山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100300063 | 姓名：李彦浩 | | 班级：工软 21 |
| 学号：202100300340 | 姓名：黄幸兒 | | 班级：工软 21 |
| 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：2023.11.2 | |
| 实验目的：  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。 | | | |
| 硬件环境：  联想Thinkpad笔记本  Intel Core i5-8250 CPU 8核  8GB内存  1907GB SSD | | | |
| 软件环境：  宿主机：Windows 10 21H2 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： 目录 [1. 分析说明Nachos信号量是如何实现的 2](#_Toc3510)  [2. 在Nachos中是如何创建及运行并发线程的 4](#_Toc12704)  [3. 按照The Little Book of Semaphores-3.6.4节实现N线程屏障，并探查一个BUG 14](#_Toc28480)  [3.1. N线程屏障实现 15](#_Toc20217)  [3.2. last thread 16](#_Toc26613)  [4. 修改代码消除Q3中出现的现象 18](#_Toc11958)  [5. 尝试不同随机种子，发现各线程执行顺序就是线程被创建顺序。简述原因。试验一下在各线程开始执行前加上延迟是否能解决这个问题，如果能，简述原因 20](#_Toc14019)  [5.1. 打印顺序问题 20](#_Toc5469)  [5.2. 加入延迟 22](#_Toc12856)  [5.3. 为什么加入延迟就解决上述问题了？ 24](#_Toc6879) 分析说明Nachos信号量是如何实现的数据结构  |  | | --- | | char\* name; // useful for debugging  int value; // semaphore value, always >= 0  List \*queue; // threads waiting in P() for the value to be > 0 |   name:用来debugg打印调试的，不太重要。  value:信号量值，如果>0表示还有临界资源可以使用，否则没有  queue:等待临界资源的线程列表 成员函数  |  | | --- | | void  Semaphore::P()  {  IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts    while (value == 0) { // semaphore not available  queue->Append((void \*)currentThread); // so go to sleep  currentThread->Sleep();  }  value--; // semaphore available,  // consume its value    (void) interrupt->SetLevel(oldLevel); // re-enable interrupts  } |   P操作：又称为wait操作。当前线程访问临界资源时，尝试P操作，如果没有临界资源，他将被挂到信号量的等待列表里，随后进入睡眠。需要注意的是nachos中实现信号量的细节与The Little Book Of Semaphore中的不同，后者先做的value自减，而前者在确保了不会线程不会阻塞后才自减。因此Nachos中的Semaphore的value最小是0。   |  | | --- | | void  Semaphore::V()  {  Thread \*thread;  IntStatus oldLevel = interrupt->SetLevel(IntOff);  thread = (Thread \*)queue->Remove();  if (thread != NULL) // make thread ready, consuming the V immediately  scheduler->ReadyToRun(thread);  value++;  (void) interrupt->SetLevel(oldLevel);  }} |   V操作：又称为signal操作。当前线程访问完临界资源后，尝试V操作。此时如果在临界资源等待列表中有等待的线程，声明这个线程想要访问临界资源。  注意不是直接让这个线程执行！当前线程访问完临界资源后可能还有别的任务！ 示例 假设现在有两个线程t1,t2想要对某个临界资源进行访问。   1. t1调用P操作，此时信号量值=1-1=0 2. 假设此时进行了线程切换，t1进入readyList，t2开始执行，它首先调用P操作。然后发现此时没有临界资源，被挂在了等待队列里。并Sleep()。 3. t2睡眠会从readyList找一个准备好的线程继续执行，那么此时readyList里只有t1，t1执行。注意Sleep中不会将当前线程置入readyList，因为此时t2的状态是Blocked而不是Ready！因此此时readyList中没有线程。  |  | | --- | | void  Thread::Sleep ()  {  Thread \*nextThread;    ASSERT(this == currentThread);  ASSERT(interrupt->getLevel() == IntOff);    DEBUG('t', "Sleeping thread \"%s\"\n", getName());  status = BLOCKED;  while ((nextThread = scheduler->FindNextToRun()) == NULL)  interrupt->Idle(); // no one to run, wait for an interrupt    //执行下一个线程前并没有把当前睡眠的线程置入readyList中！  scheduler->Run(nextThread); // returns when we've been signalled  } |  1. t1上一次停在了V操作前，这次它继续执行，调用V操作。它从临界资源等待列表中挑了一个出来，等待列表只有一个t2，那么t2被拿出来，放到了readyList中，随后t1继续做自己的任务。 2. 假设此时发生了线程切换，t1进入readyList，换t2执行，对于t2来说，它刚刚执行完:  |  | | --- | | currentThread->Sleep(); |  1. 然后它进行下一轮while循环的判断，但是此时信号量的值由于t1释放资源变成了1，不会进入while循环，它会继续执行自己剩下的代码部分。（访问临界资源，然后V操作）  在Nachos中是如何创建及运行并发线程的 既然要多线程并发，那么在虚拟机的某个进程中需要开辟至少两个线程。在这点上，发行的nachos在内核程序的启动入口中提供了一个样例。   |  | | --- | | //main.cc  int  main(int argc, char \*\*argv)  {  int argCount; // the number of arguments  // for a particular command  DEBUG('t', "Entering main");  (void) Initialize(argc, argv); //in system.cc  #ifdef THREADS  ThreadTest(); //in threadtest.cc  #if 0  SynchTest();  #endif  #endif  //...  //system.cc  void  Initialize(int argc, char \*\*argv)  {  //...  // We didn't explicitly allocate the current thread we are running in.  // But if it ever tries to give up the CPU, we better have a Thread  // object to save its state.  currentThread = new Thread("main");  currentThread->setStatus(RUNNING);  //...  //threadtest.cc  void  SimpleThread(\_int which)  {  int num;    for (num = 0; num < 5; num++) {  printf("\*\*\* thread %d looped %d times\n", (int) which, num);  currentThread->Yield();  }  }  //----------------------------------------------------------------------  // ThreadTest  // Set up a ping-pong between two threads, by forking a thread  // to call SimpleThread, and then calling SimpleThread ourselves.  //----------------------------------------------------------------------  void  ThreadTest()  {  DEBUG('t', "Entering SimpleTest");  Thread \*t = new Thread("forked thread");  t->Fork(SimpleThread, 1);  SimpleThread(0);  } |   在main中，如果运行的是THREADS中的可执行产物，那么将执行ThreadTest()这个函数。这就是一个样例，创建一个新的线程，然后令该线程执行SimpleThread函数，随后在当前线程中也同样执行一遍这个SimpleThread函数。  这里要注意的是Nachos中只有兄弟线程概念，没有父子线程概念。一个线程的Fork操作会调用StackAllocate函数，也就是说在栈空间上实际产生了一个同名但完全独立的线程，而非共享执行Fork操作的线程的栈空间。