

**Lab Report**

|  |  |
| --- | --- |
| **Course**: | Operating System Principle |
| **Semester**: | 2nd semester of the academic year **2023-2024** |
| **Major**: | Software Engineering |
| **Class**: | 2022 |
| **Student Name**: | 吴孜远 |
| **Student ID:** | 222022321062009 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Process Synchronization --- Sleeping TA and Thread Pool | | | |
| Date | | May, 2024 | Type | | □Confirmatory  √ Design  √ Comprehensive |
| 1. **Objective & Requirements**    1. Understanding the concept and application of process synchronization    2. Learn how to use the synchronization mechanisms provided in Linux and pthread library, i.e. the mutex lock and semaphore    3. Grasp the use of process synchronization tools in the solving of real word problems, e.g. the sleeping TA problem or the thread pool problem. | | | | | |
| 1. **Experimental environment (**platform and software**)**   Ubuntu 18.04 or higher versions | | | | | |
| 1. **Experimental content and design** (Main Content, Procedure, Codes and Results) 2. Task 1   **The Sleeping Teaching Assistant**  A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA’s office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time. Using POSIX threads, mutex locks, and/or semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.  Using Pthreads, begin by creating N students. Each will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping. Perhaps the best option for simulating students programming—as well as the TA providing help to a student—is to have the appropriate threads sleep for a random period of time using the sleep() API:     1. Please provide your procedure to perform the tasks and source codes.   我们先来看整个逻辑，整理核心代码如下，包括详细注释：  学生：  while (1) {  // 问问题前先自己编程随机的时间  int ran = rand() % 10 + 1;  prinf("student %d is programming for %d seconds...\n", i, ran);  sleep(ran);    // 有问题来问老师  sem\_wait(&mutex); //互斥锁，保证一个座位只由一个人获取  if (seats\_ava > 0) { // 有空余座位，可以坐下  seats\_ava--;  sem\_post(&waiting); // 坐下后发出通知已有学生等待，唤醒老师  sem\_post(&mutex);  sem\_wait(&ta\_ava); // 等待老师有空（不在睡觉或不在辅导学生）    sem\_wait(&mutex1); // 互斥锁，保证同时只有一个学生占有老师（离开座位和去办公室）  seats\_ava++; //空出一个座位  printf("\tstudnt %d is getting help for %d seconds\n", i, teach\_time);  sleep(teach\_time); // 学生也进入辅导  printf("\t\tThe student %d's TA process done!\n", i);  sem\_post(&mutex1);  } else { // if there is no seats  sem\_post(&mutex);  printf("No seats! student %d go back\n",i);  }  }  老师：  while (1) {  sem\_wait(&waiting);// 没有学生就sleep，直到学生将其唤醒  teach\_time = rand() % 10 + 1;  sem\_post(&ta\_ava); // ta发出通知，可以进行辅导了  sleep(teach\_time); // 辅导（线程休眠）    // 若已经没有等待的学生  if(seats\_ava == N\_SEATS)  {  printf("no students waiting, sleep!