操作系统,作业 4

此作业仅包括几个编程问题,以及一个简单的激活帐户的作业,您将需要进行家庭作业 5. 请记住,您的程序需要注释并始终如一地缩进,并且您必须记录您的来源并至少写下自己从头开始编写一半的代码。

1. (10 分)Arc 帐户激活。

   在我们的下一个家庭作业中,您将需要使用一个名为 arc 的高性能计算集群。
   在您执行此操作之前,您需要按照步骤在 arc 上激活您的帐户并设置一个密钥对以便您可以登录。如果您没有完成此任务的 arc 激活,则存在您不会遇到的问题能够在下一次作业中完成。为了帮助确保每个人都准备好,我在这个任务上有一个低点值的问题,要求你在 arc 上激活你的帐户并在 arc 集群的头节点上登录。

   在这项任务到期前大约五天,您应该会收到系统管理员在 arc 上发来的电子邮件。它将包含一个指向有关 arc 集群的更多信息的链接(其中大部分信息不适用于您)和一个用于输入公钥以激活您的帐户并允许您登录的链接。该电子邮件将在大约一周内过期,所以你需要在拿到arc账号后尽快激活,否则下一次作业你将无法做其中一题。

   首先,您需要生成一个 RSA 密钥对。这将在您的主目录中名为 .ssh 的目录下创建一对文件。每当您想登录到 arc 时,您都需要访问这些文件,并且您只能从大学网络上的计算机登录到 arc。从可以访问您的 AFS 文件空间的机器上执行此操作可能很好,这样您就可以从几乎任何大学机器上获得新的密钥对。从其中一台 remote.eos.ncsu.edu 机器创建密钥对应该可以正常工作。不要在 VCL 映像上创建密钥对。许多 VCL 图像是具有临时文件系统的临时系统。当机器离开时,该文件系统将被删除。如果您在这样的系统上创建您的密钥对,那么在您的 VCL 映像被删除后,您将无法在以后使用它。

   如果您以前从未生成过 ssh 密钥对,那很容易。如果您以前做过,则可以使用您已经创建的密钥对,或者为不同的目的保留多个密钥对。如果您想生成与我在这里建议的不同的密钥(例如,将其命名为不同的名称),那很好,但是您需要负责了解自己在做什么。

   要生成您的密钥对,请登录其中一个 EOS Linux 系统,然后输入以下命令。它可能会问您一些有关如何存储密钥的问题。您可以只采用这些的默认答案。

   ssh-keygen -t rsa -b 4096

   此命令将创建一个 4096 位 RSA 密钥对,并将其存储在主目录下 .ssh 目录中的几个文件中。

   看看这对的公钥:

   猫 ~/.ssh/id_rsa.pub

   要使用 arc 系统注册此公钥,您需要单击 arc 管理员在电子邮件中收到的超链接。要与此链接末尾的机器通信,您似乎需要一个校园 IP 地址,因此您可以从校园进行此操作,也可以在按照其余说明操作之前通过 VPN 进入校园。如果您以前从未使用过 VPN 连接到校园,可以在以下位置找到安装所需软件和连接的说明:

https://oit.ncsu.edu/campus-it/campus-data-network/vpn/现在,单击电子邮

件中的链接以激活您的帐户。您的 Web 浏览器可能会抱怨它看到的是自签名证书;没关系。到达目标页面后,将 .ssh/id_rsa.pub 文件的内容粘贴到电子邮件链接带您进入的文本框中。请务必获取 .ssh/id_rsa.pub 文件的全部内容,包括开头的 ssh-rsa 部分。然后,按下表格底部的按钮,您的帐户就会被激活。 ...但是您还没有完成这部分作业。

要获得这部分作业的学分,您只需要再做一件事。从其中一台大学 Linux 机器上,您应该能够通过 ssh 连接到 arc 系统上的头节点:

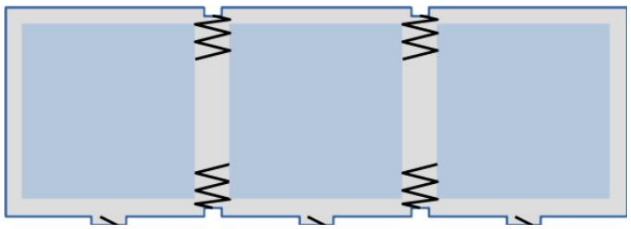
```
ssh arc.csc.ncsu.edu 网站
```

如果这有效,您将在 arc 系统上得到一个 shell 提示符。您需要尽可能在 arc 上获得此 shell 提示才能解决此问题。当我在 arc 系统上时,我得到如下提示。你的看起来应该是一样的,尽管你会看到一个不同的统一 ID。

```
[sjiao2@login ~]$
```