而线程需要结束时，都是下一个要调度的线程清理其栈空间，这一点我们可以从scheduler模块下的Run方法中看出来。   |  | | --- | | //scheduler.cc :: Run  //...  SWITCH(oldThread, nextThread);  //...  if (threadToBeDestroyed != NULL) {  delete threadToBeDestroyed;  threadToBeDestroyed = NULL;  }  //... |   这就能看出来了，先执行SWITCH上下文切换到一个新线程上，然后由新线程负责析构上一个结束的线程。  兄弟线程的部分只是为了澄清概念，算是半个题外话，因为直接和线程运行并切换的部分其实是scheduler模块中的东西。   |  | | --- | | void  Thread::Fork(VoidFunctionPtr func, \_int arg)  {  #ifdef HOST\_ALPHA  DEBUG('t', "Forking thread \"%s\" with func = 0x%lx, arg = %ld\n",  name, (long) func, arg);  #else  DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",  name, (int) func, arg);  #endif    StackAllocate(func, arg);  IntStatus oldLevel = interrupt->SetLevel(IntOff);  scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts  // are disabled!  (void) interrupt->SetLevel(oldLevel);  } |   之前提到，Fork在SimpleThread之前，但是Fork出来的线程的执行可并不在SimpleThread函数之前，如上边的代码，Fork本身是一个分配栈空间的函数，随后把Fork出的线程加入到ready list中，而并没有立刻执行这个线程。  现在理一下思路，nachos中有一个全局的指针，currentThread，指示了当前正在运行的线程，而同时还有一个ready list，里面存放了准备运行的线程。而当前currentThread在Initialize时指向的main线程，在ThreadTest函数中Fork出来的线程叫做“forked thread”。如下图：  Thread_space  随后main函数按照它的逻辑一直执行下去，直到最后（或者时间片过期，时钟中断及其实现，上两个实验我都详细叙述过了，这里不再赘述）需要Finisih的时候，为什么最后一定会调用Finish呢？这个之前写过，在StackAllocate时，为其栈空间底部注入了一个ThreadFinish的回调，其中调用了Finsih。   |  | | --- | | // thread.cc :: StackAllocate  machineState[WhenDonePCState] = (\_int) ThreadFinish;  //thread.cc  static void ThreadFinish() { currentThread->Finish(); } |   当main函数执行了这个Finish时，会在其中调用Sleep函数，然后发现ready list中还有一个forked thread等着呢，于是调用Run方法跑forked thread线程。   |  | | --- | | //thread.cc :: Finish  //...  threadToBeDestroyed = currentThread;  Sleep();  //...  //thread.cc :: Sleep  //...  while ((nextThread = scheduler->FindNextToRun()) == NULL)  interrupt->Idle(); // no one to run, wait for an interrupt    scheduler->Run(nextThread); // returns when we've been signalled  //... |   那么上下文切换的部分是怎样实现的呢？  在线程控制对象中，如果该对象控制一个用户线程，那么将开辟一组寄存器存放数据（有别于一般的寄存器）。   |  | | --- | | //thread.h  //defination of Thread  #ifdef USER\_PROGRAM  // A thread running a user program actually has \*two\* sets of CPU registers --  // one for its state while executing user code, one for its state  // while executing kernel code.  int userRegisters[NumTotalRegs]; // user-level CPU register state  public:  void SaveUserState(); // save user-level register state  void RestoreUserState(); // restore user-level register state  AddrSpace \*space; // User code this thread is running.  #endif |   nachos编写了一个Machine类来模拟CPU（包括CPU的寄存器组）。其中包括了一个字段：   |  | | --- | | int registers[NumTotalRegs]; // CPU registers, for executing user programs |   官方对其解释是：用来执行用户程序的CPU寄存器组。  那么在运行新的线程前，我们需要将这组寄存器的内容保存到线程控制对象中保存用户程序数据的寄存器组。  随后检查栈溢出。然后是最关键的部分，切换线程：SWITCH函数   |  | | --- | | ;scheduler.cc  SWITCH(oldThread, nextThread);  ;线程切换时，保存现场的步骤和参数对于每种不同的处理器是不一样的，所以这个函数是由  ;汇编语言编写的  ;以MIPS架构为例  /\* Symbolic register names \*/  #define z $0 /\* zero register \*/  #define a0 $4 /\* argument registers \*/  #define a1 $5  #define s0 $16 /\* callee saved \*/  #define s1 $17  #define s2 $18  #define s3 $19  #define s4 $20  #define s5 $21  #define s6 $22  #define s7 $23  #define sp $29 /\* stack pointer \*/  #define fp $30 /\* frame pointer \*/  #define ra $31 /\* return address \*/  # a0 -- pointer to old Thread  # a1 -- pointer to new Thread  .globl SWITCH  .ent SWITCH,0  SWITCH:  sw sp, SP(a0) # save new stack pointer  sw s0, S0(a0) # save all the callee-save registers  sw s1, S1(a0)  sw s2, S2(a0)  sw s3, S3(a0)  sw s4, S4(a0)  sw s5, S5(a0)  sw s6, S6(a0)  sw s7, S7(a0)  sw fp, FP(a0) # save frame pointer  sw ra, PC(a0) # save retu rn address    lw sp, SP(a1) # load the new stack pointer  lw s0, S0(a1) # load the callee-save registers  lw s1, S1(a1)  lw s2, S2(a1)  lw s3, S3(a1)  lw s4, S4(a1)  lw s5, S5(a1)  lw s6, S6(a1)  lw s7, S7(a1)  lw fp, FP(a1)  lw ra, PC(a1) # load the return address  j ra  .end SWITCH |   关于sw,lw这些汇编指令的用法：  lw:load word  sw:store word  语法:   |  | | --- | | lw/sw des,offset(source)  ; MIPS指令集中的官方写法如下  lw/sw rt, imm(rs) |   意即:将source寄存器中的值+偏移量offset得到对应内存地址，将该内存地址处的一个字节加载/保存到一个指定寄存器中。  用公式表达为: rt = mem[rs+E(imm)]  但观察nachos中的汇编指令(这里以SWITCH函数为例)   |  | | --- | | SWITCH:  sw sp, SP(a0) # save new stack pointer  sw s0, S0(a0) # save all the callee-save registers  sw s1, S1(a0)  sw s2, S2(a0)  sw s3, S3(a0)  sw s4, S4(a0)  sw s5, S5(a0)  sw s6, S6(a0)  sw s7, S7(a0)  sw fp, FP(a0) # save frame pointer  sw ra, PC(a0) # save return address    lw sp, SP(a1) # load the new stack pointer  lw s0, S0(a1) # load the callee-save registers  lw s1, S1(a1)  lw s2, S2(a1)  lw s3, S3(a1)  lw s4, S4(a1)  lw s5, S5(a1)  lw s6, S6(a1)  lw s7, S7(a1)  lw fp, FP(a1)  lw ra, PC(a1) # load the return address  j ra  .end SWITCH |   这里是进行线程切换的函数，而a0,a1分别为旧线程与新线程的指针。