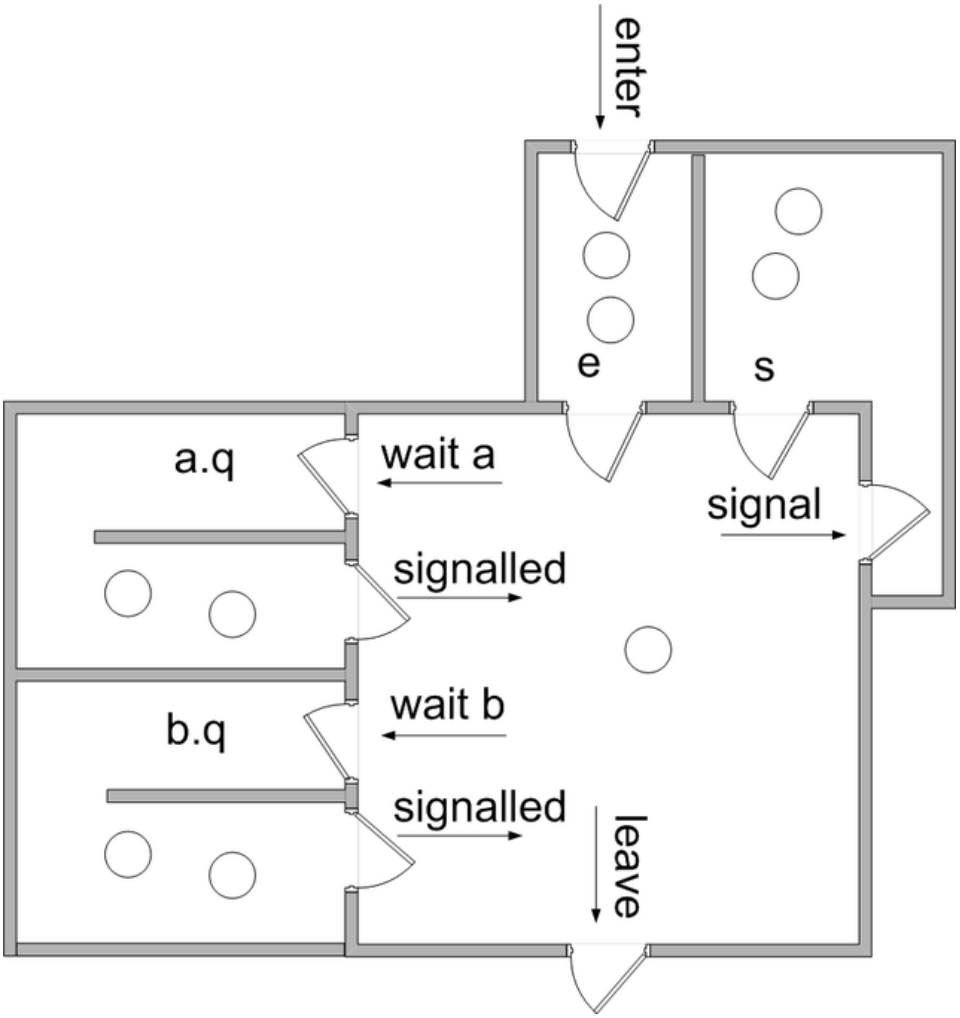
Monitor (synchronization)

Connected to:

Per Brinch Hansen Thread (computer science) Thread (computing)

From Wikipedia, the free encyclopedia



This article contains instructions, advice, or how-to content. The purpose of Wikipedia is to present facts, not to train. Please help improve this article either by rewriting the how-to content or by moving it to Wikiversity, Wikibooks or Wikivoyage. (January 2014)

In concurrent programming, a **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signalling other threads that their condition has been met. A monitor consists of a mutex (lock) object and **condition variables**. A **condition variable** is basically a container of threads that are waiting for a certain condition. Monitors 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.

Another definition of **monitor** is a **thread-safe** class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in

time, at most one thread may be executing any of its methods. Using a condition variable(s), it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor"). For the rest of this article, this sense of "monitor" will be referred to as a "thread-safe object/class/module".

