Monitor (synchronization)

In concurrent programming, a **monitor** is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.

Monitors were invented by C. A. R. Hoare [1] and Per Brinch Hansen, [2] and were first implemented in Brinch Hansen's Concurrent Pascal language.

Mutual exclusion

As a simple example, consider a monitor for performing transactions on a bank account.

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    if balance < amount then return false
    else { balance := balance - amount ; return true }
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    balance := balance + amount
  }
}
```

While a thread is executing a method of a monitor, it is said to *occupy* the monitor. Monitors are implemented to enforce that *at each point in time, at most one thread may occupy the monitor*. This is the monitor's mutual exclusion property.

Upon calling one of the methods, a thread must wait until no other thread is executing any of the monitor's methods before starting execution of its method. Note that without this mutual exclusion, in the present example, two threads could cause money to be lost or gained for no reason. For example two threads withdrawing 1000 from the account could both return true, while causing the balance to drop by only 1000, as follows: first, both threads fetch the current balance, find it greater than 1000, and subtract 1000 from it; then, both threads store the balance and return.

In a simple implementation, mutual exclusion can be implemented by the compiler equipping each monitor object with a private lock, often in the form of a semaphore. This lock, which is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

Waiting and signaling

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition P holds true. A busy waiting loop

```
   while not( P ) do skip
```

will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true.

The solution is **condition variables**. Conceptually a condition variable is a queue of threads, associated with a monitor, on which a thread may wait for some condition to become true. Thus each condition variable c is associated with an assertion P_c. While a thread is waiting on a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state. In most types of monitors, these other threads may signal the condition variable c to indicate that assertion P_c is true in the current state.

Thus there are two main operations on condition variables:

- **wait** `c` is called by a thread that needs to wait until the assertion P_c is true before proceeding. While the thread is waiting, it does not occupy the monitor.
- **signal** `c` (sometimes written as **notify** `c`) is called by a thread to indicate that the assertion P_c is true.

As an example, consider a monitor that implements a semaphore. There are methods to increment (V) and to decrement (P) a private integer `s`. However, the integer must never be decremented below 0; thus a thread that tries to decrement must wait until the integer is positive. We use a condition variable `sIsPositive` with an associated assertion of $P_{sIsPositive} = (s > 0)$.

```
monitor class Semaphore
{
  private int s := 0
  invariant s >= 0
  private Condition sIsPositive /* associated with s > 0 */

  public method P()
  {
    if s = 0 then wait sIsPositive
    assert s > 0
    s := s - 1
  }

  public method V()
  {
    s := s + 1
    assert s > 0
    signal sIsPositive
  }
}
```

When a **signal** happens on a condition variable that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor: the thread that signals and any one of the threads that is waiting. In order that at most one thread occupies the monitor at each time, a choice must be made. Two schools of thought exist on how best to resolve this choice. This leads to two kinds of condition variables which will be examined next:

- *Blocking condition variables* or *Signal and Wait* give priority to a signaled thread.
- *Nonblocking condition variables* or *Signal and Continue* give priority to the signaling thread.

Blocking condition variables

The original proposals by C. A. R. Hoare and Per Brinch Hansen were for *blocking condition variables*. Monitors using blocking condition variables are often called *Hoare style* monitors. With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition variable.

We assume there are two queues of threads associated with each monitor object

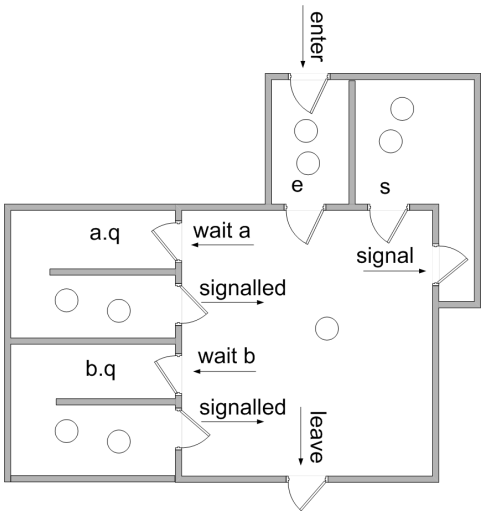
- `e` is the entrance queue
- `s` is a queue of threads that have signaled.

In addition we assume that for each condition variable c, there is a queue

- c`.q`, which is a queue for threads waiting on condition variable c

All queues are typically guaranteed to be fair (in all futures, each thread that enters the queue will be chosen infinitely often) and, in some implementations, may be guaranteed to be first in first out.

The implementation of each operation is as follows. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)



A Hoare style monitor with two condition variables `a` and `b`. After Buhr *et al.*

