South China University of Technology

《Operating System》Experiment Report

Experiment Title： Session 2: Solving IPC Problems

Name： 谭演锋 Student ID： 202130100456

Class： 计算机全英联合班 Group：

Collaborator：

Teacher： 钟竞辉

|  |
| --- |
| **Description** |
| 【Objective and Requirement】  **Objective:** Learn how to code the Barber problem and Reader & Writer Problem by semaphore  **Requirement:**  **Task 1: Sleeping Barber Problem**.  The first line of input consists of four integers, representing: the number of customers (n), the number of hairdressers, the number of chairs, and the haircut time (in seconds). The following n lines (2^n+1) provide information about each customer, with each line containing two variables: the i-th customer and the arrival time of the customer. If there is no customer, the barber falls asleep; If a customer come in the shop:   1. If all chairs are occupied, the customer leaves the shop; 2. If the barber is busy and there are free chairs, the customer sits in one of the free chairs; 3. If the barber is asleep, the customer wakes up the barber.   When a customer enters the shop and there is an opportunity for a haircut (either there is an available seat or the haircut can begin immediately), print: “customer i: there are n customers in front waiting for a haircut”  If the remaining number of chairs is 0, the customer will leave directly, and the output should be:“customer i: no more empty chairs, customer leaves”  When the customer has finished the haircut, print:“customer i finished haircut”  **Task 2: Reader & Writer Problem**   1. The first line of input consists of two integers. The first integer indicates the priority order of readers and writers (1 for reader priority, 2 for writer priority), and the second integer indicates the number of readers and writers to be entered next. Following that, each line contains 4 variables, with the data entered in the following order: process number i, reader or writer indicator, the time when process i starts execution, and the duration of process i. Pressing Enter on the last line signifies the end of input.New reader come in and spend 1 second to read the data. New writer come in and spend 6 seconds to update the data; 2. If readers are reading data, the writers must wait until all readers finish their jobs. 3. If writers are updating data, the readers must wait until the writers finish his job.   Every time a reader or writer applies for reading or writing, print: writer(/reader) i waiting to write(/read)  When the reader or writer begins reading or writing, print: writer(/reader) i starts to write(/read)  When the reader or writer completes reading or writing, print: writer(/reader) i ends writing(/read)  【Environment】  Operating System：CentOS7 |
| **Content** |
| 【Procedure】  **Task 1**  We have 3 variables of semaphores for synchronization. The barber\_ready represent the free barber number, the customer\_ready represent the number of customers waiting for the barbering. Finally, the mutex1 is used for protect the variable num\_chair which will be used to judge whether customer leave or stay.    Fig.1 Global variable  First, we set the function down and up for a good looking of the the semaphore operation. We define two function to simulate the behavior of customers and barber. For the barber, the function **down(customer\_ready)** waits for the customer, if a customer is ready, it will wake up to work, else it will be blocked . Then we use the **mutex1** to protect the variable **number\_chair**. When it finishes, it will release ‘**barber\_ready**’ semaphore to let next ready customer getting on the barber seat, which is waking up the barber. Then the barber gives haircut(sleeping for several seconds). In order to get the customer ID we use a global vector to store sequence of lining customer.  For a customer, it sleeps for the arrival time before attempting to enter the barber shop (**std::this\_thread::sleep\_for(std::chrono::seconds(arrival\_time))**). Then it acquires a lock (**down(mutex1)**) to check if there is an available chair, which is used to protect the judge statement and the operation of **num\_chair.** If there is a chair, it prints the status and updates the number of chairs. It signals that the customer is ready (**up(customer\_ready)**) to wake up the barber, releases the lock, and waits for the barber (**down(barber\_ready)**).  If there is no chair, the customer leaves, prints a message, and releases the lock.t    Fig.2 Implement detail of customer and barber function  First we read input for the number of customers, barbers, chairs, and haircut time. Then we use the pair variable to get the customer and their arrive time.  Then we initializes semaphores (**sem\_init**) and sets initial values, where **barber\_ready** is equal to the number of barber indicating the available barber. The **customer\_ready** equal to 0 indicating no customer now. Then the mutex1 is equal to 1. Here the 0 in the **sem\_init** indicating the type of semaphores.  Then we create a thread for the barber **(std::thread barber\_thread(barber)**) and create customer threads using a lambda function.  Finally we join customer threads and barber thread to the main thread. After finishing the task we destroys semaphores (**sem\_destroy**).    Fig.3 Implement detail of main function  The running detail is showed in Fig.4. We have 5 customer, 1 barber, 1 seat and 3 second for haircut.  In the 2 second the customer 3 first comes in, then the barber start to work.  In the 3 second the customer 2 comes in, then he seat down.  In the 4 second the customer 4 comes in, but there are no seats, so he leaves.  In the 5 second the customer 3 finish the haircut, then customer 2 start to haircut.  In the 7 second the customer 1 comes in, then he seat down.  