而不可能在sw到ra之后立即lw到ra。这个在逻辑上讲不通(要不然不是白存了)。翻阅了SDU CS学院ZHL的指导书，发现在nachos里应该是:   |  | | --- | | lw/sw Rsrc,const(Rindex) |   意思是,取mem[$Rindex+E(const)]的值，存入$Rsrc;  或将$Rsrc的值，存入mem[$Rindex+E(const)]  这里要注意一件事，SWITCH函数保存的现场和SaveUserState保存的现场不是一个东西！  在userRegisters这个类字段上有这样的注释说明：   |  | | --- | | // A thread running a user program actually has \*two\* sets of CPU registers --  // one for its state while executing user code, one for its state  // while executing kernel code.  int userRegisters[NumTotalRegs]; // user-level CPU register state |   实际上，无论什么线程都有machineState这个寄存器组，用来执行kernel code。  但只有用户线程有userRegisters这个寄存器组，用来执行user code。  SWITCH函数本质上做的是对kernel code的现场保存。  这一通sw,lw，意思就是把当前kernel寄存器组中的值保存到对应线程控制对象的machineState数组里，然后把新线程的machineState数组中的数据加载到kernel寄存器组里。  随后是一个跳转指令   |  | | --- | | j ra |   ra是return address的缩写，也就是函数返回地址寄存器。j在MIPS指令集里是无条件跳转的指令，也就是说这里简单地把程序指针跳到了ra寄存器中对应的内存地址上。  那么现在ra里面的地址是谁？注意lw ra, PC(a1)这个指令，意思就是把a1寄存器中的值加上PC，得到一个新的内存地址，然后保存到ra寄存器里，所以ra = mem[$a1 + E(PC)]。  那么mem[$a1+E(PC)]到底是什么呢？a1其实是\*newThread，也就是新线程栈空间的开始部分   |  | | --- | | # a0 -- pointer to old Thread  # a1 -- pointer to new Thread |   而PC这个立即数的含义并非是程序指针的内存地址，而是代表了newThread这个线程的PC指针在kernel code对应的寄存器组的偏移量。  这一点可以从switch.h头文件中看出来   |  | | --- | | /\* Registers that must be saved during a context switch.  \* These are the offsets from the beginning of the Thread object,  \* in bytes, used in switch.s  \*/  #define SP 0  #define S0 4  #define S1 8  #define S2 12  #define S3 16  #define S4 20  #define S5 24  #define S6 28  #define S7 32  #define FP 36  #define PC 40 |   注意官方给的注释:in bytes。这其实就意味着0,4,8,…这些数本质上是字节数。比如S1这个寄存器在对应寄存器组中的字节偏移量是8，而一个\_int(分不同机器，可能是条件编译了int，也可能是long)是4个字节，也就是说其数组元素索引为8/4=2。其余同理。  所以$a1是基址，PC是偏移量，加上之后我们就得到了新线程程序指针的真实内存空间了。  那么也就是说ra里的值是新线程PC指针的内存地址。当无条件跳转到该地址后，接下来PC顺序执行，就应该执行新线程了。具体内容需要根据Fork中传递的函数来决定。  这样就切换到forked thread上了，然后它会按上面步骤一直执行，直到forked thread又调用了Finish，然后调用了Sleep。  随后它会发现while一直成立（ready list没线程了），因此会调用interrupt→Idle()。而Idle函数处理完应当处理的中断后，将停机。   |  | | --- | | //interrupt.cc :: Idle  if (CheckIfDue(TRUE)) { // check for any pending interrupts  while (CheckIfDue(FALSE)) // check for any other pending  ; // interrupts  yieldOnReturn = FALSE; // since there's nothing in the  // ready queue, the yield is automatic  status = SystemMode;  return; // return in case there's now  // a runnable thread  }  // if there are no pending interrupts, and nothing is on the ready  // queue, it is time to stop. If the console or the network is  // operating, there are \*always\* pending interrupts, so this code  // is not reached. Instead, the halt must be invoked by the user program.  DEBUG('i', "Machine idle. No interrupts to do.\n");  printf("No threads ready or runnable, and no pending interrupts.\n");  printf("Assuming the program completed.\n");  Halt(); |   对于其中CheckIfDue的解析，实验一中已经详述了，不加赘述。  这样两个线程都执行完了。  当然现在讲的是FCFS的例子，有些人可能觉得FCFS虽然在概念上讲也叫线程并发，但总归没有那么”并发“，他们想要上下文切换更频繁复杂的例子。那我这里再举一个带时间片/时钟中断的例子好了。  首先我们确保线程执行时是会推进系统时间的，不然如果系统时间永远处于0处，那也没有时间片过期或者时钟中断这个说法了，因此这个例子的前提必须是线程执行时我们要推进系统时钟，无论你用什么方式，只要是合法的推进时钟的手段就可以。  有了之前的例子，这里我们的叙述可以稍微简单一点，不妨直接从时间片过期前夕开始讲起吧。时间片过期前夕，也即还差一个操作的时间就过期的时间点。此时系统时钟推进，势必要调用OneTick（不论你是什么方法推进的系统时钟）   |  | | --- | | // interrupt.cc :: OneTick  // advance simulated time  if (status == SystemMode) {  stats->totalTicks += SystemTick;  stats->systemTicks += SystemTick;  } |   随后检查是否有时间片过期，如果有，调用回调函数。这个时钟中断的回调之前解析过，就是声明一个yieldOnReturn的切换开关，如果为开，那么OneTick中将Yield此线程。   |  | | --- | | // interrupt.cc :: OneTick  while (CheckIfDue(FALSE))  // interrupt.cc :: CheckIfDue  inHandler = TRUE;  status = SystemMode; // whatever we were doing,  // we are now going to be  // running in the kernel  (\*(toOccur->handler))(toOccur->arg); // call the interrupt handler  status = old; // restore the machine status  inHandler = FALSE;  delete toOccur;  return TRUE;  // interrupt.cc :: OneTick  if (yieldOnReturn) { // if the timer device handler asked  // for a context switch, ok to do it now  yieldOnReturn = FALSE;  status = SystemMode; // yield is a kernel routine  currentThread->Yield();  status = old;  } |   有些人可能会问：”不是说线程不能主动Yield放弃CPU吗？这里怎么Yield了？”很简单，这里线程是被动的Yield放弃CPU，如果没有这个时钟中断，那么它将继续执行下去，但时钟中断发生了，它不得不被动地放弃了CPU。  这就是RR/Priority等一系列带时间片的调度算法下的线程并发执行。 按照The Little Book of Semaphores-3.6.4节实现N线程屏障，并探查一个BUGN线程屏障实现  * 编写屏障类Barrier * 编写测试程序   Barrier类数据结构:   |  | | --- | | private:  int n;//the number of threads  int count;  char\* name;  Semaphore\* turnstile;  Lock\* mutex; //mutex lock to access the count register |   成员函数：（主要的几个，构造和析构函数这里不放了）这里就是完全按照3.6.4节实现的，之前分析过很多了，不再赘述。   |  | | --- | | void Barrier::Arrive(int index) {  // revert the counter  if(count>=n){  count=0;  }  mutex->Acquire();  count = count +1;  printf("Thread [%d] has arrived\n",index);  mutex->Release();  if(count==n){  turnstile->V();  printf("Thread [%d] is the last\n", index);  }  turnstile->P();  turnstile->V();  //simulate critical point  printf("Thread [%d] cross through Critical Section\n",index);  } |   测试程序：   |  | | --- | | // global variables to realize N thread barrier test  Barrier\* barrier;  Thread\* threads[N\_THREADS];  char threads\_name[N\_THREADS][MAX\_NAME];  void BarThread(\_int which)  {  printf("Thread [%d] rendezvous\n", which);  barrier->Arrive(which);  }  void ThreadsBarrier()  {  barrier = new Barrier("N\_thread",N\_THREADS);  // Create and fork N\_THREADS threads  for(int i = 0; i < N\_THREADS; i++) {  sprintf(threads\_name[i], "Thread %d", i);  threads[i] = new Thread(threads\_name[i]);  threads[i]->Fork(BarThread, i);  };  } |   但这样是会出问题的。 last thread 采用随机数种子12，附加选项rs，该选项将生成随机长度的时钟中断时间片，也就是说每个线程在其对应时间片中可以执行操作的次数是随机的。这一点通过注入一个带有随机种子的Timer实现。   |  | | --- | | //main.cc :: main  else if (!strcmp(\*argv, "-rs")) {  ASSERT(argc > 1);  RandomInit(atoi(\*(argv + 1))); // initialize pseudo-random  // number generator  randomYield = TRUE;  argCount = 2;  }  //system.cc :: Initialize  if (randomYield) // start the timer (if needed)  timer = new Timer(TimerInterruptHandler, 0, randomYield);  //timer.cc :: Timer  Timer::Timer(VoidFunctionPtr timerHandler, \_int callArg, bool doRandom)  {  randomize = doRandom;  handler = timerHandler;  arg = callArg;  // schedule the first interrupt from the timer device  interrupt->Schedule(TimerHandler, (\_int) this, TimeOfNextInterrupt(),  TimerInt);  }  //timer.cc :: TimeOfNextInterrupt  int  Timer::TimeOfNextInterrupt()  {  if (randomize)  return 1 + (Random() % (TimerTicks \* 2));  else  return TimerTicks;  } |   简单来说就是在nachos启动时，软件模拟的时钟模块将根据是否添加rs选项来注入。如果添加rs选项，那么它将注入一个每次都生成随机长度的时间片的时钟模块。schedule后面的函数以前分析过了，这里不再赘述。   |  | | --- | | ./nachos -rs 12 //-rs causes Yield to occur at random (but repeatable) spots |   可以看到现在有三个线程:t1、t5和t9都认为他们是最后一个到达的。  0   |  | | --- | | 1 void Barrier::Arrive(int index) {  2 // revert the counter  3 if(count>=n){  4 count=0;  5 }  6 mutex->Acquire();  7 count = count +1;  8 printf("Thread [%d] has arrived\n",index);  9 mutex->Release();  10 if(count==n){  11 turnstile->V();  12 printf("Thread [%d] is the last\n", index);  13 }  14 turnstile->P();  15 turnstile->V();  16 //simulate critical point  17 printf("Thread [%d] cross through Critical Section\n",index);  18} |   这个错误的发生是合理的，假想一下这个情况：   1. t1执行完第9行，发生线程切换，不管切换到谁吧，反正那个线程继续往下执行。 2. 在切换回t1之前，假设t5得到了时间片，开始执行。假设它也在第9行后被切换了。 3. 随后，t9得到时间片，它先执行到第10行，发现此时确实count==n成立，然后令turnstile执行了V操作。此时turnstile对应的semaphore为1。（为什么是1之前分析过了） 4. 随后t9继续执行（注意时间片是随机的，t9随机到的时间片比t1和t5长！），打印了一下自己是最后一个到的，然后执行P操作，但此时Semaphore的value是1，不会阻塞，所以它通过了14行，然后来到15行，唤醒了另外一个线程。 5. 假设这个线程就是t1吧，t1之前被切换时执行到第10行了，它继续执行，但别忘了**count变量对于不同线程来说是共享的**，也就是**t1此时看到的count变量的值为n**！那么t1就会执行第10、11行了。这个错误原因对于t5来说是一样的。  修改代码消除Q3中出现的现象 这个现象的本质是：这个条件触发是”一次性“的。简单来说就是某个时间点count==n后，之后count==n恒成立，那么我们必须确保**更新count时就顺带检查count==n是否成立**，不然此时如果一旦线程切换之后再来检查，很可能就恒成立了。  那么怎么做到更新count时就顺带检查count==n是否成立，比较容易想到的其实就是在互斥锁里顺带检查。这样即便切换了线程，其他线程由于得不到互斥锁，也不会更新count的值。   |  | | --- | | void Barrier::Arrive(int index) {  // revert the counter  if(count>=n){  count=0;  }  mutex->Acquire();  count = count +1;  printf("Thread [%d] has arrived\n",index);  if(count==n){  turnstile->V();  printf("Thread [%d] is the last\n", index);  }  mutex->Release();    turnstile->P();  turnstile->V();  //simulate critical point  printf("Thread [%d] cross through Critical Section\n",index);  } |   核心代码改成这样。再测一下，就没问题了。  1 尝试不同随机种子，发现各线程执行顺序就是线程被创建顺序。简述原因。试验一下在各线程开始执行前加上延迟是否能解决这个问题，如果能，简述原因打印顺序问题 其实还是有一些随机数种子表现没有那么糟糕（即执行顺序就是创建顺序），但大部分都符合题目描述。这里我贴一张表现比较好的种子。  2  这里0→2→7→9→1→8。也不能说完全按创造顺序来的，不过基本上也差不多了。  确实很奇怪，为什么加上了随机数种子去生成时间片，执行顺序基本上依然是创建顺序？首先我们从创建线程的部分开始看起。   |  | | --- | | void ThreadsBarrier()  {  barrier = new Barrier("N\_thread",N\_THREADS);  // Create and fork N\_THREADS threads  for(int i = 0; i < N\_THREADS; i++) {  sprintf(threads\_name[i], "Thread %d", i);  threads[i] = new Thread(threads\_name[i]);  threads[i]->Fork(BarThread, i);  };  } |   这是创建线程部分的代码。但与常识相悖的是，看似这个创建线程的部分执行了n次，但系统时钟根本没有推进。还记得Lab2时就探讨过这个问题吗？系统时钟只能由OneTick推进，但OneTick只有两个时机会被执行，这个在Nachos源码中就提到了：   |  | | --- | | // Two things can cause OneTick to be called:  // interrupts are re-enabled  // a user instruction is executed |   只有重新打开中断或用户指令才能引发OneTick。而这部分代码完全没有重开中断（当然更不可能有用户指令），因此直到n个线程创建完，当前系统时间还是0。  