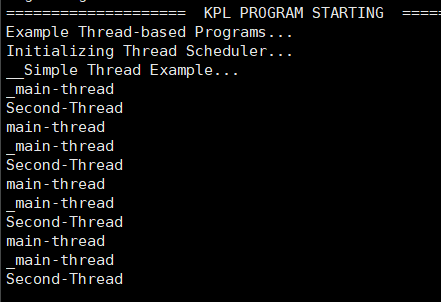


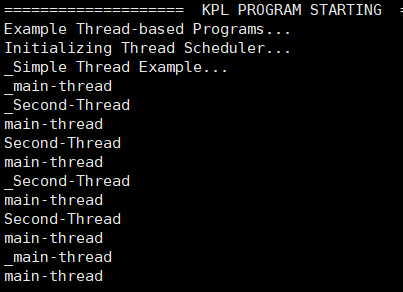
取消注释’\_’后，可以看到在什么时候程序运行了中断函数。

效果如下：

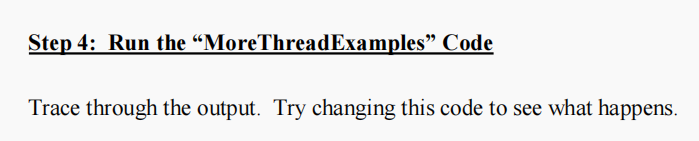


可以看到操作系统在定时执行着中断，在main-thread出现中断后，下一次执行的变成了second-thread这种机制避免了一个程序连续执行。

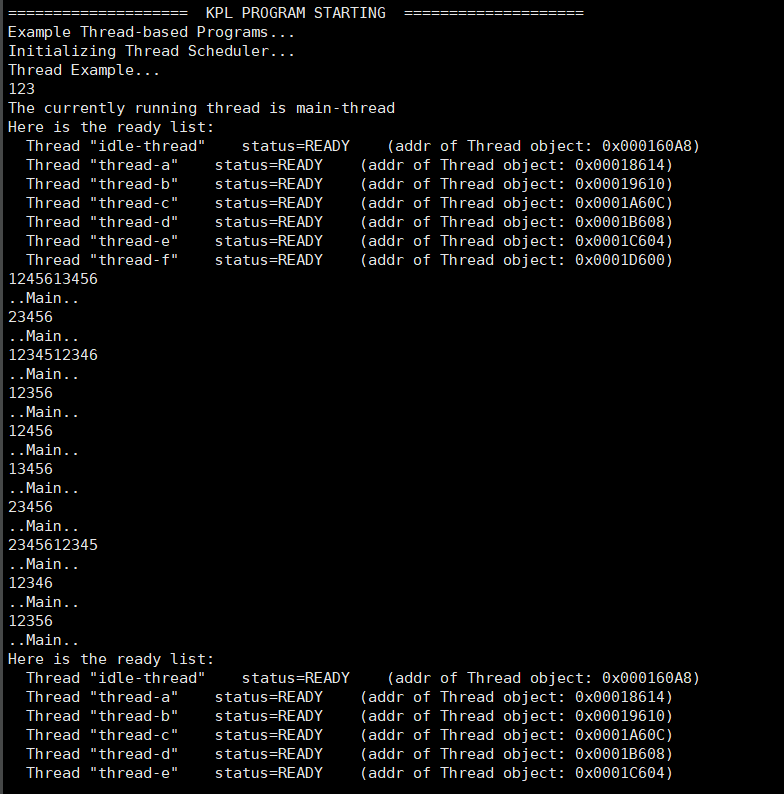
如果注释掉其中的yield函数，可以发现：

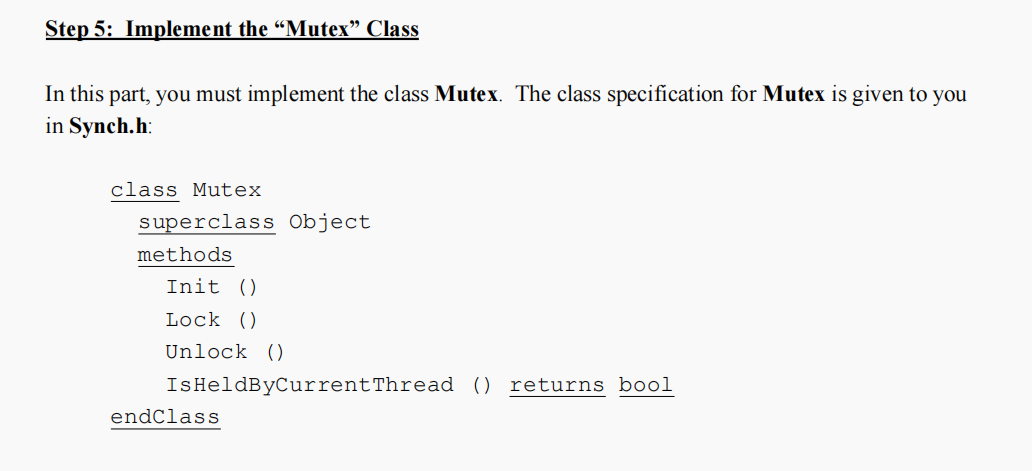


此时已经无法调度，起一个切换线程的功能。因为TimerInterruptHandler中的yield已经被注释掉了。

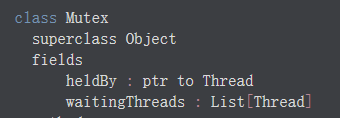


运行后结果如下：

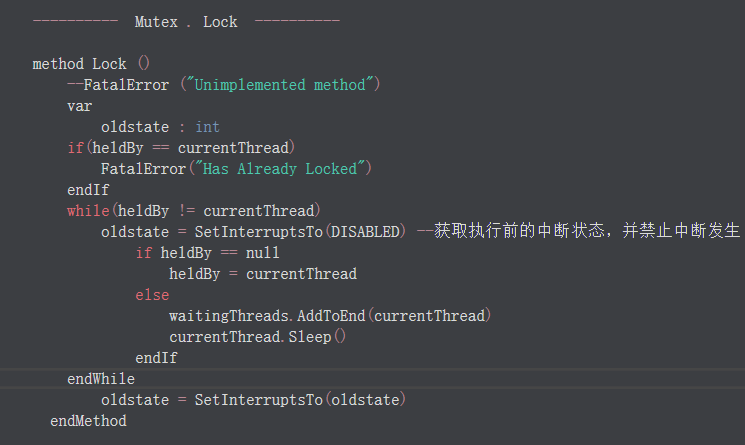




首先在Sync.h中添加需要的变量



而后补全Mutex关于上锁和解锁的函数：



首先，它检查是否已经有一个线程持有了这个锁。如果heldBy等于currentThread，表示当前线程已经持有锁，那么会引发致命错误，因为一个线程不应该重复获得同一个锁。

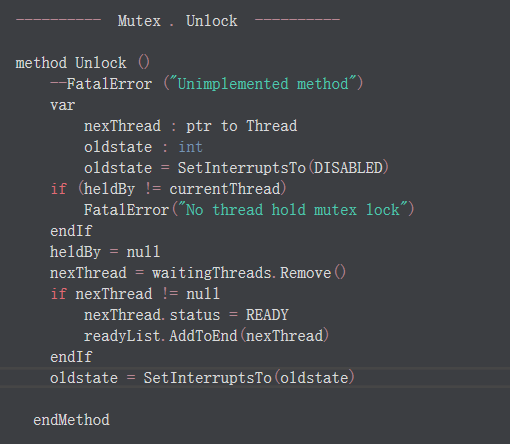
