Review of lab2

step 1

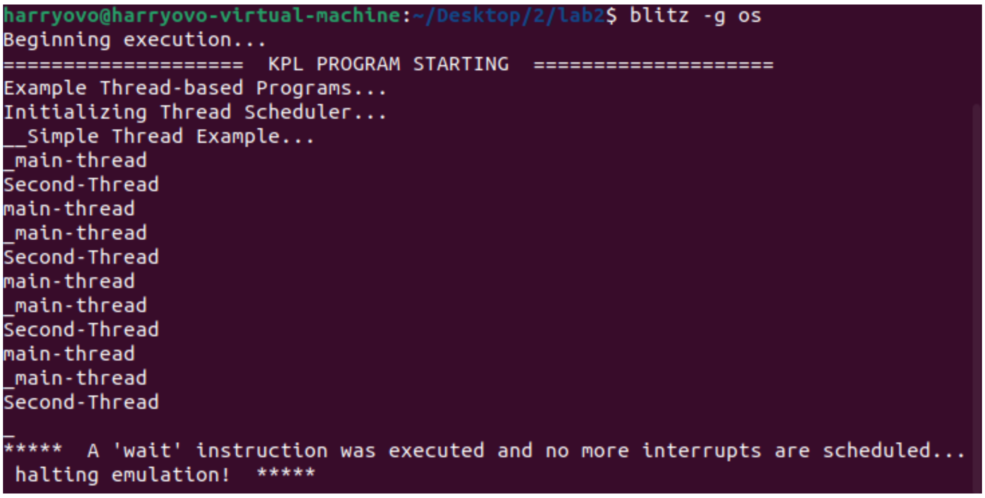
略。

step 2

代码主要组成如下所述：

1. 在内核中启动时分配对象，此后不能再分配。
2. Runtime.s 包含用汇编语言编写的例程，包含程序入口与中断向量
3. 例程执行时调用main.c的main函数
4. List实现了操作系统的链表
5. List主要的键类名为Thread.它是线程的核心，包含线程调度（状态转换）和时间切片相关函数
   　Thread中的FatalError函数就是进程终止函数，打印可能的错误。随后进入blitz的命令行模式，可能需要输入st查看函数和方法处于活动状态。
   　Thread中的SetInterruptsTo用于改变CPU的中断标志（开中断或关中断），返回一个变量提示现在是什么状态。
6. 线程五种状态：JUST_CREATED、READY、RUNNING、BLOCKED和UNUSED
7. 每个线程都有堆栈（系统堆栈），堆栈放置在systemStack字段中的thread对象中。该堆栈只用于内核例程，而之后的实验中用户程序在各自的虚拟空间中有属于自己的堆栈。
8. Thread对象还存储CPU的状态，在线程切换时所有数据保存在thread对象，这些字段（regs和stackTop）由名为Switch的代码例程使用。（PCB？）
9. Thread对象还存储指向函数的指针(initialFunction)和函数的实参(initialArgument字段)，该指针指向现成的main函数，不同的线程执行不同的main函数。
10. initialArgument字段表示线程的编号。
11. 使用Thread对象中的Fork方法启动一个初始化的新线程，这会使他进入ready状态（加入readyList，这是一个全局变量）。
12. currentThread：全局变量，表示状态为RUNNING的唯一一个线程。
13. Thread对象中的Yield方法：只能在当前运行的线程上调用，作用为切换到其他线程。
    ① 禁用中断：防止干扰。
    ② 在readylist找下一个线程（若没有其他线程，则yield实际上是nop）
    ③ 使当前线程变为READY
    ④ 将当前线程添加到readyList尾部
    ⑤ 运行下一个线程(Run方法)
14. Thread对象中的Run方法：先检查线程堆栈溢出，再调用Switch方法执行线程切换。
15. Thread对象中的Switch方法：返回发生在另一个进程的另一个函数。
16. 其他类与方法：下面会提及。

step 3

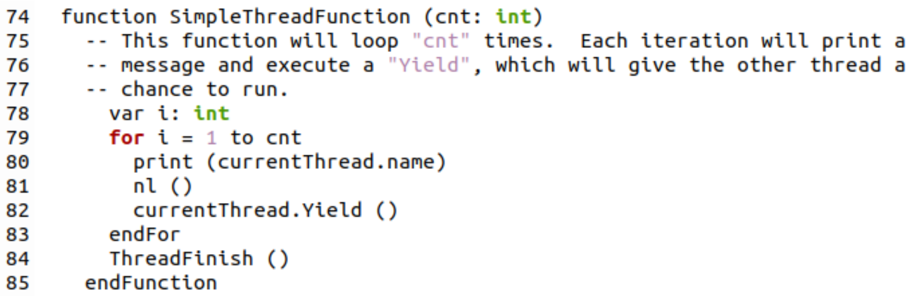


```
harryovo@harryovo-virtual-machine:~/Desktop/2/lab2$ blitz -g os
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Example Thread-based Programs...
Initializing Thread Scheduler...
__Simple Thread Example...
_main-thread
Second-Thread
main-thread
_main-thread
Second-Thread
main-thread
_main-thread
Second-Thread
main-thread
_main-thread
Second-Thread
_
*****  A 'wait' instruction was executed and no more interrupts are scheduled...
 halting emulation!  *****
```

解释：

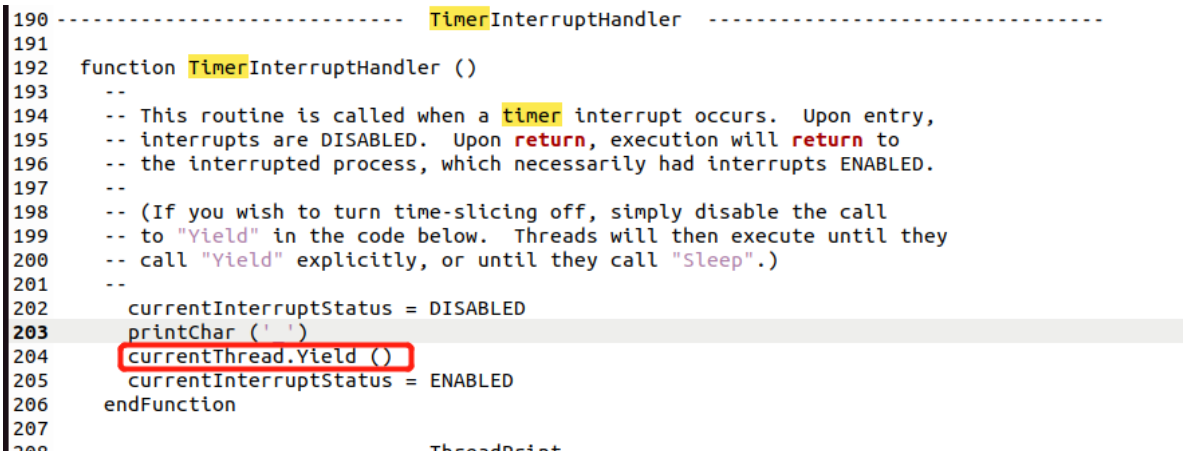
当进程中的线程所使用的时间超过了时间片所定义的宽度，则会发生时间中断，时间中断的处理发生在TimerInterruptHandler方法中。取消注释，输出"_"是为了更加清晰地看出线程超时的发生。

