山东大学 软件 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202000300358 | 姓名：梁思睿 | | 班级：20.6 |
| 学号： | 姓名：王丁 | | 班级：20.6 |
| 实验题目： 实验三 使用信号量解决N线程屏障问题(Lab3) | | | |
| 实验学时：2 | | 实验日期：2022.10.26 | |
| 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：  设备名称 Three  处理器 Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz  机带 RAM 16.0 GB (15.8 GB 可用)  设备 ID 8CA8307A-62B3-42B1-AEE1-871F308ACDF3  产品 ID 00342-35669-83174-AAOEM  系统类型 64 位操作系统, 基于 x64 的处理器  笔和触控 为 10 触摸点提供触控支持 | | | |
| 软件环境：  宿主机：Windows 10 21H1 64位  虚拟机软件：WSL2  Linux:Ubuntu20.04.1LTS (GNU/Linux 5.4.72-microsoft-standard-WSL2 x86\_64)  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-LW | | | |
| 实验步骤与内容：  实验内容：   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。   **步骤一：分析说明Nachos的信号量是如何实现的**  Nachos信号量实现过程分析：  主要根据synch.h, synch.cc这两个文件中对信号量的定义进行信号量的实现。这两个文件中定义了信号量、锁和条件变量。  Nachos已经实现了Semaphore，基本结构为：  **class** Semaphore {  **public:**  void P(); // 信号量的 P 操作  void V(); // 信号量的 V 操作  **private:**  int value; // 信号量值 ( >=0)  List \*queue; // 线程等待队列  };  信号量的私有属性由信号量的值，它是一个阀门。线程等待队列中存放所有等待该信号量的线程。信号量有两个操作：P操作和V操作，这两个操作都是原子操作。  P操作   1. 当value=0时： 2. 将当前运行线程放入线程等待队列 3. 当前运行线程进入睡眠状态，并切换到其他线程运行 4. 当value>0时：value--     V操作：   1. 如果线程等待队列中有等待该信号量的线程，取出其中一个将其设置成就绪态，准备运行。 2. value++   在 threads/synch.h 中可以找到 Nachos 的信号量的定义：  **class** Semaphore {  **public**:  Semaphore(**const char**\* debugName, **int** initialValue); *// set initial value*  ~Semaphore(); *// de-allocate semaphore*  **char**\* getName() { **return** name;} *// debugging assist*  **void** P(); *// these are the only operations on a semaphore*  **void** V(); *// they are both \*atomic\**  **private**:  **char**\* name; *// useful for debugging*6  **int** value; *// semaphore value, always >= 0*  List \*queue; *// threads waiting in P() for the value to be > 0*  };  可见 value 就是其值，queue 就是等待队列，PV 操作直接用函数 P()和函数 V()实现（这两个函数比较简单，不再过多分析），初始化的时候要给定 value的大小，比如 mutex 互斥锁就只应该有一个。  信号量创建时需要名称和初始值两个部分，其中名称仅用于调试，因此叫做“debugName”。信号量结构体中包含getName( )方法和P( )、V( )两个操作，用于减少或增加信号量的值。List类型的queue用于存放被阻塞的线程，当信号量的值>0时，队列中的一个线程被移出并唤醒。  不可以用线程直接读取信号量的值，因为当把值放入寄存器时，可能已经发生了上下文的切换，并且其他线程可能也调用了P、V，因此现在真正的信号量的值可能已经变化。  synch.cc中相关源码   1. 信号量初始化   Semaphore::Semaphore(const char\* debugName, int initialValue)  {  name = (char\*)debugName;  value = initialValue;  queue = new List;  }   1. 信号量释放   Semaphore::~Semaphore()  {  delete queue;  }   1. P( )操作   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. V( )操作   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);  }  **步骤二：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。**  Nachos系统的线程没有一个统一的管理，创建线程后，线程就游离在整个系统中，只有线程就绪队列维护着所有就绪态的线程，任何时刻，我们都没有办法知道系统中到底有多少线程。当线程就绪队列为空并且中断队列中除了时钟中断之外没有其他中断时，Nachos就退出。  系统主要通过各种线程控制结构队列对线程进行管理，线程队列的作用是加速各种需要对线程进行检索的算法。