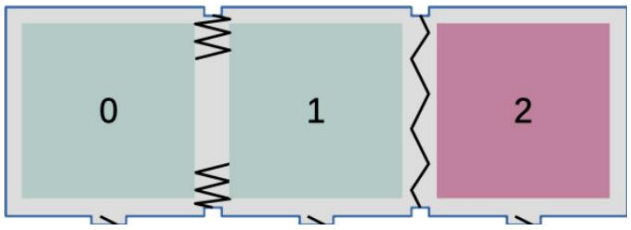
如果这行得通,那么您应该能以良好的状态迎接下一次任务。您可以通过输入 exit 退出 arc 系统。

2. (40 分) 对于这个问题,你将使用POSIX 互斥锁和条件变量来实现会议厅分配系统的部分模拟。下图显示了我们想象中的房间类型。这是一个长长的大厅,间隔一定的间隔。



大厅中的隔板可用于将其分成单独的空间。所以它可以同时被多个组织使用,每个组织使用一个或多个空间,它们之间有分区。

下图左边是两个空格,右边是一个空格。



大厅可以包含任意数量的空间,n,在启动时给定。空格从左侧的零编号到右侧的 n-1。当一个组织想要使用大厅的一部分时,他们会指定它需要多少空间。然后,他们等待直到有那么多连续的空间可用。一旦他们被分配了他们请求的空间数量,他们就可以想用多久就用多久。当一个组织使用大厅的一部分时,其他组织不能使用这些

相同的空间,但其他空间可以被另一个组织使用。这就是分区的用途。例如,在上图中,一个组织可能使用空间 0 和 1,而另一个组织使用空间 2。

当一个组织使用完大厅的一部分时,他们会释放它。然后,它可供其他组织使用。

空间分配

在我们的模拟中,组织被实现为线程。他们在分配和释放大厅空间时调用我们系统上的功能。如果一个线程不能马上得到它请求的空间,系统会让它们等待直到有足够的空间可用。

系统将使用先适应策略来分配空间。它总是会响应每个请求给出最小编号的空格。例如,想象一个 10 个空间宽的大厅,其中有 2、3、6、7 和 8 个可用空间,另外 5 个空间正在使用中。如果一个组织需要一个宽度为 2 的空间,系统会给它空间 2 和 3;这是足够大的请求的最小编号的连续空间。但是,如果组织需要宽度为 3 的空间,则系统将不得不分配空间 6、7 和 8。

系统可以按任何顺序授予请求;它不必按照请求的顺序授予它们。但是(除非你做了额外的功劳)如果空间可以满足等待线程的请求,它不应该让空间空闲。

监控实施

您将要开发一个名为 monitor 的同步机制来实现会议厅分配系统。某些编程语言(如 Java 和 C++)提供了内置监视器,但 C 没有。提供了监视器的部分实现和一些测试程序。头文件 hall.h 定义了您将实现的接口。在 hall.c 中,您将实现监视器的功能。在 hall.c 中,您还可以为监视器创建所需的任何状态。这将包括一个用于控制对监视器的访问的互斥锁。可能您还需要一个或多个条件变量,以使线程在需要时在监视器中等待。您还可以使用其他变量来跟上监视器的当前状态(例如,大厅的哪些部分正在使用)。在监视器实现文件中将这些变量定义为静态全局变量。这样,您可以从监视器的任何功能访问它们,但它们不会与该组件外部使用的名称冲突。

对于某些功能,监视器需要打印出分配报告。这是一个包含 n 个字符的序列,其中 n 是大厅的宽度。报告中的每个字符从左到右显示其中一个空格的当前状态。星号表示空间是空闲的。对于已分配的空间,相应的字符应为当前使用该空间的组织名称的首字母。例如,报表“**aabb*”代表一个有 7 个空格宽的大厅。

左边的两个空格和最右边的空格是可用的。空间 2 和 3 由以“a”开头的组织使用,空间 4 和 5 由以“b”开头的组织使用。

您的显示器将提供以下功能:

- void initMonitor(int n)
  此函数初始化监视器,分配它需要的任何堆内存并在必要时初始化它的状态。 n参数是大厅的总宽度,它包含的空格数。

- void destroyMonitor()
  在我们使用完监视器后调用此函数。它应该释放监视器使用的任何资源。

- int allocateSpace(char const *name, int width)
  当线程想要使用大厅中的空间时调用此函数。第一个参数是线程的组织名称,第二个width参数是它需要的连续空间的宽度。该函数返回一个名为 start 的值,如下所述。

