since $\neg$ *empty* (*OKtoWrite*) by assumption, `signalC(OKtoWrite)` makes $W \neq 0$ true. Executing `EndWrite` for the last (only) writer could falsify $W \neq 0$, but one of the `signalC` statements will make either $R \neq 0$ or $W \neq 0$ true.

7.4. Theorem

Algorithm 7.4 is free from starvation of readers.

Proof: If a reader is starved is must be blocked indefinitely on `OKtoRead`. We will show that a `signal(OKtoRead)` statement must eventually be executed, unblocking the first reader on the queue. Since the condition queues are assumed to be FIFO, it follows by (numerical) induction on the position of a reader in the queue that it will eventually be unblocked and allowed to read.

To show

$\neg$ *empty* (*OKtoRead*) $\longrightarrow \diamond$ *signalC*(*OKtoRead*),

assume to the contrary that

$\neg$ *empty*(*OKtoRead*) $\wedge$ 2 $\neg$ *signalC*(*OKtoRead*).

By the first invariant of Lemma 7.3, $(W \neq 0) \vee \neg$ *empty* (*OKtoWrite*). If $W \neq 0$, by progress of the writer, eventually `EndWrite` is executed; `signalC(OKtoRead)` will be executed since by assumption $\neg$ *empty* (*OKtoRead*) is true (and it remains true until some `signalC(OKtoRead)` statement is executed), a contradiction.

If $\neg$ *empty* (*OKtoWrite*) is true, by the second invariant of Lemma 7.3, $(R \neq 0) \vee (W \neq 0)$. We have just shown that $W \neq 0$ leads to a contradiction, so consider the case that $R \neq 0$. By progress of readers, if no new readers execute `StartRead`, all readers eventually execute `EndReader`, and the last one will execute `signalC (OKtoWrite)`; since we assumed that $\neg$ *empty* (*OKtoWrite*) is true, a writer will be unblocked making $W \neq 0$ true, and reducing this case to the previous one. The assumption that no new readers successfully execute `StartRead` holds because $\neg$ *empty* (*OKtoWrite*) will cause a new reader to be blocked on `OKtoRead`.

‹ PREVIOUS NEXT ›

‹ PREVIOUS NEXT ›

7.8. A Monitor Solution for the Dining Philosophers

Algorithm 6.10, an attempt at solving the problem of the dining philosophers with semaphores, suffered from deadlock because there is no way to test the value of two fork semaphores simultaneously. With monitors, the test can be encapsulated, so that a philosopher waits until both forks are free before taking them.

Algorithm 7.5 is a solution using a monitor. The monitor maintains an array `fork` which counts the number of free forks *available to* each philosopher. The `take-Forks` operation waits on a condition variable until two forks are available. Before leaving the monitor with these two forks, it decrements the number of forks available to its neighbors. After eating, a philosopher calls `releaseForks`, which, in addition to updating the array `fork`, checks if freeing these forks makes it possible to signal a neighbor.

Let *eating* [*i*] be true if philosopher *i* is eating, that is, if she has successfully executed `takeForks (i)` and has not yet executed `releaseForks(i)`. We leave it as an exercise to show that *eating* [*i*]$\leftrightarrow$(*forks*[*i*] = 2) is invariant. This formula expresses the requirement that a philosopher eats only if she has two forks.

Algorithm 7.5. Dining philosophers with a monitor

```
monitor ForkMonitor
  integer array[0..4] fork ← [2, . . . , 2]
  condition array[0..4] OKtoEat
  operation takeForks(integer i)
    if fork[i] ≠ 2
       waitC(OKtoEat[i])
    fork[i+1] ← fork[i+1] - 1
    fork[i-1] ← fork[i-1] - 1
  operation releaseForks(integer i)
    fork[i+1] ← fork[i+1] + 1
    fork[i-1] ← fork[i-1] + 1
    if fork[i+1] = 2
       signalC(OKtoEat[i+1])
    if fork[i-1] = 2
       signalC(OKtoEat[i-1])

                    philosopher i

    loop forever
p1:    think
p2:    takeForks(i)
p3:    eat
p4:    releaseForks(i)
```

7.5. Theorem

Algorithm 7.5 is free from deadlock.

Proof: Let *E* be the number of philosophers who are eating, In the exercises, we ask you to show that the following formulas are invariant:

7-1.

$$\neg\, empty(OKtoEat[i]) \rightarrow (fork[i] < 2).$$

7-2.

$$\sum_{i=0}^{4} fork[i] = 10 - 2 * E.$$

Deadlock implies *E* = 0 and all philosophers are enqueued on `OKtoEat`. If no philosophers are eating, from (7.2) we conclude $\sum fork[i] = 10$. If they are all enqueued waiting to eat, from (7.1) we conclude $\sum fork[i] \leq 5$ which contradicts the previous formula.

Starvation is possible in Algorithm 7.5. Two philosophers can conspire to starve their mutual neighbor as shown in the following scenario (with the obvious abbreviations):

n	phil1	phil2	phil3	*f*0	*f*1	*f*2	*f*3	*f*4
1	**take(1)**	take(2)	take(3)	2	2	2	2	2
2	release(1)	take(2)	**take(3)**	1	2	1	2	2
3	release(1)	**take(2)** to **waitC(OK[2])**	release(3)	1	2	0	2	1
4	**release(1)**	(blocked)	release(3)	1	2	0	2	1
5	**take(1)**	(blocked)	release(3)	2	2	1	2	1
6	release(1)	(blocked)	**release(3)**	1	2	0	2	1
7	release(1)	(blocked)	**take(3)**	1	2	1	2	2

Philosophers 1 and 3 both need a fork shared with philosopher 2. In lines 1 and 2, they each take a pair of forks, so philosopher 2 is blocked when trying to take forks in line 3. In lines 4 and 5 philosopher 1 releases her forks and then immediately takes them again; since *forks* [2] does not receive the value 2, philosopher 2 is not signaled. Similarly, in lines 6 and 7, philosopher 3 releases her forks and then immediately takes them again. Lines 4 through 7 of the scenario can be continued indefinitely, demonstrating starvation of philosopher 2.

< PREVIOUS NEXT >

< PREVIOUS NEXT >

7.9. Monitors in BACI^L

The monitor of the C implementation of Algorithm 7.4 is shown in Listing 7.1. The program is written in the dialect supported by BACI. The syntax and the semantics are very similar to those of Algorithm 7.4, except that the condition queues are not FIFO, and boolean variables are not supported. A Pascal program is also available in the archive.

< PREVIOUS NEXT >

< PREVIOUS NEXT >

7.10. Protected Objects