操作系统,作业 3

这个作业包括两个编程问题。通常,我们将使用以下命令编译您的 C 程序
gcc 和以下命令行选项。一些程序需要如下所述的额外选项。

gcc -Wall -g -std=c99 -o 程序 program.c -lpthread

另外,请记住您的程序需要注释和一致的缩进,而且您必须
记录您的来源并从头开始自己编写至少一半的代码。

1. (12 pts) 假设你有四个信号量初始化如下:

w = 0; x = 0;
em = 1;西门子
= 1;

以下每个函数将由不同的线程同时运行:

```
无效线程1（）{
    获得（w）;打印（“一
    个”）;获取（y）;打
    印（“b”）;释放
    （y）;打印（“c”）;
    获取（y）;

}
```

```
无效线程2（）{
    释放（w）;打印
    （“d”）;释放（x）;
    释放（z）;打印
    （“e”）;获取（z）;
    打印（“f”）;

}
```

```
无效thread3（）{
    打印（“克”）;获取
    （z）;打印（“h”）;
    释放（z）;打印
    （“我”）;释放
    （w）;

}
```

```
无效 thread4() {
    获得（w）;打印
    （“j”）;释放（y）;
    获得（x）;打印
    （“k”）;释放（y）;
    打印（“我”）;

}
```

在以下字符串中,此代码可以打印出哪些字符串（如果不同的线程正在运行每个函数）?将您的答案写在一个文件中,如果可以打印,则在每个字符串上标记“是”,如果不能打印,则标记“否”。课程 Moodle 站点上发布了一个名为 orders.txt 的起始文件。

它有每个字符串的副本,顺序如下。只需在每个字符串后加上“是”或“否”即可表明您的答案。在 Gradescope 上名为 HW3 Q1 的作业下,将您的文件转换并提交为名为 hw3 p1.pdf 的 PDF 文件。

(a) gabcdhefijkl (b)
abcdefgjklhi (c)
abcdefghijkl (d)
gjabchdiefkl (e)
deagjbcfklhi (f)
jgdhiefkablc

2. (40 pts) 对于这道题,你要解决我们在作业中做的最大和问题 作为符号 1 和 2。输入和输出格式将与上次相同,但我们将使用不同的技术来读取输入和分发工作。这一次,您将在 C 中解决它,使用多个工作线程在具有最大和的整数序列中找到连续的非空子序列。然而,我们不是静态地划分工作(每个工人总是被分配相同的问题子任务),而是动态地划分工作(因此不同的工人可能从执行到执行得到不同的子任务)。您将使用匿名的 POSIX 信号量进行同步。我给你一个框架来帮助你开始,maxsum-sem.c,发布在课程主页上。

与之前的作业一样,将有一个命令行参数给出要使用的工作线程数和一个可选的“报告”参数,该参数将使工作人员报告他们从他们的工作部分中找到的最大总和。值序列仍将从标准输入中读取。

这个版本不是一开始就读取所有输入,而是先启动 worker,然后使用主线程读取值列表。最初,工人将无事可做。但是,当主线程读取输入时,它们将使用基于信号量的同步代码为自己分配越来越多的工作。如果当前没有工作要发出,空闲的工作人员将不得不等待主线程读入更多。这将一直持续到主线程完成读取所有输入。

然后,一旦所有工作完成,工作线程就可以终止,主线程可以通过比较工作线程从他们的部分工作中找到的最大总和来报告最终的最大总和。与我们以前的版本相比,如果读取输入真的很慢或者整个输入列表在程序启动时不可用(例如,如果输入正在下载或者它是其他一些结果),这将是一个很好的方法计算)。工作人员可以从到目前为止读入的列表部分开始,然后在读入时开始处理更多输入。

能够使用信号量将使更新所有 worker 的最终最大总和变得容易得多。我们可以只使用一个全局变量。工作人员完成后,他们可以将他们发现的最大总和与这个全局值进行比较,如果他们的发现更大,则更新它。由于每个工人都会修改这个变量,我们需要使用基于信号量的互斥代码来确保他们中的两个不会同时尝试修改它(这不太可能,但它也很容易防止).

