COMP2046 Coursework Autumn 2024

Weight: 20% module marks

Deadline: 27th December 2024, 5pm Beijing time

Submission: Create a single scyXXX.zip (Student account) file containing your source code files and files provided along with this coursework. We will need to rebuild your code to test your implementation. You should submit your single zip file through Moodle.

Overview

The goal of this coursework is to make use of operating system APIs (specifically, the POSIX API in Linux) and simple concurrency directives to solve a number of synchronisation problems that occur on scheduling systems that are similar to the ones that you may find in fairly simple operating systems.

Completing all tasks will give you a good understanding of:

- The use of operating system APIs in Linux.
- Critical sections, semaphores, mutexes, and the principles of synchronisation/mutual exclusion.
- The implementation of linear bounded buffers of a fixed size using linked lists.
- The basics of concurrent/parallel programming using an operating system's facilities (e.g. semaphores, mutex).
- Basic process creation, memory image overwriting and shared memory usage.

Getting Help

You MAY ask Dr Qian Zhang and Dr Fiseha B. Tesema for help in understanding coursework requirements if they are not clear (i.e. what you need to achieve). Any necessary clarifications will then be added to the Moodle page so that everyone can see them.

You may NOT get help from anybody else to actually do the coursework (i.e. how to do it), including ourselves. You may get help on any of the code samples provided, since these are designed to help you to do the coursework without giving you the answers directly.

Background Information

All code should be implemented in C and tested on the school's Linux servers. Code that cannot be successfully compiled on the school's Linux servers using the provided source/header files **will affect your marks.**

Additional information on programming in Linux, the use of POSIX APIs, and the specific use of threads and concurrency directives in Linux can be found, e.g., here (it is my understanding that this book was published freely online by the authors and that there are no copyright violations because of this).

Additional information on the bounded buffer problem can be found in, e.g.:

- Tanenbaum, Andrew S. 2014 Modern Operating Systems. 4th ed. Prentice Hall Press, Upper Saddle River, NJ, USA., Chapter 2, section 2.3.4
- Silberschatz, Abraham, Peter Baer Galvin, Greg Gagne. 2008. Operating System Concepts. 8th ed. Wiley Publishing, Chapter 4 and 5
- Stallings, William. 2008. Operating Systems: Internals and Design Principles. 6th ed. Prentice Hall Press, Upper Saddle River, NJ, USA, Chapter 5
- In addition to these books, much of the information in the lecture notes should be extremely helpful

Please use only the system calls or POSIX libraries discussed in the lab. The use of any other system calls or POSIX libraries is not allowed.

Coding and Compiling Your Coursework

You are free to use a code editor of your choice, but your code MUST compile and run on the school's Linux servers. It will be tested and marked on these machines.

For task 1

We have provided a header file, `linkedlist.h`, which contains three function prototypes. The source file, `linkedlist.c`, includes the implementation of these functions. This code implements a basic singly linked list with the following three functions:

1. **addLast**: Adds a new element to the end of the list.
2. **addFirst**: Adds a new element to the beginning of the list.
3. **removeFirst**: Removes and returns the first element from the list.

It dynamically manages memory for the list nodes and handles edge cases, such as empty lists. The list structure is defined using a struct element that holds a data pointer and a next pointer, making the implementation flexible for various data types.

To ensure consistency across all students, changes are not to be made on the given source files. Note that, in order to use these files with your own code, you will be required to specify the file on the command line when using the gcc compiler (e.g. (updated Dec 4)

```
gcc –std=c99 TASKX.c -o TASKX linkedlist.c -pthread
```

and include the linkedlist.c file in your code (using #include "linkedlist.h").

For task 2

You should ensure that your code compiles using the GNU C-compiler, e.g., with the command:

(updated Dec 4)

```
gcc TASK2.c -o TASK2output -lrt -pthread
```

Copying Code and Plagiarism

You are allowed to freely copy and adapt code samples provided in lab exercises or lectures. Additionally, you may use examples from Linux/POSIX websites, which offer many code snippets to help with specific tasks. This coursework assumes and encourages the use of these resources as part of the learning process. Since you are not claiming this code as your own original work, this does not constitute plagiarism.