仔细观察代码，执行输出"Simple Thread Example..."时进程已经开启，已知blitz操作系统中进程开启后会有一个基本线程："main-thread"（他并未在main.c定义却出现了），时间片已经开始计时，所以线程还没有运行就出现了"_"。 随后观察代码：



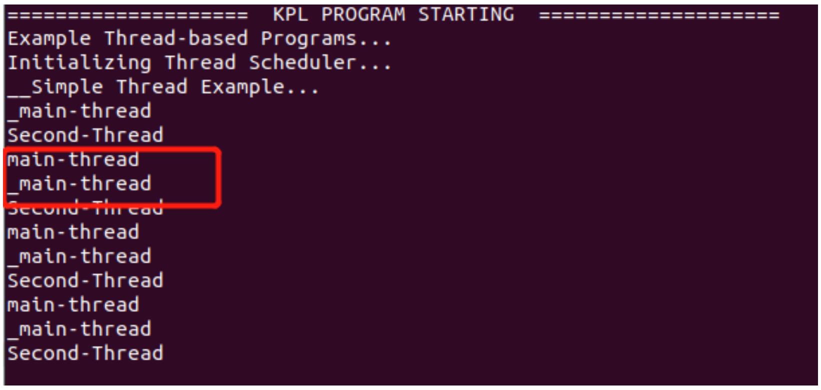
```
function SimpleThreadFunction (cnt: int)
  -- This function will loop "cnt" times.  Each iteration will print a
  -- message and execute a "Yield", which will give the other thread a
  -- chance to run.
    var i: int
    for i = 1 to cnt
      print (currentThread.name)
      nl ()
      currentThread.Yield ()
    endFor
    ThreadFinish ()
  endFunction
```

若没有时间中断，将会交替出现"main-thread" 和 "second-thread".但存在时间中断，所以在中断后根据TimerInterruptHandler，会切换到下一个线程开始执行：



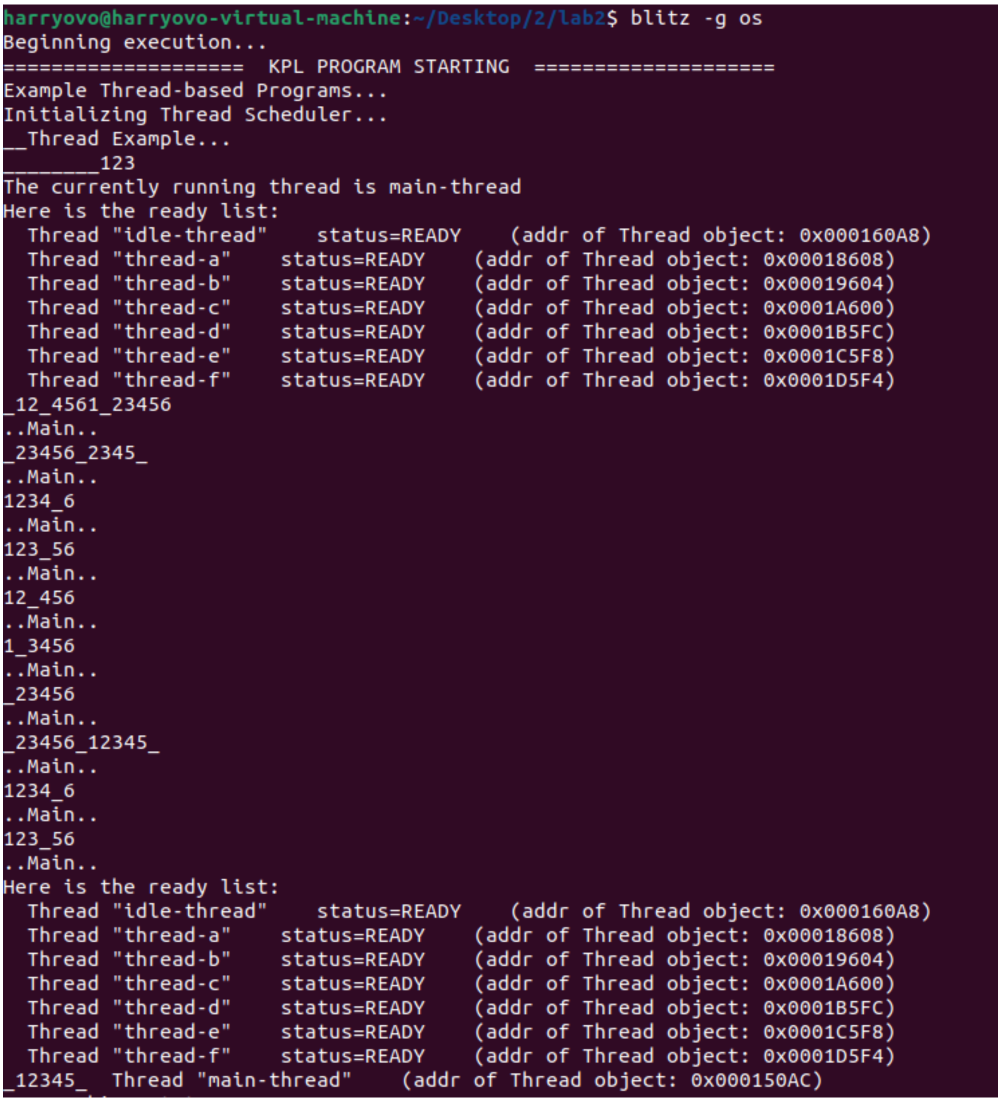
```
----------------------------  TimerInterruptHandler  --------------------------------

  function TimerInterruptHandler ()
    --
    -- This routine is called when a timer interrupt occurs.  Upon entry,
    -- interrupts are DISABLED.  Upon return, execution will return to
    -- the interrupted process, which necessarily had interrupts ENABLED.
    --
    -- (If you wish to turn time-slicing off, simply disable the call
    -- to "Yield" in the code below.  Threads will then execute until they
    -- call "Yield" explicitly, or until they call "Sleep".)
    --
      currentInterruptStatus = DISABLED
      printChar ('_')
      currentThread.Yield ()
      currentInterruptStatus = ENABLED
    endFunction

                                  ThreadPrint
```