系统中的线程队列主要有：   1. 系统的线程队列，线程一经创建就插入这个队列，知道线程的控制结构被释放才从该队列移出，该队列提供了遍历所有线程的手段，比如每次中断时，可以通过全局的线程队列对所有的定时线程进行一次时间检查判断是否需要唤醒。该队列的队首和队尾指针定义为全局变量globalThreadHead和globalThreadTail，线程结构中有两个指针nextThread和prevThread用来维护这个双向队列。 2. 每个线程管理两个线程队列，该线程活动的子线程队列和终止的子线程队列，这两个队列用以加速父子线程之间的通信算法，如Wait。线程结构中指针Parent指向父线程，指针firstChild指向活动子线程队列，而指针exitChild指向终止的子线程队列，类中还定义了指针leftSibling和rightSibling用来维护父子线程队列 3. 系统调度队列，该队列用于实现系统的线程调度算法。这个队列由类Scheduler进行管理 4. 各种系统资源上的等待队列，当线程等待分配系统资源而进入睡眠状态时，由系统资源的管理部件对睡眠的线程进行管理，这些管理一般都是用线程队列实现   在Nachos中用户线程是从核心线程继承而来的。  userRegisters数组是用户存储用户寄存器值的数组。其大小由NumTotalRegs确定。  MachineState存储在内核状态下运行的线程的状态。而用userRegisters数组存储在用户模式下运行的线程状态。  在Nachos中，用户线程都是以内核线程的方式开始的，当加载用户程序且创建地址空间之后，内核线程就转变成了用户线程。  **与线程相关的类分析**  **Scheduler类**  Nachos中作业调度程序，是一个Scheduler类的对象实现的，它的方法提供了所有对线程或进程调度的功能。当系统启动Scheduler对象会以一个全局变量scheduler的方式被定义。  void ReadyToRun(Thread\* thread); 将一个线程添加到就绪队列的尾部  Thread\* FindNextToRun 返回队首线程指针。  void Run(Thread\* nextThread); 该方法调用使用汇编写成的SWITCH函数来将当前线程上下文切换到另外一个线程的上下文。  **Scheduler类源代码分析**  **Scheduler.cc**  Scheduler的唯一数据成员是就绪队列，在schduler.h中定义。它存储所有处于READY（就绪）状态的线程。scheduler.cc中包含的几个重点函数如下：  **ReadyToRun**  void  Scheduler::ReadyToRun (Thread \*thread)  {  DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());  thread->setStatus(READY);  readyList->Append((void \*)thread);  }  将线程的状态设为就绪态，并且将其加入到就绪队列的尾部  **FindNextToRun**  Thread \*  Scheduler::FindNextToRun ()  {  return (Thread \*)readyList->Remove();  }  队首线程出队，返回队首线程指针  **Run**  void  Scheduler::Run (Thread \*nextThread)  {  Thread \*oldThread = currentThread;    #ifdef USER\_PROGRAM // ignore until running user programs  if (currentThread->space != NULL) { // if this thread is a user program,  currentThread->SaveUserState(); // save the user's CPU registers  currentThread->space->SaveState();  }  #endif    oldThread->CheckOverflow(); // check if the old thread  // had an undetected stack overflow  currentThread = nextThread; // switch to the next thread  currentThread->setStatus(RUNNING); // nextThread is now running    DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",  oldThread->getName(), nextThread->getName());    // This is a machine-dependent assembly language routine defined  // in switch.s. You may have to think  // a bit to figure out what happens after this, both from the point  // of view of the thread and from the perspective of the "outside world".  SWITCH(oldThread, nextThread);    DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());  // If the old thread gave up the processor because it was finishing,  // we need to delete its carcass. Note we cannot delete the thread  // before now (for example, in Thread::Finish()), because up to this  // point, we were still running on the old thread's stack!  