待所有worker都完成后,主线程就可以只看这个全局变量的值来报告最终的最大总和。

这个程序的棘手部分是在工人完成他们之前的工作时将工作分配给他们。这实际上是生产者/消费者问题的一个例子。读取输入的主线程就像生产者,每个工人就像消费者,在可用时获得更多工作,并等待所有当前可用的工作都已经发出。

与此程序的先前版本一样,您可以决定如何拆分工作并将其分配给工人。您可以考虑为序列的每个元素分配不同的工作人员。该工作人员可以检查以该元素结尾的所有范围。这种方法的优点是,如果一个工人负责某个值xi ,那么如果xi已被读入,则意味着序列中的所有先前值都已被读入。您可以按自己的方式分配工作想要,但请记住,工作人员不能使用尚未读入的值。 ,

在我自己的解决方案中,我将大部分分配工作的逻辑放在一个名为 getWork() 的函数中。工人在闲置时调用此函数。如果还有更多工作要做,它会立即返回,或者,如果所有可用的工作都已经完成,该函数将阻塞工作线程,直到主线程读取了更多输入。我的 getWork() 函数返回工作人员负责的值的索引。您的解决方案可能会执行类似这样的操作,也可能会返回其他内容,具体取决于您如何划分工作。如果我们将线程视为生产者和消费者,则 getWork() 函数有点像消费者可能用来从缓冲区中获取下一项的代码。如果还有更多

要完成的工作,它得到它;如果没有,它会等待。

在我们读取所有输入并完成所有工作后,我的 getWork() 函数开始向工作人员返回一个特殊的标记值。这表明没有更多的工作要做。一旦完成所有工作,就好像主线程（生产者)开始将虚拟值（哨兵)放入缓冲区。当工作人员获得虚拟值时,他们知道是时候退出了。您不需要以这种方式实现您的解决方案,但这不是一个糟糕的思考方式,它可以帮助您简化代码。

一旦您的程序开始运行,您应该能够像下面这样运行它,让它处理来自先前作业的输入文件之一。和以前一样,当您在最大的输入文件上运行您的解决方案时,您应该获得加速,并且有多个 worker（前提是您有多个核心）。

在我自己的解决方案中,我在至少有四个内核的系统上与四个工作人员一起运行的速度几乎提高了 4 倍。您的性能结果可能有点嘈杂,具体取决于您运行的系统。为了更好地了解您的表现,您可能需要运行几次并检查中位数。