Monitors were invented by Per Brinch Hansen[1] and C. A. R. Hoare,[2] and were first implemented in Brinch Hansen's Concurrent Pascal language.[3]

Mutual exclusion

As a simple example, consider a thread-safe object 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 {
      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 thread-safe object, it is said to *occupy* the object, by holding its mutex (lock). Thread-safe objects are implemented to enforce that *at each point in time, at most one thread may occupy the object*. The lock, which is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

Upon calling one of the methods, a thread must wait until no other thread is executing any of the thread-safe object'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.

The syntactic sugar "monitor class" in the above example is implementing the following basic representation of the code, by wrapping each function's execution in mutexes:

```
class Account {
  private lock myLock

  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      if balance < amount {
        return false
      } else {
        balance := balance - amount
        return true
      }
    } finally {
      myLock.release()
    }
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      balance := balance + amount
    } finally {
      myLock.release()
    }
  }
}
```

Condition variables

Problem statement

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. Other "solutions" exist such as having a loop that unlocks the monitor, waits a certain amount of time, locks the monitor and check for the condition *P*. Theoretically, it works and will not deadlock, but issues arise. It's hard to decide an appropriate amount of waiting time, too small and the thread will hog the CPU, too big and it will be apparently unresponsive. What is needed is a way to signal the thread when the condition P is true (or *could* be true).

Case study: classic bounded producer/consumer problem

A classic concurrency problem is that of the **bounded producer/consumer**, in which there is a queue or ring buffer of tasks with a maximum size, with one or more threads being "producer" threads that add tasks to the queue, and one or more other threads being "consumer" threads that take tasks out of the queue. The queue is assumed to be non–thread-safe itself, and it can be empty, full, or between empty and full. Whenever the queue is full of tasks, then we need the producer threads to block until there is room from consumer threads dequeueing tasks. On the other hand, whenever the queue is empty, then we need the consumer threads to block until more tasks are available due to producer threads adding them.

As the queue is a concurrent object shared between threads, accesses to it must be made atomic, because the queue can be put into an **inconsistent state** during the course of the queue access that should never be exposed between threads. Thus, any code that accesses the queue constitutes a **critical section** that must be synchronized by mutual exclusion. If code and processor instructions in critical sections of code that access the queue could be **interleaved** by arbitrary **context switches** between threads on the same processor or by simultaneously-running threads on multiple processors, then there is a risk of exposing inconsistent state and causing race conditions.

Incorrect without synchronization

A naïve approach is to design the code with **busy-waiting** and no synchronization, making the code subject to race conditions:

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.
        while(queue.isFull()){} // Busy-wait until the queue is non-full.
        queue.enqueue(myTask); // Add the task to the queue.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){
        while (queue.isEmpty()){} // Busy-wait until the queue is non-empty.
        myTask=queue.dequeue(); // Take a task off of the queue.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This code has a serious problem in that accesses to the queue can be interrupted and interleaved with other threads' accesses to the queue. The *queue.enqueue* and *queue.dequeue* methods likely have instructions to update the queue's member variables such as its size, beginning and ending positions, assignment and allocation of queue elements, etc. In addition, the *queue.isEmpty()* and *queue.isFull()* methods read this shared state as well. If producer/consumer threads are allowed to be interleaved during the calls to enqueue/dequeue, then inconsistent state of the queue can be exposed leading to race conditions. In addition, if one consumer makes the queue empty in-between another consumer's exiting the busy-wait and calling "dequeue", then the second consumer will attempt to dequeue from an empty queue leading to an error. Likewise, if a producer makes the queue full in-between another producer's exiting the busy-wait and calling "enqueue", then the second producer will attempt to add to a full queue leading to an error.

Spin-waiting

One naive approach to achieve synchronization, as alluded to above, is to use "**spin-waiting**", in which a mutex is used to protect the critical sections of code and busy-waiting is still used, with the lock being acquired and released in-between each busy-wait check.

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while(queue.isFull()){ // Busy-wait until the queue is non-full.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a consumer might take a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isFull()".
        }

        queue.enqueue(myTask); // Add the task to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
```

```
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){
        queueLock.acquire(); // Acquire lock for initial busy-wait check.
        while (queue.isEmpty()){ // Busy-wait until the queue is non-empty.
            queueLock.release();
            // Drop the lock temporarily to allow a chance for other threads
            // needing queueLock to run so that a producer might add a task.
            queueLock.acquire(); // Re-acquire the lock for the next call to "queue.isEmpty()".
        }
        myTask=queue.dequeue(); // Take a task off of the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This method assures that inconsistent state does not occur, but wastes CPU resources due to the unnecessary busy-waiting. Even if the queue is empty and producer threads have nothing to add for a long time, consumer threads are always busy-waiting unnecessarily. Likewise, even if consumers are blocked for a long time on processing their current tasks and the queue is full, producers are always busy-waiting. This is a wasteful mechanism. What is needed is a way to make producer threads block until the queue is non-full, and a way to make consumer threads block until the queue is non-empty.