if (threadToBeDestroyed != NULL) {  delete threadToBeDestroyed;  threadToBeDestroyed = NULL;  }    #ifdef USER\_PROGRAM  if (currentThread->space != NULL) { // if there is an address space  currentThread->RestoreUserState(); // to restore, do it.  currentThread->space->RestoreState();  }  #endif  }  把当前线程的状态设为运行状态，调用使用汇编写成的SWITCH函数来将当前线程上下文切换到另外一个线程的上下文。每次执行线程上下文切换后，新线程都会检查threadToBeDestroyed，将老线程清理掉。  **thread.h**  // Thread state  enum ThreadStatus { JUST\_CREATED, RUNNING, READY, BLOCKED };  // external function, dummy routine whose sole job is to call Thread::Print  extern void ThreadPrint(\_int arg);  线程的状态存储在ThreadStatus 类型的status数据成员中，线程的状态必须是以上枚举类型之一，当线程状态改变时，status值会有相应改变。每个线程有自己的线程栈和寄存器。  **Thread类**  class Thread {  private:  // NOTE: DO NOT CHANGE the order of these first two members.  // THEY MUST be in this position for SWITCH to work.  int\* stackTop; // the current stack pointer  \_int machineState[MachineStateSize]; // all registers except for stackTop  public:  Thread(const char\* debugName); // initialize a Thread  ~Thread(); // deallocate a Thread  // NOTE -- thread being deleted  // must not be running when delete  // is called  // basic thread operations  void Fork(VoidFunctionPtr func, \_int arg); // Make thread run (\*func)(arg)  void Yield(); // Relinquish the CPU if any  // other thread is runnable  void Sleep(); // Put the thread to sleep and  // relinquish the processor  void Finish(); // The thread is done executing    void CheckOverflow(); // Check if thread has  // overflowed its stack  void setStatus(ThreadStatus st) { status = st; }  char\* getName() { return (name); }  void Print() { printf("%s, ", name); }  private:  // some of the private data for this class is listed above    int\* stack; // Bottom of the stack  // NULL if this is the main thread  // (If NULL, don't deallocate stack)  ThreadStatus status; // ready, running or blocked  char\* name;  void StackAllocate(VoidFunctionPtr func, \_int arg);  // Allocate a stack for thread.  // Used internally by Fork()  #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  };  设置一个当前栈顶指针，以及放着除了栈顶指针寄存器外的其他寄存器的数组。以及，设置一个Thread类的构造函数和析构函数。  其中：  Fork函数用于产生从 JUST\_CREATE 到 READY 的状态转换，并生成线程实例可运行的环境；  Yield函数将线程从运行状态进入就绪状态，并从就绪队列选择一个线程运行；Sleep函数将线程从运行状态进入阻塞状态，并从就绪队列选择一个线程运行；  Finish函数说明这个线程以及执行完成，终止当前线程；  CheckOverflow函数检查线程是否已溢出栈。  线程的状态分为刚被创建（just\_created）、就绪（ready）、运行（running）和阻塞（blocked）四部分，StackAllocate函数为线程分配栈。  在Nachos中，用户线程是从核心线程继承而来的。运行用户程序的线程实际上有两组CPU寄存器 —— 一组记录执行用户代码时的状态，另一组用于记录执行内核代码时的状态。  extern "C" {  // First frame on thread execution stack;  // enable interrupts  // call "func"  // (when func returns, if ever) call ThreadFinish()  void ThreadRoot();  // Stop running oldThread and start running newThread  void SWITCH(Thread \*oldThread, Thread \*newThread);  }  **ThreadRoot函数**  ThreadRoot是以汇编形式写成的，它是在线程运行前第一个被运行的函数。