Lab 5 (CS): Interprocess Communication (IPC)

Innopolis University
Course of Operating Systems

Lab Objectives

- IPC (Inter-process Communication)
  - Create processes which communicate and share data.
- Learn to create threads using pthread.h library.
- Review synchronization mechanisms among threads and how to solve critical region problem.
- Learn to work with diverse synchronization mechanisms (mutexes, condition variables, ...etc).

IPC

It provides a mechanism to exchange data across multiple processes. We have different ways to implement IPC.

- Pipes
  - a half-duplex method (or one-way communication). It has a read end `(pipefd[0])` and a write end `(pipefd[1])`. Data written to the write end of a pipe can be read from the read end of the pipe. `(man 7 pipe)`
  - lasts only as long as the process created it.
  - **int pipe(int pipefd[2])** system call. `(man 2 pipe)`
- Named pipes (FIFO files)
  - An extension to the traditional pipe and it can last as long as the system is up (beyond the life of the process).
  - Similar to regular files but it follows FIFO.
  - **mkfifo()** library function.
- Message Queues
- Shared Memory
- ...etc

Source: https://www.scaler.com/topics/operating-system/inter-process-communication-in-os/

IPC

Example:

- What are file descriptors?
  - File descriptors provide a primitive, low-level interface to input and output operations and are represented as objects of type `int`. They can represent a connection to a device (such as a terminal), or a pipe for communicating with another process.
- Write a program ex1.c that creates a (unnamed) pipe and then forks a child process.
- The parent reads a message as a command line argument, sends it via the pipe, then the child process reads the message from the pipe and prints it to its stdout. Make sure that the child process has opened the pipe before writing to it.
- Modify the previous program to let the child process send the length of the message (`strlen`) to the parent process.
- Write a script which reads the message as a command line argument and runs the program for 10 times.

POSIX Threads

A *langauge-independent* thread model (although its API is provided in C) developed by IEEE.

In C language, it provides functions and macros for:

- Thread management (creation, deletion, cancellation, etc).
- Mutexes
- Condition variables
- Advanced thread synchronization mechanisms.

Semaphores are not part of the standard, but they are provided as an appendix.

Thread Creation and Deletion

```
#include <pthread.h>

pthread_t pthread_self(void);

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

void pthread_exit(void *retval);
```

Thread Creation and Deletion

Example:

- Write a C program that demonstrates thread creation and deletion using the `pthread_create` function.
- Create two threads.
- Inside each thread's start routine function, use `printf` to display a message containing the thread id (`pthread_t`).
- After a brief execution, terminate the threads. Ensure that the program prints the messages from both threads correctly.
- **Hint:** use `gcc -pthread` to compile.

Critical region

Example:

- Write a C program that demonstrates a race condition between threads trying to update the value of a global variable `counter`.
- Create two threads.
- Both threads attempt to update the counter variable simultaneously without proper synchronization.
- Explain where the critical region is in this code and why a race condition occurs?

Thread Synchronization

The race condition is an undesirable situation in OS, and we need to find a way to prevent the threads/processes from entering the critical region/section. We can achieve this desire by synchronizing the threads/processes.

We will learn about 3 thread synchronization mechanisms:

- Joins
- Mutexes
- Condition variables (Optional)

Semaphores are also a widely used thread synchronization mechanism.

Joins

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

`pthread_join` blocks the calling thread until the specified thread has finished executing, storing its return value.

Thread joins

Example:

- Solve the race condition problem in counter variable using thread joins.
- Explain how using thread joins can help solve the problem of race conditions in multi-threaded programs.
- Provide a simple example where thread joins are used to ensure that a critical section of code is executed by only one thread at a time, preventing race conditions.
- Describe the problem scenario, the threads involved, and how thread joins are applied to resolve the issue.

Mutexes

Mutual exclusions: used to protect against race conditions.

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

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

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutexes

Example:

- Let's create a race condition!
- Write a program with a global variable `counter`.
- Spawn 1000 threads. Each of these threads increments `counter` 1000 times (one by one).
- Print the final value of `counter`. Explain the result.
- Repeat the same procedure, this time protecting `counter` with a mutex.

Exercise 1 (1/3)

- In this exercise, you will work on implementing a simple publish/subscribe messaging system via pipes and named pipes. This exercise consists of two parts (I and II).

Part I:

- write a program *channel.c* which runs two processes (publisher and subscriber) and one (unnamed) pipe between them.
- The publisher process reads a message from stdin and sends it to the subscriber who prints the message to stdout.
- The maximum size of a single message is 1024 bytes.
- Submit **channel.c**.

Part II:

- Write a program *publisher.c* which forks n child process which in turn open n named pipes in the path (/tmp/ex1). The parent process reads messages from stdin and the child processes publish the messages to the subscribers.