现在来看Fork函数，它干了一件这样的事。   |  | | --- | | //Fork  scheduler->ReadyToRun(this);  //ReadyToRun  readyList->Append((void\*)thread); |   相当于新创建的线程会被添加到readyList队尾，则readyList中线程序号完全按照我们的创建顺序来，如下图：   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 就绪线程序号 | t0 | t1 | t2 | ... | t(n-1) |   但别忘了，Fork里面开关中断了！   |  | | --- | | //Timer  int  Timer::TimeOfNextInterrupt()  {  if (randomize)  return 1 + (Random() % (TimerTicks \* 2));  else  return TimerTicks;  } |   也就是说生成的大小介于0-2\*TimerTicks之间。所以一个线程的时间片很可能非常小，甚至是0！  现在我们假想一个情况：   1. t0创建出来并执行，打印了一个rendezvous，由于现在n个线程就到了它一个，它将在turnstile.P()时阻塞。 2. 然后t1创建出来了，假设它分配到的时间片是1。而OneTick一次推进10。在Fork里重开中断了，因此调用OneTick，OneTick里将系统时间加至10。然后CheckIfDue检查，发现时间片截止到1，那么就声明自己时间片过期要关中断。返回到OneTick，发现当前线程已经声明为时间片过期了，因此要切换线程，比如切换出来是t2。 3. t2执行，打印了rendezvous。   这样打印rendezvous的顺序就变成t0→t2了。而不是t0→t1！ 加入延迟 指导书中给我们的提示是通过软件空循环来消耗时间，造成延迟，以此来打乱rendezvous的输出顺序。注意，这里的消耗时间消耗的不是物理上的时间！而是软件中模拟的系统时间！所以我们在这个空循环里是要推时钟的！不然毫无意义。  我根据参考资料，加入了以下代码：   |  | | --- | | #define N\_TICKS 1000 // the number of ticks to advance simulated time  Semaphore \*advance;  void MakeTicks(int n) // advance n ticks of simulated time  {  //use the IntOff and IntOn to advance the clock  advance = new Semaphore("advance",1);  for(int i=n;i>0;i--){  advance->P();  advance->V();  }  }  void BarThread(\_int which)  {  MakeTicks(N\_TICKS);  printf("Thread [%d] rendezvous\n", which);  barrier->Arrive(which);  } |  1. 定义了一个空循环要消耗系统时间的次数，1000次。 2. 定义了一个信号量，专门用来空循环消耗系统时间。看这个信号量的名字就知道了，advance，顾名思义，将系统时钟提前的信号量。为什么能这样呢？因为信号量PV操作时是可以开关中断的。下面放个P操作的，V的我就不放了，占地方。  |  | | --- | | void  Semaphore::P()  {  IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts    while (value == 0) { // semaphore not available  queue->Append((void \*)currentThread); // so go to sleep  currentThread->Sleep();  }  value--; // semaphore available,  // consume its value    (void) interrupt->SetLevel(oldLevel); // re-enable interrupts  } |  1. 然后我们在推时钟的函数MakeTicks中重复无意义地不断PV，这样就是在推进系统时钟，除此之外没有任何意义。 2. 这样相当于加上了延迟，之后我们再执行原来每个线程要执行地东西就行了，比如打印rendezvous，还有N线程屏障的核心代码。   测试一下：  3  这下没问题了。 为什么加入延迟就解决上述问题了？ 现在假想这样一种情况：   1. 本来t1是第一个打印rendzvous的线程。但是我们现在给所有线程都加了个延迟=Δt。假设t1分配到的时间片 ts(1) < Δt，那么在它的时间片过期时就要切换线程，假设是t2。 2. 虽然t2和t1的延迟都是Δt，但是它们的时间片不一样。很可能会出现t2的时间片比t1的长的情况，这样同样的Δt可以更少的t2的时间片，这样t2就会更早执行。   我举个例子，Δt = 120，ts(2) = 60，ts(1) = 20，OneTick = 10，来看看会发生什么。   1. t1空循环一次，系统时钟往前推两次（P,V各一次），检查到过期，切换到t2。延迟还剩100 2. t2空循环一次，没到期，时间片剩 60 - 20 = 40。再空循环一次，没到期，时间片剩20。再空循环一次，到期了，时间片剩0。延迟还剩120 - 60 = 60，切换到t1。 3. t1空循环一次，延迟还剩80，切换到t2 4. t2空循环三次，延迟结束，切换到t1 5. t1空循环一次，延迟剩60，切换到t2 6. t2延迟已经结束，打印rendzvous   看到了吗？**t2由于时间片更长，所以提前结束了延迟，导致提前打印rendzvous**。  就像上面测试结果吧，打印顺序是：0→7→6→1→8→4→2→9→5→3  那么时间片长度从大到小排列就是：3→5→9→2→4→8→1→6→7→0  时间片长短完全和打印顺序倒过来。 | | | |
| 结论分析与体会：  本次实验我接触到了并行计算以及相关研究中的一个很重要的概念——“N线程屏障”，简单来说就是有N个线程，只有当N个线程均到达后他们才能逐个通过临界区，否则阻塞。我自学了老师提供的“The Little book of Semaphore”这本书，从中了解到了不止实验目的中的turnstile，还有诸如two-phase turnstile等解决方案。并且我阅读了Nachos作者提供的源码，了解到了互斥锁以及条件变量这两种线程同步机制。下面我将摘录一份自己的学习笔记，以展现我的收获和体会。 同步机制 线程同步机制共有三种：   1. Semaphore 2. Lock 3. Conditional Variable   在Nachos中，这三种机制的实现是一层套一层的（2用1，3用2）。其实比较难的是Conditional Variable。但为了笔记的连续性，三个都写一点。但因为上文我已经写过信号量的实现了，下面我就只写互斥锁和条件变量的实现。 1.2. Lock数据结构  |  | | --- | | char\* name; // for debugging  Thread \*owner; // remember who acquired the lock  Semaphore \*lock; |   name:debug用  owner:持有该锁的线程  lock:信号量，初始值为1  其实Lock本质上就是个二值信号量，用来做Mutex（互斥）的。 成员函数  |  | | --- | | void Lock::Acquire()  {  IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts  lock->P(); // procure the semaphore  owner = currentThread; // record the new owner of the lock  (void) interrupt->SetLevel(oldLevel); // re-enable interrupts  } |   Acquire:获取锁的行为。可以把锁当成唯一一个临界资源，PV都是对这一个临界资源的操作。   |  | | --- | | void Lock::Release()  {  IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts  // Ensure: a) lock is BUSY b) this thread is the same one that acquired it.  ASSERT(currentThread == owner);  owner = NULL; // clear the owner  lock->V(); // vanquish the semaphore  (void) interrupt->SetLevel(oldLevel);  } |   Release:释放锁。但是需要注意的是，释放锁的时候你得真的有这把锁，这就是ASSERT在断言的事情。因此有了：   |  | | --- | | bool Lock::isHeldByCurrentThread()  {  bool result;  IntStatus oldLevel = interrupt->SetLevel(IntOff);  result = currentThread == owner;  (void) interrupt->SetLevel(oldLevel);  return(result);  } |   isHeldByCurrenThread:判断当前线程是否持有锁。  示例就不说了，上面Semaphore那个就可以用。 1.3. Conditional Variable 线程在访问临界资源时，需要满足/遵守一些条件，（比如一个空列表是不能Remove元素的，此时条件就为列表不能为空）如果条件不满足，那么线程将阻塞在这个条件下，直到条件满足可以继续执行为止。将线程阻塞在某一条件下的列表叫做条件变量(Conditional Variable)。 数据结构  |  | | --- | | char\* name;  List\* queue; // threads waiting on the condition  Lock\* lock; // debugging aid: used to check correctness of  // arguments to Wait, Signal and Broadcast |   name:debug用。  