  如果没有足够的可用空间,此函数将使线程等待,直到它可以拥有所需的空间。如果线程必须在此函数中等待,则监视器应打印如下消息,其中 name 是线程给出的名称,current-allocation 是大厅的分配报告。只有在线程必须等待时才打印这一行。如果线程可以立即拥有它请求的空间,则忽略它。

  名称等待:当前分配

  一旦线程可以拥有其请求的空间,allocateSpace() 函数应该打印如下一行,给出线程的名称和大厅的（更新的)分配报告。

  分配的名称:当前分配

  此函数返回的起始值应该是分配给线程的最左边空间的索引。这告诉线程它被赋予了大厅的哪一部分。当释放空间时,线程应将此值（连同它的名称和分配的宽度)传递回监视器。

- void freeSpace(char const *name, int start, int width)
  一个线程在使用它在先前调用 allocateSpace() 时提供的空间完成时调用此函数。 name字段是线程的组织名称,start是分配给线程的最左边空间的索引,width是分配给线程的连续空间数。调用此函数后,从 start up 到 start + width - 1 的空间被认为是可用的,可能会被分配给其他线程。

  此函数应打印出如下一行,其中 name 是释放空间的线程的名称,current-allocation 是大厅的分配报告（空间被释放后立即）。

  名称释放:当前分配

额外学分

这个监视器有饿死的危险。需要非常宽的空间的线程可能会饿死。为了获得最多 8 分的额外分数,构建您的显示器以防止饥饿。

为此,您将使用老化。任何线程的年龄 t 将被定义为当 t 在 allocateSpace() 中等待时从 allocateSpace() 返回的其他线程的数量（即,在 t 之前获得空间的其他线程的数量）。因此,由于线程必须等待,它们的年龄会增加。

如果你做了额外的功劳,你将需要在 allocateSpace() 打印的分配消息的输出中报告一个额外的值。您的消息应如下所示。括号中的值 a 应该给出每个线程在获得其请求的空间时的年龄。这将帮助我们检查您的额外信用,并且它可能会帮助您在开发解决方案的额外信用部分时进行调试。

分配的名称 (a):当前分配

每次线程调用 allocateSpace() 时,线程的年龄从零开始。因此,线程不会从一个请求到下一个请求累积年龄。

当有一个更旧的线程在等待时,我们将通过让线程等待它们请求的空间（即使空间可用)来防止饥饿。具体来说,只要有另一个线程ta的年龄比 tb 大 100 以上,监视器就应该让线程tb等待空间（即使空间可用） 。

例如,如果线程ta在年龄为 101 的 allocateSpace() 中等待,则调用 allocateSpace() 的新线程tb将无法立即获得其空间（即使空间可用）。新线程tb的年龄为零,而ta的年龄大于 100。 tb线程将不得不等待ta。当然,其他一些等待线程tc仍然可以立即获得它的空间,前提是tc的年龄为 1 或更大。

对于额外的信用,您不必更改驱动程序。您应该能够仅使用监视器中的代码来实现老化。事实上,您的无饥饿监视器应该与其他未随任务分配的测试驱动程序一起工作。

驱动程序

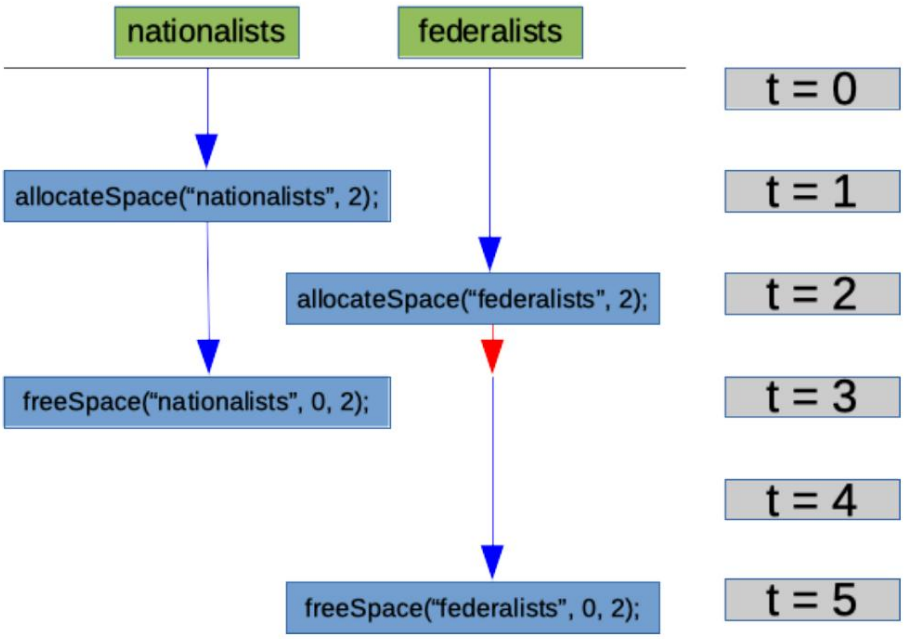
我们提供了四个测试驱动程序来帮助您锻炼您的显示器,driver1.c、driver2.c、driver3.c 和driver4.c。如果您正在尝试额外信用选项,还有一个可以使用的驱动程序。
还有其他我们没有提供的驱动程序。我们将在测试您的显示器时使用这些。查看驱动程序 1、2 和 3,您会发现可以轻松编写自己的驱动程序来测试监视器的特定行为。这就是我们在测试您的解决方案时要做的事情。