Exercise 1 (2/3)

- The publisher program accepts n (command line argument) as the maximum number of subscribers and creates n named pipes in the path ($/tmp/ex1/s\{i\}$) where $i \in [1, n]$. For example, the publisher creates the named pipe ($/tmp/ex1/s1$) for the first subscriber.
- Write another program *subscriber.c* which accepts a number *id* (command line argument) indicates the subscriber's *index* and opens the named pipe ($/tmp/ex1/s\{id\}$), reads the messages and prints to its stdout.
- Write a script **ex1.sh** to run one publisher and **n** subscriber where **n** is read as a command line argument for the script where ($0 < n < 4$). Run all subscribers and the publisher in separate shell windows (https://askubuntu.com/a/46630). Look at the output of all running subscribers and check whether they all received the message or some of them. Why we need to create n named pipes for n subscribers? Add the answer to the file **ex1.txt**.

Exercise 1 (3/3)

- Assume that all subscribers had subscribed to the channel. Do not create threads here, but only processes.
- The maximum size of a single message is 1024 bytes.
- The publisher/subscriber should always be ready to read/write messages from/to stdin/stdout. We can terminate them by sending termination signals such as **SIGINT** (Ctrl+C).
- Add your explanation to **ex1.txt** about your findings in this exercise with respect to inter-process communication via pipes and fifo files.
- Submit **publisher.c**, **subscriber.c**, **ex1.txt** and **ex1.sh**.
- **Note:** Ensure that the subscriber opens the pipe before you start writing to it, otherwise you would get **SIGPIPE** error.

Exercise 2

- Create a struct **Thread** which contains three fields *id* which holds the thread id, an integer number *i* and *message* as a string of size 256.
- Write a program **ex2.c** that creates an array of n threads using the struct **Thread**. The field *i* should store the index of the created thread (0 for the first thread and so on) whereas the field *id* contains the thread id and the field *message* contains the message "Hello from thread *i*" where *i* is the value of the field *i*. Each thread should output its *id* and *message*, then exit. Main program should inform about thread creation by displaying the message "Thread *i* is created".
- Do the threads print messages in the same order you created?
- Fix the program to force the order to be strictly Thread 1 is created, Thread 1 prints message, Thread 1 exits and so on.
- **Hint:** use gcc -pthread **ex2.c** to compile.
- Submit **ex2.c** and **ex2.sh** which runs the program.

Exercise 3 (1/3)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
// primality test
bool is_prime(int n){
    if (n <= 1) return false;
    int i = 2;
    for (; i * i <= n; i++){
        if (n % i == 0)
            return false;
    }
    return true;
}

// Primes counter in [a, b)
int primes_count(int a, int b)
{
    int ret = 0;
    for (int i = a; i < b; i++)
        if (is_prime(i) != 0)
            ret++;
    return ret;
}
```

```c
// argument to the start_routine
// of the thread
typedef struct prime_request
{
    int a, b;
} prime_request;


// start_routine of the thread
void * prime_counter(void *arg)
{
    // get the request from arg
    prime_request req = ...

    // perform the request
    int *count = ...

    return ((void *)count);
}
```

Exercise 3 (2/3)

- Given the code snippets in the previous slide, write a multi threaded C program **ex3.c** that accepts two integers as command line arguments: n and m. The program prints the number of primes in the range $[0, n)$, distributing the computation *equally* onto m threads.
- Divide the interval $[0, n)$ into m equal sub intervals (except maybe for the last one), spawn a separate thread to count the number of primes in each of those subintervals, and sum their individual results to get the final answer. The final answer will be the number of primes in the range [0, n).
- In addition to **ex3.c**, write a shell script **ex3.sh** that compiles the program and calculates its execution time with $n = 10,000,000$ and $m \in \{1, 2, 4, 10, 100\}$. Store the 5 timing results in **ex3_res.txt**. Finally, add a brief explanation of the findings in **ex3_exp.txt**.

Exercise 3 (3/3)

- Submit **ex3.c**, **ex3.sh**, **ex3_exp.txt** and **ex3_res.txt**.
- **Hint:** use `pthread_join` and **time** shell command.
- **Note:** Your main tasks in this exercise are as follows:
  1. Complete the start routine of the threads.
  2. Set up the threads.
  3. Set up the threads arguments (prime requests).
  4. Pass requests to the threads.
  5. Collect prime count from each thread.
  6. Aggregate the prime counts.
  7. Print the final result.
  8. Clean up!

Exercise 4 (1/3)

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

// primality test (from ex3)
bool is_prime(int n);

// The mutex
pthread_mutex_t global_lock =
    PTHREAD_MUTEX_INITIALIZER;

// Do not modify these variables
    directly, use the functions
    on the right side.
int k = 0;
int c = 0;

// input from command line
int n = 0;
```

```c
// get next prime candidate
int get_number_to_check()
{
    int ret = k;
    if (k != n)
        k++;
    return ret;
}

// increase prime counter
void increment_primes()
{
    c++;
}

// start_routine
void *check_primes(void *arg)
{
    while (1){
        // TODO
    }
}
```