```
enter the monitor:
   enter the method
   if the monitor is locked
       add this thread to e
       block this thread
   else
       lock the monitor
```

```
leave the monitor:
   schedule
   return from the method
```

```
wait c :
   add this thread to c.q
   schedule
   block this thread
```

```
signal c :
   if there is a thread waiting on c.q
       select and remove one such thread t from c.q
       (t is called "the signaled thread")
       add this thread to s
       restart t
```

```
        (so t will occupy the monitor next)
        block this thread
```

```
 schedule :
   if there is a thread on s
     select and remove one thread from s and restart it
     (this thread will occupy the monitor next)
   else if there is a thread on e
     select and remove one thread from e and restart it
     (this thread will occupy the monitor next)
   else
     unlock the monitor
     (the monitor will become unoccupied)
```

The `schedule` routine selects the next thread to occupy the monitor or, in the absence of any candidate threads, unlocks the monitor.

The resulting signaling discipline is known a *"signal and urgent wait,"* as the signaler must wait, but is given priority over threads on the entrance queue. An alternative is *"signal and wait,"* in which there is no `s` queue and signaler waits on the `e` queue instead.

Some implementations provide a **signal and return** operation that combines signaling with returning from a procedure.

signal c **and return :**

```
   if there is a thread waiting on c.q
       select and remove one such thread t from c.q
       (t is called "the signaled thread")
       restart t
       (so t will occupy the monitor next)
   else
       schedule
   return from the method
```

In either case ("signal and urgent wait" or "signal and wait"), when a condition variable is signaled and there is at least one thread on waiting on the condition variable, the signaling thread hands occupancy over to the signaled thread seamlessly, so that no other thread can gain occupancy in between. If P_c is true at the start of each **signal** c operation, it will be true at the end of each **wait** c operation. This is summarized by the following contracts. In these contracts, I is the monitor's invariant.

```
enter the monitor:
   postcondition I
```

```
leave the monitor:
   precondition I
```

wait c **:**

```
   precondition I
   modifies the state of the monitor
   postcondition P_c and I
```

```
signal c :
   precondition P_c and I
   modifies the state of the monitor
   postcondition I
```

```
signal c and return :
   precondition P_c and I
```

In these contracts, it is assumed that I and P_c do not depend on the contents or lengths of any queues.

(When the condition variable can be queried as to the number of threads waiting on its queue, more sophisticated contracts can be given. For example, a useful pair of contracts, allowing occupancy to be passed without establishing the invariant, is

```
wait c :
   precondition I
   modifies the state of the monitor
   postcondition P_c
```

```
signal c
   precondition (not empty(c) and P_c) or (empty(c) and I)
   modifies the state of the monitor
   postcondition I
```

See Howard[3] and Buhr *et al.*,[4] for more).

It is important to note here that the assertion P_c is entirely up to the programmer; he or she simply needs to be consistent about what it is.

We conclude this section with an example of a blocking monitor that implements a bounded, thread safe stack.

```
monitor class SharedStack {
  private const capacity := 10
  private int[capacity] A
  private int size := 0
  invariant 0 <= size and size <= capacity
  private BlockingCondition theStackIsNotEmpty /* associated with 0 < size and size <= capacity */
  private BlockingCondition theStackIsNotFull /* associated with 0 <= size and size < capacity */
```

```
  public method push(int value)
  {
    if size = capacity then wait theStackIsNotFull
    assert 0 <= size and size < capacity
    A[size] := value ; size := size + 1
    assert 0 < size and size <= capacity
    signal theStackIsNotEmpty and return
  }
```