除了main线程外，所有其它线程都是从ThreadRoot开始运行的。也就是说，一个线程所干的所有事都在这个ThreadRoot函数中，相当于一个封装。根据查找资料，可以了解到它的语法是：  ThreadRoot(intInitialPC,int InitialArg, int WhenDonePC,int StartupPC)  其中，InitialPC指明新生成线程的入口函数地址，InitialArg是该入口函数的参数；StartupPC是在运行该线程是需要做的一些初始化工作，指向InterruptEnable函数，比如开中断；而WhenDonePC是当该线程运行结束时需要作的一些后续工作，指向ThreadFinish函数。  **SWITCH函数**  进行线程上下文切换，停止运行旧线程并开始运行新线程。SWITCH首先保存所有重要的寄存器的值到当前线程的线程控制块中。即第一个私有的线程类成员stackTop及其后的machineState[machineStatesize]数组中。stackTop和machineState[]数组元素的摆列位置要和对应的寄存器的编排位置严格对应。  在保存的所有这些寄存器中，ra寄存器用于存放调用函数的返回地址。对于当前的ra，它包含了调用SWITCH后函数应返回的正确的地址。放弃了CPU的当前线程将会由其他上下文切换事件再次获得 CPU。当它被切换回来时，所有保存在 stackTop 和 machineState[]数组中的内容都将恢复到对应的寄存器中，包括返回地址寄存器 ra，第 113 行上的指令使得控制跳转到 ra 所保存的地址上，则当前线程又重新获得执行。  整个 Run 函数运行于内核，因为它属于 Nachos内核进程。  **thread.cc**  实现了用于管理线程事务的具体方法。该文件中有几个主要的操作：Fork、Yield、Sleep和Finish，下面根据源码分别进行理解：  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);  }  func是线程函数入口地址，arg是线程函数。Fork函数中，先调用了StackAllocate函数，分配线程栈并初始化machineState数组，然后关中断，将线程加入到就绪队列，之后恢复中断状态。  **StackAllocate 函数**  void  Thread::StackAllocate (VoidFunctionPtr func, \_int arg)  {  stack = (int \*) AllocBoundedArray(StackSize \* sizeof(\_int));  #ifdef HOST\_SNAKE  // HP stack works from low addresses to high addresses  stackTop = stack + 16; // HP requires 64-byte frame marker  stack[StackSize - 1] = STACK\_FENCEPOST;  #else  // i386 & MIPS & SPARC & ALPHA stack works from high addresses to low addresses  #ifdef HOST\_SPARC  // SPARC stack must contains at least 1 activation record to start with.  stackTop = stack + StackSize - 96;  #else // HOST\_MIPS || HOST\_i386 || HOST\_ALPHA  stackTop = stack + StackSize - 4; // -4 to be on the safe side!  #ifdef HOST\_i386    #endif  #endif // HOST\_SPARC  \*stack = STACK\_FENCEPOST;  #endif // HOST\_SNAKE    machineState[PCState] = (\_int) ThreadRoot;  machineState[StartupPCState] = (\_int) InterruptEnable;  machineState[InitialPCState] = (\_int) func;  machineState[InitialArgState] = arg;  machineState[WhenDonePCState] = (\_int) ThreadFinish;  }  宏PCState、StartupPCState、InitialPCState、InitialArgState、WhenDonePCState分别代表9、3、0、1、2。ThreadRoot是一个函数名，它是由汇编实现。InterruptEnable和ThreadFinish是两个静态函数名称。它们都被存储在machineState数组中。代表各个寄存器的值。同时，线程入口函数地址被存储在以InitialPCState为下标的数组中。线程函数参数被存储在以InitialArg为下表的machineState数组中。当线程开始运行时machineState[InitialPCState]会被加载到返回地址寄存器，存储线程函数的第一条指令开始的位置。  **Yield函数**  void  Thread::Yield ()  {  Thread \*nextThread;  IntStatus oldLevel = interrupt->SetLevel(IntOff);    ASSERT(this == currentThread);    DEBUG('t', "Yielding thread \"%s\"\n", getName());    nextThread = scheduler->FindNextToRun();  if (nextThread != NULL) {  scheduler->ReadyToRun(this);  scheduler->Run(nextThread);  }  (void) interrupt->SetLevel(oldLevel);  }  关中断，找到下一个运行的线程，之后当前线程进入就绪状态，并通过线程上下文转换把就绪队列中的下一个运行的线程变为运行状态，最后恢复中断状态。