(N.B.: Mutexes themselves can also be **spin-locks** which involve busy-waiting in order to get the lock, but in order to solve this problem of wasted CPU resources, we assume that *queueLock* is not a spin-lock and properly uses a blocking lock queue itself.)

Condition variables

The solution is to use **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, m`, where c is a condition variable and `m` is a mutex (lock) associated with the monitor. This operation 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. The function, and fundamental contract, of the "wait" operation, is to do the following steps:
    1. Atomically:

        a. release the mutex m,
        b. move this thread from the "ready queue" to c's "wait-queue" (a.k.a. "sleep-queue") of threads, and
        c. sleep this thread. (Context is synchronously yielded to another thread.)

    2. Once this thread is subsequently notified/signalled (see below) and resumed, then automatically re-acquire the mutex m.

    Steps 1a and 1b can occur in either order, with 1c usually occurring after them. While the thread is sleeping and in c's wait-queue, the next program counter to be executed is at step 2, in the middle of the "wait" function/subroutine. Thus, the thread sleeps and later wakes up in the middle of the "wait" operation.
    The atomicity of the operations within step 1 is important to avoid race conditions that would be caused by a preemptive thread switch in-between them. One failure mode that could occur if these were not atomic is a *missed wakeup*, in which the thread could be on c's sleep-queue and have released the mutex, but a preemptive thread switch occurred before the thread went to sleep, and another thread called a signal/notify operation (see below) on c moving the first thread back out of c's queue. As soon as the first thread in question is switched back to, its program counter will be at step 1c, and it will sleep and be unable to be woken up again, violating the invariant that it should have been on c's sleep-queue when it slept. Other race conditions depend on the ordering of steps 1a and 1b, and depend on where a context switch occurs.

- `signal c`, also known as `notify c`, is called by a thread to indicate that the assertion P_c is true. Depending on the type and implementation of the monitor, this moves one or more threads from `c`'s sleep-queue to the "ready queue" or another queues for it to be executed. It is usually considered a best practice to perform the "signal"/"notify" operation before releasing mutex m that is associated with c, but as long as the code is properly designed for concurrency and depending on the threading implementation, it is often also acceptable to release the lock before signalling. Depending on the threading implementation, the ordering of this can have scheduling-priority ramifications. (Some authors instead advocate a preference for releasing the lock before signalling.) A threading implementation should document any special constraints on this ordering.
    - `broadcast c`, also known as `notifyAll c`, is a similar operation that wakes up all threads in c's wait-queue. This empties the wait-queue. Generally, when more than one predicate condition is associated with the same condition variable, the application will require **broadcast** instead of **signal** because a thread waiting for the wrong condition might be woken up and then immediately go back to sleep without waking up a thread waiting for the correct condition that just became true. Otherwise, if the predicate condition is one-to-one with the condition variable associated with it, then **signal** may be more efficient than **broadcast**.

As a design rule, multiple condition variables can be associated with the same mutex, but not vice versa. (This is a one-to-many correspondence.) This is because the predicate P_c is the same for all threads using the monitor and must be protected with mutual exclusion from all other threads that might cause the condition to be changed or that might read it while the thread in question causes it to be changed, but there may be different threads that want to wait for a different condition on the same variable requiring the same mutex to be used. In the producer-consumer example described above, the queue must be protected by a unique mutex object, m. The "producer" threads will want to wait on a monitor using lock m and a condition variable c_{full} which blocks until the queue is non-full. The "consumer" threads will want to wait on a different monitor using the same mutex m but a different condition

variable c_{empty} which blocks until the queue is non-empty. It would (usually) never make sense to have different mutexes for the same condition variable, but this classic example shows why it often certainly makes sense to have multiple condition variables using the same mutex. A mutex used by one or more condition variables (one or more monitors) may also be shared with code that does *not* use condition variables (and which simply acquires/releases it without any wait/signal operations), if those critical sections do not happen to require waiting for a certain condition on the concurrent data.