要使用监视器实现编译其中一个驱动程序,您应该能够使用如下命令。这会使用您的监视器实现在 driver1.c 中进行编译,并编写一个名为 driver1 的可执行文件。要试用不同的驱动程序,只需更改驱动程序源文件的名称和可执行文件的名称（在 -o 选项后给出的名称）。

```
gcc -Wall -std=c99 -D_XOPEN_SOURCE=500 hall.c driver1.c -o driver1 -lpthread
```

driver1.c 程序旨在确保不允许两个组织同时使用同一空间。下图显示了当您使用工作监视器运行它时应该发生的情况。这些组织以历史上的政党命名。我们建了一个三格宽的大厅。

两条垂直线显示两条线的行动,民族主义者和联邦主义者。蓝色框显示这些线程何时调用监视器中的函数,红色箭头显示线程在返回之前必须在监视器函数中等待的位置。预期行为按时间从上到下排序（右侧显示特定的秒数）。一秒钟后,nationalists 线程尝试分配一个宽度为 2 的空间。它应该立即获得左边的两个空间（0 和 1）。一秒钟后,联邦主义者线程试图分配一个宽度为 2 的空间,但它必须等待。当民族主义者线程在时间 3 释放其空间时,联邦主义者线程应该获得两个空间（同样,由于先适应策略,最左边的两个空间）。它使用这些空间两秒钟然后释放它们。



您可以运行程序并在生成时观察输出以检查时间。我发现在数秒时按回车很有用。这会在输出中以（大约）一秒的间隔放置空行。它使查看终端输出和查看每条消息何时打印以及多条消息何时几乎同时打印变得容易。这是您应该从第一个驱动程序获得的输出（以及您应该在括号中看到每条消息的大约时间）。