$ time ./maxsum-sem 1 < input-5.txt 最大总和:227655

真实的 0m15.086s
用户 0m15.099s
系统 0m0.003s

$ time ./maxsum-sem 4 < input-5.txt 最大总和:227655

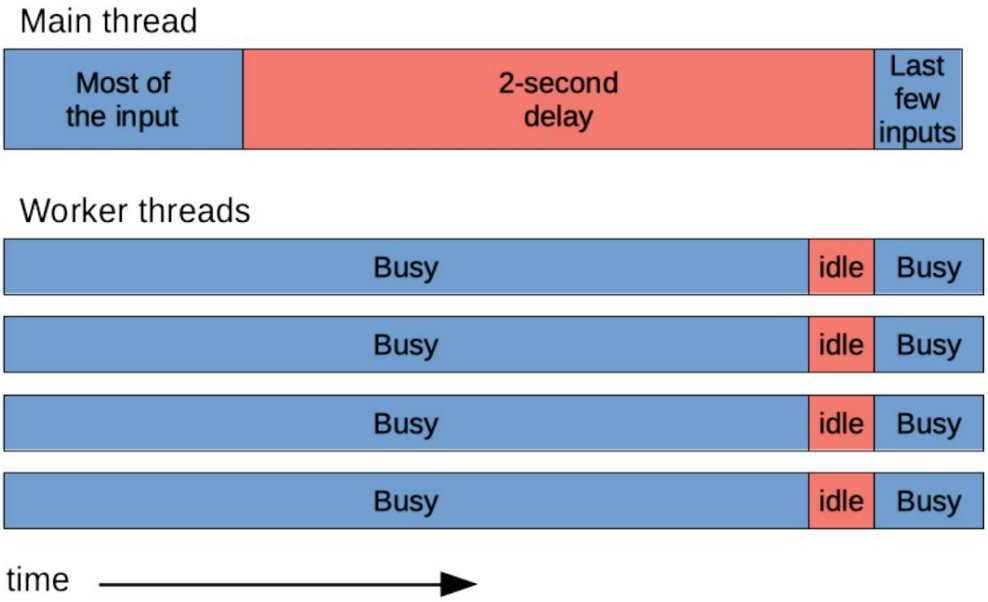
真实的 0m3.784s
用户 0m15.118s
系统 0m0.014s

即使程序首次启动时某些输入不存在,此解决方案也应该可以正常工作。要获得更有趣的结果,请尝试以下操作。在这里,我们在输入的交付中引入了一些延迟。下面 shell 命令中的注释告诉您我们正在做什么以及您应该期待什么。在下面的示例中,我在输入的传递中使用了两秒的延迟,因为我的程序在最大输入上花费了大约四秒。如果您的程序在此测试用例上速度不快,您可以尝试更长的延迟。

在这里,我们使用 sed（流编辑器)命令将输入分成两部分。我们在开始时一次性给出大部分输入（除最后 # 5000 个值外）,然后
我们在 2 秒延迟后给出剩余的 # 输入。大量的初始输入让 # 工人在开始时有很多工作要做。因此,他们可以在主线程等待 2 秒以获取
最后的 # 小批输入时保持忙碌。当最后 5000 个值到达时,工作人员可以 # 快速完成剩余的问题。 # 工作人员可能会一直很忙,并且
经过的时间大约与我们一次获得所有输入时的时间相同。

$ (sed -n 1,95000p input-5.txt; sleep 2; sed -n 95001,$p input-5.txt) |时间-p ./maxsum-sem 4
我最高:227655

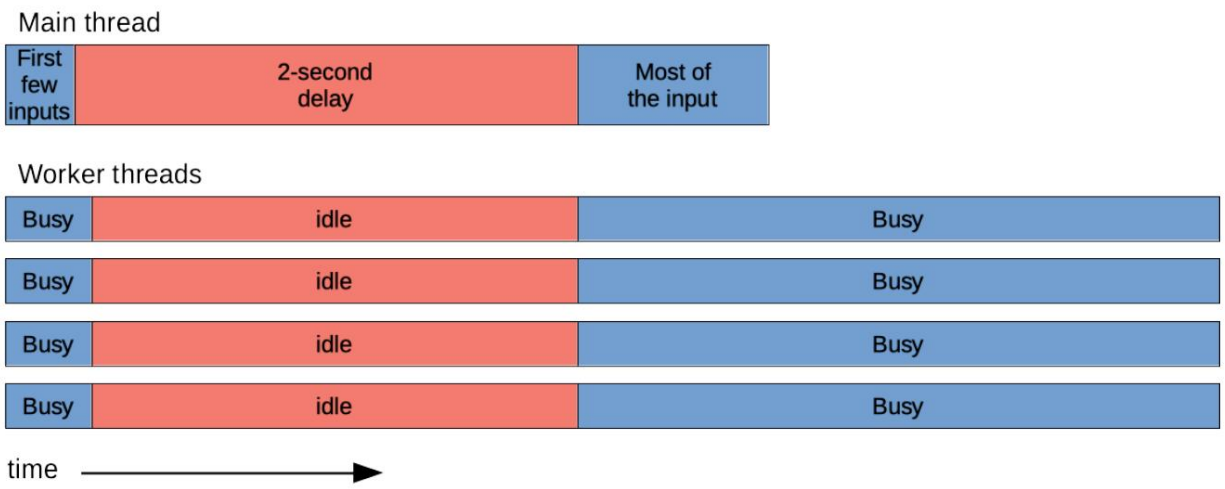
真实 3.78 用户
15.11 系统 0.02



下面,我们为程序提供相同的输入,但我们将其分成 5000 个元素的小部分,然后在 2 秒延迟后为剩余的 #95000 个元素。该程序应该能够 # 处理这个问题,但它应该比前面的例子慢大约 2 秒（测量经过的实时时间）。处理列表的前 #5000 个元素只需要很少的时间。 # 工人们会很快完成这个。然后,主线程和工作人员都将 # 必须等待 2 秒才能获得大部分工作,即 95000 # 剩余值。当他们得到输入文件的其余部分时,#仍然需要他们一段时间来处理它。因此,经过的 # 时间将增加大约 2 秒,因为没有足够的工作 # 来让工作人员忙于接近开始的 2 秒延迟。