\n");  }  }  这里我们一共用到了四个x信号量，一个全局变量：  int seats\_ava：用于记录当前剩余座位数，主要用于判断。  sem\_t ta\_ava：用于标记ta是否可用，初始值为0，表示一开始就是在睡觉，不可用。  sem\_t waiting：用于标记是否有学生在座位上等待，初始值为0，表示没有。它可以随着等待学生数目的增加而增加。  sem\_t mutex：互斥锁，用于防止多个学生同时修改一个座位变量，初始值为1。  sem\_t mutex1：为什么要加这个锁mutex1呢？  原因是我们将辅导的过程简化为两个线程同时休眠，但是我经过实验发现不能保证同时休眠，也就是说会发生如下情况：老师先结束休眠，继续下一个循环，发现还有学生在等待，向学生发出通知表明自己可辅导，那么这个学生就直接获取老师开始辅导，此时上一个学生还在进行辅导（线程还在休眠）。我们发现这时就有两个学生同时在辅导，所以我们直接加锁保证同时只有一个学生能获取老师的辅导（P(ta)）。  至于main函数，值得一提的就是命令行传参的方法：  N\_STUDENTS = atoi(argv[1]);  N\_SEATS = atoi(argv[2]);  Argc是参数个数，argv[0]是我们输入的可执行程序的名字。  我们直接看编译运行结果吧：    可以看到没有什么问题，各个线程有序进行，没有顺序错乱等错误。我这里没有设置终止条件，学生在编程和辅导之间来回切换。  学生4先来问问题，唤醒老师，开始辅导，这期间学生3、0、2依次开始等待，学生4辅导完之后，又回去自己编程，接下来学生3开始辅导，这期间学生1开始等待，学生4又来问问题发现没座位让它等待了，所以又回去……  接下来附上完整代码：   1. #include <stdio.h> 2. #include <stdlib.h>    // For rand() and srand() 3. #include <time.h>      // For time(0) 4. #ifndef \_\_USE\_GNU 5. #define \_\_USE\_GNU 6. #endif 7. #include <pthread.h> 8. #include <unistd.h>    // for sleep 9. #include <semaphore.h> //semaphore 10. *//shared semaphore* 11. sem\_t ta\_ava; 12. sem\_t waiting; 13. sem\_t mutex; 14. sem\_t mutex1; 15. int N\_STUDENTS; 16. int N\_SEATS; 17. int seats\_ava; 18. int teach\_time; 19. void \*student(void \*param); */\* threads call this function \*/* 20. void \*ta(void \*param); */\* threads call this function \*/* 21. int main(int argc, char \*argv[]) 22. { 23. srand(time(0)); 24. N\_STUDENTS = atoi(argv[1]); 25. N\_SEATS = atoi(argv[2]); 27. seats\_ava = N\_SEATS; 29. pthread\_t \*students = (pthread\_t \*)malloc(N\_STUDENTS \* sizeof(pthread\_t)); 30. pthread\_t ta\_pid; 32. pthread\_attr\_t attr; */\* set of thread attributes \*/* 33. pthread\_attr\_init(&attr);*/\* get the default attributes \*/* 34. *//initialize semaphore* 35. sem\_init(&ta\_ava, 0, 0); 36. sem\_init(&waiting, 0, 0); 37. sem\_init(&mutex, 0, 1); 38. sem\_init(&mutex1, 0, 1); 39. for(int i = 0; i < N\_STUDENTS; i++) 40. { 41. int \*arg = malloc(sizeof(\*arg)); *// 为每个参数分配内存* 42. \*arg = i; 43. pthread\_create(&students[i], &attr, student, arg); 44. } 45. pthread\_create(&ta\_pid, &attr, ta, NULL); 47. for(int i = 0; i < N\_STUDENTS; i++) 48. { 49. pthread\_join(students[i], NULL); 50. } 51. pthread\_join(ta\_pid, NULL); 52. *//destroy* 53. sem\_destroy(&ta\_ava); 54. sem\_destroy(&waiting); 55. sem\_destroy(&mutex); 56. sem\_destroy(&mutex1); 57. return 0; 58. } 59. void \*student(void \*param) 60. { 62. int i = \*(int \*)param; *//type conversion* 63. cpu\_set\_t mask; 64. CPU\_ZERO(&mask);     *//clear* 65. CPU\_SET(0, &mask);   *//set core i* 66. sched\_setaffinity(0, sizeof(mask), &mask); 68. while (1) { 69. *// random programing time* 70. int ran = rand() % 10 + 1; 71. printf("student %d is programming for %d seconds...