民族主义者分配:nn*

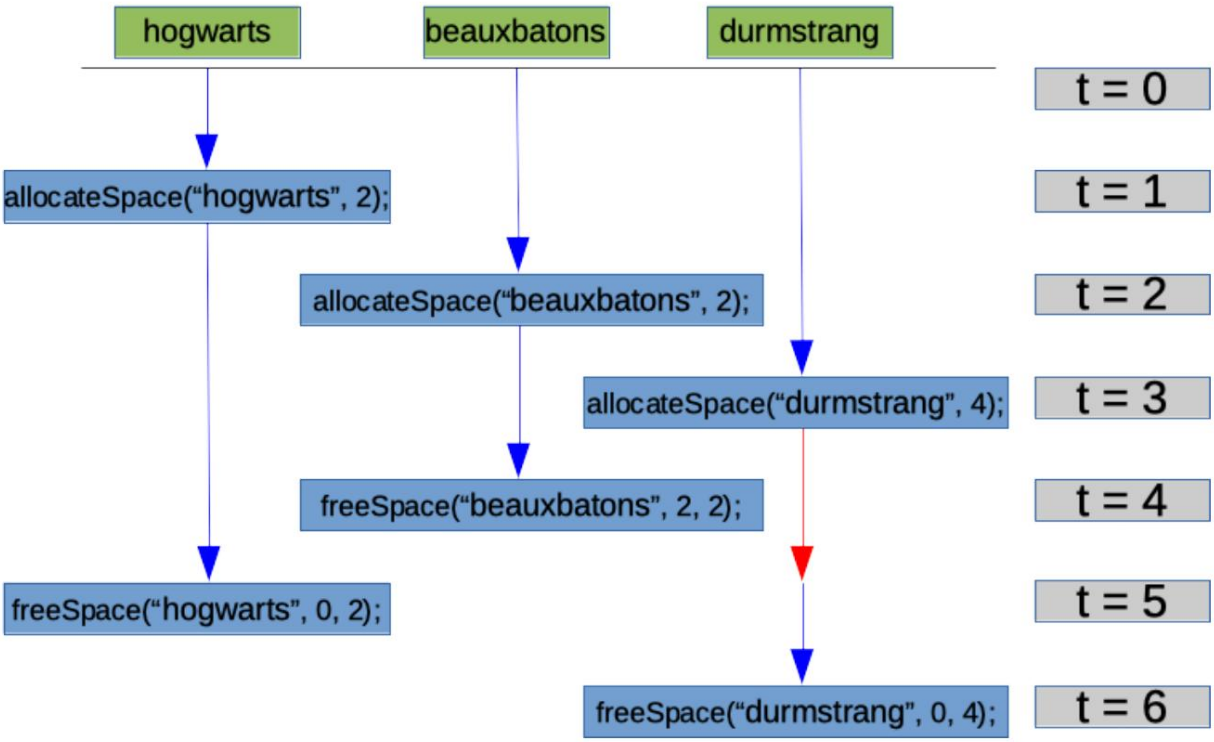
联邦主义者等待:nn* 民族主义者获释:***

联邦主义者分配:ff*

联邦党人获释:***

（在 1 秒时）（在 2 秒时）（在 3 秒时）（在 3 秒时）（在 5 秒时）

driver2.c 程序检查以确保仅在所有其他组织都让开后才将空间分配给组织。它以魔法学校为组织机构，大厅宽4格。一秒钟后，hogwarts 分配了一个宽度为 2 的空间。然后，beauxbatons 分配了剩下的两个空间。在时间 3，durmstrang 尝试分配宽度为 4 的空间，但没有可用空间，因此他们必须等待。在时间 4，beauxbatons 释放了他们的空间，但 durmstrang 仍然需要再等一秒钟让 hogwarts 放弃剩余的两个空间。德姆斯特朗将所有四个空间保持一秒钟，然后释放它们。

