|  |  |
| --- | --- |
| 实验编号：**Lab3** | |
| 实验题目：使用信号量解决N线程屏障问题 | |
| 实验学时：4 | 实验日期：2022.12.4 |
| 实验目的：  使用Nachos操作系统信号量机制，编写程序解决N线程屏障问题。 | |
| 硬件环境：  联想品牌 thinkbook14+ 型号笔记本  AMD Ryzen7-6800H CPU  32GB内存  512GB SSD | |
| 软件环境：  宿主机：Windows 10 22H2 64位  虚拟机软件：VMware Workstation Pro 16.1.2 build-17966106  Linux：Ubuntu 14.04.6 LTS Desktop i386 (Trusty Tahr)  gcc/g++：(Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4  MIPS交叉编译器：gcc-2.8.1-mips.tar.gz  Nachos：Nachos-3.4-UALR-2022 | |
| 实验步骤与内容：  实验具体内容：   1. 分析说明Nachos的信号量是如何实现的。 2. 在Nachos中是如何创建及运行并发(而非线程自己主动调用Yield放弃CPU)线程的。 3. 先按“The Little Book of Semaphores”中3.6.4小节中的代码实现N线程屏障。用不同的随机数种子seed测试(./nachos -rs seed)，是否会发现有可能多个线程均判定自己为最后一个到达的线程，这个现象是什么原因造成的？该现象会导致N线程屏障出现与有题目要求不一致的错误码？ 4. 请修改代码消除上面3中出现的现象。 5. 用不同的随机数种子测试，是否会发现各线程打印输出的rendezvous行的顺序，基本就是线程被创建的顺序(0,1,2…9)的现象？这是为什么，难道-rs选项没有起作用？试验在打印输出rendezvous之前加延迟(用软件空循环耗时)或Linux的sleep能否解决此问题，并解释为什么。 6. 请试着修改代码解决上面5出现的现象。提示：不用修改Nachos的核心实现代码，修改的是我们编写的N线程屏障的代码。  *1.分析说明Nachos的信号量是如何实现的* **（1）信号量的结构**  信号量的数据结构定义在./threads目录下的synch.h中   1. class Semaphore { 2. public: 3. Semaphore(const char debugName, int initialValue); // set initial value 4. ~Semaphore(); // de-allocate semaphore 5. char getName() { return name;} // debugging assist 7. void P(); // these are the only operations on a semaphore 8. void V(); // they are both atomic 10. private: 11. char name; // useful for debugging 12. int value; // semaphore value, always >= 0 13. List queue; // threads waiting in P() for the value to be > 0 14. };   定义了PV操作；  成员变量有name-信号量名，value-可用资源数，queue-阻塞队列  **（2）信号量中的PV操作**  *P(),V()方法在文件synch.cc中实现*   1. // Semaphore::P 2. // Wait until semaphore value > 0, then decrement. Checking the 3. // value and decrementing must be done atomically, so we 4. // need to disable interrupts before checking the value. 5. // 6. // Note that Thread::Sleep assumes that interrupts are disabled 7. // when it is called. 8. void 9. Semaphore::P() 10. { 11. IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts 13. while (value == 0) { // semaphore not available 14. queue->Append((void )currentThread); // so go to sleep 15. currentThread->Sleep(); 16. } 17. Value--; // semaphore available, 18. // consume its value 19. (void) interrupt->SetLevel(oldLevel); // re-enable interrupts 20. } 22. // 23. // Semaphore::V 24. // Increment semaphore value, waking up a waiter if necessary. 25. // As with P(), this operation must be atomic, so we need to disable 26. // interrupts. Scheduler::ReadyToRun() assumes that threads 27. // are disabled when it is called. 