\n", i, ran); 72. sleep(ran); 74. *// if there have available seats* 75. sem\_wait(&mutex); 76. if (seats\_ava > 0) { 77. seats\_ava--; 78. sem\_post(&waiting); *// sit and wait* 79. sem\_post(&mutex); 81. sem\_wait(&mutex1); 82. sem\_wait(&ta\_ava); 83. seats\_ava++; 84. printf("\tstudent %d is getting help for %d seconds\n", i, teach\_time); 85. *//sem\_wait(&ta\_ava);* 86. sleep(teach\_time); 87. printf("\t\tThe student %d's TA process done!\n", i); 88. sem\_post(&mutex1); 89. } else { *// if there is no seats* 90. sem\_post(&mutex); 91. printf("No seats! student %d go back\n",i); 92. } 93. } 95. pthread\_exit(0); 96. } 97. void \*ta(void \*param) 98. { 99. cpu\_set\_t mask; 100. CPU\_ZERO(&mask);     *//clear* 101. CPU\_SET(1, &mask);   *//set core i* 102. sched\_setaffinity(0, sizeof(mask), &mask); 104. while (1) { 105. *// student wake ta up* 106. sem\_wait(&waiting); 107. teach\_time = rand() % 10 + 1; 108. *// ta available now* 109. sem\_post(&ta\_ava); 111. sleep(teach\_time); 112. *//sem\_post(&ta\_ava);* 114. if(seats\_ava == N\_SEATS) 115. { 116. printf("no students waiting, sleep!\n"); 117. } 118. } 120. pthread\_exit(0); 121. } | | | | | |
| 1. Result analysis and discussion   In this part, you are required to provide your analysis of experimental results and summing up the harvest and the existing problems; besides, you are required to provide your thinkings about the questions:   * Investigate the concept of Thread Pool and the underlying techniques in the implementation of Thread Pool. Try to get the similarities between Thread Pool and the problem of Sleeping TA or Sleeping Barber. Describe how you can implement a thread pool according to your way of solving the Sleeping TA problem.   本实验的ta问题其实就是换了个背景的理发师问题，主要看能不能合理地利用信号量避免冲突，将线程排个序。  思考题：  线程池（Thread Pool）是一种多线程处理形式，它允许在后台线程中执行任务，而不是创建和销毁线程。线程池的主要优势在于减少了在处理大量任务时频繁创建和销毁线程所产生的性能开销。  线程池的基本组成部分通常包括：  工作队列：用于存放待执行任务的队列。  工作线程：一组预先创建的线程，它们不断地从工作队列中取出任务并执行。  同步机制：用于控制线程对工作队列的访问，确保线程安全。  任务提交机制：用于向工作队列提交新任务。  线程池的实现通常涉及以下技术：  互斥锁（Mutex）：用于保护共享资源，如工作队列。  条件变量（Condition Variable）：用于线程间的协调，比如当工作队列为空时让工作线程等待新任务的到来。  信号量（Semaphore）：用于控制对共享资源的访问数量，如限制同时执行任务的线程数。  线程池与 Sleeping TA 问题或 Sleeping Barber 问题有相似之处，主要体现在资源管理和线程间的协调上。比如说在我们做的 Sleeping TA 问题中，学生线程需要等待 TA 线程的帮助，这与线程池中工作线程等待新任务类似。  在 Sleeping TA 问题中，使用等待队列来模拟线程池中的任务队列，学生线程在请求帮助时进入队列。使用互斥锁和条件变量来控制对任务队列的访问，类似于线程池中对工作队列的同步。  如何根据解决 Sleeping TA 问题的方法来实现线程池呢？  创建固定数量的工作线程，并让它们等待新任务。  使用互斥锁保护任务队列，并将新任务安全地添加到队列中。  每一个工作线程循环：（类似于TA）  使用互斥锁进入临界区。  检查任务队列是否为空。  如果队列不为空，取出一个任务并离开临界区。  如果队列为空，使用条件变量让工作线程等待，直到有新任务到达。  执行任务：工作线程执行取出的任务。 | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |