are available for non-Unix systems as well. The basic thread creation, joining and destruction calls are:

Call	Description
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg)	Creates a new thread. Returns 0 if successful. Arguments: thread - A data structure which will contain information about the created thread. attr - Thread attributes. Can be NULL. start_routine - Pointer to the thread's starting function. Must be declared as void *fun(void *) arg - Argument passed to the starting routine.
void pthread_exit(void *value_ptr)	Exits from a thread. The value in value_ptr is passed to another thread that joins with this exiting thread.

void pthread_join(pthread_t thread, void **value_ptr)	Suspends execution until the thread specified by "thread" completes execution. If value_ptr is not NULL, it will point to a location containing the value passed by "thread" when it exits using pthread_exit. pthread_join returns 0 if successful.

You need to `#include <pthread.h>` to use these. We will now look at an example of how to use threads. Type out the program below on a Unix machine with pthreads installed (if unsure do this lab on Sunfire) and call it assg2p3.c. Compile it using "gcc assg2p3.c -o assg2p3", and execute using "./assg2p3". Run this program several times (at least 8-10 times) and observe the output.

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

// Global variable.
int ctr=0;
pthread_t thread[10];

void *child(void *t)
{
      // Print out the parameter passed in, and the current value of ctr.
      printf("I am child %d. Ctr=%d\n", t, ctr);
      // Then increment ctr
      ctr++;
      pthread_exit(NULL);
}

int main()
{
      int i;

      // Initialize ctr
      ctr=0;

      // Create the threads
      for(i=0; i<10; i++)
            pthread_create(&thread[i], NULL, child, (void *) i);

      // And print out ctr
      printf("Value of ctr=%d\n", ctr);
      return 0;
}
```

Question 4 (2 marks)

Do the threads print out in order? I.e. does it go "I am child 1.", "I am child 2.", etc? Or are the thread outputs mixed up? In either case explain why.

Question 5 (3 marks)

Based on your observation on the values of ctr printed by each thread, do threads share memory or do they each have their own portions of memory? Explain your answer, with reference to ctr.

Question 6 (3 marks)

Are the values of ctr as printed out by the child threads correct? Explain why or why not.

Question 7 (4 marks)

The variable "i" in main is effectively the index number of the child thread. Explain why it must be cast to (void *) before passing to the child thread, and why the child thread can successfully print out "i" without recasting it back to an int.

You will notice that the "printf("Value of ctr=%d\n", ctr);" statement in main sometimes executes even before the last thread completes. We will now use "pthread_join" to ensure that the 10th thread completes before this statement executes. To do this, add the following statement to just before the printf:

```
  // wait for the 10th thread
    pthread_join(thread[9], null);
```

Our main would therefore now look like this, with the added lines in **bold.**

```
int main()
{

      ...

      // wait for the 10th thread
      pthread_join(thread[9], NULL);
      // And print out ctr
      printf("Value of ctr=%d\n", ctr);
      return 0;
}
```

Compile and run the program, and verify that the printf no longer executes until the 10th thread completes. However you will notice that the threads still do no execute in order.

Question 8 (4 marks)

Modify the program so that ALL the threads execute in order. I.e. thread 0 executes first, then thread 1, etc. Compile your program and run it several times to verify that it works correctly, then describe the changes that you made, and cut-paste the code into your answer book. You will demo this program at the start of the next lab session.

5. Introduction to Mutexes

In multi-threaded applications there are sections of code where no more than one thread can execute at a time. For example, code that updates global variables should not be executed by more than one thread or the updating may go wrong. Such sections of code are called "critical sections".

The word "mutex" stands for "Mutual Exclusion", and the idea here is that when a thread wants to enter a critical section the thread must first successfully obtain a lock called a "mutex". When it has the mutex, the thread can safely enter the critical section. Once it exits the critical section, the thread frees the mutex.

Only one thread can obtain this mutex, and other threads that are trying will block until the mutex is freed.

The mutex must be shared by several threads and must therefore be "global", and of type pthread_mutex_t. So to create a mutex call "my_mutex", the statement would be:

```
pthread_mutex_t my_mutex=PTHREAD_MUTEX_INITIALIZER;
```

This creates a new mutex lock, and initializes it to a default starting value. The functions below are used to manipulate the mutex:

Call	Description
int pthread_mutex_lock(pthread_mutex_t *mutex)	Locks the mutex. Blocks and does not return if mutex is already locked. If mutex locks successfully, this function returns with a 0. This function returns a non-0 value if something goes wrong.
int pthread_mutex_unlock(pthread_mutex_t *mutex)	Unlocks the mutex. Returns 0 if successful.
int pthread_mutex_destroy(pthread_mutex_t *mutex)	Destroys the mutex. Returns 0 if successful.

Type out the program below, calling it assg2p4.c.

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

int glob;

void *child(void *t)
{
      // Increment glob by 1, wait for 1 second, then increment by 1 again.
      printf("Child %d entering. Glob is currently %d\n", t, glob);
      glob++;
      sleep(1);
      glob++;
      printf("Child %d exiting. Glob is currently %d\n", t, glob);
}

int main()
{
      int i;
      glob=0;

      for(i=0; i<10; i++)
            child((void *) i);

      printf("Final value of glob is %d\n", glob);
      return 0;
}
```

Question 9 (1 mark)

What is the value of glob printed at the end of main?

Question 10 (3 marks)

Now modify main so that it spawns each call to "child" as a thread, giving us 10 threads in total. Describe the changes you made to the program.

Question 11 (3 marks)

Are the values of glob now correct? Explain why or why not.

Now modify your program as shown below. Statements in **<u>bold underline</u>** are newly added.

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

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

int glob;

void *child(void *t)
{
      // Increment glob by 1, wait for 1 second, then increment by 1 again.
      printf("Child %d entering. Glob is currently %d\n", t, glob);
      pthread_mutex_lock(&mutex);
      glob++;
      sleep(1);
      glob++;
      pthread_mutex_unlock(&mutex);
      printf("Child %d exiting. Glob is currently %d\n", t, glob);

      ... Other code you may have added ...
}

int main()
{
      int i, quit=0;
      glob=0;

      ... Codes and declarations that make your program multi-threading

      printf("Final value of glob is %d\n", glob);
      pthread_mutex_destroy(&mutex);
      ... Other code you may have added ...
}
```

Question 12 (3 marks)
Do your threads now update glob correctly? Explain your answer, and why the updates are correct/incorrect.

Question 13 (4 marks)

You will notice that the statement "printf("Final value of glob=%d\n", glob);" in main often executes before all the threads are done, causing the wrong value of glob to be printed. Modify your program so that this statement executes only after all threads complete.

Describe the modifications made, and cut and paste the code into your answer book. You will demo that your program works at the start of the next lab session.