如果没有线程持有这个锁，它会尝试获取锁。进入一个while循环，该循环会一直运行，直到当前线程成功获取锁。

在while循环中，首先通过SetInterruptsTo(DISABLED)禁止中断的发生。这是为了确保在获取锁的过程中，其他中断或线程的干扰不会导致问题。获取禁止中断的状态（oldstate）是为了在最后将中断状态还原。

接下来，它检查heldBy是否为null，如果是，则表示锁当前没有被其他线程持有。在这种情况下，当前线程可以安全地获得锁。因此，heldBy会被设置为currentThread，表示当前线程持有了锁。

如果heldBy不是null，则表示锁已经被其他线程持有。在这种情况下，当前线程将被添加到等待线程列表（waitingThreads）中，并通过currentThread.Sleep()来阻塞当前线程。这使得当前线程在等待其他线程释放锁时不会消耗CPU资源。

循环将一直执行，直到当前线程成功获取锁，这通常发生在其他线程释放锁时，被阻塞的线程被唤醒，然后它再次尝试获取锁。获取锁后，中断状态将被还原为之前的状态（oldstate）。



首先，它会通过SetInterruptsTo(DISABLED)禁止中断的发生。这是为了确保在解锁的过程中，其他中断或线程的干扰不会导致问题。获取禁止中断的状态（oldstate）是为了在最后将中断状态还原。

接下来，它检查当前线程是否持有这个锁。如果heldBy不等于currentThread，表示当前线程没有持有锁，它会引发致命错误，因为只有持有锁的线程才能释放它。