这也是为什么会出现下列情况的原因：



```
====================  KPL PROGRAM STARTING  ====================
Example Thread-based Programs...
Initializing Thread Scheduler...
__Simple Thread Example...
_main-thread
Second-Thread
main-thread
_main-thread
Second-Thread
main-thread
_main-thread
Second-Thread
main-thread
_main-thread
Second-Thread
```

step 4



```
harryovo@harryovo-virtual-machine:~/Desktop/2/lab2$ blitz -g os
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Example Thread-based Programs...
Initializing Thread Scheduler...
__Thread Example...
________123
The currently running thread is main-thread
Here is the ready list:
  Thread "idle-thread"    status=READY    (addr of Thread object: 0x000160A8)
  Thread "thread-a"    status=READY    (addr of Thread object: 0x00018608)
  Thread "thread-b"    status=READY    (addr of Thread object: 0x00019604)
  Thread "thread-c"    status=READY    (addr of Thread object: 0x0001A600)
  Thread "thread-d"    status=READY    (addr of Thread object: 0x0001B5FC)
  Thread "thread-e"    status=READY    (addr of Thread object: 0x0001C5F8)
  Thread "thread-f"    status=READY    (addr of Thread object: 0x0001D5F4)
_12_4561_23456
..Main..
_23456_2345_
..Main..
1234_6
..Main..
123_56
..Main..
12_456
..Main..
1_3456
..Main..
_23456
..Main..
_23456_12345_
..Main..
1234_6
..Main..
123_56
..Main..
Here is the ready list:
  Thread "idle-thread"    status=READY    (addr of Thread object: 0x000160A8)
  Thread "thread-a"    status=READY    (addr of Thread object: 0x00018608)
  Thread "thread-b"    status=READY    (addr of Thread object: 0x00019604)
  Thread "thread-c"    status=READY    (addr of Thread object: 0x0001A600)
  Thread "thread-d"    status=READY    (addr of Thread object: 0x0001B5FC)
  Thread "thread-e"    status=READY    (addr of Thread object: 0x0001C5F8)
  Thread "thread-f"    status=READY    (addr of Thread object: 0x0001D5F4)
_12345_  Thread "main-thread"    (addr of Thread object: 0x000150AC)
```