```
  public method int pop()
  {
    if size = 0 then wait theStackIsNotEmpty
    assert 0 < size and size <= capacity
    size := size - 1 ;
```

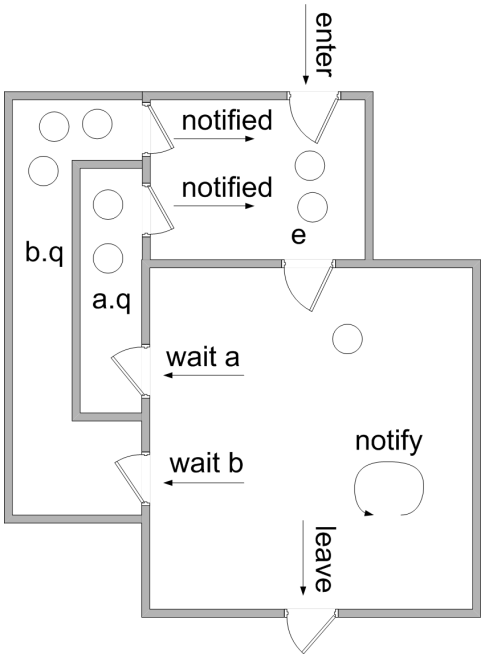
```
    assert 0 <= size and size < capacity
    signal theStackIsNotFull  and return A[size]
  }
}
```

Nonblocking condition variables

With *nonblocking condition variables* (also called *"Mesa style"* condition variables or *"signal and continue"* condition variables), signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the `e` queue. There is no need for the `s` queue.

With nonblocking condition variables, the **signal** operation is often called **notify** — a terminology we will follow here. It is also common to provide a **notify all** operation that moves all threads waiting on a condition variable to the `e` queue.

The meaning of various operations are given here. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)



A Mesa style monitor with two condition variables `a` and `b`

```
enter the monitor:
   enter the method
   if the monitor is locked
     add this thread to e
     block this thread
   else
     lock the monitor
```

```
leave the monitor:
   schedule
   return from the method
```

```
wait c :
   add this thread to c.q
   schedule
   block this thread
```

```
notify c :
   if there is a thread waiting on c.q
       select and remove one thread t from c.q
       (t is called "the notified thread")
       move t to e
```

```
notify all c :
   move all threads waiting on c.q to e
```

```
 schedule :
   if there is a thread on e
     select and remove one thread from e and restart it
   else
     unlock the monitor
```

As a variation on this scheme, the notified thread may be moved to a queue called `w`, which has priority over `e`. See Howard[5] and Buhr *et al.*[6] for further discussion.

It is possible to associate an assertion P_c with each condition variable c such that P_c is sure to be true upon return from **wait** c. However, one must ensure that P_c is preserved from the time the **notify**ing thread gives up occupancy until the notified thread is selected to re-enter the monitor. Between these times there could be activity by other occupants. Thus it is common for P_c to simply be *true*.
For this reason, it is usually necessary to enclose each **wait** operation in a loop like this

```
  while not( P ) do wait c
```

where P is some condition stronger than P_c. The operations **notify** c and **notify all** c are treated as "hints" that P may be true for some waiting thread. Every iteration of such a loop past the first represents a lost notification; thus with nonblocking monitors, one must be careful to ensure that too many notifications can not be lost.
As an example of "hinting" consider a bank account in which a withdrawing thread will wait until the account has sufficient funds before proceeding

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0
  private NonblockingCondition balanceMayBeBigEnough
```

```
  public method withdraw(int amount)
     precondition amount >= 0
  {
    while balance < amount do wait balanceMayBeBigEnough
    assert balance >= amount
    balance := balance - amount
  }
```

```
  public method deposit(int amount)
     precondition amount >= 0
  {
    balance := balance + amount
    notify all balanceMayBeBigEnough
```

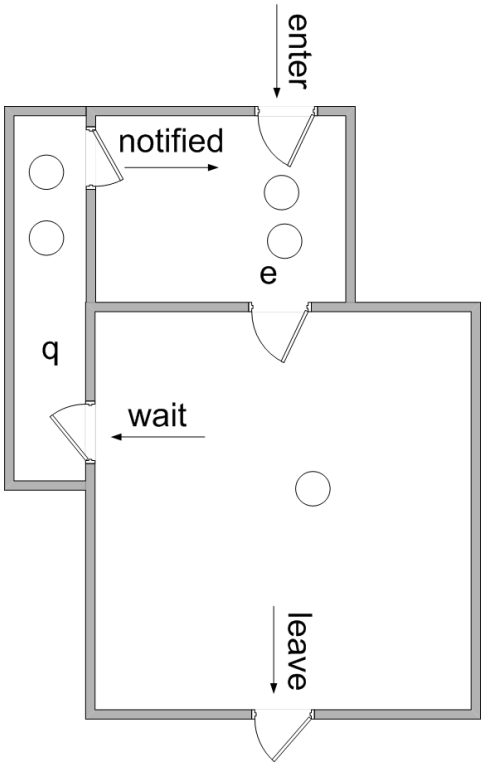
```
    }
}
```

In this example, the condition being waited for is a function of the amount to be withdrawn, so it is impossible for a depositing thread to *know* that it made such a condition true. It makes sense in this case to allow each waiting thread into the monitor (one at a time) to check if its assertion is true.

