山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202000300125 | 姓名：贾星宇 | | 班级：计软20.5 |
| 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年11月7日 | |
| 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：  HUAWEI matebook14 2020笔记本  Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz CPU  8GB内存  512GB SSD | | | |
| 软件环境：  宿主机：Windows 10 21H2 64位  虚拟机软件：VMware Workstation Pro 16.2.4 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：   1. 分析说明Nachos的信号量是如何实现的     Nachos信号量的实现集中于synch代码文件中的Semmaphore类中实现，主要分为如下三个部分：  成员变量：字符串类型的name用于debugging；int类型的value为关键元素信号量值，后续将利用此值实现信号量的屏障；List类型的queue，记录等待的线程。  P()函数。P()函数代表了同步机制中的P操作，主要用于线程申请临界区资源，其工作方式如下：首先利用while循环判断，如果当前信号量的value为0，则说明临界区资源被占用，则将当前线程放入此信号量的等待队列queue，并利用Sleep()函数使此线程休眠。在休眠操作中函数寻找下一个线程作为nextThread并运行。当信号量不为0时，说明临界区资源可以使用，则将信号量value--，随后使用。    V()函数：此函数代表了同步机制中的V操作，主要用于释放临界区资源。当执行V操作时，首先利用Remove()函数从当前信号量的线程等待队列中移出一个线程，表示此线程可以使用临界区资源。如果没有线程等待使用则正常运行下一个线程。随后将信号量的value++，表示此临界区资源占有少一。    为了保证上述两操作的原子性，这两个函数都用关中断和开中断扩起。至于如何使用上述两函数，仿照上学期所学，假设存在一值a不能同时访问，为临界区资源，那么需要为该值设置一信号量，如semaphoreA。当其他方法要改变a的值时，需要有如下操作：  **semaphoreA.P()//申请资源**  **//change a’s value，如果申请到了**  **semaphoreA.V()//使用结束释放资源**   1. 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。   在Nachos中，我们可以通过for循环+Fork函数实现并发进程的创建和运行，如demo1中的prodcons++源代码文件中，在ProdCons方法中创建了N\_PROD个并发的生产者进程的代码：    也就是说，我们在创建线程时，通过new Thread()方法创建，随后调用Fork方法将线程所使用的函数和参数传入线程中。  如果nachos中存在多个线程，那么这些线程会并发执行，也就是轮流使用处理机（CPU）。正常情况下，当一个线程发生中断后，处理器会利用线程调度策略调取另一个线程使用CPU，这个方法的实现是通过Interrupt类中，每次执行OneTick函数之后检查是否中断实现的：如果存在中断，则调用Yield函数让当前线程放弃CPU并进行上下文切换，实现了线程的并发执行。   1. 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线程屏障问题指的是只有当n个线程都汇合（rendezvous）后，才可以开始运行临界区的程序。因此我们需要有几个变量：  n，需要汇合的线程的个数；count，已经汇合的线程的个数；mutex，临界资源count的信号量；barrier，能否运行临界区程序的信号量，一开始为0，直到所有线程汇合后其value变为1，表示可以访问临界资源。  根据MakeFile中的提示，我需要为lab3添加文件threadsbar.cc，按照readme文件中的模板写入，并添加至Makefile文件中。      随后开始编写threadsbar.cc文件。  首先初始化需要的变量，其中count、mutex、barrier在先前已描述，threads为参与运行的进程：    随后是MakeTicks函数，在经过苦苦思索和资料查阅后，猜测此函数的作用为定时，类似于js中的settimeout，可以让时间在这个函数中停留，也是处理器走过的时钟打点数。在此函数中N\_TICKS设置为1000，也就是1000ms。或者说，每次BarThread函数要运行至少1s。因此选择C++语言的time函数完成ticks的操作：循环，如果开始循环时间与当前时间相差不到1s，则继续循环：    随后是BarThread函数，在此函数内完成线程屏障问题。首先让线程隔1s运行，随后打印消息，表示线程汇合，随后利用PV操作修改count令其自增，随后判断，如果count为10，表示所有线程已汇合，则用V函数允许访问临界区代码，随后经过P和V函数访问。如果没有到，那么线程将会阻塞在下方的P函数上，不会访问到临界区（后来发现，是否间隔1s也没有什么影响）：    随后是运行函数ThreadsBarrier，在此函数内首先初始化mutex和barrier，并设置其初始值分别为1和0，随后利用Fork开启每一个线程：    打开命令行，make无误后输入./nachos -rs 1:    发现程序正常运行。  利用不同的随机种子运行：  -rs 3：    -rs 10      **当测试到-rs 12时发现问题：**    可以看到，当第9个线程汇合后报告为最后一个，但后续在各个线程运行临界区程序时，线程1、5又分别报告了为最后一个。  让我们重新回到BarThread方法中寻找问题：    设想每一个进程状态，当进程进入函数后，便可以打印“汇合”消息，随后进入临界区修改count，随后判断修改后的count是否到达N\_THREADS。但可能出现如下极端情况：进程0~10皆执行完了第37行，也就是改变了count值，随后又开始执行下方if语句，那么他们都会满足if条件，并进入语句块打印消息。这样就出现了多个进程都认为自己是最后一个进程的问题。  这样不会影响题目的要求，因为只要进程进入if语句块，不管进入几个，由于先前的PV操作，都可以说明所有进程已经汇合，而且不管对barrier进行几次V操作，都可以保证所有进程可以访问临界区代码，因此不影响N线程屏障的问题。但是，显然我们需要修改这个问题。   1. 对代码做出修改以消除上述现象。   在3中已分析知道出现问题的原因是出现了部分线程在if语句块前面滞留的现象，导致那部分线程等到了其他线程也执行完加操作。因此我们要做的是让线程执行count++操作后立刻执行if语句块，换个角度考虑，我们要做的是让线程在执行++操作和if语句块内容的过程中不受其他线程的打扰。  这样思路就比较清晰了，我们只需要让if语句块和上述count++操作都放到PV临界区内即可：    再次执行随机种子为12的操作，发现问题消失：     1. 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。   可以很清楚地发现，大部分随机种子下打印的rendezvous行的顺序，基本就是线程被创建的顺序(0,1,2…9)的现象，如前四个：      仅有第三个不是线程的创建顺序。  实验表示，无论是否注释掉printf前的空循环耗时，都不会影响打印的结果。随后尝试利用Linux的sleep（1）来延时1s，发现结果和上述情况完全相同，因此上述方法不是问题的关键所在。  但是，-rs选项是有作用的，因为存在部分种子（如3）会导致结果的不一致，只是大部分打印输出顺序为创建顺序。  经过大量的实践和检验，考虑了多种因素后，我决定回到出发点，从线程同步的角度审视这个问题。这时候我发现，在ThreadsBarrier方法中用Fork创建线程时，利用了for循环，而每个for循环中先是运行相应线程创建类的构造函数，随后创建一个线程，随后再次运行新fork类的构造函数并再次创建线程。由于main线程创建新线程后可以和新线程同步运行，因此极有可能出现如下情况：  Main线程For循环创建出第一个线程，随后开始创建第二个线程。而在创建第二个线程的时候第一个线程已经开始运行并进行打印，因此出现了打印顺序基本上是创建顺序这种情况。     1. 请试着修改代码解决上面5出现的现象。   起初我认为是由于构造函数占用了较多的时间，因此我先一次for循环构造，随后用for循环只执行fork函数，但依旧不起效果，所以我们必须引入一个新信号量test，用这个信号量的PV保证在for循环过程中新创建出的线程被阻塞无法运行，而等到创建结束后新线程才可以运行。具体代码如下：  我们加入信号量test来解决问题。首先是创建线程函数：    随后是线程执行函数，在开头引入PV阻塞：    效果检验：  我们取随机种子是7，8，9，10的情况作比较，在PV限制前：      添加test信号量PV后：      可以显著的看到，添加PV后，进程打印的顺序已经基本上不同于创建进程的顺序。  在后来的讨论中，我们对于这个问题有了更深入的理解：  首先，加入空循环耗时无法解决问题的原因，是因为nachos寄居于宿主机linux上，因此它的每一个系统线程是由linux上的用户线程相对应的。我们在利用while循环消耗时间的时候，实质上运行在linux上，而不是调用nachos自带的线程切换和时间片轮转来完成的，因此-rs对空循环消耗的时间长短几乎是没有影响的，因此无法改变现状。  如果想要改变现状，还是需要从根本上做起。我们在执行打印之前，首先调用nachos的模拟时钟，利用模拟时钟实现调用interrupt方法OneTick，而随机种子-rs可以影响的是OneTick，因此就可以实现打印顺序与线程创建顺序不同的效果。 | | | |
| 结论分析与体会：  此次实验令我收获颇丰。  在代码的阅读和理解能力上，我掌握了更多C语言的语法以及与操作系统有关的各种函数的使用方法。  在审视所写代码的优缺点上，我拥有了更强的测试及思考能力。当我根据大纲的指示去一点点测试、一点点发现问题并最终修正后，无论是成就感还是自己的能力都有了很大的提高  当遇到问题时，应该广泛地查阅资料，并从根源上去看这个问题的解决方法，同时不应停留在表层，应该多加测试，多尝试才可以成功。 | | | |