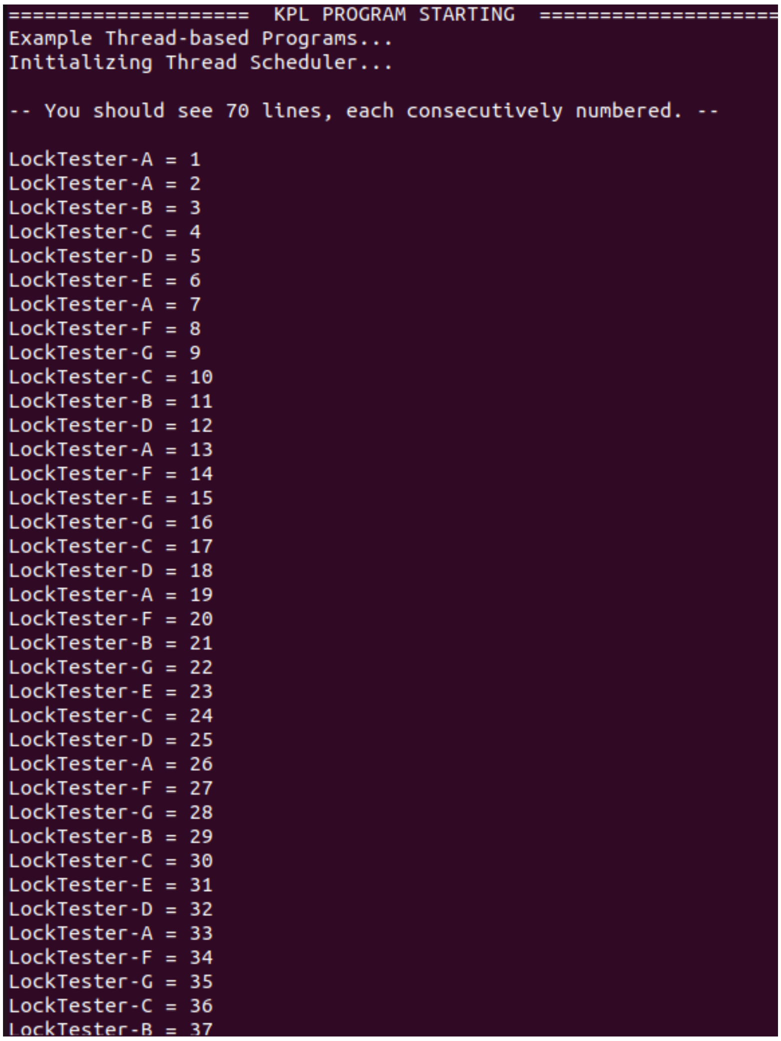
解释：

与step3类似，但出现了没有缺数字的情况。原因是在print执行完毕后才发生时间中断。

在for循环稳定后，就出现了上图中每一个循环超时的线程往前挪一个的情况。这也间接说明了时间片每一次计时是等长的。

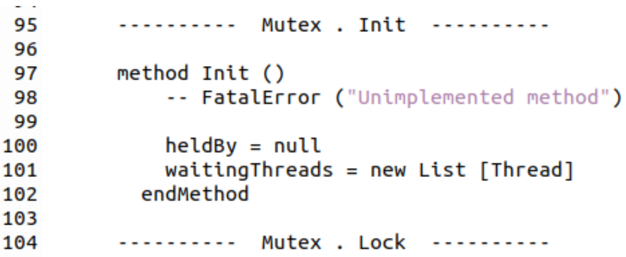
step 5

结果：



```
====================  KPL PROGRAM STARTING  ====================
Example Thread-based Programs...
Initializing Thread Scheduler...

-- You should see 70 lines, each consecutively numbered. --

LockTester-A = 1
LockTester-A = 2
LockTester-B = 3
LockTester-C = 4
LockTester-D = 5
LockTester-E = 6
LockTester-A = 7
LockTester-F = 8
LockTester-G = 9
LockTester-C = 10
LockTester-B = 11
LockTester-D = 12
LockTester-A = 13
LockTester-F = 14
LockTester-E = 15
LockTester-G = 16
LockTester-C = 17
LockTester-D = 18
LockTester-A = 19
LockTester-F = 20
LockTester-B = 21
LockTester-G = 22
LockTester-E = 23
LockTester-C = 24
LockTester-D = 25
LockTester-A = 26
LockTester-F = 27
LockTester-G = 28
LockTester-B = 29
LockTester-C = 30
LockTester-E = 31
LockTester-D = 32
LockTester-A = 33
LockTester-F = 34
LockTester-G = 35
LockTester-C = 36
LockTester-B = 37
```

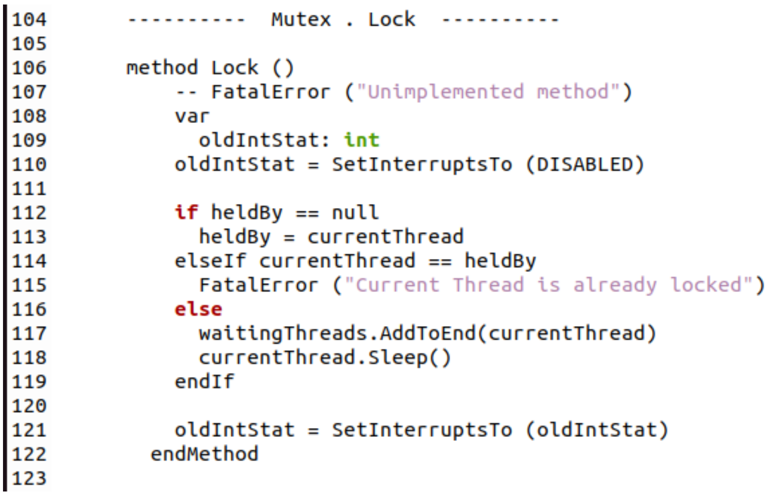
但我们并不关心main.c的测试函数。我们重点考虑Synch.c中的Mutex类的实现：



```
    ----------  Mutex . Init  ----------

    method Init ()
        -- FatalError ("Unimplemented method")

        heldBy = null
        waitingThreads = new List [Thread]
      endMethod

    ----------  Mutex . Lock  ----------
```

heldBy：当前正在使用资源的线程。

waitingTheards：等待使用资源的线程的队列。



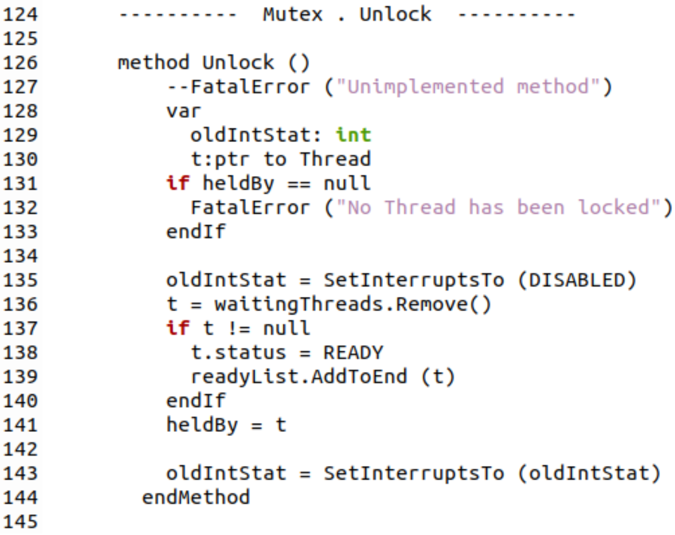
```
104    ----------  Mutex . Lock  ----------
105
106    method Lock ()
107        -- FatalError ("Unimplemented method")
108        var
109          oldIntStat: int
110        oldIntStat = SetInterruptsTo (DISABLED)
111
112        if heldBy == null
113          heldBy = currentThread
114        elseIf currentThread == heldBy
115          FatalError ("Current Thread is already locked")
116        else
117          waitingThreads.AddToEnd(currentThread)
118          currentThread.Sleep()
119        endIf
120
121        oldIntStat = SetInterruptsTo (oldIntStat)
122      endMethod
123
```

如果heldBy为空则说明没有线程正在占用资源，则一定也没有线程在等待。所以直接将当前进程变为正在使用资源的运行态；

若heldBy为当前线程，抛出错误；

若当前已经有线程占用资源，则当前线程需要进入等待队列，并进入睡眠。

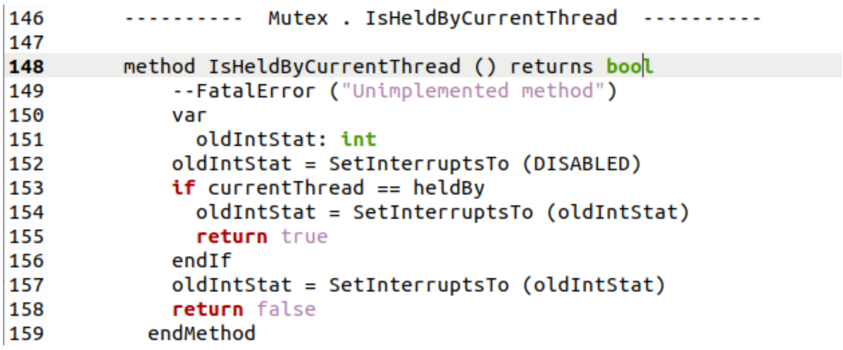
由于如果中断会出现很大的问题，所以不允许中断(原语)，事先关中断(110行)，执行完后再开中断(121行)。



```
----------  Mutex . Unlock  ----------

method Unlock ()
    --FatalError ("Unimplemented method")
    var
      oldIntStat: int
      t:ptr to Thread
    if heldBy == null
      FatalError ("No Thread has been locked")
    endIf

    oldIntStat = SetInterruptsTo (DISABLED)
    t = waitingThreads.Remove()
    if t != null
      t.status = READY
      readyList.AddToEnd (t)
    endIf
    heldBy = t

    oldIntStat = SetInterruptsTo (oldIntStat)
  endMethod
```