Implicit condition variable monitors

In the Java language, each object may be used as a monitor. (However, methods that require mutual exclusion must be explicitly marked as **synchronized**.) Rather than having explicit condition variables, each monitor (i.e. object) is equipped with a single wait queue, in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notify all** operations apply to this queue.

This approach has also been adopted in other languages such as C#.



A Java style monitor

Implicit signaling

Another approach to signaling is to omit the **signal** operation. Whenever a thread leaves the monitor (by returning or waiting) the assertions of all waiting threads are evaluated until one is found to be true. In such a system, condition variables are not needed, but the assertions must be explicitly coded. The contract for wait is

```
wait P:
    precondition I
    modifies the state of the monitor
    postcondition P and I
```

History

C. A. R. Hoare and Per Brinch Hansen developed the idea of monitors around 1972, based on earlier ideas of their own and of E. W. Dijkstra. [7] Brinch Hansen was the first to implement monitors. Hoare developed the theoretical framework and demonstrated their equivalence to semaphores.

Monitors were soon used to structure inter-process communication in the Solo operating system.

Programming languages that have supported monitors include

- Ada since Ada 95 (as protected objects)
- C# (and other languages that use the .NET Framework)
- Concurrent Euclid
- Concurrent Pascal
- D

- Delphi (Delphi 2009 and above, via TObject.Monitor)
- Java (via the wait and notify methods)
- Mesa
- Modula-3
- Python (via threading.Condition [8] object)
- Ruby
- Squeak Smalltalk
- Turing, Turing+, and Object-Oriented Turing
- μC++

A number of libraries have been written that allow monitors to be constructed in languages that do not support them natively. When library calls are used, it is up to the programmer to explicitly mark the start and end of code executed with mutual exclusion. Pthreads is one such library.

Bibliography

- Monitors: an operating system structuring concept, C. A. R. Hoare – Communications of the ACM, v.17 n.10, p. 549-557, Oct. 1974 [9]
- Monitor classification P.A. Buhr, M. Fortier, M.H. Coffin – ACM Computing Surveys, 1995 [10]

External links

- Java Monitors (lucid explanation) [11]
- "Monitors: An Operating System Structuring Concept [12]" by C. A. R. Hoare
- "Signalling in Monitors [13]" by John H. Howard (computer scientist)
- "Proving Monitors [14]" by John H. Howard (computer scientist)
- "Experience with Processes and Monitors in Mesa [15]" by Butler W. Lampson and David D. Redell
- pthread_cond_wait [16] – description from the Open Group Base Specifications Issue 6, IEEE Std 1003.1
- "Block on a Condition Variable [17]" by Dave Marshall (computer scientist)
- "Strategies for Implementing POSIX Condition Variables on Win32 [18]" by Douglas C. Schmidt and Irfan Pyarali
- Condition Variable Routines [19] from the Apache Portable Runtime Library
- wxCondition description [20]
- Boost Condition Variables Reference [21]
- ZThread Condition Class Reference [22]
- Wefts::Condition Class Reference [23]
- ACE_Condition Class Template Reference [24]
- QWaitCondition Class Reference [25]
- Common C++ Conditional Class Reference [26]
- at::ConditionalMutex Class Reference [27]
- threads::shared [28] – Perl extension for sharing data structures between threads
- Tutorial multiprocessing traps [29]
- http://msdn.microsoft.com/en-us/library/ms682052(VS.85).aspx
- Monitors [30] in Visual Prolog.

Notes

[1] Hoare, C. A. R. (1974), "Monitors: an operating system structuring concept". Comm. A.C.M. **17(10)**, 549–57. (http://doi.acm.org/10.1145/355620.361161)

[2] Brinch Hansen, P. (1975). "The programming language Concurrent Pascal". IEEE Trans. Softw. Eng. **2** (June), 199–206.