queue:阻塞线程的队列，也就是线程等待某一条件的队列。  lock:这个是用来debug的。作者的重点在于，条件变量一定要和一把lock一起使用，这个重点说。 为什么要有lock? 这里的lock指的不是数据结构里的那个lock成员变量，而是和条件变量一起使用的lock。  本质上是因为：条件变量的成员函数里有个列表，列表本身作为临界资源，访问是需要同步的，而这个临界资源是唯一的，所以需要互斥锁。  你可能会说：”啊？我用一个同步列表作为条件变量中的成员变量列表不就可以了？“这简直是在颠倒黑白，因为同步列表本身就是用条件变量和lock实现的，这么说就完全陷入循环论证了。  先来看看不用lock会发生什么。  假设现在有两个线程t1,t2。他们在访问临界资源前都要考虑一个条件C。   1. 假设t1先执行，此时t1查看条件C，发现成立。 2. 但就在t1更新临界区状态前（更新完后可能条件C就不成立了！），发生了上下文切换。切换到了线程t2。 3. t2也查看条件C，注意在第2步中t1还没更新临界区状态！所以条件C此时止于t2一定是成立的！ 4. t2一看条件C成立，它就进入临界区访问了。假设在访问过程中发生了上下文切换，切换回了t1。 5. 别忘了t1在第2步就查询完了条件C了，它只会顺序往下执行。所以即便t2在第4步中更新了临界区状态导致条件C不成立了，t1也不会知道这个更新。 6. t1一看条件C成立，它就也进入了临界区。 7. 临界区同时存在t1,t2两个线程。   但如果代码中有互斥锁，切换上下文到t2后，t2会发现此时有人(t1)在对条件C进行判断，他就会等着先不判断条件C。直到t1进入临界区并将条件C更改为最新的状态，他才会去查看。  这个后面看条件变量的应用时再举例子。 成员函数  |  | | --- | | void Condition::Wait(Lock\* conditionLock)  {  IntStatus oldLevel = interrupt->SetLevel(IntOff);  ASSERT(conditionLock->isHeldByCurrentThread()); // check pre-condition  if(queue->IsEmpty()) {  lock = conditionLock; // helps to enforce pre-condition  }  ASSERT(lock == conditionLock); // another pre-condition  queue->Append(currentThread); // add this thread to the waiting list  conditionLock->Release(); // release the lock  currentThread->Sleep(); // goto sleep  conditionLock->Acquire(); // awaken: re-acquire the lock  (void) interrupt->SetLevel(oldLevel);  } |   wait:经过上面那一通为什么要把lock和条件变量一起用的分析（包括举的counter example)，应该可以认识到lock的重要性了。所以我们在把一个线程挂在条件等待队列中时：   1. 检查这把锁是否真的是当前线程拥有的，如果它声称自己拥有这把锁但实际上并没有，那也没什么继续的意义了。 2. 查看当前线程传进来的锁是否真的是和条件变量配合使用的锁，如果它拥有的甚至不是和这个条件变量配合使用的那把锁，它也没资格查看条件是否成立。   随后在主要逻辑中：   1. 由于条件不成立，所以要把它挂到等待条件队列里 2. 但是当前线程获取了锁，不能让其他线程失去查看条件是否成立的资格，因此要先释放这把锁 3. 既然条件不成立，那就让当前线程睡眠，切换到其他线程上，看执行完了条件能不能过。 4. 当前线程睡醒起来，它并不清楚当前条件过不过（这也是为什么一般条件变量wait操作都写在循环里），为了得知条件是否成立，它得重新检查，但是别忘了，检查条件是否成立是需要锁的！所以它必须重新获取一下锁，然后尝试去获取条件的最新情况。  |  | | --- | | void Condition::Signal(Lock\* conditionLock)  {  Thread \*nextThread;  IntStatus oldLevel = interrupt->SetLevel(IntOff);  ASSERT(conditionLock->isHeldByCurrentThread());  if(!queue->IsEmpty()) {  ASSERT(lock == conditionLock);  nextThread = (Thread \*)queue->Remove();  scheduler->ReadyToRun(nextThread); // wake up the thread  }  (void) interrupt->SetLevel(oldLevel);  } |   signal:唤醒条件等待队列中的线程有两个方式，这里先讲signal。它每次只唤醒一个线程。一样的，wait的两次检查它也没理由不做。  在主要逻辑中，由于signal并不是在检查条件之后做的，更重要的是，signal时我们是确保条件一定成立的！因此我们可以立即找一个准备好的线程，声明它准备执行（至于为什么不立即执行，首先我们是Mesa semantic，其次在Semaphore中讲过了）。并且由于我们现在已经处于获取到锁的状态，因此不需要重新acquire了。   |  | | --- | | void Condition::Broadcast(Lock\* conditionLock)  {  Thread \*nextThread;  IntStatus oldLevel = interrupt->SetLevel(IntOff);  ASSERT(conditionLock->isHeldByCurrentThread());  if(!queue->IsEmpty()) {  ASSERT(lock == conditionLock);  while( (nextThread = (Thread \*)queue->Remove()) ) {  scheduler->ReadyToRun(nextThread); // wake up the thread  }  }  (void) interrupt->SetLevel(oldLevel);  } |   这个和signal逻辑一样，只是唤醒了所有等待在该条件下的线程。只有你确保了这个条件在之后一定成立再这么做！ 2. 同步机制的应用2.1. 同步列表(SyncList) 上面条件变量的部分提到过，只要是个列表，就有线程同步问题，并且列表作为唯一的临界资源，必定要有互斥锁和条件变量配合使用。 数据结构  |  | | --- | | List \*list; // the unsynchronized list  Lock \*lock; // enforce mutual exclusive access to the list  Condition \*listEmpty; // wait in Remove if the list is empty |   listEmpty是在同步列表为空时的条件等待队列。比如我们想要在同步列表中Remove一个函数时，需要保证它不为空，如果为空就得把当前线程挂在条件等待队列下面。  lock不多说了，和条件变量配合使用的互斥锁。  list是同步列表的实体，其实我们只要给线程不安全的列表加上同步机制就变成同步列表了。 成员函数 主要是两个   |  | | --- | | void  SynchList::Append(void \*item)  {  lock->Acquire(); // enforce mutual exclusive access to the list  list->Append(item);  listEmpty->Signal(lock); // wake up a waiter, if any  lock->Release();  } |   首先，访问列表要互斥锁。  其次，通知可能存在的等待在条件等待列表中的线程它可以继续操作了，对于同步列表来说，条件等待列表对应于列表是否为空。  最后，让其他线程也有能力访问列表或探查条件是否成立，释放锁。   |  | | --- | | void \*  SynchList::Remove()  {  void \*item;  lock->Acquire(); // enforce mutual exclusion  while (list->IsEmpty())  listEmpty->Wait(lock); // wait until list isn't empty  item = list->Remove();  ASSERT(item != NULL);  lock->Release();  return item;  } |   首先，访问列表要互斥锁。  其次，探查列表是否为空，为空则挂在条件等待列表下面，然后睡眠，醒过来再探查一遍，直到条件成立为止。  最后，让其他线程也有能力访问列表或探查条件是否成立，释放锁。 示例 假设两个线程t1,t2想要对同一个同步列表进行一些操作。t1想移除头部的元素，而t2想在列表末尾插入一个元素。同步列表初始状态为空。（没有元素）   1. 假设t1先执行。t1获取锁，然后探查条件：列表不为空。它发现条件不成立（列表初始状态为空），随后wait操作。    1. 为了让其他线程能够访问列表或探查条件，它先释放了锁。    2. 然后它把自己拉到条件等待队列中。    3. 随后它睡眠了。注意睡眠时会切换线程。 2. 线程切换到t2。现在锁已经被t1释放了，t2获取锁成功。    1. t2的任务是向列表尾部插入一个元素，这根本不需要探查条件，它插入了一个元素。    2. t2通知条件等待列表中的t1现在条件成立了，你可以动了。它把t1拉到readyList中去。    3. 为了让t1能访问列表或探查条件，t2释放锁。    4. t2终止，finish调用sleep，sleep切换线程。 3. 现在线程切换回t1了，此时t2已经释放了锁，t1为了继续探查条件，它获取了锁。 4. t1重新判断条件，发现成立。它进入列表，删了头部的元素。 5. 为了让可能存在的其他线程可以访问列表或探查条件，它释放了锁。 