Exercise 4 (2 / 3)

- In this exercise we are going to solve the problem from ex3 in a different way. We will distribute the primality checking computation of integers in $[0, n)$ between the m threads in a different manner.
- We will have a global state for the next number to check for primality, k, and the number of primes discovered thus far, c.
- Each of the m threads reads the global k, checks it for primality, and increments it. If it turned out to be prime, it increments the global c.
- It must be the case that no two threads check the same integer for primality, and the total of the thread checks must account for the interval $[0, n)$. Also, we want the thread execution to be as parallel as possible.

Exercise 4 (3 / 3)

You are given the function that checks an integer for primality, the functions for reading and incrementing the global k and c, and a mutex for handling k and c. Your task is as follows.

- Write the threads start_routine.
- Utilize the mutexes so that:
  - No two threads check the same integer for primality.
  - No integer is left unchecked by any of the threads.
  - The execution is as parallel as possible.
- Set up the threads and join them.
- Print the final result.
- Clean up!

Exercise 4 (3 / 3)

Note that you should not directly manipulate the global k and c. You are already given function to handle that for you. It is however your responsibility that the mutexes are set so that to prevent race conditions.

Write the functionality in `ex4.c`. Also, write a shell script `ex4.sh` that compiles the program and times its execution with $n = 10,000,000$ and $m \in \{1, 2, 4, 10, 100\}$. Store the 5 timing results in `ex4_res.txt`. Finally, provide a brief explanation of the results in `ex4_exp.txt` and compare them to the results from exercise 3.

Submit **ex4.c**, **ex4.sh**, **ex4_exp.txt** and **ex4_res.txt**

Useful tools:

- `pthread_mutex_t`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Exercise 5 (Optional)

In this exercise we are going to solve the previous primes counting problem by framing it as a producer consumer problem.

There is a global queue of fixed size and a global counter for the total number of primes. Threads are one of two types: producers or consumers. For simplicity, we will create only one producer and m consumers.

- The producer pushes prime candidates into the queue. A prime candidate is an integer that is not divisible by 2 or 3.
- Consumers pop a prime candidate from the queue and perform a complete primarlity test on them (optimized by skipping checks for divisibility by 2 or 3). If it turns out to be primes, they increment the global counter.

Exercise 5 (Optional)

You are given functions for producing into and consuming from the global queue, a function for incrementing the global prime counter, a mutex for handling it, and a pair of condition variables (and an associated mutex). Your task is as follows.

- Write thread procedures for producers and consumers.
- Utilize condition variables (and the associated mutex) to synchronize producers and consumers (a standard problem in computer science).
- Set up the threads for the one producer and m consumers.
- Join the threads.
- Clean up!

Again, don't manipulate the global variables directly. You are given functions to manipulate them. However, it is your responsibility to utilize mutexes and condition variables to ensure proper synchronization.

Exercise 5 (Optional)

Write the functionality in `ex5.c`. Also, write a shell script `ex5.sh` that compiles the program and times its execution with $n = 10,000,000$ and $m \in \{1, 2, 4, 10, 100\}$. Store the 5 timing results in `ex5_res.txt`.
Finally, provide a brief explanation of the results in `ex5_exp.txt` and compare them to the results from exercises 3 and 4.
Useful tools:

- `pthread_cond_wait`
- `pthread_cond_broadcast`

Exercise 6 (Optional)

Implement exercise 3 with semaphores instead of condition variables.

Exercise 7 (Optional)

- Write a multi-threaded solution for computing the least common multiple (also called the lowest common multiple or smallest common multiple) of a series of positive integers
- Your program will take two arguments from the command line. The first argument is the name of the file that contains the positive integer numbers, and the second argument is the number of threads T to be created. We assume that the number of threads is always smaller than the number of integers given in the input file.
- **Note:** For a single run, a single thread computes the least common multiple for a single integer. You need to synchronize the threads using one of the synchronization methods in a way to not duplicate the job and avoid race condition.
- Calculate the execution time of the program via *time* command by varying the number of threads and positive integers.
- Create a sample of positive integers and run your program.

References

- https://cyriacjames.files.wordpress.com/2015/05/assignment2-duejune9.pdf
- Threads Wikipedia page
- Tutorial on POSIX threads
- Example on the producer consumer problem
- The manual pages on POSIX threads contain examples.
- Tanenbaum is the best for gaining an intuition on threads.

End of lab 5 (CS)