Monitor usage

The proper basic usage of a monitor is:

```
acquire(m); // Acquire this monitor's lock.
while (!p) { // While the condition/predicate/assertion that we are waiting for is not true...
        wait(m, cv); // Wait on this monitor's lock and condition variable.
}
// ... Critical section of code goes here ...
signal(cv2); -- OR -- notifyAll(cv2); // cv2 might be the same as cv or different.
release(m); // Release this monitor's lock.
```

To be more precise, this is the same pseudocode but with more verbose comments to better explain what is going on:

```
// ... (previous code)
// About to enter the monitor.
// Acquire the advisory mutex (lock) associated with the concurrent data that is shared between threads,
// to ensure that no two threads can be preemptively interleaved or run simultaneously on different cores
// while executing in critical sections that read or write this same concurrent data.
// If another thread is holding this mutex, then this thread will be sleeped (blocked) and placed on
// m's sleep queue.  (Mutex "m" shall not be a spin-lock.)
acquire(m);
// Now, we are holding the lock and can check the condition for the first time.

// The first time we execute the while loop condition after the above "acquire", we are asking,
// "Does the condition/predicate/assertion we are waiting for happen to already be true?"

while ( ! p() ) // "p" is any expression (e.g. variable or function-call) that checks the condition
                                // and evaluates to boolean.  This itself is a critical section, so you *MUST*
                                // be holding the lock when executing this "while" loop condition!

// If this is not the first time the "while" condition is being checked, then we are asking the question,
// "Now that another thread using this monitor has notified me and woken me up and I have been
// context-switched back to, did the condition/predicate/assertion we are waiting on stay true between
// the time that I was woken up and the time that I
// re-acquired the lock inside the "wait" call in the last iteration of this loop,
// or did some other thread cause the condition to become false again in the meantime
// thus making this a spurious wakeup?

{
        // If this is the first iteration of the loop, then the answer is "no" -- the condition is not ready yet.
        // Otherwise, the answer is: the latter.  This was a spurious wakeup, some other thread occurred first
        // and caused the condition to become false again, and we must wait again.

        wait(m, cv);
                // Temporarily prevent any other thread on any core from doing operations on m or cv.
                // release(m)   // Atomically release lock "m" so other code using this concurrent data
                //                              // can operate, move this thread to cv's wait-queue so that it will be notified
                //                              // sometime when the condition becomes true, and sleep this thread.
                //                              // Re-enable other threads and cores to do operations on m and cv.
                //
                // Context switch occurs on this core.
                //
                // At some future time, the condition we are waiting for becomes true,
                // and another thread using this monitor (m, cv) does either a signal/notify
                // that happens to wake this thread up, or a notifyAll that wakes us up, meaning
                // that we have been taken out of cv's wait-queue.
                //
                // During this time, other threads may be switched to that caused the condition to become
                // false again, or the condition may toggle one or more times, or it may happen to
                // stay true.
                //
                // This thread is switched back to on some core.
                //
                // acquire(m)   // Lock "m" is re-acquired.

        // End this loop iteration and re-check the "while" loop condition to make sure the predicate is
        // still true.

}

// The condition we are waiting for is true!
// We are still holding the lock, either from before entering the monitor or from the
// last execution of "wait".

// Critical section of code goes here, which has a precondition that our predicate
// must be true.
// This code might make cv's condition false, and/or make other condition variables'
// predicates true.
```

```
// Call signal/notify or notifyAll, depending on which condition variables' predicates
// (who share mutex m) have been made true or may have been made true, and the monitor semantic type
// being used.

for (cv_x in cvs_to_notify){
        notify(cv_x); -- OR -- notifyAll(cv_x);
}
// One or more threads have been woken up but will block as soon as they try
// to acquire m.

// Release the mutex so that notified thread(s) and others can enter
// their critical sections.
release(m);
```

Solving the bounded producer/consumer problem

This section may be too technical for most readers to understand. Please help improve this section to make it understandable to non-experts, without removing the technical details. The talk page may contain suggestions. (January 2014) (Learn how and when to remove this template message)