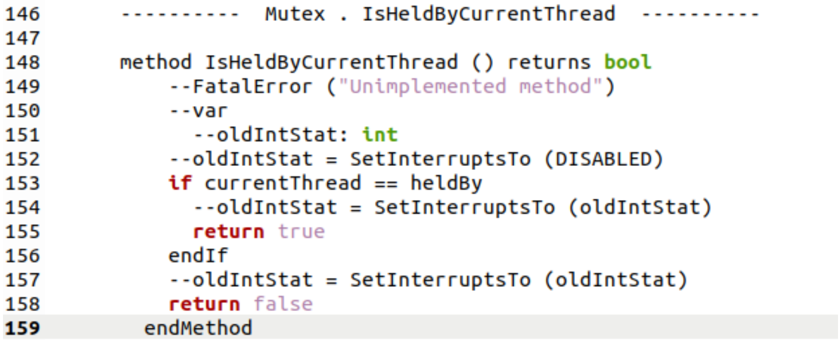
若当前没有锁，抛出错误；

从等待队列中移出最先进入的线程(FIFO)，若非空则将其放入READY状态。若队列中已经没有等待的线程，则将heldBy赋值为null。



```
----------  Mutex . IsHeldByCurrentThread  ----------

method IsHeldByCurrentThread () returns bool
    --FatalError ("Unimplemented method")
    var
      oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if currentThread == heldBy
      oldIntStat = SetInterruptsTo (oldIntStat)
      return true
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
    return false
  endMethod
```

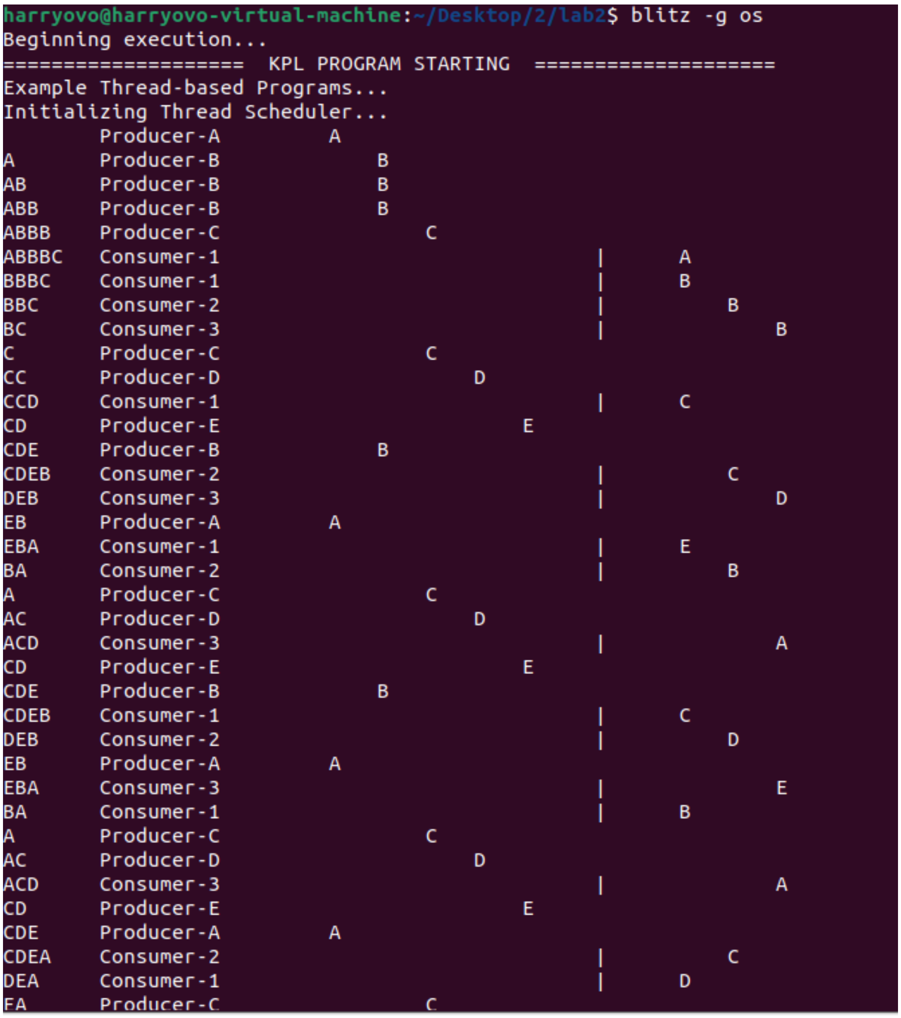
也可以不加锁：



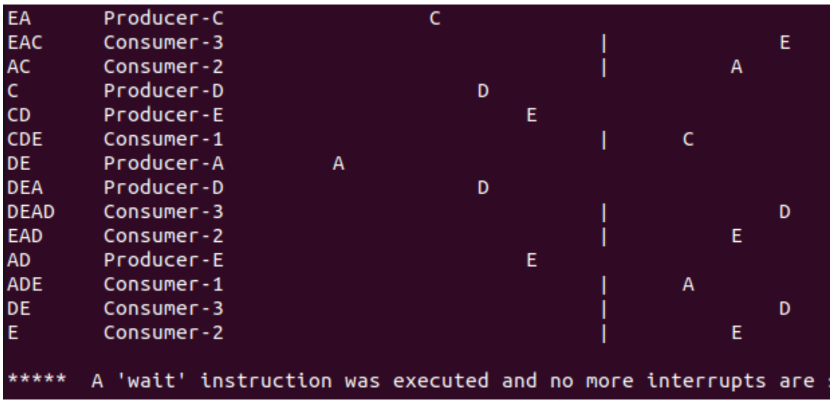
```
----------  Mutex . IsHeldByCurrentThread  ----------

method IsHeldByCurrentThread () returns bool
    --FatalError ("Unimplemented method")
    --var
      --oldIntStat: int
    --oldIntStat = SetInterruptsTo (DISABLED)
    if currentThread == heldBy
      --oldIntStat = SetInterruptsTo (oldIntStat)
      return true
    endIf
    --oldIntStat = SetInterruptsTo (oldIntStat)
    return false
  endMethod
```