28. // 30. void 31. Semaphore::V() 32. { 33. Thread thread; 34. IntStatus oldLevel = interrupt->SetLevel(IntOff); 36. thread = (Thread )queue->Remove(); 37. if (thread != NULL) // make thread ready, consuming the V immediately 38. scheduler->ReadyToRun(thread); 39. value++; 40. (void) interrupt->SetLevel(oldLevel); 41. }   **P()方法为请求资源操作**  P操作请求资源。先检查变量value的值，如果value的值等于0，说明没有可分配的资源，于是将当前线程加入queue阻塞队列，并将该线程置为睡眠（阻塞）状态。当value的值大于0时，该线程消耗value值，进入就绪队列  **V()方法为释放资源操作**  V操作释放资源，并从阻塞队列中取出一个线程，如果取出的线程不为空，则将该线程转为就绪态，并将value的值加1，表示释放一个资源  一个信号量通过调用P(),V()方法，对value值进行加减，就表示资源的消耗与释放。 *2.在Nachos中是如何创建及运行并发(而非线程自己主动调用Yield放弃CPU)线程的* 对./threads目录下的thread.cc和scheduler.cc文件进行分析  **（1）创建线程：fork()**   1. void 2. Thread::Fork(VoidFunctionPtr func, int arg) 3. { 4. #ifdef HOSTALPHA 5. DEBUG('t', "Forking thread "%s" with func = 0x%lx, arg = %ldn", 6. name, (long) func, arg); 7. #else 8. DEBUG('t', "Forking thread "%s" with func = 0x%x, arg = %dn", 9. name, (int) func, arg); 10. #endif 12. StackAllocate(func, arg); 14. IntStatus oldLevel = interrupt->SetLevel(IntOff); 15. scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts 16. // are disabled! 17. (void) interrupt->SetLevel(oldLevel); 18. } 19. void 20. Scheduler::ReadyToRun (Thread thread) 21. { 22. DEBUG('t', "Putting thread %s on ready list.n", thread->getName()); 24. thread->setStatus(READY); 25. readyList->Append((void )thread); 26. }   首先为线程分配栈资源（StackAllocate()），然后由全局调度者scheduler将线程放入就绪队列中（scheduler->ReadyToRun(this)），等待被调度即可。  **（2）线程切换：yield()**   1. void 2. Thread::Yield () 3. { 4. Thread nextThread; 5. IntStatus oldLevel = interrupt->SetLevel(IntOff); 7. ASSERT(this == currentThread); 9. DEBUG('t', "Yielding thread "%s"n", getName()); 11. nextThread = scheduler->FindNextToRun(); 12. if (nextThread != NULL) { 13. scheduler->ReadyToRun(this); 14. scheduler->Run(nextThread); 15. } 16. (void) interrupt->SetLevel(oldLevel); 17. }   *\*ASSERT()的作用是对括号中表达式的真值进行验证，如果为假，会打印一条出错信息并中止程序，如果为真则继续运行。*  在线程进行切换时，scheduler在就绪队列中寻找下一个新的线程（scheduler->FindNextToRun()）进行调度，如果就绪队列中有线程，将当前运行的线程放入就绪队列中（scheduler->ReadyToRun(this)），同时执行下一个线程（scheduler->Run(nextThread)）。  **（3）线程阻塞：sleep（）**   1. void 2. Thread::Sleep () 3. { 4. Thread nextThread; 6. ASSERT(this == currentThread); 7. ASSERT(interrupt->getLevel() == IntOff); 9. DEBUG('t', "Sleeping thread "%s"n", getName()); 11. status = BLOCKED; 12. while ((nextThread = scheduler->FindNextToRun()) == NULL) 13. interrupt->Idle(); // no one to run, wait for an interrupt 15. scheduler->Run(nextThread); // returns when we've been signalled 16. }   先将状态设置为阻塞态，然后从就绪队列中寻找新的线程来调度（FindNextToRun()），如果就绪队列中有线程就可以执行该线程（run()），如果就绪队列中没有线程则阻塞等待时钟中断到来（idle()）。与线程切换不同，线程阻塞会进入阻塞等待状态。  