Having introduced the usage of condition variables, let's use it to revisit and solve the classic bounded producer/consumer problem. The classic solution is to use two monitors, comprising two condition variables sharing one lock on the queue:

```
global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock;  // A mutex for the ring-buffer of tasks.  (Not a spin-lock.)
global CV queueEmptyCV; // A condition variable for consumer threads waiting for the queue to become non-empty.
                        // Its associated lock is "queueLock".
global CV queueFullCV; // A condition variable for producer threads waiting for the queue to become non-full.
                       //Its associated lock is also "queueLock".

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while(queue.isFull()){ // Check if the queue is non-full.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto queueFullCV, and sleep this thread.
            wait(queueLock, queueFullCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        queue.enqueue(myTask); // Add the task to the queue.

        // Now the queue is guaranteed to be non-empty, so signal a consumer thread
        // or all consumer threads that might be blocked waiting for the queue to be non-empty:
        signal(queueEmptyCV); -- OR -- notifyAll(queueEmptyCV);

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while (queue.isEmpty()){ // Check if the queue is non-empty.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto queueEmptyCV, and sleep this thread.
            wait(queueLock, queueEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }
        // Critical section that requires the queue to be non-empty.
        // N.B.: We are holding queueLock.
        myTask=queue.dequeue(); // Take a task off of the queue.
        // Now the queue is guaranteed to be non-full, so signal a producer thread
        // or all producer threads that might be blocked waiting for the queue to be non-full:
        signal(queueFullCV); -- OR -- notifyAll(queueFullCV);

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.

        doStuff(myTask); // Go off and do something with the task.
    }
}
```

This ensures concurrency between the producer and consumer threads sharing the task queue, and blocks the threads that have nothing to do rather than busy-waiting as shown in the aforementioned approach using spin-locks.

A variant of this solution could use a single condition variable for both producers and consumers, perhaps named "queueFullOrEmptyCV" or "queueSizeChangedCV". In this case, more than one condition is associated with the condition variable, such that the condition variable represents a weaker condition than the conditions being checked by individual threads. The condition variable represents threads that are waiting for the queue to be non-full *and* ones waiting for it to be non-empty. However, doing this would require using *notifyAll* in all the threads using the condition variable and cannot use a regular *signal*. This is because the regular *signal* might wake up a thread of the wrong type whose condition has not yet been met, and that thread would go back to sleep without a thread of the correct type getting signalled. For example, a producer might make the queue full and wake up another producer instead of a consumer, and the woken producer would go back to sleep. In the complementary case, a consumer might make the queue empty and wake up another consumer instead of a producer, and the consumer would go back to sleep. Using *notifyAll* ensures that some thread of the right type will proceed as expected by the problem statement.

Here is the variant using only one condition variable and notifyAll:

```
global volatile RingBuffer queue; // A thread-unsafe ring-buffer of tasks.
global Lock queueLock; // A mutex for the ring-buffer of tasks.  (Not a spin-lock.)
global CV queueFullOrEmptyCV; // A single condition variable for when the queue is not ready for any thread
                              // -- i.e., for producer threads waiting for the queue to become non-full
                              // and consumer threads waiting for the queue to become non-empty.
                              // Its associated lock is "queueLock".
                              // Not safe to use regular "signal" because it is associated with
                              // multiple predicate conditions (assertions).

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while(queue.isFull()){ // Check if the queue is non-full.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto the CV, and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }

        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        queue.enqueue(myTask); // Add the task to the queue.

        // Now the queue is guaranteed to be non-empty, so signal all blocked threads
        // so that a consumer thread will take a task:
        notifyAll(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another producer instead).

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to add the next task.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){

        queueLock.acquire(); // Acquire lock for initial predicate check.
        while (queue.isEmpty()){ // Check if the queue is non-empty.
            // Make the threading system atomically release queueLock,
            // enqueue this thread onto the CV, and sleep this thread.
            wait(queueLock, queueFullOrEmptyCV);
            // Then, "wait" automatically re-acquires "queueLock" for re-checking
            // the predicate condition.
        }
        // Critical section that requires the queue to be non-full.
        // N.B.: We are holding queueLock.
        myTask=queue.dequeue(); // Take a task off of the queue.

        // Now the queue is guaranteed to be non-full, so signal all blocked threads
        // so that a producer thread will take a task:
        notifyAll(queueFullOrEmptyCV); // Do not use "signal" (as it might wake up another consumer instead).

        // End of critical sections related to the queue.
        queueLock.release(); // Drop the queue lock until we need it again to take off the next task.

        doStuff(myTask); // Go off and do something with the task.
    }
}
```