即：如果其他线程已准备好运行，则当前放弃CPU，加入就绪列表的末尾，以便之后重新被调度。  **sleep函数**  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    scheduler->Run(nextThread); // returns when we've been signalled  }  与Thread::Yield ()类似，但有些不同。先检验是否已经关闭中断。因为当前线程在等待同步变量（信号量等）时被阻塞，因此放弃 CPU。当就绪队列空的时候，cpu保持空闲状态，直到有一个线程就绪为止。之后某个线程将唤醒此线程，并将其放回就绪队列，以便可以重新调度它。Sleep方法会在执行IO操作时或者是等待一个事件时经常被调用。在调用Sleep之前，线程经常把它自己放入IO设备等待队列。  **根据上述源代码分析，可以得出Nachos创建及运行并发线程的过程：**   1. 创建多线程   Nachos中的线程是以创建一个Thread类的对象方式实现，如果需要创建线程，需要new一个新的Thread类对象，创建并发线程就要创建多个Thread对象。  但是当一个Thread类构造函数被调用时，仅生成了线程控制块，将线程状态变为JUST\_CREATED状态：并没有为线程分配栈空间，也没有初始化寄存器组，即如果此时要调度还不知道从哪里开始运行。   1. 运行并发线程   要将线程函数的入口地址和线程号作为参数，调用Fork函数，其调用函数StackAllocate为线程分配线程栈并初始化machineState数组，之后调度对象将自己推入就绪队列，这时线程具备了运行条件。当一个线程第一次被调度时，上下文切换历程将machineState中的值加载到返回地址寄存器，存储ThreadRoot第一条可执行指令的地址，因此一个新线程执行的第一个历程总是ThreadRoot  这样，既有了线程栈，又有了程序开始执行的位置，当该线程获得CPU后，线程就进入运行状态并自动调用自己的运行方法。在 Nachos 中，线程调度是由定义在 Threads/scheduler.h 和 scheduler.cc 的 Scheduler类的一个全局对象来完成的。这个类的方法提供了线程和进程的所有调度功能。例如，demo1中的prodcons++.cc调用Fork函数的方法如下：  DF`BSBKXNYBG7(~_XEL0U~4  对每个线程都执行Fork函数，使他们都具备运行条件。这时，系统就可以从就绪队列中挑选一个线程，该线程就可进入运行状态，在 Nachos 中上下文切换是由调用 Scheduler 类的 Run(Thread \*)函数开始的，直到该线程①运行方法执行结束或②线程中出现未经处理的异常或③调用其他方法或④发生阻塞而终止。当该线程执行结束后，系统会再从就绪队列中选择一个线程，使其进入运行状态；线程会进入阻塞状态时，让出CPU并暂时中止自己的执行，之后运行下一个线程。当引起该线程阻塞的原因解除后，线程被重新加入到就绪队列中，进入就绪状态，被CPU调度选中后它会从原来停止的位置开始，继续执行。最后，当线程需要终止时，ThreadFinish调用当前进程的中止函数 Finish。但是，线程不能自身析构，所以Finish仅是设置全局变量 ThreadToBeDestroyed 为当前进程，并调用 Sleep 函数将自身状态置为阻塞。该线程的真正终止实际上是由下一次线程上下文切换时完成的。  如果设置一个缓冲区，可以使主体的运行分离，减少彼此之间的交互，这样可以实现线程的并发执行。以demo1中的生产者消费者问题为例，有了缓冲区以后，生产者线程只需要往缓冲区里面放置数据，而不需要了解消费者此时消费的情况；同样，消费者只需要从缓冲区里拿出数据即可，也不需要了解生产者当前生产的情况。这样，就从逻辑上实现了“生产者线程”和“消费者线程”的分离，两类线程可以并发执行。  **步骤三：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？**  **N线程屏障问题分析：**  屏障允许每个线程等待，直到所有的合作线程都达到某一点，然后从该点继续执行。  旋转门：会有一个wait和一个signal快速连续，可以允许一个线程一次通过，它可以被锁定以阻止所有线程。在其初始状态（零）时，旋转门被锁定。第n个线程解锁它后，所有的n个线程都可以通过。    可以观察到：先试用mutex互斥夹紧，各个线程访问count变量，等到count增大到与线程数量相等后释放barrier信号量，旋转门锁定。进入屏障前将barrier信号量初始化为0，前（n-1）个线程都调用barrier.wait()，进入阻塞状态，这样使得线程都会合在一处。直到第n的线程将barrier释放后所有线程被释放，旋转门被解锁，之后的线程可以一次性通过旋转门。  实现代码：  #define N\_THREADS 10 // the number of threads  #define N\_TICKS 1000 // the number of ticks to advance simulated time  #define MAX\_NAME 16 // the maximum lengh of a name  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include "copyright.h"  #include "system.h"  #include "synch.h"  Thread \*threads[N\_THREADS];  char threads\_names[N\_THREADS][MAX\_NAME];  Semaphore \*barrier,\*mutex;  int count = 0;  void BarThread(\_int which)  {  printf("Thread %d rendezvous\n", which);  mutex->P();  count = count+1;  mutex->V();  if(count == 10){  barrier->V();  printf("Thread %d is the last\n", which);  }  barrier->P();  barrier->V();  printf("Thread %d critical point\n", which);  }  void ThreadsBarrier()  {  mutex = new Semaphore("mutex", 1);  barrier = new Semaphore("barrier", 1);  barrier->P();  // create and fork N\_THREADS of consumer threads  for (int i=0; i < N\_THREADS; i++) {  // this statemet is to form a string to be used as the name for thread i.  sprintf(threads\_names[i], "%d", i);  threads[i] = new Thread(threads\_names[i]);  threads[i]->Fork(BarThread, i);  };  }  运行结果：    **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？**  可以观察到有多个线程均判定自己为最后一个到达的线程（线程8和线程9）  出现这个现象的原因可能是：  线程8与线程9先后加入就绪线程队列，在线程8互斥完成count+1语句后线程9紧接着完成这样的语句，两个线程共用全局变量count，因此在线程8和线程9判断count==10时判断结果均为ture，因此两个线程都认为自己是最后一个线程。  **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？**  根据“The Little Book of Semaphores”中3.6.4小节中的代码实现N线程屏障的描述：在障碍物上等待后的另一个信号。现在，当每个线程通过时，它发出信号，以便下一个线程可以通过。这种模式，一个等待和一个快速连续的信号，经常发生，它有一个名字；它被称为旋转门，因为它允许一个线程一次通过，并且它可以被锁定以阻止所有的线程。在其初始状态（零）时，旋转门被锁定。第n个线程解锁它，然后所有的n个线程都通过。在互斥锁之外读取计数的值似乎很危险。  上述现象会导致多个线程均对barrier信号量解锁，导致barrier的value值>0，从而导致后续的barrier作为信号量的PV操作失效。  验证想法：    添加测试语句后可以观察到Thread8 和 Thread9经过barrier->V()操作后使得barrier的value值大于0，后续线程的一次barrier->P()操作将不会导致barrier的value值小于零，从而导致barrier丧失了作为屏障的作用  **步骤4：请修改代码消除上面3中出现的现象**  修改后代码：  #define N\_THREADS 10 // the number of threads  #define N\_TICKS 1000 // the number of ticks to advance simulated time  #define MAX\_NAME 16 // the maximum lengh of a name  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include "copyright.h"  #include "system.h"  #include "synch.h"  Thread \*threads[N\_THREADS];  char threads\_names[N\_THREADS][MAX\_NAME];  Semaphore \*barrier,\*mutex;  //void MakeTicks(int n) {} // advance n ticks of simulated time  int count = 0;  void BarThread(\_int which)  {  // MakeTicks(N\_TICKS);  printf("Thread %d rendezvous\n", which);  mutex->P();  count = count+1;  if(count == 10){  barrier->V();  printf("Thread %d is the last\n", which);  }  mutex->V();    barrier->P();  barrier->V();  printf("Thread %d critical point\n", which);  }  void ThreadsBarrier()  {  mutex = new Semaphore("mutex", 1);  barrier = new Semaphore("barrier", 1);  barrier->P();  // create and fork N\_THREADS of consumer threads  for (int i=0; i < N\_THREADS; i++) {  // this statemet is to form a string to be used as the name for thread i.  sprintf(threads\_names[i], "%d", i);  threads[i] = new Thread(threads\_names[i]);  threads[i]->Fork(BarThread, i);  };  }  将count==10的判断过程加入临界区，从而只有一个线程能够判断count==10为true，从而只有一个线程能够作为最后一个线程  修改后运行结果：    可以观察到只有线程9认为自己是最后一个线程  **步骤五：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，**  测试过程：  随机种子seed=1：    随机种子seed=2：    随机种子seed=3：    可以观察到各线程打印输出的rendezvous行的顺序，基本就是线程被创建的顺序(0,1,2…9)的现象，出现这种现象的原因为：  for (int i=0; i < N\_THREADS; i++) {  // this statemet is to form a string to be used as the name for thread i.  sprintf(threads\_names[i], "%d", i);  threads[i] = new Thread(threads\_names[i]);  threads[i]->Fork(BarThread, i);  };  线程通过for循环以0-9的顺序被创建并且通过Fork方法初始化并添加到就绪队列中。