继续分析system.cc文件  使用**timer**来设置时间片进行轮转   1. scheduler = new Scheduler(); // initialize the ready queue 2. if (randomYield) // start the timer (if needed) 3. timer = new Timer(TimerInterruptHandler, 0, randomYield); 4. static void 5. TimerInterruptHandler(int dummy) 6. { 7. if (interrupt->getStatus() != IdleMode) 8. interrupt->YieldOnReturn(); 9. }   所以，在Nachos中，系统会设置一个timer作为时间片，线程的**时间片耗完**后，**sheduler会进行线程切换**，将当前进程转为就绪态，并从就绪队列中唤醒一个新线程，**实现了Nachos中的线程并发** *3. 先按“The Little Book of Semaphores”中3.6.4小节中的代码实现N线程屏障。用不同的随机数种子seed测试(./nachos -rs seed)，是否会发现有可能多个线程均判定自己为最后一个到达的线程，这个现象是什么原因造成的？该现象会导致N线程屏障出现与有题目要求不一致的错误码？* **（1）根据3.6.4的实现思路编写了代码文件threadsbar.cc**   1. #include <stdio.h> 2. #include <stdlib.h> 3. #include <sys/types.h> 4. #include <sys/stat.h> 5. #include <fcntl.h> 6. #include <unistd.h> 7. #include "copyright.h" 8. #include "system.h" 10. #include "synch.h" 12. #define NTHREADS 10 // the number of threads 13. #define NTICKS 1000 // the number of ticks to advance simulated time 15. Thread producers[NPROD]; //array of pointers to the producer 16. Thread consumers[NCONS]; // and consumer threads;  19. Semaphore mutex; //semaphore for the mutual exclusion 20. Semaphore barrier; //屏障信号量 22. Thread threads[NTHREADS]; 23. int count=0; 25. void MakeTicks(int n) // advance n ticks of simulated time将模拟时间提前到下一个预定的硬件中断 26. { 28. }  31. void BarThread(int which) 32. { 33. //MakeTicks(NTICKS); 34. printf("Thread %d rendezvousn", which);  37. mutex->P(); 38. count=count+1; 39. mutex->V(); 41. if(count==NTHREADS){ 42. printf("Thread %d is the lastn", which); 43. barrier->V(); 44. } 45. barrier->P(); 46. barrier->V(); 47. printf("Thread %d critical pointn", which); 48. }  51. void ThreadsBarrier() 53. { 54. //printf("enter 1n"); 55. mutex=new Semaphore("mutex", 1); 56. barrier=new Semaphore("mutex", 0); 57. int i; 59. // Create and fork NTHREADS threads 60. for(i = 0; i < NTHREADS; i++) { 61. // printf("enter for loopn"); 62. threads[i]=new Thread(""+i); 63. threads[i]->Fork(BarThread, i); 64. }; 65. }   **（2）编译后运行**  **①输入：./nachos -rs 1**    ……顺序将随机数种子值从1递增逐个输入，在输入种子值为12时出现了多个线程均判定自己为最后一个线程的情况，如下  **②输入：./nachos -rs 12**  线程9、1、5都认为自己是最后一个到达的线程    **（3）问题原因分析**    如图，代码中对count的数值进行判断时是在**互斥锁之外**进行的，如果n个线程之前的线程在count+1操作之后，if判断之前**产生中断**，当第n个线程到达，使count的值变为希望的线程总数后，**之前产生中断的线程**此时再继续执行判断语句，就会**认为自己是最后一个线程**，产生图中的结果。这个现象会导致if语句块中的**信号量的V()操作执行多于一次**，**barrier的值也会产生大于1的现象**。  而如果第n个线程和之前的某个线程（如第n-1个线程）都在count+1操作后发生了中断，当出现第n个线程仍处于中断，但第n-1个线程中断结束了的情况时，第n-1个线程就会进行判断并认为自己是最后一个线程，同时解开屏蔽，这就会导致出现与题目要求不一致的错误。  ***4.请修改代码消除上面3中出现的现象。