解释略。

step 6



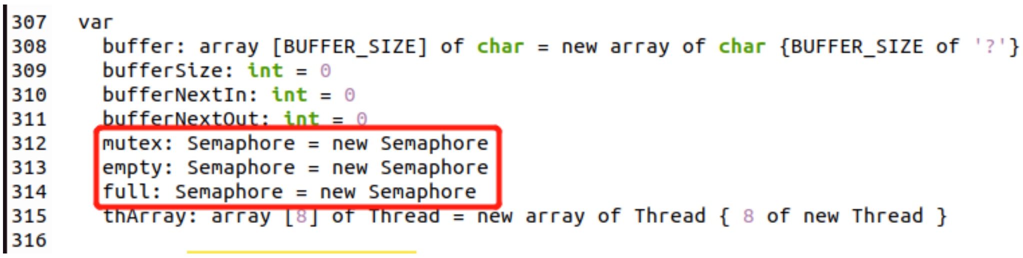
```
harryovo@harryovo-virtual-machine:~/Desktop/2/lab2$ blitz -g os
Beginning execution...
====================  KPL PROGRAM STARTING  ====================
Example Thread-based Programs...
Initializing Thread Scheduler...
        Producer-A         A
A       Producer-B             B
AB      Producer-B             B
ABB     Producer-B             B
ABBB    Producer-C                 C
ABBBC   Consumer-1                               |      A
BBBC    Consumer-1                               |      B
BBC     Consumer-2                               |          B
BC      Consumer-3                               |              B
C       Producer-C                 C
CC      Producer-D                     D
CCD     Consumer-1                               |      C
CD      Producer-E                         E
CDE     Producer-B             B
CDEB    Consumer-2                               |          C
DEB     Consumer-3                               |              D
EB      Producer-A         A
EBA     Consumer-1                               |      E
BA      Consumer-2                               |          B
A       Producer-C                 C
AC      Producer-D                     D
ACD     Consumer-3                               |              A
CD      Producer-E                         E
CDE     Producer-B             B
CDEB    Consumer-1                               |      C
DEB     Consumer-2                               |          D
EB      Producer-A         A
EBA     Consumer-3                               |              E
BA      Consumer-1                               |      B
A       Producer-C                 C
AC      Producer-D                     D
ACD     Consumer-3                               |              A
CD      Producer-E                         E
CDE     Producer-A         A
CDEA    Consumer-2                               |          C
DEA     Consumer-1                               |      D
EA      Producer-C                 C
```



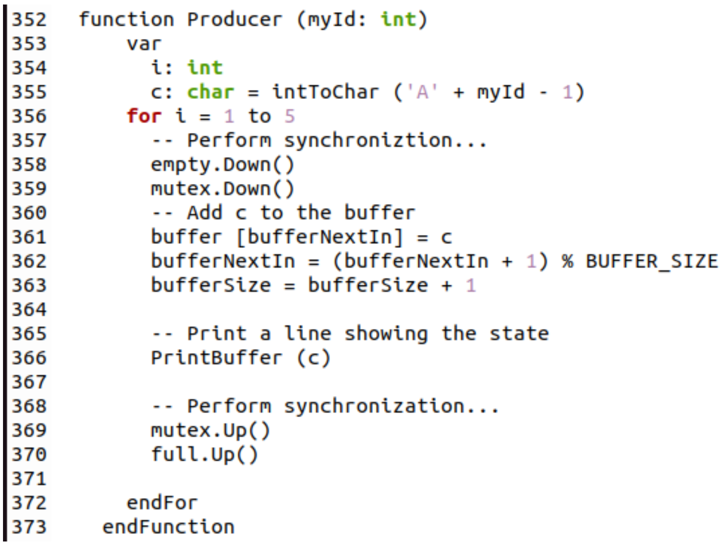
```
EA      Producer-C                 C
EAC     Consumer-3                               |              E
AC      Consumer-2                               |          A
C       Producer-D                     D
CD      Producer-E                         E
CDE     Consumer-1                               |      C
DE      Producer-A         A
DEA     Producer-D                     D
DEAD    Consumer-3                               |              D
EAD     Consumer-2                               |          E
AD      Producer-E                         E
ADE     Consumer-1                               |      A
DE      Consumer-3                               |              D
E       Consumer-2                               |          E

*****  A 'wait' instruction was executed and no more interrupts are
```

注：顺序无关紧要，只需要每个producer都输出5次即正确。

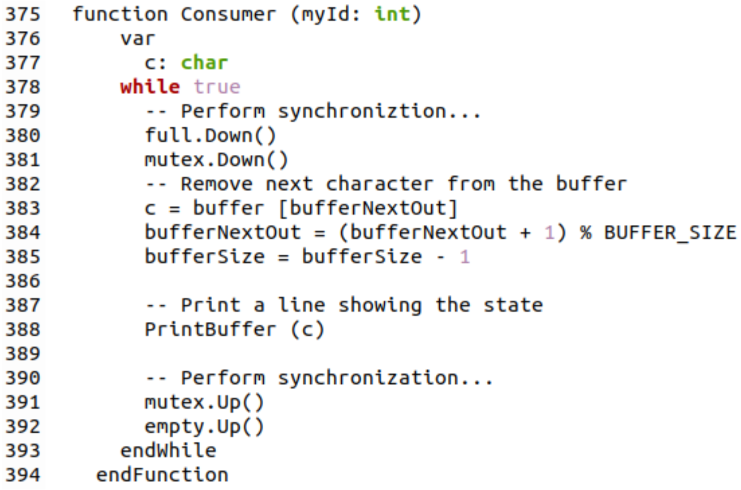
知识点：信号量与前后（同步）关系、互斥关系



```
  var
    buffer: array [BUFFER_SIZE] of char = new array of char {BUFFER_SIZE of '?'}
    bufferSize: int = 0
    bufferNextIn: int = 0
    bufferNextOut: int = 0
    mutex: Semaphore = new Semaphore
    empty: Semaphore = new Semaphore
    full: Semaphore = new Semaphore
    thArray: array [8] of Thread = new array of Thread { 8 of new Thread }

