華中科技大學

研究生课程报告

哲学家就餐问题实验报告

学　　号　M202470678

姓　　名　梅硕

专　　业　机械工程

课程指导老师　周健

院（系、所）　机械科学与工程学院

2024 年 12 月 26 日

目　　录

1 问题描述

哲学家就餐问题（The Dining Philosophers Problem）是一个经典的计算机科学问题，最初由计算机科学家 Edsger Dijkstra 提出，用于研究并发编程中的同步问题。它的意义在于，通过一个简单但抽象的场景，展示了多线程或多进程环境中资源竞争与死锁问题的核心挑战，并为解决这些问题提供启发。在问题中，五位哲学家围坐在圆桌周围，他们在思考和进餐之间交替进行。每位哲学家需要两只叉子才能进餐，而每只叉子都位于两位哲学家之间。如果哲学家同时尝试拿取左右两只叉子，就可能引发以下问题：

- **死锁**：所有哲学家都拿起了右手的叉子，等待左边的叉子，导致所有人无法继续操作。
- **资源饥饿**：某些哲学家可能永远无法获得两只叉子，因为其他哲学家始终占用。
- **并发性低**：哲学家可能采取不必要的等待，降低资源的利用效率。

本实验中，通过哲学家就餐这一经典问题进行研究，了解并发计算中多线程竞争时产生的上述问题，并探索解决他们的方案。

2 “不太聪明”的哲学家

本章节的任务是模拟线程并发中产生的产生的死锁问题，因此本章节中哲学家可能表现得不那么“聪明”。

他们中的所有人均在饥饿时首先尝试拿起左手的叉子，然后拿起右手的叉子，如果拿起叉子的过程失败，那么他会一直等待，直到成功。同时，他们表现出对旁人的漠不关心，只考虑自己的事情，因此，当他们在就餐时，如果身旁的哲学家准备拿起在他们手中的叉子，他们不予理会。上述哲学家行为可以用伪代码表示如算法 2.1 ：

算法 2.1. “不太聪明”的哲学家

此版本的“筷子”使用互斥锁表示即可，当某个哲学家（线程）占用筷子时，

Algorithm 2.1 stupid philosopher

Output: State of philosopher

```
1:  Initialize philosopher's state as thinking, random choose thingkingtime
2:  while true do
3:      if state is hungry then
4:          repeat   try get left chopstick
5:          until    sccess
6:          repeat   try get right chopstick
7:          until    success
8:          change state to eating
9:      end if
10:     if state is eating then occupy chopsticks for eatingtime random choose eating-
        time
11:         change state to thinking
12:     end if
13:     if state is thinking then do nothing for thinkingtime
14:         change state to hungry
15:     end if
16: end while
17: return
```

该线程获得代表筷子的互斥锁。哲学家尝试拿起筷子并一直等待的过程即是上锁阻塞，成功后即是占用互斥锁。

显然，这些不太聪明并对他人漠不关心的哲学家很快进入了每个人都拿起左手筷子的状态，并进入了永无止境的饥饿中（即死锁）。

3 “比较聪明”的哲学家

本章节的任务是解决模拟线程并发中产生的产生的死锁问题，解决的方法是经典的资源层次解决方案。

编号非 0 的哲学家行为与第 2 章所述相同。然而其中编号 0 的哲学家，他首先尝试拿起自己右手的筷子。该哲学家行为可以用伪代码表示如算法 3.1 ：

算法 3.1. “比较聪明”的哲学家

Algorithm 3.1 philosopher

Output: State of philosopher

```
1:  Initialize philosopher's state as thinking, random choose thinkingtime
2:  while true do
3:      if state is hungry then
4:          repeat   try get right chopstick
5:          until    success
6:          repeat   try get left chopstick
7:          until    sccess
8:      end if
9:      if state is eating then
10:         occupy chopsticks for eatingtime
11:         random choose eatingtime
12:     end if
13:     if state is thinking then do nothing for thinkingtime
14:     end if
15: end while
16: return
```

算法 3.1 几乎是将算法 2.1 重复了一遍，仅在 4~7 行有顺序上的调整。然而这样微小的调整，却确实保证了程序不会陷入死锁，因为哲学家 0 不能够与哲学家 4 同时拿起第一根需要拿起的筷子，而陷入无限的等待。

然而，这些“比较聪明”的哲学家虽然不再陷入死锁状态，但出现了新的问题：其中几位哲学家陷入了长久的饥饿。例如，在 0 号哲学家进食时，1~3 号哲学家拿起了自己左手的筷子，4 号哲学家未拿起筷子。在这样的情况产生后，如果 0 号哲学家的动作远远快于 4 号和 1 号，那么 4 号哲学家将始终无法拿起筷子，而 1~3 号哲学家也不会产生任何进展。

