South China University of Technology

《Operating System》Experiment Report

Experiment Title： Session 2: Solving IPC Problems

Name： 向天翼 Student ID： 201836020389

Class： 18计算机联合班 Group：

Collaborator：

Teacher： 钟竞辉

|  |
| --- |
| **Description** |
| 【Objective and Requirement】  **Objective:** Synchronization and mutual exclusion of Process and Thread.  **Requirement:**  **Task 1: Sleeping Barber Problem**.  The barber shop has 6 chairs, 1 barber and 1 barber chair; 20 customers come in the barber shop randomly; 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.   **Task 2: Reader & Writer Problem**   1. 10 readers and 10 writers try to access data S; 2. New reader come in and spend 1 second to read the data. New writer come in and spend 6 seconds to update the data; 3. If readers are reading data, the writers must wait until all readers finish their jobs. 4. If writers are updating data, the readers must wait until the writers finish his job.   【Environment】  Operating System：Ubuntu 18.04.4 LTS |
| **Content** |
| 【Procedure】  **Task 1**  We define two function to simulate the behavior of customers and barber. For the barber, it first waits for the customer, if a customer is ready, it will wake up to work. Then it will update the number of vacant seats. This operation is protected by the ‘mutex’ semaphore. When he finishes, it will release ‘barber\_ready’ semaphore to let next ready customer getting on the barber seat. For a customer, to see whether there are any vacant seats, it will first wait to access ‘free\_seat’. If there is no vacant it will exit. Otherwise, it will occupy one seat and awake the barber.    Fig.1 Implement detail of customer and barber function  The running detail is showed in Fig.2. When the program is running, we keep track of the number of free spaces in the barber shop. When a person enters the barbershop, they will check whether there is a vacancy, and if there is no vacancy, they will leave directly. From the running results, we can see that starting from the customer with id 12, they all left the barber shop. When the barber finishes the haircut, he will continue to serve the next customer, and the vacant position will increase by 1 at the same time, which can also be seen from the result.    Fig.2 Running result of task 1  **Task 2**  In order to allow readers and writers to alternately enter the thread pool, we define a linked list that contains the same number of 1 as the writer and the same number of 0 as the reader. This linked list is shuffled every time the program is started, and then started according to the order of the linked list. Threads for readers and writers. This can achieve the purpose of allowing readers and writers to enter alternately.  For readers, it will first check whether it is the first reader in the current reader queue. If it is, it will prevent the writer from entering. Then update the number of readers. After reading, it will check if it is the last reader in the current reader queue, and if it is, it will allow the writer to enter. For writers, it only needs to wait when there are readers, and then start writing after all readers exit. The running detail is showed in Fig.4.    Fig.3 Implement detail of writer and reader function  When the working time for a single writer is longer than the reader, just like the requirement given by the problem. Since the working time for a single writer is longer than the reader, readers will be collectively blocked when they arrive. However, due to the shorter working hours of readers, readers can read immediately after the writer is finished, so multiple readers will complete the reading at the same time, as the Figure 4 shows.    Fig. 4 Running result of task 2 (Working time: Writer > Reader)  When the working time for a single writer is equal to the reader, there is almost never the situation of too many readers completing tasks at the same time as mentioned before, which is showed in Figure 5.    Fig. 5 Running result of task 2 (Working time: Writer = Reader)  When the working time for a single writer is shorter than the reader, we can see that the situation that too many readers completing tasks at the same time has occurred again. Lots of writers are collectively blocked when the readers are reading. When the readers are finished, they start to write the data one by one, which is showed in Figure 6.    Fig. 6 Running result of task 2 (Working time: Writer < Reader)  **Appendix: Source code for every task.