Please note that some provided examples may omit error checking for clarity. It is your responsibility to include appropriate error checking in your implementation.

However, you must not copy code from any other sources, including:

- Another student, whether from this course or any other course.
- Any third-party sources, such as ChatGPT or similar tools.

If you do so, it will be considered an attempt to pass someone else's work as your own, which is plagiarism. The University treats plagiarism very seriously. Consequences may include receiving a zero for the coursework, failing the module, or facing more severe penalties.

Task 1: Concurrency problem: Bounded buffers (60 %)

In this task you will implement a bounded buffer (i.e. of fixed size) of characters in two different ways.

In task 1a, you will:

- Use a single producer and single consumer.
- Synchronise using binary semaphores (or mutexes where appropriate).

In task 1b, you will:

- Use multiple producers and multiple consumers.
- Synchronise using counting semaphores (or mutexes where appropriate).

In both cases, you will produce/consume a pre-determined number of elements. Both tasks will give you good insights into the different synchronisation approaches for bounded buffers.

Task 1a: Bounded buffer with binary semaphores (30 marks)

You are asked to implement a FCFS bounded buffer (represented as linked list). The buffer can contain at most MAX_BUFFER_SIZE (defined in the task1a.c code templet) elements, and each element contains one character (a '*' in this case). You are asked to implement a single producer and a single consumer. The producer generates MAX_NUMBER_OF_JOBS (defined in the task1a.c code templet) '*' characters and adds them to the end of the buffer (provided that empty spaces are available). The consumer removes '*' characters from the start of the buffer (provided that full elements are available). Each time the producer (consumer) adds (removes) an element, the number of elements currently in the buffer is shown on the screen as a line of stars (see output sample provided task1a.txt). Different implementations and synchronisation approaches are possible, however, we are asking you to use only binary semaphores (or mutexes where appropriate) in your implementation.

The final version of your code will include:

- A linked list of characters representing the bounded buffer utilised in the correct manner. The maximum size of this list should be configured to not exceed, e.g., 50 elements.
- A producer function that generates stars ('*') and adds them to the end of the buffer (using addLast) as soon as space is available.
- A consumer function that removes elements from the start of the buffer using removeFirst (one at a time).
- A visualisation function that displays the exact number of elements currently in the buffer (using a line of stars) on the screen every time an element is added to or removed from the buffer.
- The code to:
    - Declare all necessary semaphores/mutexes and initialise them to the correct values.
    - Create the producer/consumer threads.
    - Join all threads with the main thread to prevent the main thread from finishing before the consumers/producers have ended.

- Generate output similar in format to the example provided for this requirement (task1a.txt) (for 100 jobs, using a buffer size of 10).

Task 1b: Bounded buffer with counting semaphores (30 marks)

You are asked to, similar to task1a, implement a bounded buffer of characters, however, this time with multiple producers (NUMBER_OF_PRODUCERS defined in task1b.c code templet), multiple consumers (NUMBER_OF_CONSUMERS defined in task1b.c code templet), and counting semaphores. The producers generate a total of MAX_NUMBER_OF_JOBS (defined in the task1b.c code templet) elements, which are removed by multiple consumers. Every time one of the producers or consumers adds or removes an element, the number of elements currently in the buffer is shown as a line of '*' characters (see output sample on Moodle). Note that every producer/consumer has a unique Id assigned to them in the output.