```



```
function Producer (myId: int)
    var
      i: int
      c: char = intToChar ('A' + myId - 1)
    for i = 1 to 5
      -- Perform synchroniztion...
      empty.Down()
      mutex.Down()
      -- Add c to the buffer
      buffer [bufferNextIn] = c
      bufferNextIn = (bufferNextIn + 1) % BUFFER_SIZE
      bufferSize = bufferSize + 1

      -- Print a line showing the state
      PrintBuffer (c)

      -- Perform synchronization...
      mutex.Up()
      full.Up()

    endFor
  endFunction
```



```
function Consumer (myId: int)
    var
      c: char
    while true
      -- Perform synchroniztion...
      full.Down()
      mutex.Down()
      -- Remove next character from the buffer
      c = buffer [bufferNextOut]
      bufferNextOut = (bufferNextOut + 1) % BUFFER_SIZE
      bufferSize = bufferSize - 1

      -- Print a line showing the state
      PrintBuffer (c)

      -- Perform synchronization...
      mutex.Up()
      empty.Up()
    endWhile
  endFunction
```

也就是所谓的PV关系。P：减少量 V：增加量。

注意：

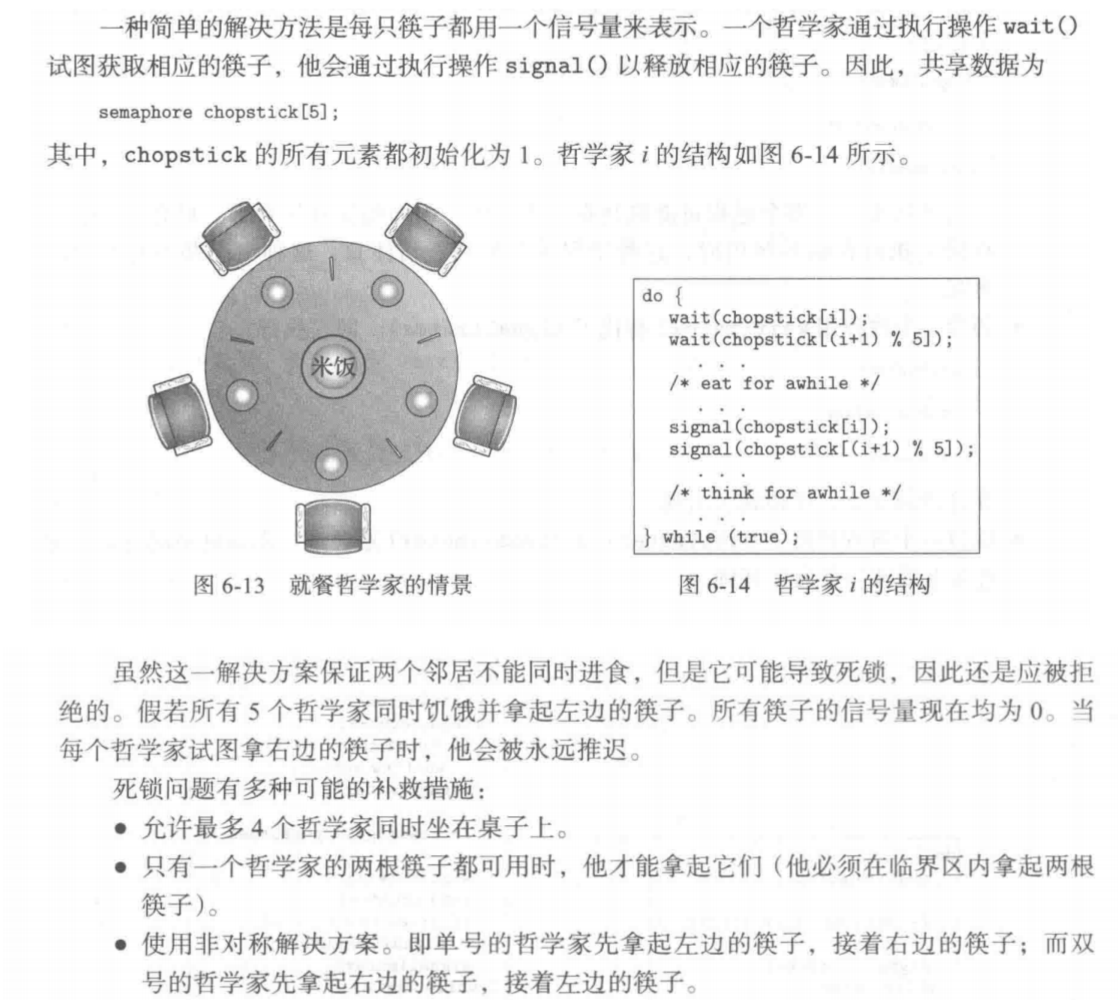
① mutex的PV在同一个函数内是成对出现的(mutex.down()、mutex.up())

② 必须先执行完P的同步关系再执行P的互斥关系，不然会造成死锁。

③ 实际上这里的mutex也可以用step5实现的互斥锁mutex类实现，只是信号量类为二值时等价为mutex类而已。这也是为什么说mutex类可以仿照semaphore类写的原因。

step 7

先介绍非管程的考虑方式与解决方案：



一种简单的解决方法是每只筷子都用一个信号量来表示。一个哲学家通过执行操作 wait() 试图获取相应的筷子，他会通过执行操作 signal() 以释放相应的筷子。因此，共享数据为

```
semaphore chopstick[5];
```

其中，chopstick 的所有元素都初始化为 1。哲学家 *i* 的结构如图 6-14 所示。

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

图 6-13　就餐哲学家的情景

图 6-14　哲学家 *i* 的结构

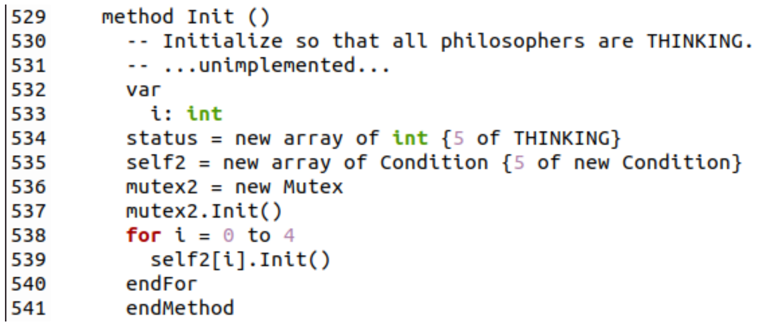
虽然这一解决方案保证两个邻居不能同时进食，但是它可能导致死锁，因此还是应被拒绝的。假若所有 5 个哲学家同时饥饿并拿起左边的筷子。所有筷子的信号量现在均为 0。当每个哲学家试图拿右边的筷子时，他会被永远推迟。

