Lab 5: Threads & Interprocess Communication (IPC)

Shokhista Ergasheva, Muwaffaq Imam, Artem Kruglov,
Nikita Lozhnikov, Giancarlo Succi, Xavier Vasquez
Herman Tarasau, Firas Jolha

Innopolis University
Course of Operating Systems

September 2022

Recap.

- How can we create n processes of the program ex.out?
- How did we share data between process last week?
- How to find the current process id of any running program ex.out using a shell command?
- How to find and kill a zombie process?
- What is the difference between fork and sleep functions w.r.t. function source (kernel or library) and usage (what is it used for)?
- What is the return type of fork system call?
- How can we avoid the orphan process in the operating systems?
- Who will be the new parent for the orphan process?

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) used for IPC between two related processes.
  - **pipe()** system call.
- Named pipes (FIFO files)
  - Full-duplex communication method.
  - Similar to regular files but it follows FIFO.
    - Similar to what you did last week in ex2.
  - **mkfifo()** library function.
- Message Queues
- Shared Memory
- ...etc

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

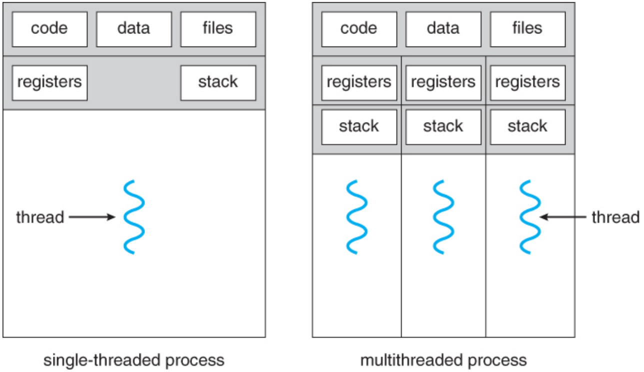
Exercise 1 (1/2)

- In this exercise, you will work on implementing a simple publish/subscribe messaging system. Your solution will emulate the functionality of a Telegram channel and provide the capability of broadcasting messages to its subscribers.
- Write a program *channel.c* which runs two processes (publisher and subscriber) and a single pipe between them.
- The publisher process reads some message from stdin and sends to the subscriber who prints the message to stdout.
- Write a program *publisher.c* which opens a named pipe (/tmp/ex1), reads messages from stdin and publishes the messages to the subscribers.
- Write another program *subscriber.c* which opens the named pipe, reads the messages and prints to 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. $(0 < n < 4)$

Exercise 1 (2/2)

- Run all subscribers and publisher in separate shell windows. (https://askubuntu.com/a/46630). Look at the output of all running processes and check if they all received the message or some of them. Check the case when you run only one subscriber.
- 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 for reading/writing messages from/to stdin/stdout. We can terminate them by terminating the programs (Ctrl+C).
- Submit **channel.c**, **publisher.c**, **subscriber.c** and **ex1.sh**.
- **Hints:** Make sure that you publish enough copies of the message for the subscribers (**n** copies). Make sure that the publisher can send only one message every second.

Threads



For more information, return to the lecture and tutorial materials and reference books.

POSIX Threads

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

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

- `pthread_t`
- `pthread_create`
- `pthread_exit`

Let us check the manual pages!

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.
- Create 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.

Critical region

Example:

- Write a shell script that produces a file of sequential numbers by reading the last number in the file, adding 1 to it, and then appending it to the file. Run one instance of the script in the background and one in the foreground, each accessing the same file
- How long does it take before a race condition manifests itself? What is the critical region?
- Modify the script to prevent the race
- Hint: use ln file file.lock to lock the data file

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

- `pthread_join`

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

Let us check the manual page!

Exercise 3 (1/2)

Write a C program **ex3.c** that accepts two command line arguments: n and m. The program prints the number of primes numbers in the range $[0, n)$, distributing the computation onto m threads.

Divide the interval $[0, n)$ into several m equal sub intervals (except maybe for the last), spawn a separate thread to count the number of primes in each of those intervals, and sum their individual results to get the final answer.

You are given a function that counts the number of prime numbers between two numbers. Your work with the C code will be the following.

- Write the threads procedure (where you will use the provided function).
- Set up the threads.
- Set up the threads arguments (utilizing the given interval structure).

Exercise 3 (2/2)

Write the functionality in `ex3.c`. Also, write a shell script `ex3.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 `ex3.txt`. Finally, provide a brief explanation of the results in `ex3.explanation.txt`.

Submit **ex3.c**, **ex3.sh**, **ex3.explanation.txt** and **ex3.txt**.

Useful tools:

- `pthread_join`
- **time** shell command.

Mutexes

Mutual exclusions: used to protect against race conditions.

- `pthread_mutex_t`
- `PTHREAD_MUTEX_INITIALIZER`
- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Let us read the man pages!

Exercise 4 (1 / 3)

In this exercise we are going to solve the problem in 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. (not necessarily in that order!).

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 (2 / 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 procedure.
- Utilize the mutexes in such a way that no two threads check the same integer for primality.
- Utilize the mutexes in such a way that no integer is left unchecked by any of the threads.
- Utilize the mutexes in a such that 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 don't have to 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 in a such a way 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.txt`. Finally, provide a brief explanation of the results in `ex4.explanation.txt` and compare them to the results from exercise 3.

Submit **ex4.c**, **ex4.sh**, **ex4.explanation.txt** and **ex4.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.txt`. Finally, provide a brief explanation of the results in `ex5.explanation.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.

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)