[3] John Howard (1976), "Signaling in monitors". *Proceedings of the 2nd International Conference on Software Engineering*, 47–52

[4] Buhr, P.H; Fortier, M., Coffin, M.H. (1995). "Monitor classification". ACM Computing Surveys (CSUR) **27(1)**. 63–107. (http://doi.acm.org/10.1145/214037.214100)

[5] John Howard (1976), "Signaling in monitors". *Proceedings of the 2nd International Conference on Software Engineering*, 47–52

[6] Buhr, P.H; Fortier, M., Coffin, M.H. (1995). "Monitor classification". ACM Computing Surveys (CSUR) **27(1)**. 63–107. (http://doi.acm.org/10.1145/214037.214100)

[7] Brinch Hansen, P. (1993). "Monitors and concurrent Pascal: a personal history", *The second ACM SIGPLAN conference on History of programming languages* 1–35. Also published in *ACM SIGPLAN Notices* **28(3)**, March 1993. (http://doi.acm.org/10.1145/154766.155361)

[8] http://docs.python.org/library/threading.html#condition-objects

[9] http://doi.acm.org/10.1145/355620.361161

[10] http://doi.acm.org/10.1145/214037.214100

[11] http://www.artima.com/insidejvm/ed2/threadsynch.html

[12] http://www.acm.org/classics/feb96/

[13] http://portal.acm.org/citation.cfm?id=807647

[14] http://doi.acm.org/10.1145/360051.360079

[15] http://portal.acm.org/citation.cfm?id=358824

[16] http://www.opengroup.org/onlinepubs/009695399/functions/pthread_cond_wait.html

[17] http://gd.tuwien.ac.at/languages/c/programming-dmarshall/node31.html#SECTION003125000000000000000

[18] http://www.cs.wustl.edu/~schmidt/win32-cv-1.html

[19] http://apr.apache.org/docs/apr/group__apr__thread__cond.html

[20] http://wxwidgets.org/manuals/2.6.3/wx_wxcondition.html

[21] http://www.boost.org/doc/html/thread/synchronization.html#thread.synchronization.condvar_ref

[22] http://zthread.sourceforge.net/html/classZThread_1_1Condition.html

[23] http://wefts.sourceforge.net/wefts-apidoc-0.99c/classWefts_1_1Condition.html

[24] http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE__Condition.html

[25] http://doc.trolltech.com/latest/qwaitcondition.html

[26] http://www.gnu.org/software/commoncpp/docs/refman/html/class_conditional.html

[27] http://austria.sourceforge.net/dox/html/classat_1_1ConditionalMutex.html

[28] http://perldoc.perl.org/threads/shared.html

[29] http://www.asyncop.net/MTnPDirEnum.aspx?treeviewPath=%5bd%5d+Tutorial%5c%5ba%5d+Multiprocessing+Traps+%26+Pitfalls%5c%5bb%5d+Synchronization+API%5c%5bg%5d+Condition+Variables

[30] http://wiki.visual-prolog.com/index.php?title=Language_Reference/Monitors

Article Sources and Contributors

Monitor (synchronization) *Source*: http://en.wikipedia.org/w/index.php?oldid=467659487 *Contributors*: AKEB, Abdull, Anna Lincoln, Asafshelly, Assimil8or, AxelBoldt, Charlesb, Chruck, Cleared as filed, Craig Pemberton, DARTH SIDIOUS 2, Dcoetzee, Delldot, Flex, Franl, Gazpacho, Gennaro Prota, Gonnet, Herrturtur, Ianb1469, Intgr, Ipatrol, JLaTondre, Jacosi, Jeroen74, Jerryobject, Jfmantis, John Vandenberg, Joy, Kislay kishore2003, Leszek Jańczuk, Marudubshinki, Miym, Mormegil, Moshewe, Moulaali, Musiphil, Niyue, Pacman128, Parklandspanaway, Pburka, Pdcook, Raul654, Rich Farmbrough, RickBeton, Rv74, TPReal, Tcncv, Theodore.norvell, Thiagomacieira, Thomas Linder Puls, Torc2, Voskanyan, Waqas1987, WikHead, Yukoba, 104 anonymous edits

Image Sources, Licenses and Contributors

Image:Monitor (synchronization)-SU.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Monitor_(synchronization)-SU.png *License*: Creative Commons Attribution 3.0 *Contributors*: Theodore.norvell

Image:Monitor (synchronization)-Mesa.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Monitor_(synchronization)-Mesa.png *License*: Creative Commons Attribution 3.0 *Contributors*: Theodore.norvell (talk)

Image:Monitor (synchronization)-.Java.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Monitor_(synchronization)-Java.png *License*: Creative Commons Attribution 3.0 *Contributors*: Theodore.norvell (talk)

License