In the 8 second the customer 2 finish the haircut, then customer 1 start to haircut.  In the 10 second the customer 5 comes in, then he seat down.  In the 11 second the customer 1 finish the haircut, then customer 5 start to haircut.  In the 14 second the customer 5 finish the haircut, the program end.    Fig.4 Running result of task 1  **Task 2**  In order to allow the priority setting, we add variables **priority**, **write\_count** and **reader\_wait** to accomplish the priority function. The **muetx1** is used to protect the **read\_count** and the **write\_count**. **write\_mutex** is used to block the writer, the **reader\_wai**t is used to block **reader\_wait**.    Fig.5 Global variable  For the reader, it sleeps for a specified start\_time before attempting to read. Then it uses semaphores (**reader\_wait**, **mutex1**, and **write\_mutex**) to synchronize access to shared resources. **reader\_wait** is used to block the reader if the **priority = 2**, which represent the writer first getting access to the data. **mutex1**is used to protect the read\_count and the judgement. **write\_mutex** is used to block the writer when the reader is reading data, it is down when the first reader comes in, which also means **priority =1**. After finishing reading, it do the some process with **mutex1** to decrement read\_count and releases the **write\_mutex** if it was the last reader.    Fig.6 Implement detail of reader function  For the writer, the process is similar. It sleeps for a specified start\_time before attempting to read. Then it uses semaphores (**reader\_wait**, , and **write\_mutex**) to synchronize access to shared resources. **down(reader\_wait)** is used to block the reader when the **priority = 2**, therefore the writer first getting accessing to the data. Then the **write\_count** ++, since one time there is only a writer process, so we do not need **mutex1** to protect data. After finishing writing, it releases the **reader\_wait** if it was the last writer and the priority =2. Then finally release the **write\_mutex**.    Fig.7 Implement detail of writer function  In the main function, it first reads the priority and the number of processes from the input. Then it initializes three semaphores (mutex1, write\_mutex, and reader\_wait) using sem\_init and setting them all equal to 1 since they are all mutex semaphores. Then i reads input for each process, including the process ID, type ('R' for reader, 'W' for writer), start time, and duration. Based on the type of process, it creates a thread for either the reader or writer function and adds it to the vector of threads. Finally, Wwits for all threads to finish using join.  Destroys the semaphores using sem\_destroy    Fig.8 Implement detail of main function  The running detail is showed in Fig.9. We have 5 processes and the reader first.  In the 3 second the reader 1 comes in, waits and stars reading.  In the 4 second the writer 2 comes in, waits since the 3 reader is reading.  In the 5 second the reader 3 comes in, waits and stars reading.  In the 6 second the reader 4 comes in, waits and stars reading.  In the 7 second the reader 3 finished.  In the 8 second the reader 1 finished.  In the 9 second the writer 5 comes in, waits since the reader 4 is reading.  In the 11 second the reader 4 finished and release the lock.  In the 11 second the writer 2 start to write first.  In the 16 second the writer 2 finished and release the lock.  In the 16 second the writer 5 start to write.  In the 19 second the writer 5 finished and the program ends.    Fig. 9 Running result of task 2 |
| **Conclusion** |
| In the experiment, I met difficulty in how to utilize synchronization and mutual exclusion techniques to solve problem. I do not know how to use the **semaphore** in c++. Then I also met problem in getting the input value and use the input value to create threads. I also met problem of how to make the priority in the task of Reader & Writer problem.  In completing the assigned tasks of implementing the Barber Problem and the Reader-Writer Problem in C++, I have gained valuable experience in concurrent programming and synchronization. The Barber Problem simulation involved managing a scenario where customers arrive, wait, and get haircuts, considering factors such as available chairs and haircut duration.I have learned the using of library **<semaphore.h>** to create semaphore and use the down and up operation for IPC. I also learn to using the library **<thread>** to create multiple thread.  Additionally, implementing the Reader-Writer Problem provided insights into handling shared resources among multiple threads. I learn to use the **mutex** for shared variables.I also had to consider different priorities for readers and writers, leading to the development of a synchronization mechanism to control access to the shared resource. Therefore, I have learned to using a mores semaphore **reader\_wait** than the classical reader and writer problem.  Throughout both tasks, I applied concepts such as semaphores, thread management, and synchronization to create robust and efficient solutions. These exercises enhanced my understanding of concurrent programming challenges and provided practical experience in solving them using C++. The skills developed through these tasks will undoubtedly prove valuable in addressing similar challenges in future projects or real-world scenarios involving concurrent systems.  The experiment most impressed me that this experiment meaningfully enhances my understanding of the principle and usage of semaphore in solving inter-process communication problems in the computer operating system. |
| **Teacher’s Comments and Score** |
| Comment：  Score：           Signature：                                                 Date： |