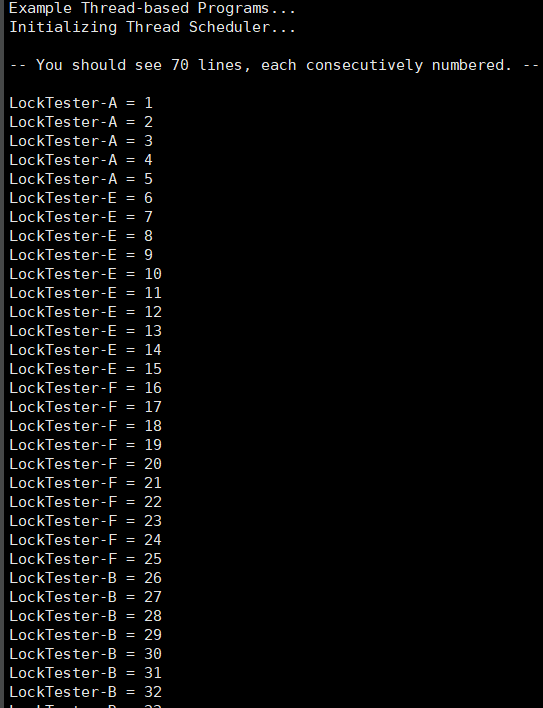
如果当前线程持有锁，它将heldBy设置为null，表示锁现在没有被任何线程持有。

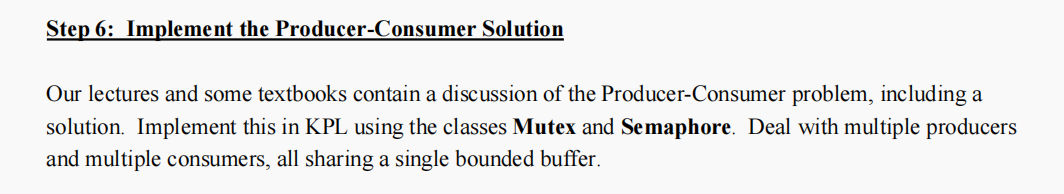
接下来，它尝试从等待线程列表（waitingThreads）中获取一个等待的线程，这通常是由于其他线程之前尝试获取锁时被阻塞。如果有等待的线程（nexThread != null），则将它从等待线程列表中移除。

如果获取到了等待的线程，它将该线程的状态设置为READY，表示该线程已准备好运行，然后将该线程添加到就绪线程列表（readyList）中。这样，该线程将有机会获得锁并执行。

最后，它将中断状态还原为之前的状态（oldstate），以确保中断处理机制在函数执行结束后能够继续运行。

最终，运行TextMutex验证结果：





分析问题：

任何时刻只能有一个线程操作缓冲区，说明操作缓冲区是临界代码，需要互斥；

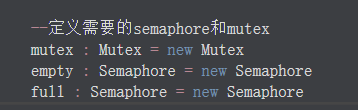
缓冲区空时，消费者必须等待生产者生成数据；缓冲区满时，生产者必须等待消费者取出数据。说明生产者和消费者需要同步。

那么我们需要一个互斥锁和两个信号量，分别是：

互斥锁 mutex：用于互斥访问缓冲区；

资源信号量 fullBuffers：用于消费者询问缓冲区是否有数据，有数据则读取数据，初始化值为 0（表明缓冲区一开始为空）；

资源信号量 emptyBuffers：用于生产者询问缓冲区是否有空位，有空位则生成数据，初始化值为 n （缓冲区大小）；



代码实现：

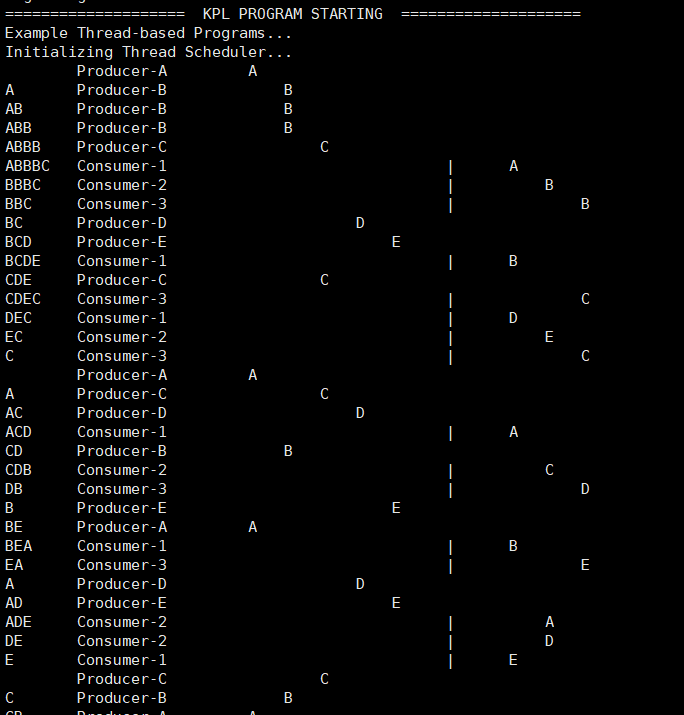
1. function Producer (myId: **int**)
2. var
3. i: **int**
4. c: **char** = intToChar ('A' + myId - 1)
5. **for** i = 1 to 5
6. -- Perform synchroniztion...
7. empty.Down()
8. mutex.Lock()
9. -- Add c to the buffer
10. buffer [bufferNextIn] = c
11. bufferNextIn = (bufferNextIn + 1) % BUFFER\_SIZE
12. bufferSize = bufferSize + 1
14. -- Print a line showing the state
15. PrintBuffer (c)
17. -- Perform synchronization...
18. mutex.Unlock()
19. full.Up()
20. endFor
21. endFunction
23. function Consumer (myId: **int**)
24. var
25. c: **char**
26. **while** **true**
27. -- Perform synchroniztion...
28. full.Down()
29. mutex.Lock()
30. -- Remove next character from the buffer
31. c = buffer [bufferNextOut]
32. bufferNextOut = (bufferNextOut + 1) % BUFFER\_SIZE
33. bufferSize = bufferSize - 1
35. -- Print a line showing the state
36. PrintBuffer (c)
38. -- Perform synchronization...
39. mutex.Unlock()
40. empty.Up()
41. endWhile
42. endFunction