6. t1结束   转换成代码为：   |  | | --- | | //t1  void \*  SynchList::Remove()  {  void \*item;  lock->Acquire(); // enforce mutual exclusion  while (list->IsEmpty())  listEmpty->Wait(lock); // wait until list isn't empty  //t1  void Condition::Wait(Lock\* conditionLock)  {  IntStatus oldLevel = interrupt->SetLevel(IntOff);  ASSERT(conditionLock->isHeldByCurrentThread()); // check pre-condition  if(queue->IsEmpty()) {  lock = conditionLock; // helps to enforce pre-condition  }  ASSERT(lock == conditionLock); // another pre-condition  queue->Append(currentThread); // add this thread to the waiting list  conditionLock->Release(); // release the lock  currentThread->Sleep(); // goto sleep  //t2  void  SynchList::Append(void \*item)  {  lock->Acquire(); // enforce mutual exclusive access to the list  list->Append(item);  listEmpty->Signal(lock); // wake up a waiter, if any  //t2  void Condition::Signal(Lock\* conditionLock)  {  Thread \*nextThread;  IntStatus oldLevel = interrupt->SetLevel(IntOff);  ASSERT(conditionLock->isHeldByCurrentThread());  if(!queue->IsEmpty()) {  ASSERT(lock == conditionLock);  nextThread = (Thread \*)queue->Remove();  scheduler->ReadyToRun(nextThread); // wake up the thread  }  (void) interrupt->SetLevel(oldLevel);  }  //t2  lock->Release();  }  //t1  conditionLock->Acquire();  (void) interrupt->SetLevel(oldLevel);  }  //t1  item = list->Remove();  ASSERT(item != NULL);  lock->Release();  return item;  } |  2.2. 汽车过桥问题问题背景 有一座桥，单行道，最大容量3，实现汽车过桥管理系统。 问题分析 我们把桥看作一个容器（类似于上一个例子中的同步列表是元素的容器一样），这个容器有一个方向标识，容器内所有元素的方向要和这个标识相同。此外容器容量最大为3。  仿照上一个例子，条件变量对应桥的容量是否满（注意我们不会主动去移除桥上的车，所以不用查桥是否为空这个条件变量），然后搭一个可以配合的锁。 数据结构  |  | | --- | | private:  int numCars;  int currentDirec;  Condition \*bridgeFull;  Lock \*lock; |   numCars标识当前车辆数  currentDirec标识当前容器方向  bridgeFull代表容量满且方向对这个条件对应的条件等待列表  lock为条件变量配对的互斥锁 成员函数 主要有三个。   |  | | --- | | void  Bridge::Arrive(int direc)  {  DEBUG('t', "Arriving at bridge. Direction [%d]", direc);  lock->Acquire();  // We can proceed if:  // a) the bridge is empty  // or b) the bridge has less than 3 cars and traffic is flowing  // in the direction we want  // Otherwise, wait  while((numCars > 0) && ((numCars >= 3) || (direc != currentDirec))) {  bridgeFull->Wait(lock);  }  numCars++; // reserve a spot on the bridge  currentDirec = direc; // make sure the direction matches  lock->Release();  DEBUG('t', "Direction [%d], ready to cross bridge now", direc);  } |  1. 访问容器或探查条件：获取锁 2. 桥已满或方向不对，挂在桥满条件变量下面    1. 其实我思考过要不要额外开一个方向对的条件变量，后来我想明白了，我们的条件就是 （容量没满且方向对）。这是一整个布尔表达式，不用拆。 3. 直到容量不满且方向对，上桥。 4. 重置当前方向为汽车行驶方向。 5. 让其他车访问桥或者探查条件，释放锁。  |  | | --- | | void  Bridge::Cross(int direc)  {  DEBUG('t', "Direction [%d], crossing bridge", direc);  } |   标识一下真上桥了，不然测试程序都看不出这个过程。 测试程序  |  | | --- | | void  SynchThread(\_int which)  {  int num;  int direc;    for (num = 0; num < 5; num++) {  direc = num % 2; // set direction (alternates)  // The following printf's need fixing for the different types of \_int of which  printf("Direction [%d], Car [%d], Arriving...\n", direc, which);  bridge->Arrive(direc);  currentThread->Yield();  printf("Direction [%d], Car [%d], Crossing...\n", direc, which);  bridge->Cross(direc);  currentThread->Yield();  printf("Direction [%d], Car [%d], Exiting...\n", direc, which);  bridge->Exit(direc);  currentThread->Yield();  }  }  void  SynchTest()  {  const int maxCars = 7; // How much traffic?  int i = 0;  Thread \*ts[maxCars];  for(i=0; i < maxCars; i++) {  ts[i] = new Thread("forked thread");  ts[i]->Fork(SynchThread, i);  }  } |   每个线程对应一辆车。每辆车5轮过桥，每轮选定一个方向过一次桥，分三步：   1. 上桥 2. 过桥 3. 下桥   每一步结束切换到另一辆车上去，执行这个任务。 2.N thread barrier 假设有正整数n=N。N线程屏障做了这么一件事情：   1. 当目前囤积的线程数量小于n时，所有已经到来的线程将阻塞。 2. 当目前到来的线程数量等于n时，所有已经到来的线程将通过。   简单来说就是线程是一批一批通过的。  一些基本的定义如下：   |  | | --- | | 1 n = the number of threads  2 count = 0  3 mutex = Semaphore (1)  4 barrier = Semaphore (0) |   书中给出了一种实现方式：（伪代码）   |  | | --- | | 1 rendezvous  2  3 mutex . wait ()  4 count = count + 1  5 mutex . signal ()  6  7 if count == n: barrier . signal ()  8  9 barrier . wait ()  10  11 critical point |   自己模拟一下就知道肯定和要求实现的功能不一致了。   1. 每个线程到来时通过互斥锁访达count计数，这个是没问题的 2. 前n-1个线程走不了if，全部阻塞在了第9行上 3. 第n个线程通过了if，然后唤醒了前n-1个线程中的某一个线程，然后自己卡在了第9行上 4. 然后就没有了（被唤醒的那个走了临界区，但其他n-1个全卡在了第9行）   这很明显是不对，因为我们是想第n个线程来了之后所有的线程都可以通过。  因此又更新了一版：   |  | | --- | | 1 rendezvous  2  3 mutex . wait ()  4 count = count + 1  5 mutex . signal ()  6  7 if count == n: barrier . signal ()  8  9 barrier . wait ()  10 barrier . signal ()  11  12 critical point |   其实就只加了一行（第10行）。但解决了上述的死锁问题：   1. 第n个线程到来，唤醒前n-1中的某一个，然后自己卡在第9行 2. 被唤醒的线程走到第10行，然后唤醒其他n-1中的某一个（注意第n个也被卡在了第9行，阻塞的仍有n-1个） 3. 第二个被唤醒的走到第10行，唤醒其他n-2中的某一个 4. …… 5. 这个过程持续到所有的线程都被唤醒，然后通过临界区   这种解决方案有一个专门的术语，叫做：turnstile 。