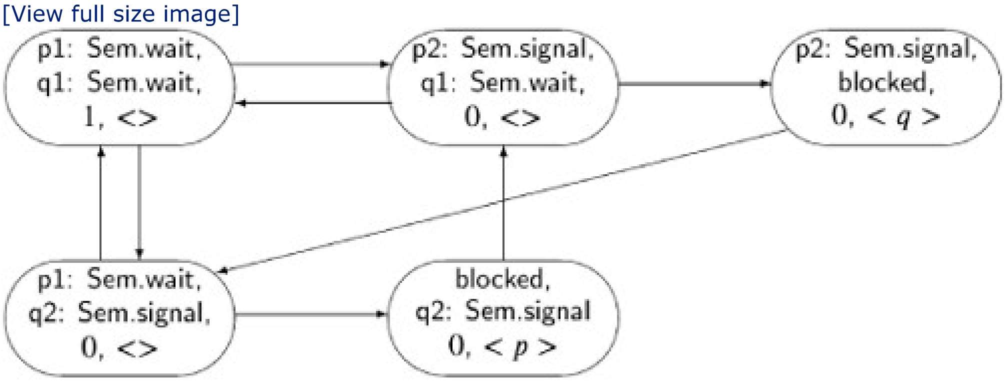
When constructing a state diagram for a program with monitors,*all* the statements of a monitor operation can be considered to be a single step because they are executed under mutual exclusion and no interleaving is possible between the statements. In Algorithm 7.2, each state has four components: the control pointers of the two processes `p` and `q`, the value of the monitor variable `s` and the queue associated with the condition variable `notZero`. Here is the state diagram assuming that the value of `s` has been initialized to 1:

[View full size image]



Consider, first, the transition from the state (`p2: Sem.signal`, `q1: Sem.wait`, 0,<>) at the top center of the diagram to the state (`p2: Sem.signal`, `blocked`, 0,< *q* >) at the upper right. Process `q` executes the `Sem.wait` operation, finds that the value of `s` is 0 and executes the `waitC` operation; it is then blocked on the queue for the condition variable `notZero`.

Consider, now, the transition from (`p2: Sem.signal`, `blocked`, 0,< *q* >) to the state (`p1: Sem.wait`, `q2: Sem.signal`, 0,<>) at the lower left of the diagram. Process `p` executes the `Sem.signal` operation, incrementing `s`, and then `signalC` will unblock process `q`. As we shall discuss in Section 7.5, process `q` *immediately resumes* the execution of its `Sem.wait` operation, so this can be considered as part of the same atomic statement. `q` will decrement `s` back to zero and exit the monitor. Since `signalC(nonZero)` is the last statement of the `Sem.signal` operation executed by process `p`, we may also consider that that process exits the monitor as part of the atomic statement. We end up in a state where process `q` is in its critical section, denoted in the abbreviated algorithm by the control pointer indicating the next invocation of `Sem.signal`, while process `p` is outside its critical section, with its control pointer indicating the next invocation of `Sem.wait`.

There is no state of the form (`p2: Sem.signal`, `q2: Sem.signal`, · · ·, · · ·) in the diagram, so the mutual exclusion requirement is satisfied.

The state diagram for a monitor can be relatively simple, because the internal transitions of the monitor statements can be grouped into a single transition.

PREVIOUS NEXT

PREVIOUS NEXT

Parallel or Concurrent Programming Programming M. Ben-Ari Addison-Wesley Principles of Concurrent and Distributed Programming, Second Edition

7.4. The Producer–Consumer Problem

Algorithm 7.3 is a solution for the producer–consumer problem with a finite buffer using a monitor. Two condition variables are used and the conditions are explicitly checked to see if a process needs to be suspended. The entire processing of the buffer is encapsulated within the monitor and the buffer data structure is not visible to the producer and consumer processes.

Algorithm 7.3. producer–consumer (finite buffer, monitor)

```
monitor PC
  bufferType buffer ← empty
  condition notEmpty
  condition notFull

  operation append(datatype v)
    if buffer is full
       waitC(notFull)
    append(v, buffer)
    signalC(notEmpty)

  operation take()
    datatype w
    if buffer is empty
       waitC(notEmpty)
    w ← head(buffer)
    signalC(notFull)
    return w
```

producer	consumer
datatype d loop forever	datatype d loop forever
p1: d ← produce p2: PC.append(d)	q1: d ← PC.take q2: consume(d)

< PREVIOUS NEXT >

< PREVIOUS NEXT >

Parallel or Concurrent Programming Programming M. Ben-Ari Addison-Wesley Principles of Concurrent and Distributed Programming, Second Edition

7.5. The Immediate Resumption Requirement

The definition of `signalC(cond)` requires that it unblock the first process blocked on the queue for `cond`. When this occurs, the signaling process (say `p`) can now continue to execute the next statement after `signalC(cond)`, while the unblocked process (say `q`) can execute the next statement after the `waitC(cond)` that caused it to block. But this is not a valid state, since the specification of a monitor requires that at most one process at a time can be executing statements of a monitor operation. Either `p` or `q` will have to be blocked until the other completes its monitor operation. That is, we have to specify if signaling processes are given precedence over waiting processes, or vice versa, or perhaps the selection of the next process is arbitrary. There may also be processes blocked on the entry to the monitor, and the specification of precedence has to take them into account.

Note that the terminology is rather confusing, because the waiting processes are processes that have just been *released* from being blocked, rather than processes that are waiting because they are blocked.

The following diagram shows the states that processes can be in: waiting to enter the monitor,