***  使用**两个旋转栅门**信号量（信号量小书将前面用到的这种一对快速连续的wait()和signal()函数称为旋转栅门）进行控制，代码如下   1. #define N\_THREADS 10 // the number of threads 2. #define N\_TICKS 1000 // the number of ticks to advance simulated time  5. Semaphore mutex; //semaphore for the mutual exclusion 6. Semaphore turnstile1,turnstile2; //屏障信号量 8. Thread threads[N\_THREADS]; 9. int count=0; 11. void MakeTicks(int n) // advance n ticks of simulated time将模拟时间提前到下一个预定的硬件中断 12. { 14. }  17. void BarThread(int which) 18. { 19. //MakeTicks(NTICKS); 20. printf("Thread %d rendezvousn", which); 21. mutex->P(); 22. count=count+1; 23. if(count==N\_THREADS){ 24. printf("Thread %d is the lastn", which); 25. turnstile2->P(); //锁定栅门2 26. turnstile1->V(); //解锁栅门1 27. } 28. mutex->V(); 30. turnstile1->P(); 31. turnstile1->V(); 32. printf("Thread %d critical pointn", which); 34. mutex->P(); 35. count=count-1; 36. if(count==0){ 37. turnstile1->P(); 38. turnstile2->V(); 39. } 40. mutex->V(); 42. turnstile2->P(); 43. turnstile2->V(); 44. }  47. void ThreadsBarrier() 49. { 50. //printf("enter 1n"); 51. mutex=new Semaphore("mutex", 1); 52. turnstile1=new Semaphore("turnstile1", 0); 53. turnstile2=new Semaphore("turnstile2", 1); 54. int i; 56. // Create and fork NTHREADS threads 57. for(i = 0; i < NTHREADS; i++) { 58. // printf("enter for loopn"); 59. threads[i]=new Thread(""+i); 60. threads[i]->Fork(BarThread, i); 61. }; 62. }     再次执行：./nachos -rs 12 运行结果如下  **异常现象消除**    ***5.用不同的随机数种子测试，是否会发现各线程打印输出的rendezvous行的顺序，基本就是线程被创建的顺序(0,1,2…9)的现象？这是为什么，难道-rs选项没有起作用？试验在打印输出rendezvous之前加延迟(用软件空循环耗时)或Linux的sleep能否解决此问题，并解释为什么。***      ……  测试了1~5的随机种子值，其中**只有3的顺序不是0-9**，**其余随机种子值**的各线程打印输出的rendezvous行的顺序都是线程创建的顺序（0-9）  **尝试在打印前加入sleep**    再次执行了1~5的随机种子数的情况，**所得结果与之前的一样**。Sleep并不能解决这个问题。因为sleep函数是将当前线程挂起，而因为每个线程在打印前都有sleep函数，且挂起时间相同，所以sleep结束继续执行的顺序仍然是线程创建的顺序，因此打印的顺序并没有变。  我认为是**每个线程执行的时间过短**，在大部分情况下**执行时间**都**小于**随机产生的**时间片**，所以不会执行时间片结束时的中断操作和线程切换，所以基本上打印的rendezous的顺序就是线程创建的顺序。  ***6.请试着修改代码解决上面5出现的现象。提示：不用修改Nachos的核心实现代码，修改的是我们编写的N线程屏障的代码。***  在打印rendezous的代码前**加入了强制提前系统模拟时间的方法**，让时间片提前走完，**使线程进入中断**。通过不断调用timer中的OneTick()方法，使模拟时间前进，从而进入中断    N\_TICKS设置为较大的1000，使提前的时间很大概率大于时间片大小，从而使线程进入中断   1. void MakeTicks(int n) // advance n ticks of simulated time将模拟时间提前到下一个预定的硬件中断 2. { 3. for (int i = 1; i<=n;++i) 4. interrupt->OneTick(); 5. } 7. }   再次测试，**发现打印rendezous的顺序与线程创建的顺序不同了** | |
| 结论分析与体会：  通过本次实验，我对nachos中的信号量、线程、中断等方面的知识有了更多的了解，通过阅读*信号量小书*并动手编写代码的方式，理解了N线程屏蔽问题以及该问题在nachos中的具体实现方法，并在*信号量小书*的启发与引导下，对代码进行了改进，解决了第一次代码运行时出现的多个进程都认为自己是最后一个线程的问题。在第二个问题的解决过程中，结合本实验中第二点的“线程创建以及并发”问题，对nachos的线程切换方式进行了初步的了解，采用提前系统时间至时间片末并进入中断的方式，使线程进行了切换，从而解决了代码运行出现的打印结果不随机的问题。  同时，在实验代码编写与修改过程中，也使我对于makefile文件、gdb调试方法等linux下编程的基础知识有了更深入的认识，收获很多。 | |