Lab 3

Interprocess Communication and Concurrency

3.1 Objectives

This lab is to learn about, and gain practical experience in interprocess communication and concurrency control in a general Linux environment. Shared memory allows multiple processes to share a given region of memory. It is the fastest form for different processes to communicate. Processes need to take care of the shared memory conflicting operations. The operating system provides concurrency control facility such as semaphore API.

After this lab, students will

- Design and implement a multi-processes concurrent program by using the producer-consumer pattern.
- Program with the fork() system call to create a new child process.
- Use the Linux shared memory API to allow processes to communicate and the Linux semaphore facility to synchronize processes.

3.2 Starter Files

The starter files are on GitHub at url: https://git.uwaterloo.ca/ece252-s25/ece252_starter/-/tree/main/lab3/starter.It contains the following sub-directories where we have example code to help you get started:

- the fork has example code of creating multiple processes and time the total execution time; it also demonstrate how a zombie process is created when the parent process does not call `wait` family calls;

- the sem has example code of using POSIX semaphore shared between processes;
- the shm has example code of using System V shared memory; and
- the cURL_IPC has example code of using a shared memory region as a cURL call back function buffer to download one image segment from a lab server by the child process and writing the downloaded image segment to a file by the parent process; and
- the tools has a shell script to compute statistics of timing data and a shell script to clean IPC facilities.

The lab3_eceubunt1.csv is the template file that you will need for submitting timing results (see Section 3.5.2).

3.3 Pre-lab Preparation

Read the entire lab3 manual to understand what the lab assignment is about. Build and run the starter code to see what they do. You should work through the provided starter code to understand how they work. The following activities will help you to understand the code.

1. Execute `man fork` to read the man page of `fork(2)`.
2. Execute `man 2 wait` to read the man page of `wait(2)` family system calls.
3. Execute `man ps` to read the man page of the `ps` command.
4. Execute `man shm_overview` to read Linux man page of POSIX shared memory API overview. At the bottom of the man page, it talks about system V shared memory facilities. Read the corresponding man pages of the system V shared memory API.
5. Execute `man sem_overview` to read Linux man page of POSIX semaphore API overview.
6. Execute `man ipcs` and `man ipcrm` to read the man pages of Linux IPC facility commands. You will find the `-s` and the `-m` options are helpful in this lab.

Linux man pages are also available on line at https://linux.die.net/.

The main data structure to represent the fixed size buffer is a queue. A circular queue is one commonly seen implementation of a fixed size buffer if FIFO is required. A stack is another implementation if LIFO is required. You can either create the data structure yourself or use one from an existing library. If you want to explore the C library queue facilities, check out the man pages of insque(3), remque(3) and queue(3). There are example code at the end of the man pages.

3.4 Lab Assignment

3.4.1 The Producer Consumer Problem

A producer-consumer problem is a classic multi-tasking problem. There are one or more tasks that create data and they are referred to as *producers*. There are one or more tasks that use the data and they are referred to as *consumers*. We will have a system of P producers and C consumers. Producers and consumers do not necessarily complete their tasks at the same speed. How many producers should be created and how many consumers should be created to achieve maximum latency improvement[1]? What if the buffer receiving the produced data has a fixed size? Another problem to think about is that when we fix the number of producers and consumers, how big the bounded buffer size should be? Is it true the bigger the buffer size is, the more latency improvement we will get, or there is a limit beyond which the bigger buffer size will not bring any further latency improvement? We will do some experiments to answer these questions by solving a similar problem that we solved in lab2 with some additional assumptions.

In lab2 we used multi-threading to download image segments from the web server and then paste all the segments together. This falls into the unbounded buffer producer consumer problem pattern. We can let producers download the image segments (i.e. creating data) and let consumers extract the image pixel data information (i.e. processing data) for future processing. One easy solution to lab 2 (also commonly seen) is to have one thread that does both the producer and the consumer jobs. This implicitly assumes that the number of producers and consumers are equal. But what if data creation and data processing are running at different speeds[2]? It may take more time to download data than to process data or vice versa. Then having the same number of producers and consumers are not optimal. In addition, in lab2, we did not restrict the receiving data buffer size. In a real world, resources are limited and the situation that a fixed size of buffer space to receive the incoming data is more realistic. In this lab, we have the additional constraint that the buffer to receive the image segments from the web server has a fixed size. So the problem we are solving is a bounded buffer producer-consumer problem[3].

[1] You probably have already noticed in lab2 that once the number of threads reaches a certain number, you reach the maximum performance improvement.

[2] For example, the data processing part could be more involved such as doing some image transformation. It could also be that the network bandwidth is tight or the lab server is slow so that it takes long to download the image segment.