**  //Task 1 barber.cpp  #include<pthread.h>  #include<stdio.h>  #include<stdlib.h>  #include<semaphore.h>  #include<unistd.h>  const int SEAT\_MAX = 6;  const int CUSTOMER\_MAX = 20;  void down(sem\_t &x){sem\_wait(&x);}  void up(sem\_t &x){sem\_post(&x);}  sem\_t barber\_ready,cust\_ready,mutex;  int free\_seat;  void init(){  sem\_init(&barber\_ready,0,1);  sem\_init(&cust\_ready,0,0);  sem\_init(&mutex,0,1);  free\_seat = SEAT\_MAX;  }  void\* \_barber(void\* arg){  while(true){  down(cust\_ready);  down(mutex);  free\_seat++;  up(mutex);  sleep(5);  printf("Barber has finished work from one customer!\n");  up(barber\_ready);  }  }  void\* \_customer(void\* arg){  down(mutex);  if(free\_seat>0){  free\_seat--;  printf("Customer whose id = %d has sit down! Free seat = %d\n",\*((int\*)arg),free\_seat);  up(mutex);  up(cust\_ready);  down(barber\_ready);  }else{  up(mutex);  printf("Customer whose id = %d has gone away!\n",\*((int\*)arg));  }  }  int main(){  init();  pthread\_t barber\_thread;  pthread\_t customer\_threads[CUSTOMER\_MAX];  int customer\_id[CUSTOMER\_MAX];  pthread\_create(&barber\_thread,NULL,\_barber,NULL);  for (int i = 0; i < CUSTOMER\_MAX; i++)  {  //sleep(rand()%10);  usleep((rand()%10)\*500000);  //sleep(1);  customer\_id[i] = i;  pthread\_create(&customer\_threads[i],NULL,\_customer,&customer\_id[i]);  }  pthread\_join(barber\_thread,NULL);  for (int i = 0; i < CUSTOMER\_MAX; i++)  {  pthread\_join(customer\_threads[i],NULL);  }  }  }  //Task 2: writer\_reader.cpp  #include<pthread.h>  #include<stdio.h>  #include<stdlib.h>  #include<semaphore.h>  #include<unistd.h>  #include<vector>  #include<algorithm>  #include<random>  #include<chrono>  const int WRITER\_MAX = 10;  const int READER\_MAX = 10;  const int IS\_READER = 0;  const int IS\_WRITER = 1;  const int WRITER\_WORKING\_TIME = 6;  const int READER\_WORKING\_TIME = 1;  void down(sem\_t &x){sem\_wait(&x);}  void up(sem\_t &x){sem\_post(&x);}  sem\_t mutex,wmutex;  int reader\_count;  void init(){  sem\_init(&mutex,0,1);  sem\_init(&wmutex,0,1);  reader\_count = 0;  }  void\* \_reader(void \*arg){  down(mutex);  if(reader\_count == 0) down(wmutex);  reader\_count++;  up(mutex);  sleep(READER\_WORKING\_TIME);  printf("Reader whose id = %d has finished reading!\n",\*((int\*)arg));  down(mutex);  reader\_count--;  if(reader\_count == 0) up(wmutex);  up(mutex);  }  void\* \_writer(void \*arg){  down(wmutex);  sleep(WRITER\_WORKING\_TIME);  printf("Writer whose id = %d has finished writing!\n",\*((int\*)arg));  up(wmutex);  }  std::vector<int> get\_order(){  std::vector<int> reader\_order(READER\_MAX,IS\_READER);  std::vector<int> writer\_order(WRITER\_MAX,IS\_WRITER);  writer\_order.insert(writer\_order.end(),reader\_order.begin(),reader\_order.end());  unsigned seed = std::chrono::system\_clock::now().time\_since\_epoch().count();  std::shuffle(writer\_order.begin(),writer\_order.end(),std::default\_random\_engine(seed));  return writer\_order;  }  int main(){  init();  pthread\_t reader\_threads[READER\_MAX];  int reader\_id[READER\_MAX];  pthread\_t writer\_threads[WRITER\_MAX];  int writer\_id[WRITER\_MAX];  std::vector<int> order = get\_order();  printf("Reading time for a single reader is %d second(s).\n", READER\_WORKING\_TIME);  printf("Writing time for a single writer is %d second(s).\n", WRITER\_WORKING\_TIME);  printf("\n");  for (int i = 0,j = 0; i + j < order.size();)  {  sleep(1);  if(order.at(i+j) == IS\_WRITER){  writer\_id[i]=i;  pthread\_create(&writer\_threads[i],NULL,\_writer,&writer\_id[i]);  i++;  } else if(order.at(i+j) == IS\_READER){  reader\_id[j]=j;  pthread\_create(&reader\_threads[j],NULL,\_reader,&reader\_id[j]);  j++;  }  }    for (int i = 0; i < WRITER\_MAX; i++)  {  pthread\_join(writer\_threads[i],NULL);  }  for (int i = 0; i < READER\_MAX; i++)  {  pthread\_join(reader\_threads[i],NULL);  }  } |
| **Conclusion** |
| From this lab session I ‘ve learned the how to utilize synchronization and mutual exclusion techniques to solve “Sleeping Barber Problem” and “Reader and Writer problem” by C++ in Linux operating system. In the Readers and Writers problem, I also learned that the time relationship between their respective work has a great impact on the final result. If the working hours of the two differ by too much, the situation of collective blocking will occur. This process 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： |