Similar to task1a, a correct implementation of this requirement includes:

- A linked list of characters representing the bounded buffer utilised in the correct manner. The maximum size of this list should be configured to not exceed, e.g., 50 elements.
- Multiple producer threads that generate a total number of MAX_NUMBER_OF_JOBS '*' characters, and not more. That is, the number of elements produced by the different threads sums up to MAX_NUMBER_OF_JOBS. Elements are added to the buffer as soon as free spaces are available. Note that every producer may end up producing a different number of elements, depending on the way threads are interleaved, but that the total items produced should be equal to MAX_NUMBER_OF_JOBS.
- A consumer function that removes elements from the start of the buffer using removeFirst (one at a time).
- A mechanism to ensure that all consumers terminate gracefully when MAX_NUMBER_OF_JOBS have been consumed.
- A visualisation function that displays the exact number of elements currently in the buffer (using a line of stars) on the screen every time an element is added to or removed from the buffer.
- The code to:
    - Declare all necessary semaphores/mutexes and initialise them to the correct values.
    - Create the producer/consumer threads.
    - Join all threads with the main thread to prevent the main thread from finishing before the consumers/producers have ended.
    - Generate output similar in format to the example provided in this coursework (task1b.txt) for this requirement (for 100 jobs, using a buffer size of 10).

Marking criteria:

The criteria used in the marking task1a and task1b of your coursework include:

- Whether you have submitted the code and you are using the correct naming conventions and format.
- Whether the code compiles correctly, and runs in an acceptable manner.
- Whether you utilise and manipulate your linked list in the correct manner.
- Whether semaphores/mutexes are correctly defined, initialised, and utilised.
- Whether consumers and producers are joined correctly.
- Whether the correct number of producers and consumers has been utilised, as specified in the coursework description.
- Whether consumers and producers end gracefully/in a correct manner.
- Whether consumers/producers have been assigned a unique id (as can be seen from the output provided on Moodle).
- Whether the exact number of elements is produced and consumed.
- Whether your code is efficient, easy to understand, and allows for maximum parallelism (i.e. no unnecessary synchronisation is applied).
- Whether unnecessary/inefficient/excessive busy waiting is used in your code.
- Whether your code runs free of deadlocks.
- Whether the output generated by your code follows the format of the examples provided on Moodle.

Submission requirements:

- Give your solutions extending the given code template task1a.c and task1b.c
- Name your output for the tasks “task1a.txt” and “task1b.txt” respectively. Please stick rigorously to the naming conventions, including capitalisation.
- Add your task1a.c and task1b.c code files, as well as the output files task1a.txt and task1b.txt, to your submission folder. Include the provided linkedlist.c and linkedlist.h files in the submission folder as well.
- Your code must compile using (updated Nov28)
  `gcc -std=c99 task1a.c -o task1a linkedlist.c -pthread` and
  `gcc -std=c99 task1b.c -o task1b linkedlist.c -pthread` on the school’s linux servers.

Task 2: Shared Memory (40 marks)

Use the knowledge you've obtained during the labs, finish the following question. You are asked to implement a solution for using shared memory for inter-process communication. Requirements of implementation are:

- A parent process (TASK2.c) that creates two new child processes;
- The process image of the two child processes is overwritten to execute ChildP1.c and ChildP2.c respectively.
- In TASK2.c, you are required to randomly generate an integer (RandInt) within the range of 1 to 20 and print out the value of RandInt. The integer RandInt will be stored in shared memory with an appropriate structure for future usage.
- The ChildP1 will wait for a random amount of time and then double the value of RandInt. As soon as ChildP1 modified the RandInt, the other child process ChildP2 subtracting a value of 10 from the existing value.
- ChildP1 and ChildP2 need to take turn performing doubling and subtraction on RandInt 10 times. You are not permitted to use sleep() or any similar function to manually control the execution order of processes.
- Make sure that the ChildP2.c won't be able to access to the RndInt before ChildP1.c finishes its task.
- Use shared memory to store all the values (intermediate results) of RandInt.
- Once all child processes have finished their tasks, the parent process needs to print out the value of RandInt at each instance.
- Your code must be compired using: (updated Dec 4)

```
gcc -lrt ChildP1.c -o ChildP1 -pthread
```

```
gcc -lrt ChildP2.c -o ChildP2 -pthread
```

```
gcc -lrt TASK2.c -o TASK2output -plhtread
```

For this task, you are required to submit three files**: TASK2.c, ChildP1.c** and **ChildP2.c**