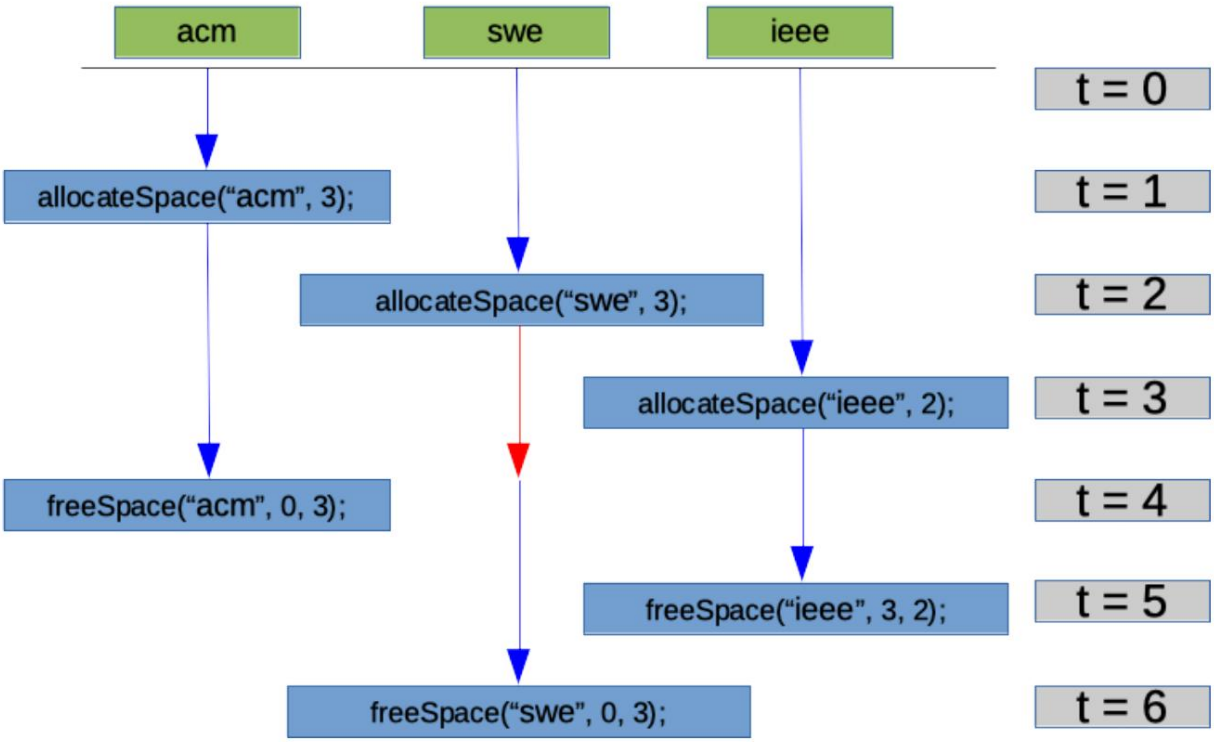


如果您使用 driver2.c 构建并运行您的监视器,您应该得到如下输出（时间显示在右侧）。

霍格沃茨分配:hh** 布斯巴顿分配:
hhbb
德姆斯特朗等待:hhbb 布斯巴顿释放:
hh**
霍格沃茨释放:**** durmstrang
分配:dddd durmstrang 释放:****

（在 1 秒） （在 2
秒） （在 3 秒） （在
4 秒） （在 5 秒）
（在 5 秒） （在 6
秒）

driver3.c 程序使用计算和工程中的专业组织来表明,在请求更多空间之前,可能会批准对少量空间的请求。它使用宽度为 5 的大厅。一秒钟后，acm 线程获得前三个空间。一秒钟后,swe 线程请求宽度为 3 的空间,但它必须等待,因为只剩下两个空间。一秒钟后,ieee 线程能够立即获得其请求的批准,因为它只需要两个空格。稍后,swe 线程能够在 acm 线程释放其空间后获得其请求的空间。



如果您使用 driver3.c 构建并运行您的监视器,您应该得到如下输出:

acm 分配:aaa** swe 等待:aaa** （在 1 秒） （在 2
ieee 分配:aaaii 秒） （在 3 秒） （在 4
秒） （在 4 秒） （在 5
acm 释放:***ii 秒） （在 6 秒）
我们分配:sssii
IEEE 释放:sss**
我们被释放了:*****

driver4.c 程序是一个压力测试。它使用以 1970 年代乐队命名的 20 个线程一遍又一遍地分配和释放各种宽度的空间,在操作之间随机等待几分之一秒。程序行为是随机的,时间的变化会给你每次执行不同的输出。

但是,您可以查找一些内容。首先,检查以确保您的解决方案没有死锁。前 4 个线程需要较大的空间,但后 4 个线程需要的空间要小得多,因此它们可能会更快地获得所需的内容。从我下面的示例输出中,您可以看到第一个乐队 Aerosmith 仅获得了 408 次他们请求的空间,但 Yes 获得了 968 次。总的来说,当我在一台远程 EOS Linux 机器上运行这个驱动程序时,我总共获得了近 14,000 个预订。

海洋分配:OOO**********
女王分配:OOOQQ**********
心脏分配:OOOQQHHHH**********
雷蒙斯分配:OOOQQHHHHRR*********
木匠分配:OOOQQHHHHRRCCCCC****
行程分配:OOOQQHHHHRRCCCCCJJJ*
M等待:OOOQQHHHHRRCCCCCJJJ*

```
Delfonics 等待:OOOQQHHHHRRCCCCCJJJ*
旅程释放:OOOQQHHHHRRCCCCC****
M分配:OOOQQHHHHRRCCCCCMMM*
释放的心:OOOQQ****RRCCCCCMMM*
U2 分配:OOOQQU***RRCCCCCMMM*
多多分配:OOOQQUT**RRCCCCCMMM*
创世等待:OOOQQUT**RRCCCCCMMM*
释放海洋:***QQUT**RRCCCCCMMM*
女王获释:*****UT**RRCCCCCMMM*
Delfonics 分配:DDDDDUT**RRCCCCCMMM*
旅程等待:DDDDDUT**RRCCCCCMMM*

 ...省略了很多输出...

老鹰被释放:******MMM****GGGG***
M 释放了:  **********GGGG***
创世纪释放:**********
Aerosmith 进行了 408 次预订
面包已有 409 人预订
Carpenters 进行了 397 次预订
Delfonics 进行了 411 次预订
Eagles 进行了 555 次预订
Foghat 进行了 553 次预订
Genesis 进行了 564 次预订
Heart进行了545次预订
Isis 进行了 709 次预订
Journey 已完成 698 个预订
M 进行了 707 次预订
Ocean 已完成 696 次预订
议会提出 838 项保留意见
女王预订了 830 次
雷蒙斯进行了 857 次预订
SANTANA 已有834人预订
Toto 预订了 978 次
U2 进行了 949 次预订
Wings 已有 955 人预订
YES 进行了 968 次预订
总计:13861
```

额外学分测试

driver10.c 程序用于作业的额外学分部分。它很像 driver4.c,但是一个线程 Aerosmith 重复请求所有空间,而所有其他线程只请求宽度为 1 的空间。具有宽请求的线程可能无法获得它需要的空间,直到所有其他线程出口。这就是下面的示例输出中似乎发生的事情。

```
海洋分配:O**********
拉蒙斯分配:OR**********
心脏分配:ORH**********

 ...省略了很多输出...

Aerosmith 分配:AAAAAAAAAAAAAAAAAAA
Aerosmith 被释放:**********
```

Aerosmith 已有 1 人预订
面包已预订 996 人
Carpenters 已完成 989 次预订
Delfonics 完成了 977 次预订
Eagles 进行了 1001 次预订
Foghat 进行了 991 次预订
Genesis 进行了 982 次预订
Heart进行了1009次预订
Isis 进行了 980 次预订
Journey进行了1001次预订
M 预订了 998 次
Ocean 已完成 972 次预订
议会提出 1006 项保留意见
皇后乐队进行了 987 次预订
雷蒙斯进行了 944 次预订
SANTANA 已有994人预订
托托预订了 969 次
U2 进行了 1002 次预订
Wings 已有 988 人预订
YES 进行了 999 次预订
总计:18786

如果您实施额外信用,您应该看到它解决了这个问题。它会稍微降低整体性能,但即使是 Aerosmith 线程也应该偶尔获得其请求的空间。这是我使用额外积分解决方案运行 driver10.c 时得到的结果：

海洋分配（0）:O**********
皇后配(0): OQ**********
心脏分配（0）:OQH**********

...省略了很多输出...

Aerosmith 分配 (16): AAAAAAAAAAAAAAAAAA
Aerosmith 被释放:**********
Aerosmith 进行了 140 次预订
面包已预订 892 人
Carpenters 进行了 867 次预订
Delfonics 进行了 903 次预订
老鹰队进行了 901 次预订
Foghat 进行了 890 次预订
Genesis 进行了 888 次预订
Heart进行了892次预订
Isis 进行了 894 次预订
Journey进行了901次预订
M 预订了 883 次
Ocean 进行了 902 次预订
议会作出 910 项保留
女王预订了 891 次
雷蒙斯进行了 895 次预订
SANTANA 已有890人预订
Toto 预订了 877 次
U2 进行了 891 次预订
Wings 已有 897 人预订
YES 进行了 880 次预订
总数:17084

提交您的作品

完成后,在 Gradescope 上名为 HW4 Q2 的作业下,仅在 hall.c 中提交您的监视器实现。您不需要更改头文件或驱动程序文件,因此您不需要提交这些文件的副本。

3. (40 分) 这个问题旨在帮助你思考在你自己的多线程代码中避免死锁,以及不同死锁预防技术的性能权衡。我已经为一个繁忙的厨房编写了一个模拟器,里面有很多需要使用一套共享炊具的厨师。

   这有点像哲学家用餐问题,只是厨师使用的器具数量不限于两个。

   你会在课程网站 kitchen.c 上找到我对这个问题的实现。如果您查看我的源代码,您会看到我们有 10 位厨师和 8 台设备。每位厨师在开始烹饪之前都会锁定他或她需要的器具。这是通过互斥锁实现的。在烹饪之前,每个厨师都会在他们需要的每个设备的互斥锁上执行 pthread mutex lock()。

   他们迅速准备了一道菜。然后他们松开正在使用的电器上的锁,休息一会儿,然后再准备另一道菜。他们一遍又一遍地这样做 10 秒钟;然后他们都终止了。该程序计算准备了多少道菜,并在执行结束时报告每位厨师的总数和全球总数。这就像对性能的衡量。我们希望厨师能够准备尽可能多的菜肴。

   不幸的是,我的程序在当前状态下陷入僵局。我希望你使用三种不同的技术来修复这个程序。您不会更改每位厨师使用的器具,也不会更改他们在每次迭代中需要烹饪或休息的时间。您只需更改设备的锁定方式即可。

   (a) 首先,将原来的 kitchen.c 程序复制到 global.c 中。在 global.c 中,删除每个设备的互斥变量和单独锁定设备的代码。相反,让我们实施一次只有一名厨师可以做饭的政策。只需使用一个互斥体进行同步。让每位厨师在烹饪前锁定一个互斥体,并在烹饪完成后(即,在他们休息时)解锁。这样,我们就可以确定两个厨师不能同时使用同一个器具;一次只能做一个。只有一个互斥量,我们的解决方案应该没有潜在的死锁。

      该解决方案应该可行,但并不理想。它禁止厨师同时做饭,即使他们不需要任何相同的器具。当我尝试这种技术时,我在一台远程 Linux 机器上总共只准备了大约 300 到 320 道菜。执行之间存在一些差异。

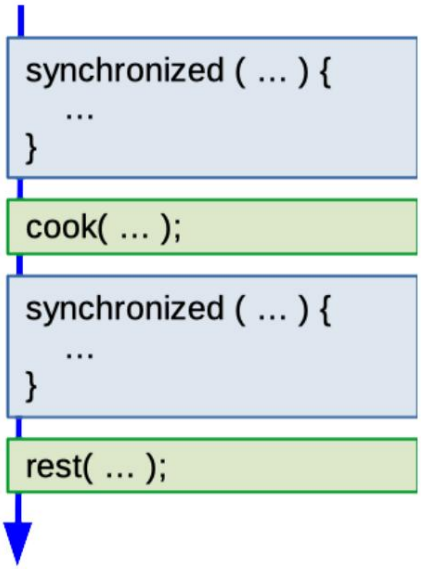
   (b) 与其强迫厨师一次只做一个,我们应该能够通过让他们以特定的顺序单独锁定电器来防止僵局。将原始 kitchen.c 程序复制到 ordered.c 并应用此死锁预防技术。为您认为可以最大化吞吐量的设备选择一个订单,一个可以使准备的菜肴总数最大化的订单。在尝试找到一个好的订单时,您可能想尝试一些不同的订单。或者,您可以考虑一下哪些电器可以早锁,哪些电器应该尽量晚锁。

      这个解决方案比 global.c 稍微复杂一点,但它的性能应该更好,因为它允许部分厨师同时做饭,前提是他们不需要任何相同的设备。当我实施这项技术时,我总共准备了大约 780 到 825 道菜(在 EOS 远程 Linux 机器上)。您至少应该尝试在您的解决方案中做到这一点。

   (c) 我们可以应用 no-hold-wait 技术,而不是应用 no-circular-wait 死锁预防技术;厨师们会立即分配所有他们需要的用具,只有在所有可用的情况下才会使用。将原始程序复制到 takeAll.c。替换代表的互斥体

每个设备的布尔变量的设备锁。我们将使用这些变量作为标志,跟上每个设备当前是否正在使用。

此外,将获取锁的代码替换为受互斥锁保护的新块。在这个街区内,您将检查特定厨师所需的所有设备是否可用。如果是,您将为所有这些设备设置标志,表明它们现在正在使用中。如果所需的设备并非全部可用,您将使用条件变量来等待设备被释放。使用互斥锁保护此代码可确保一次只有一个线程可以查看设备并更改其标志。



一旦厨师拥有他们需要的所有器具（将它们标记为正在使用）,他们就会做饭。由于互斥体块外的 cook 函数可以让多个厨师同时做饭,只要他们都有所需的设备即可。

烹饪后,厨师将输入另一个受同一互斥锁保护的代码块。在这个块中,他们会将他们正在使用的所有设备标记为可用,然后唤醒正在等待设备的任何其他厨师。要唤醒其余的厨师,您可以使用 pthread cond broadcast() 函数。我们可以不用广播来解决这个问题,但是使用这个函数会更容易。

释放所有电器后,厨师可以调用 rest()。
在性能方面,我的 takeAll.c 实现比我的其他两个解决方案都好一点。在 EOS Linux 机器上,我通常准备了大约 985 到 1010 道菜。

然而,准备很多菜的厨师和没准备那么多菜的厨师之间的差距通常更大。我会说这是该解决方案中饥饿风险的结果。没有厨师被完全阻止做饭,但有些人似乎没有经常获得他们需要的器具。

一旦所有三个无死锁程序都正常工作,分别运行它们并在名为 kitchen.txt 的文件中写一份报告。在你的报告中,你只需要（清楚地)报告你的三个程序分别准备了多少道菜。对于每个版本,还要报告任何厨师准备的最少菜肴数量（即对于准备最少菜肴的厨师,他们准备了多少？)和任何厨师准备的最大菜肴数量。这将有助于显示每个解决方案在厨师中的公平程度。

完成后,在 Gradescope 上名为 HW4 Q3 的作业下提交 global.c、ordered.c、takeAll.c 和 kitchen.txt 的电子副本。