$ (sed -n 1,5000p input-5.txt; sleep 2; sed -n 5001,$p input-5.txt) | time -p ./maxsum-sem 4 最大总和:227655

真实 5.77
用户 15.11
系统 0.00



完成后,在 Gradescope 上名为 HW3 Q2 的作业下提交源文件 maxsum-sem.c。

3. (40 分)我们的家庭作业 2 中的共享内存 reset.c / lightsout.c 程序存在竞争条件。如果两个用户同时运行 lightsout.c 程序,他们可能会同时尝试修改 GameState 结构,可能会使它处于不一致的状态。当然,如果我们试图证明这种竞争条件,我们将很难手工完成。我们不太可能同时运行 lightsout.c 的两个副本,它们之间的距离足够近,以至于它们实际上会相互干扰。

对于这个问题,你要做两件重要的事情。您要将互斥代码添加到 lightsout.c (以及一点点到 reset.c)以消除竞争条件的可能性。您还将添加一个测试界面,以非常非常快地模拟一遍又一遍地运行程序。您将在互斥代码周围包装条件编译指令,以便您可以在编译时轻松启用或禁用互斥。通过测试界面和用于打开和关闭相互包含代码的编译时选项,我们应该能够看到竞争条件实际上导致错误(希望如此),并验证启用相互排除代码可以防止错误。

使用 POSIX 命名信号量提供互斥。使用命名信号量将使 lightsout.c 程序的多个副本访问同一个信号量。当运行 lightsout.c 的进程需要使用共享的 GameState 结构时,它将获取信号量以确保没有其他进程同时使用该数据结构。完成后,它将释放信号量,以便其他进程可以访问该结构。

reset.c 程序现在需要创建共享内存段和命名信号量(初始化为 1)。共享内存和信号量将在 lightsout.c 的多次执行中继续存在。在 lightsout.c 中,在 reset.c 创建它们之后,您应该能够通过 shmget() 和 sem open() 获取共享内存和信号量。

我们都需要为我们的信号量使用不同的名称,以确保属于一个学生的程序不会干扰其他学生的程序,即使他们碰巧在同一台主机上运行。
使用 /unityid-lightsout-lock 作为信号量的名称(其中 unityid 是您的实际统一 ID)。

除了之前作业中的移动、撤消和报告命令外,您的程序还需要支持一个新的命令来访问测试界面:

- ./lightsout 测试 nrc

像这样运行 lightsout.c 将尽可能快地执行移动命令 n 次,在 r 行,c 列移动。测试命令不会打印任何东西;那只会减慢速度。

我们希望它能够尽快运行 GameState 数据结构,以帮助我们检查可能的竞争条件。

要使用 test 命令,我们将同时运行程序的两个副本,每个副本运行一个具有较大 n 值的测试。如果没有互斥代码来保护对共享内存的访问,两个试图同时快速修改 GameState 的程序可能会使它处于不一致的状态。我们将能够通过在编译期间启用或禁用互斥代码来检查这一点。

对于之前的作业,您可能已经使用 main() 中的所有代码编写了程序。对于这个任务,我们需要稍微组织一下代码,这样互斥代码就可以测试了。

您的每个命令都将在如下函数中实现。指向共享内存 GameState 的指针被传递给每个函数,并且（对于移动和撤消)返回值表明函数是否成功,因此调用者可以在需要时打印错误消息。

```
// 在给定的行、列位置移动,如果成功则返回 true。 bool move(GameState *state, int r, int c);


// 撤消最近的移动,如果成功则返回 true。布尔撤消（游戏状态*状态）；

//打印棋盘的当前状态。无效报告（游戏状态*状态）；
```

解析命令行参数后,您的主函数将调用这些函数之一来访问 GameState。在这些函数中的每一个中,您将在开始时获取信号量,然后在返回之前释放信号量。如果其中一个函数从多个地方返回,请确保在通过任何这些路径返回之前释放信号量。不要只是将信号量锁定在 main 中,然后在 main 退出之前将其解锁。这将阻止测试功能正常工作。