Synchronization primitives

Implementing mutexes and condition variables requires some kind of synchronization primitive provided by hardware support that provides atomicity. Locks and condition variables are higher-level abstractions over these synchronization primitives. On a uniprocessor, disabling and enabling interrupts is a way to implement monitors by preventing context switches during the critical sections of the locks and condition variables, but this is not enough on a multiprocessor. On a multiprocessor, usually special atomic **read-modify-write** instructions on the memory such as **test-and-set**, **compare-and-swap**, etc. are used, depending on what the ISA provides. These usually require deferring to spin-locking for the internal lock state itself, but this locking is very brief. Depending on the implementation, the atomic read-modify-write instructions may lock the bus from other cores' accesses and/or prevent re-

ordering of instructions in the CPU. Here is an example pseudocode implementation of parts of a threading system and mutexes and Mesa-style condition variables, using **test-and-set** and a first-come, first-served policy. This glosses over most of how a threading system works, but shows the parts relevant to mutexes and condition variables:

Sample Mesa-monitor implementation with Test-and-Set

This section may be too technical for most readers to understand. Please help improve this section to make it understandable to non-experts, without removing the technical details. The talk page may contain suggestions. (January 2014) (Learn how and when to remove this template message)

```
// Basic parts of threading system:
// Assume "ThreadQueue" supports random access.
public volatile ThreadQueue readyQueue; // Thread-unsafe queue of ready threads.  Elements are (Thread*).
public volatile global Thread* currentThread; // Assume this variable is per-core.  (Others are shared.)

// Implements a spin-lock on just the synchronized state of the threading system itself.
// This is used with test-and-set as the synchronization primitive.
public volatile global bool threadingSystemBusy=false;

// Context-switch interrupt service routine (ISR):
// On the current CPU core, preemptively switch to another thread.
public method contextSwitchISR(){
    if (testAndSet(threadingSystemBusy)){
        return; // Can't switch context right now.
    }

    // Ensure this interrupt can't happen again which would foul up the context switch:
    systemCall_disableInterrupts();

    // Get all of the registers of the currently-running process.
    // For Program Counter (PC), we will need the instruction location of
    // the "resume" label below.  Getting the register values is platform-dependent and may involve
    // reading the current stack frame, JMP/CALL instructions, etc.  (The details are beyond this scope.)
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.

    readyQueue.enqueue(currentThread); // Put this thread back onto the ready queue for later execution.

    Thread* otherThread=readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.

    currentThread=otherThread; // Replace the global current-thread pointer value so it is ready for the next thread.

    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread
    // (at "resume" below).  Again, the details of how this is done are beyond this scope.
    restoreRegisters(otherThread.registers);

    // *** Now running "otherThread" (which is now "currentThread")!  The original thread is now "sleeping". ***

    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.

    // Return to where otherThread left off.

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

}

// Thread sleep method:
// On current CPU core, a synchronous context switch to another thread without putting
// the current thread on the ready queue.
// Must be holding "threadingSystemBusy" and disabled interrupts so that this method
// doesn't get interrupted by the thread-switching timer which would call contextSwitchISR().
// After returning from this method, must clear "threadingSystemBusy".
public method threadSleep(){
    // Get all of the registers of the currently-running process.
    // For Program Counter (PC), we will need the instruction location of
    // the "resume" label below.  Getting the register values is platform-dependent and may involve
    // reading the current stack frame, JMP/CALL instructions, etc.  (The details are beyond this scope.)
    currentThread->registers = getAllRegisters(); // Store the registers in the "currentThread" object in memory.
    currentThread->registers.PC = resume; // Set the next PC to the "resume" label below in this method.

    // Unlike contextSwitchISR(), we will not place currentThread back into readyQueue.
    // Instead, it has already been placed onto a mutex's or condition variable's queue.

    Thread* otherThread=readyQueue.dequeue(); // Remove and get the next thread to run from the ready queue.

    currentThread=otherThread; // Replace the global current-thread pointer value so it is ready for the next thread.