具体的程序实现上，除了 0 号线程占用锁的顺序外，两个版本没有什么不同的。

4 “聪明得多”的哲学家

本章节的任务是解决模拟线程并发中产生的产生的死锁和饥饿问题，解决的方法是 Chandy/Misra solution。通过恰当的初始情况设定，该方法避免了死锁，对刚刚吃过的线程造成不利，从而避免了饥饿情况，使得任务能够一直推进。在

进一步之前，我们需要对问题进行一定的限制说明。在本次实验中，为了为了使得饥饿现象进一步解决，在原方法中加入抢占式的调度。

由于本问题实际上在研究多线程中的线程调度，那么处理过程中需要引入一些与实际更为接近的限制条件。

- 哲学家每段就餐时间对应于线程的一个任务需要完成的时间，在该任务完成前，线程不能进行下一个任务。如果任务被中断，下一次获得资源后将继续进行。也就是说，**哲学家如果就餐时被打断而退出，他将进入饥饿状态而不是思考状态**。
- **哲学家的每次进餐有最小时间限制**，这表示了进程执行任务的资源准备，也是线程调度时线程切换的开销和负载均衡的权衡结果。

这个版本下，筷子将总是被哲学家持有并在两个特定的哲学家之间传递。一个持有筷子且使用过筷子（此时筷子是脏的）的哲学家在他的邻桌哲学家需要使用筷子时，将他的筷子擦干净交给请求者，否则始终保留筷子即使是在思考的时候。上述哲学家行为可以用伪代码表示如算法 4.1 。

算法 4.1 该方法的具体操作上比算法 2.1 和算法 3.1 稍微复杂一些。首先，我们需要对表示筷子的方法进行一定的修改，单使用互斥量的方法，不足以使得哲学家线程能够相互通信了解对方是否发出请求。因此，我们的“筷子类”，需要包含有一个互斥锁表示筷子的持有，一个共享锁（使用互斥锁亦可）控制对筷子请求信息的修改，同时需要一个布尔型变量表示筷子是否已经使用，一个整形变量表示该筷子的请求者。

由于这里引入了一个新的锁，控制对筷子的请求，首先应当说明这个锁是不会锁死的，因为该锁仅需要在修改单个筷子的请求时修改，修改完成即可释放，并不存在需要在占用时等待其他资源的情况，因此没有死锁条件，不会死锁。此外在具体的实现细节上，我们需要注意的是，当发起对某个筷子的请求时，如果此时请求变量被邻居设置过，那么尝试标记另外的筷子，直到两个筷子均被标记。标记完成后再进行拾起筷子亦即占有资源的动作，当拾起筷子时，将自己的请求标记清除。

算法 4.1. “比较聪明”的哲学家（加入抢占机制的 Chandy/Misra solution）

Algorithm 4.1 smart philosopher

Output: State of philosopher

```
1: Initialize philosopher's state as thinking, random choose thingkingtime
2: Initialize all chopsticks are used, give chopstick to philosopher with low ID
3: while true do
4:     if state is hungry then
5:         request left chopstick
6:         request right chopstick
7:         pick left chopstick
8:         pick right chopstick
9:     end if
10:    if state is eating then
11:        while eatingtime ≠ 0 do
12:            if neighbor make a request then
13:                mark chopsticks as clean
14:                release corresponding chopstick
15:                break
16:            end if
17:            occupy chopsticks mini-eatingtime
18:            mark chopsticks as used
19:            eatingtime = eatingtime − mini − eatingtime
20:        end while
21:    end if
22:    if state is thinking then
23:        while looptime ≤ thinkingtime do
24:            if neighbor make a request then
25:                mark chopsticks as clean
26:                release corresponding chopstick
27:                break
28:            end if
29:            update looptime
30:        end while
31:    end if
32: end while
33: return
```

5 调试记录

5.1 C++ 的死锁预防机制

依据 cppreference 中的描述，鼓励互斥量上锁底层实现中加入死锁的检测并抛出系统异常 resource_deadlock_would_occur[1]，这一特性导致 version1 在不进行特殊处理的情况下，会出现报错并且不会正常停止，因此程序中使用 try 语句捕捉错误并使用 exit 语句终止程序。

5.2 Windows11 Clang19 多线程 bug

Windows11 使用 Clang19-msvc 作为 C++ 标准库，一定条件下对锁进行释放时会出现两次释放，并报错无法释放未持有的锁。相同代码换用换用 GNU gcc 编译没有问题。（只能说在 Windows 下不用 VS 是个错误，查一个并不存在于自己代码中的 bug 实在是难受）

参考文献

[1] CPPREFERENCE. cpp mutex lock[EB/OL]. .

https://en.cppreference.com/w/cpp/thread/mutex/lock.