[3] Here is another producer consumer problem example: you can think of the producer as a keyboard device driver and the consumer as the application wishing to read keystrokes from the keyboard; in such a scenario the person typing at the keyboard may enter more data than the consuming program wants, or conversely, the consuming program may have to wait for the person to type in characters. This is, however, only one of many cases where producer/consumer scenarios occur, so do not get too tied to this particular usage scenario.

3.4.2 Problem Statement

We are still solving an image concatenation problem. The image strips are the same ones that you have seen in lab2. In lab2, the lab web server sleeps a random seconds before it sends a random horizontal strip of an image to the client. In this lab, we have a different server running at port 2530 which sleeps for a fixed time before it sends a specific image strip requested by the client. The deterministic sleep time in the server is to simulate the time to produce the data. The image format sent by the server is still the simple PNG format (see Figure 1.2a). The PNG segment is still an 8-bit RGBA/color image (see Table 1.1 for details). The web server still uses an HTTP response header that includes the sequence number to tell you which strip it sends to you. The HTTP response header has the format of "X-Ece252-Fragment: M" where $M \in [0, 49]$. To request a horizontal strip with sequence number M of picture N, where $N \in [1, 3]$, use the following URL: `http://machine:2530/image?img=N&part=M`, where

`machine` is one of the following:

- `ece252-1.uwaterloo.ca`,
- `ece252-2.uwaterloo.ca`, and
- `ece252-3.uwaterloo.ca`.

For example, when you request data from http://ece252-1.uwaterloo.ca:2530/image?img=1&part=2, you will receive a horizontal image strip with sequence number 2 of picture 1 . The received HTTP header will contain "X-Ece252-Fragment: 2". The received data will be the image segment in PNG format. You may use the browser to view a horizontal strip of the PNG image the server sends. Each strip has the same dimensions of 400×6 pixels and is in PNG format.

Your objective is to request all horizontal strips of a picture from the server and then concatenate these strips in the order of the image sequence number from top to bottom to restore the original picture as quickly as possible for a given set of given input arguments specified by the user command. You should name the concatenated image as `all.png` and output it to the current working directory.

There are three types of work involved. The first is to download the image segments. The second is to process downloaded image data and copy the processed data to a global data structure for generating the concatenated image. The third is to generate the concatenated all.png file once the global data structure that holds the concatenated image data is filled.

The producers will make requests to the lab web server and together they will fetch all 50 distinct image segments. Each time an image segment arrives, it gets placed into a fixed-size buffer of size B, shared with the consumer tasks. When there are B image segments in the buffer, producers stop producing. When all 50 distinct image segments have been downloaded from the server, all producers will terminate. That is the buffer can take maximum B items, where each item is an

image segment. The horizontal image strips sent out by the lab servers are all less than $10,000$ bytes.

Each consumer reads image segments out of the buffer, one at a time, and then sleeps for X milliseconds specified by the user in the command line[4]. Then the consumer will process the received data. The main work is to validate the received image segment and then inflate the received IDAT data and copy the inflated data into a proper place inside the memory.

Given that the buffer has a fixed size, B, and assuming that $B < 50$, it is possible for the producers to have produced enough image segments that the buffer is filled before any consumer has read any data. If this happens, the producer is blocked, and must wait till there is at least one free spot in the buffer.

Similarly, it is possible for the consumers to read all of the data from the buffer, and yet more data is expected from the producers. In such a case, the consumer is blocked, and must wait for the producers to deposit one or more additional image segments into the buffer.

Further, if any given producer or consumer is using the buffer, all other consumers and producers must wait, pending that usage being finished. That is, all access to the buffer represents a critical section, and must be protected as such.

The program terminates when it finishes outputting the concatenated image segments in the order of the image segment sequence number to a file named all.png.

Note that there is a subtle but complex issue to solve. Multiple producers are writing to the buffer, thus a mechanism needs to be established to determine whether or not some producer has placed the last image segment into the buffer. Similarly, multiple consumers are reading from the buffer, thus a mechanism needs to be established to determine whether or not some consumer has read out the last image segment from the buffer [5].

Requirements

Let B be the buffer size, P be the number of producers, C be the number of consumers, X be the number of milliseconds that a consumer sleeps before it starts to process the image data, and N be the image number you want to get from the server. The producer consumer system is called with the execution command syntax of:

```
./paster2 <B> <P> <C> <X> <N>
```

The command will execute per the above description and will then print out the

[4]This is to simulate data processing takes time.