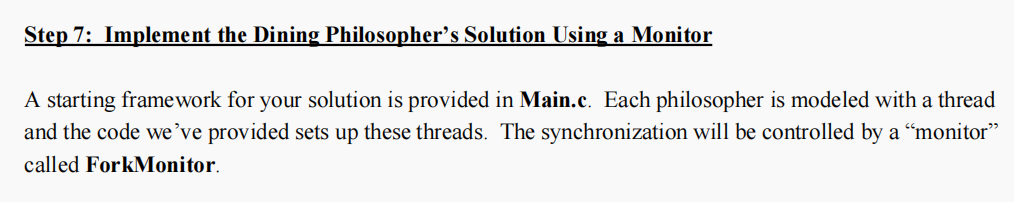
如果消费者线程一开始执行 P(fullBuffers)，由于信号量 fullBuffers 初始值为 0，则此时 fullBuffers 的值从 0 变为 -1，说明缓冲区里没有数据，消费者只能等待。

接着，轮到生产者执行 Down(emptyBuffers)，表示减少 1 个空槽，如果当前没有其他生产者线程在临界区执行代码，那么该生产者线程就可以把数据放到缓冲区，放完后，执行 Up(fullBuffers) ，信号量 fullBuffers 从 -1 变成 0，表明有「消费者」线程正在阻塞等待数据，于是阻塞等待的消费者线程会被唤醒。

消费者线程被唤醒后，如果此时没有其他消费者线程在读数据，那么就可以直接进入临界区，从缓冲区读取数据。最后，离开临界区后，把空槽的个数 + 1。

最终运行效果如下：





使用信号量和互斥锁解决问题，思路为用一个数组 state 来记录每一位哲学家的三个状态，分别是在进餐状态、思考状态、饥饿状态（正在试图拿叉子）。

那么，一个哲学家只有在两个邻居都没有进餐时，才可以进入进餐状态。

第 i 个哲学家的左邻右舍，则由宏 LEFT 和 RIGHT 定义：

LEFT : ( i + 5 - 1 ) % 5

RIGHT : ( i + 1 ) % 5

比如 i 为 2，则 LEFT 为 1，RIGHT 为 3。

代码如下：

先定义变量和函数：加入五个信号量，一个互斥锁和test函数

1. **class** ForkMonitor
2. superclass Object
3. fields
4. status: array [5] of **int**             -- For each philosopher: HUNGRY, EATING, or THINKING
5. s : array [5] of Semaphore
6. monMutex : Semaphore
7. methods
8. Init ()
9. test (p: **int**)
10. PickupForks (p: **int**)
11. PutDownForks (p: **int**)
12. PrintAllStatus ()
13. endClass

函数如下：

1. method test (p: **int**)
2. **if** status[p] == HUNGRY && status[(p+5-1)%5]!=EATING && status[(p+1)%5]!=EATING
3. status[p] = EATING
4. s[p].Up()
5. endIf
6. self.PrintAllStatus()
7. endMethod

10. method PickupForks (p: **int**)
11. -- This method is called when philosopher 'p' is wants to eat.
12. -- ...unimplemented...
13. monMutex.Down()
14. status[p] = HUNGRY
15. self.PrintAllStatus()
16. self.test(p)
17. monMutex.Up()
18. s[p].Down()
20. endMethod
22. method PutDownForks (p: **int**)
23. -- This method is called when the philosopher 'p' is done eating.
24. -- ...unimplemented...
25. monMutex.Down()
26. status[p] = THINKING
27. self.PrintAllStatus()
28. self.test((p + 5 - 1) % 5)
29. self.test((p + 1) % 5)
30. monMutex.Up()
31. endMethod

最终结果如下：