在 Nachos 系统中，有一个线程就绪队列，其中是所有就绪线程。调度算法非常简单，就是取出第一个放在处理机运行即可。因此线程的运行顺序为1-9，线程运行初始阶段没有信号量进行控制，因此打印输出的rendezvous行的顺序，基本就是线程被创建的顺序(0,1,2…9)  添加空循环：  void MakeTicks(int n) {  for(int i=0;i<n;i++){ }  } // advance n ticks of simulated time  添加sleep:  void MakeTicks(int n) {  sleep(1);  } // advance n ticks of simulated time  添加后的运行结果：  seed=32    seed=135    并没有改变线程打印输出的rendezvous行的顺序，基本就是线程被创建的顺序(0,1,2…9)的现象。即使为线程添加的休眠函数（或者通过空循环耗时），线程依然会按照添加进队列的顺序进行运行，并不会干扰到线程的运行顺序，因此无法改变打印输出的rendezvous行的顺序。  **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。**  修改后代码：  #define N\_THREADS 10 // the number of threads  #define N\_TICKS 10000 // the number of ticks to advance simulated time  #define MAX\_NAME 16 // the maximum lengh of a name  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include <time.h>  #include "copyright.h"  #include "system.h"  #include "synch.h"  Thread \*threads[N\_THREADS];  char threads\_names[N\_THREADS][MAX\_NAME];  Semaphore \*barrier,\*mutex;  Thread \*current\_thread;  void MakeTicks(\_int which){  int x = rand();  int y = x % 10;  if(y>5){  threads[which]->Yield();  }  }  int count = 0;  int count1 = 0;  void BarThread(\_int which)  {  MakeTicks(which);  printf("Thread %d rendezvous\n", which);    mutex->P();  count = count+1;  if(count == 10){  printf("Thread %d is the last\n", which);  barrier->V();  }  mutex->V();  barrier->P();  barrier->V();  printf("Thread %d critical point\n", which);  }  void ThreadsBarrier()  {  srand(time(0));  mutex = new Semaphore("mutex", 1);  barrier = new Semaphore("barrier", 1);  barrier->P();  // create and fork N\_THREADS of consumer threads  for (int i=0; i < N\_THREADS; i++) {    threads[i] = new Thread("thread"+i);  };  for (int i=0; i < N\_THREADS; i++) {  threads[i]->Fork(BarThread, i);  };  }  修改后运行效果：    修改思路：  由于线程创建后直接运行，那么线程就会按照创建的顺序运，如果我们希望线程按照一个随机的顺序运行，那么我们需要通过Yield方法将当前运行强制切换到另一个就绪线程运行。我初步尝试在线程创建时使用Yield，但是由于线程一开始就进入临界区，Yield方法无法中断其进程而导致失败。因此我再次尝试在MakeTicks方法中使用Yield，随机选取一些函数中断其进程并切换到另一个进程，如结果所示，线程可以随机运行。 | | | |
| 结论分析与体会：  这个实验使我复习了上学期学习的信号量有关知识点，并且通过运行代码真正了解了信号量的实现机制和作用。  此外通过学习生产者-消费者问题中的代码我了解到实现互斥的 P 操作一定要在实现同步的 P 操作之后，不然可能会出现死锁。如若生产者先 P(mutex)给临界区加锁，再 P(empty)但此时缓冲区为空，生产者进程卡住，而 此时消费者 P(mutex)也被卡住，两者相互等待被对方唤醒，出现死锁。V 操作不会导致进程阻塞，生产者和消费者最后两个 V 操作顺序可以交换。  通过学习屏障解决方案我了解到可以通过信号量制造屏障和旋转门。屏障允许每个线程等待，直到所有的合作线程都达到某一点，然后从该点继续执行。旋转门会有一个wait和一个signal快速连续，可以允许一个线程一次通过，它可以被锁定以阻止所有线程。在其初始状态（零）时，旋转门被锁定。第n个线程解锁它后，所有的n个线程都可以通过。  此外在实验过程中我学习到了nachos中的多线程是如何创建并运行的，对于操作系统中线程有关的知识有了进一步的了解，受益匪浅。 | | | |