[5]Due to network transmission has randomness, the order of image segments placed in the buffer may not necessarily be the same order that they have been requested by the producers. The last image segment in the buffer may not necessarily be the image segment with the biggest sequence number. We do not want to request the same image segment twice since this will bring down the performance, so both producers and consumers know the buffer in total will serve 50 image segments.

time it took to execute. You should measure the time before you create the first process and the time after the last image segment is consumed and the concatenated all.png image is generated. Use the `gettimeofday` for time measurement (see starter code under the `fork` directory) and terminal screen for display. Thus your last line of output should look like:

```
paster2 execution time: <time in seconds> seconds
```

For a set of given `(B, P, C, X, N)` tuple values, run your application and measure the time it takes. Note for a give value of `(B, P, C, X, N)`, you need to run multiple times to compute the average execution time in a general Linux environment.

Implement each producer/consumer as an individual process. You start your program with one process which then forks multiple producer processes and multiple consumer processes. The parent process will wait for all the child processes to terminate and then start to process the data structure that holds the concatenated image data and create the final all.png file. Aside from the parent process, the P producer processes that download the image segments and C consumer processes that process the image segment data, you are allowed to create extra processes to do other type of work when you see a need. Just keep in mind that having more processes is not cost free. Hence a good implementation will try to minimize system resource usage unless extra resource usage will bring meaningful improvement.

Use shared memory for processes to communicate. You may use System V shared memory API. The bounded buffer is a shared data structure such as a circular queue that all processes share access to. Note that shared memory access needs to be taken care of at the application level. The POSIX semaphore are to be used for concurrency control.

A Sample Program Run

The following is an example execution of paster2 given $(B, P, C, X, N) = (2, 1, 3, 10, 1)$. In this example, the bounded buffer size is 2. We have one producer to download the image segments and three consumers to process the downloaded data. Each consumer sleeps 10 milliseconds before it starts to process the data. And the image segments requested are from image 1 on lab servers.

```
[eceubuntu1:]./paster2 2 1 3 10 1
paster2 execution time: 100.45 seconds
```

Note that due to concurrency, your output may not be exactly the same as the sample output above. Also depending on the implementation details and the platform where the program runs, the sample system execution time is only for illustration purpose. The exact paster2 execution time value your program produces will be different than the one shown in the sample run.

3.5 Deliverables

3.5.1 Pre-lab Deliverables

None.

3.5.2 Post-lab Deliverables

Put the following items under the directory named lab3:

1. All the source code and a Makefile. The Makefile default target is `paster2` executable file. That is command `make` should generate the `paster2` executable file. We also expect that the command `make clean` will remove the object code and the default target. That is the `.o` files and the executable files should be removed.

2. A timing result .csv file named lab3_hostname.csv which contains the timing results by running paster2 on a server whose name is hostname. For example, `lab3_eceubuntu1.csv` means paster2 was executed on the server eceubuntu1 and the file contains the timing results. The first line of the file is the header of the timing result table. The rest of the rows are the timing result command line argument values and the timing results. The columns of the .csv file from left to right are values of B, P, C, X, and the corresponding paster2 average execution time. We have an example .csv file in the starter code folder named `lab3_eceubuntu1.csv` for illustration purpose.

   Run your paster2 on `eceubuntu1`. Record the average timing measurement data for the (B, P, C, X, N) values shown in Table 3.1 for a particular host. Note that for each given (B, P, C, X, N) value in the table, you need to run the program n times and compute the average time. We recommend $n = 5$.

3. Submit your lab project 3 on ECE Marmoset Submission and Testing Server.

B	P	C	X	N	Time
5	1	1	0	1	
5	1	5	0	1	
5	5	1	0	1	
5	5	5	0	1	
10	1	1	0	1	
10	1	5	0	1	
10	1	10	0	1	
10	5	1	0	1	
10	5	5	0	1	
10	5	10	0	1	
10	10	1	0	1	
10	10	5	0	1	
10	10	10	0	1	
5	1	1	200	1	
5	1	5	200	1	
5	5	1	200	1	
5	5	5	200	1	
10	1	1	200	1	
10	1	5	200	1	
10	1	10	200	1	
10	5	1	200	1	
10	5	5	200	1	
10	5	10	200	1	
10	10	1	200	1	
10	10	5	200	1	
10	10	10	200	1	
5	1	1	400	1	
5	1	5	400	1	
5	5	1	400	1	
5	5	5	400	1	
10	1	1	400	1	
10	1	5	400	1	
10	1	10	400	1	
10	5	1	400	1	
10	5	5	400	1	
10	5	10	400	1	
10	10	1	400	1	
10	10	5	400	1	
10	10	10	400	1	

Table 3.1: Timing measurement data table for given (B, P, C, X, N) values.

3.6 Marking Rubric

The Rubric for marking is listed in Table 3.2.

Points	Sub-points	Description
100		Post-lab
	10	clean project files organization
	5	`Makefile`
	85	Complete implementation of `paster2` and timing data

Table 3.2: Lab3 Marking Rubric