在 test 命令上,您的程序将只调用如下函数。检查板位置后,它只是一遍又一遍地调用 move() 函数。你可以看到,test() 本身并没有获取信号量;这是在它调用的 move() 函数中完成的。

```
// 测试接口,用于快速反复执行给定的移动。 bool test( GameState *state, int n, int r, int c ) {
    // 确保行/列有效。如果 ( r < 0 || r >= GRID_SIZE || c < 0 || c >=
    GRID_SIZE ) 返回 false；

    // 多次执行相同的操作。 for ( int i = 0; i < n; i++ ) move( state, r,
    c );

    返回真；
}
```

您获取和释放信号量的所有调用都将包含在如下所示的条件编译代码中。这将使我们通过定义预处理器常量来禁用互斥，

不安全,在 gcc 线上。您需要使用这个宏名称 UNSAFE,就像这样。这是我们将如何测试您的程序的一部分,通过使用此符号来启用或禁用您的互斥代码。

```
#ifndef 不安全
    sem_wait( mySemaphore ); #万
—
```

一旦您的程序开始运行,您应该能够像在之前的作业中那样运行它。
您还应该能够在启用和不启用互斥代码的情况下测试您的程序。当然,你的程序在互斥的情况下应该工作得很好,但如果你在没有互斥的情况下同时运行两个测试并且 n 的值足够大,它可能（可能,希望)失败。

竞争条件更有可能出现在多核系统上。对于这个程序,您不太可能在单核机器上看到它们（但是,也许）。

以下是您的程序的可能执行。作业 2 中的所有命令应该仍然有效,但在这里我试图显示测试界面。我包含了一些 shell 注释来解释我在做什么。

```
# 构建重置程序 gcc -g -Wall -std=c99
-D_XOPEN_SOURCE reset.c -o reset -lpthread

# 构建启用互斥的 lightsout 程序。 gcc -g -Wall -std=c99 -D_XOPEN_SOURCE lightsout.c -o
lightsout -lpthread

# 重置为空白板。
 ./重置board-2.txt

# 同时运行测试的两个副本。第一个在后台运行 #（使用 &）,另一个紧接着 # 在前台运行。这将需要几秒钟的时间
来运行,#n 的值足够大,以至于两个副本当时都应该有机会#运行。第一个命令在后台运行,# 你最终会从 shell 得到
一个报告,表明后台命令已经完成。没关系; shell 对在后台启动的任何命令执行此操作。 ./lightsout 测试 5000000
2 1 & ./lightsout 测试 5000000 2 1




# 运行两个测试副本后,我们可以检查板的最终状态。重复同样的动作 # 次应该给我们留下一个空白板,就像我们开
始时一样：./lightsout report


 .....
 .....
 .....
 .....
 .....

# 现在,再次编译,禁用互斥。 gcc -g -Wall -std=c99 -DUNSAFE -D_XOPEN_SOURCE
lightsout.c -o lightsout -lpthread

# 运行相同的测试并编译出互斥代码。它现在应该运行得更快,因为丢失的开销减少了
```

同步代码。但是,存在竞争条件。 # 甚至有可能运行这个测试会使你的程序崩溃,这取决于你如何实现
你的程序。 ./lightsout 测试 5000000 2 1 & ./lightsout 测试 5000000 2 1

现在,看看棋盘上发生了什么。我有几盏灯是 # 亮着的。这是因为多个并发客户端之间的竞争条件。如果
我重新设置并再次运行,我可能会得到 # 不同的结果。

./熄灯报告
.....
.*...
..*..
.....
.....

您的程序将在每次运行时使用相同的共享内存和命名信号量。如果其中任何一个处于未知状态,您可能需要删除它们,然后使用 reset.c 以已知状态重新创建它们。与之前的分配一样,您可以通过运行 ipcs 查看您的共享内存段,您可以通过输入 ipcrm -m 以及 ipcs 报告的共享内存 ID 来删除它。在 Linux 机器上,您应该能够通过输入 ls /dev/shm 查看您的信号量。

您可以通过使用 rm 命令删除 /dev/shm 下的文件来删除它,就像删除任何其他文件一样。

完成后,在 Gradescope 上名为 HW3 Q3 的作业下提交您的三个源文件。
这将包括 common.h、reset.c 和 lightsout.c 的源代码。