中文中大概是十字转门，旋转栅门的意思。  可以想象一下酒店大堂门口的那个转门：   1. 前n-1个人进去之后都是推一下，这样前面的人的位置就会离入口/出口近一点 2. 第n个人进去推一下，第一个人就进去了，同时第一个人进去前也推一下，把第二个人转到入口/出口的位置，然后之后每个人都这么进去。   这种解决方案不止一个问题：   1. 在互斥锁mutex之外访问count是ok的吗？ 2. 1带来的问题，barrier会被signal多次吗？ 3. n线程后，turnstile会变成什么样子？   如果我们把if count==n这条语句放在了互斥锁外面，可能会出现以下情况：   1. 第n-1个线程执行到if后，因为某种原因被中断切换了 2. 第n个线程来到if，发现count==n，所以他signal了 3. 第n-1个线程切换回来，到if，发现count==n了（别忘了这个时候第n个线程早就来了），所以他也signal了。   这下直接变成两个两个通过临界区了（第n个signal一个，第二个signal一个，这两个后续也会再signal两个，后续同理）如果临界资源不允许多个线程访问，这样的逻辑无疑是错误的。  为了解决上述第一个问题，又提出一个解决方案：   |  | | --- | | 1 rendezvous  2  3 mutex . wait ()  4 count = count + 1  5 if count == n: barrier . signal ()  6  7 barrier . wait ()  8 barrier . signal ()  9 mutex . signal ()  10  11 critical point |   很明显这个也是错的。  如果把所有操作都放在互斥锁中，当第一个线程执行到第7行后肯定被卡住了，他需要等后面n-1个都到了才能走。  而此时互斥锁还没被释放，也就是说剩下n-1个线程压根就进不来，count永远等于1，到不了n了。这n个线程将永远卡在这里。  这个例子是个反例，他教会我们：在持有一把锁时千万不要把线程阻塞到信号量上，不然这把锁永远释放不了。  然后他提出了一个新的需求：往往N线程屏障用于循环中，n个线程在每个循环里都要过一次屏障。  现在考虑之前提到过的第三个问题：n线程后，turnstile（或者说信号量）是怎样的？   1. 一开始turnstile是0，在第n个线程到达之前，每来一个线程都会被wait()卡住，所以第n个线程来之前，turnstile的value是-(n-1) 2. 然后第n个线程来了，从第n个线程起，每轮唤醒一个线程，到最后第一个线程也要signal一次，最终turnstile的value是-(n-1)-1+1+n = 1。其中，-(n-1)代表第n个线程到来时的信号量的值，-1代表第n个线程的wait()，+1代表if中的signal，+n代表每个线程都要signal一次   这样的逻辑是错误的。为什么？  如果我们这段代码处于一个循环中，第一次循环没问题，n个线程来齐了才给通过的。但第一次循环结束后turnstile的值可是会变成1的，而不是0。那第二次循环的时候第一个来的线程可就直接过了wait()了，不会被阻塞。  为了让第二次循环和以后的循环都不会出现这样的问题，他又更新了一个解决方案：   |  | | --- | | 1 rendezvous  2  3 mutex . wait ()  4 count += 1  5 mutex . signal ()  6  7 if count == n: turnstile . signal ()  8  9 turnstile . wait ()  10 turnstile . signal ()  11  12 critical point  13  14 mutex . wait ()  15 count -= 1  16 mutex . signal ()  17  18 if count == 0: turnstile . wait () |   这下多了一次wait，下个循环turnstile的初值就变成0了，好像没问题了。  但不在互斥锁里访问count，势必出现线程同步问题：   1. 第n-1个中断切换卡在7 2. 第n个来了 3. 第n-1个切换回来后以为自己就是第n个，和第n个通过了if判断，各执行一次signal，就会出现多个线程访问临界区   那既然不能在互斥锁外访问count，就改为在互斥锁里面访问：   |  | | --- | | 1 rendezvous  2  3 mutex . wait ()  4 count += 1  5 if count == n: turnstile . signal ()  6 mutex . signal ()  7  8 turnstile . wait ()  9 turnstile . signal ()  10  11 critical point  12  13 mutex . wait ()  14 count -= 1  15 if count == 0: turnstile . wait ()  16 mutex . signal () |   考虑这么一种情况：   1. 前n-1个线程来到了第8行，阻塞在了这里。 2. 第n个线程来了。它signal了一下，让前n-1个线程中的某一个通过了tunstile，假设这个线程是t3吧。此时turnstile上阻塞了n-1个线程，turnstile的值为-n-1。 3. t3线程一直执行，直到16行，此时由于第九行的signal，假设唤醒了t2，turnstile的值为-n-2，count的值为n-1。 4. 别忘了我们的代码会跑在一个循环里，假设此时这个线程没有被切换走，而是继续执行，那么它将进入下一次循环。 5. 它进入互斥锁，给count+1，count变为n。于是它唤醒了另外n-1个线程中的一个，假设唤醒了tn，并阻塞在turnstile上。 6. 这个时候假设发生了线程切换，还记得第三步的时候线程t3唤醒的那个线程吗（t2）？假设现在就切换到那个线程上了，t2线程原本阻塞在第八行，唤醒后执行第九行，它又signal了一下。 7. 现在最关键的来了，可能t2线程在第6步又唤醒了t3！（别忘了t3此时在turnstile的waiting list里面） 8. 假设此时t2切换为了t3，t3发现自己可以继续执行了，然后它通过了临界区，以及第二把互斥锁，进入下一轮循环。 9. 因此t3线程在第4步和第8步都进入了下一轮循环。但别忘了第1步的n-1个线程现在还在一开始那个循环。这样t线程直接就领先了一个循环。   这就是The Little Book of Semaphores中说的：  This code allows a precocious thread to pass through the second mutex, then loop around and pass through the first mutex and the turnstile, effectively getting ahead of the other threads by a lap.  意即：这段代码允许一个早熟的线程（就是我们上述描述的t线程）通过第二个互斥锁，进入下一次循环，然后通过第一把互斥锁和turnstile，这样就完全领先了其他线程一个循环。如下图所示：  4  为了解决这个“Precocious thread”的问题，就又给了一版解决方案：  首先需要两个turnstile。   |  | | --- | | turnstile = Semaphore(0)  turnstile2 = Semaphore(1)  mutex = Semaphore(1) |   一开始第一个turnstile是关的（也就是线程一来就会阻塞在这里），第二个是开的。当所有线程都抵达第一个，我们把第二个锁起来；当所有线程都抵达第二个，我们把第一个锁起来。这样上文的那个t3在抵达下一轮循环时就会被已经上锁的第一个turnstile卡住。   |  | | --- | | # rendezvous  2  3 mutex . wait ()  4 count += 1  5 if count == n:  6 turnstile2 . wait () # lock the second  7 turnstile . signal () # unlock the first  8 mutex . signal ()  9  10 turnstile . wait () # first turnstile  11 turnstile . signal ()  12  13 # critical point  14  15 mutex . wait ()  16 count -= 1  17 if count == 0:  18 turnstile . wait () # lock the first  19 turnstile2 . signal () # unlock the second  20 mutex . signal ()  21  22 turnstile2 . wait () # second turnstile  23 turnstile2 . signal () |   这种解决方案叫做two-phase barrier （二阶段屏障），所有的线程都被强制等待了两次：第一次等所有线程都到；第二次让所有线程逐一获取临界区资源。  在阅读这本参考资料时，作者的一段话很打动我：  By finding the right kinds of statements to assert and prove, you can sometimes find a concise way to convince yourself (or a skeptical colleague) that your code is bulletproof.  通过寻找一种间接又直观的证明，你总可以说服自己（或质疑你的思路的人）：我写的代码在逻辑上是严谨而自洽的。  其实很多时候我并没有尝试严格证明自己的代码是目标的充要条件，而是通过诸如：   1. 只有第n个线程可以解锁或者给两个turnstiles上锁 2. 在一个线程解锁第一个turnstile时，他必须也穿过第二个，反之亦然；因此一个线程不可能成为precocious的。   这样的符合直观感受的命题来求证自己的思路是否正确。我认为在计算机领域，尤其system层面，由于我们有大量试错然后调试的机会，直观感受往往在编码过程中起到重要的作用。  这让我想起了John Tate说过的一句名言：  **Think Geometrically, Prove Algebraically.** | | | |