死锁问题有多种可能的补救措施：

- 允许最多 4 个哲学家同时坐在桌子上。
- 只有一个哲学家的两根筷子都可用时，他才能拿起它们（他必须在临界区内拿起两根筷子）。
- 使用非对称解决方案。即单号的哲学家先拿起左边的筷子，接着右边的筷子；而双号的哲学家先拿起右边的筷子，接着左边的筷子。

但是step7所采用的是管程解决方案。

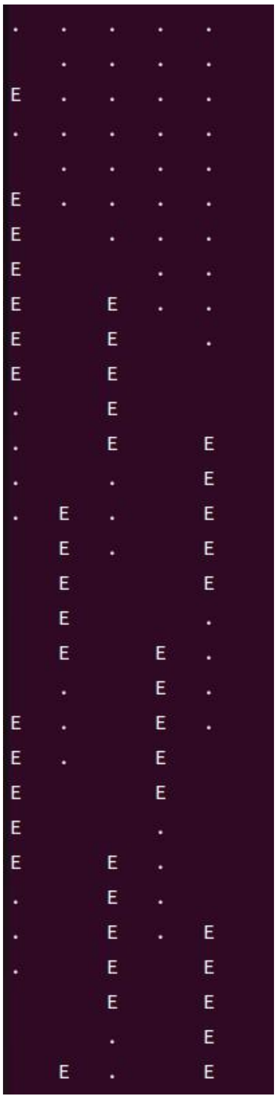
假设两根筷子都可用时才能拿起筷子：



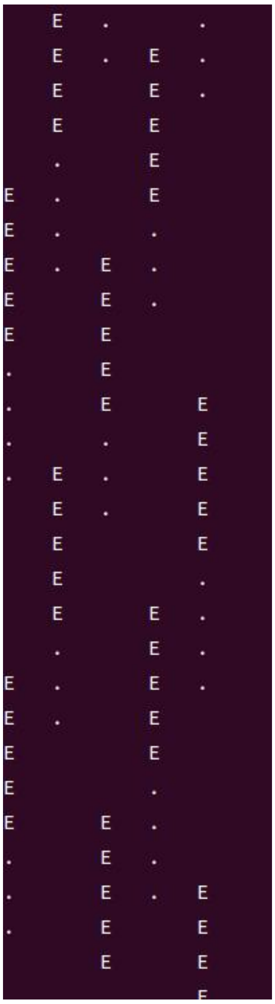
```
  method Init ()
    -- Initialize so that all philosophers are THINKING.
    -- ...unimplemented...
    var
      i: int
    status = new array of int {5 of THINKING}
    self2 = new array of Condition {5 of new Condition}
    mutex2 = new Mutex
    mutex2.Init()
    for i = 0 to 4
      self2[i].Init()
    endFor
    endMethod
```

```
method PickupForks (p: int)
  -- This method is called when philosopher 'p' is wants to eat.
  -- ...unimplemented...
  mutex2.Lock()
  status[p] = HUNGRY
  self.PrintAllStatus()
  self.test(p)
  if status[p] != EATING
    self2[p].Wait(&mutex2)
  endIf
  mutex2.Unlock()
  endMethod

method PutDownForks (p: int)
  -- This method is called when the philosopher 'p' is done eating.
  -- ...unimplemented...
  mutex2.Lock()
  status[p] = THINKING
  self.PrintAllStatus()
  self.test((p+4)%5)
  self.test((p+1)%5)
  mutex2.Unlock()
  endMethod

method test (p: int)
  if (status[(p+4)%5] != EATING && status[p] == HUNGRY && status[(p+1)%5] != EATING)
    status[p] = EATING
    self.PrintAllStatus()
    self2[p].Signal(&mutex2)
  endIf
endMethod
```

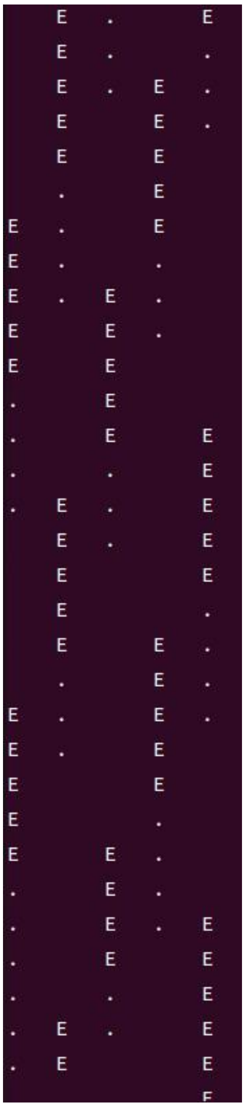
结果:



```
.   .   .   .   .
    .   .   .   .
E   .   .   .   .
.   .   .   .   .
    .   .   .   .
E   .   .   .   .
E       .   .   .
E           .   .
E       E   .   .
E       E       .
E       E
.       E
.       E       E
.       .       E
.   E   .       E
    E   .       E
    E           E
    E           .
    E       E   .
    .       E   .
E   .       E   .
E   .       E
E           E
E           .
E       E   .
.       E   .
.       E   .   E
.       E       E
        E       E
        .       E
    E   .       E
```



```
    E   .       .
    E   .   E   .
    E       E   .
    E       E
    .       E
E   .       E
E   .       .
E   .   E   .
E       E   .
E       E
.       E
.       E       E
.       .       E
.   E   .       E
    E   .       E
    E           E
    E           .
    E       E   .
    .       E   .
E   .       E   .
E   .       E
E           E
E           .
E       E   .
.       E   .
.       E   .   E
.       E       E
        E       E
                E
```



```
    E   .       E
    E   .       .
    E   .   E   .
    E       E   .
    E       E
    .       E
E   .       E
E   .       .
E   .   E   .
E       E   .
E       E
.       E
.       E       E
.       .       E
.   E   .       E
    E   .       E
    E           E
    E           .
    E       E   .
    .       E   .
E   .       E   .
E   .       E
E           E
E           .
E       E   .
.       E   .
.       E   .   E
.       E       E
.       .       E
.   E   .       E
.   E           E
                E
```