    // Restore the registers from currentThread/otherThread, including a jump to the stored PC of the other thread
    // (at "resume" below).  Again, the details of how this is done are beyond this scope.
    restoreRegisters(otherThread.registers);

    // *** Now running "otherThread" (which is now "currentThread")!  The original thread is now "sleeping". ***

    resume: // This is where another contextSwitch() call needs to set PC to when switching context back here.

    // Return to where otherThread left off.
```

```
}

public method wait(Mutex m, ConditionVariable c){
    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    assert m.held; // (Specifically, this thread must be the one holding it.)

    m.release();
    c.waitingThreads.enqueue(currentThread);

    threadSleep();

    // Thread sleeps ... Thread gets woken up from a signal/broadcast.

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // Context switches may now occur here, making the client caller's predicate false.

    m.acquire();

}

public method signal(ConditionVariable c){

    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    if (!c.waitingThreads.isEmpty()){
        wokenThread=c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken thread is not given any priority.

}

public method broadcast(ConditionVariable c){

    // Internal spin-lock while other threads on any core are accessing this object's
    // "held" and "threadQueue", or "readyQueue".
    while (testAndSet(threadingSystemBusy)){}
    // N.B.: "threadingSystemBusy" is now true.

    // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
    // the thread-switching timer on this core which would call contextSwitchISR().
    // Done outside threadSleep() for more efficiency so that this thread will be sleeped
    // right after going on the condition-variable queue.
    systemCall_disableInterrupts();

    while (!c.waitingThreads.isEmpty()){
        wokenThread=c.waitingThreads.dequeue();
        readyQueue.enqueue(wokenThread);
    }

    threadingSystemBusy=false; // Must be an atomic assignment.
    systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    // Mesa style:
    // The woken threads are not given any priority.

}

class Mutex {
    protected volatile bool held=false;
```

```
    private volatile ThreadQueue blockingThreads; // Thread-unsafe queue of blocked threads.  Elements are (Thread*).

    public method acquire(){
        // Internal spin-lock while other threads on any core are accessing this object's
        // "held" and "threadQueue", or "readyQueue".
        while (testAndSet(threadingSystemBusy)){}
        // N.B.: "threadingSystemBusy" is now true.

        // System call to disable interrupts on this core so that threadSleep() doesn't get interrupted by
        // the thread-switching timer on this core which would call contextSwitchISR().
        // Done outside threadSleep() for more efficiency so that this thread will be sleeped
        // right after going on the lock queue.
        systemCall_disableInterrupts();

        assert !blockingThreads.contains(currentThread);

        if (held){
            // Put "currentThread" on this lock's queue so that it will be
            // considered "sleeping" on this lock.
            // Note that "currentThread" still needs to be handled by threadSleep().
            readyQueue.remove(currentThread);
            blockingThreads.enqueue(currentThread);
            threadSleep();

            // Now we are woken up, which must be because "held" became false.
            assert !held;
            assert !blockingThreads.contains(currentThread);
        }

        held=true;

        threadingSystemBusy=false; // Must be an atomic assignment.
        systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    }

    public method release(){
        // Internal spin-lock while other threads on any core are accessing this object's
        // "held" and "threadQueue", or "readyQueue".
        while (testAndSet(threadingSystemBusy)){}
        // N.B.: "threadingSystemBusy" is now true.

        // System call to disable interrupts on this core for efficiency.
        systemCall_disableInterrupts();

        assert held; // (Release should only be performed while the lock is held.)

        held=false;

        if (!blockingThreads.isEmpty()){
            Thread* unblockedThread=blockingThreads.dequeue();
            readyQueue.enqueue(unblockedThread);
        }

        threadingSystemBusy=false; // Must be an atomic assignment.
        systemCall_enableInterrupts(); // Turn pre-emptive switching back on on this core.

    }

}

struct ConditionVariable {

    volatile ThreadQueue waitingThreads;

}
```

Semaphore

As an example, consider a thread-safe class 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()
  {
    while s = 0:
      wait sIsPositive
    assert s > 0
    s := s - 1
  }
```

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

Implemented showing all synchronization (removing the assumption of a thread-safe class and showing the mutex):

```
class Semaphore
{
  private volatile int s := 0
  invariant s >= 0
  private ConditionVariable sIsPositive /* associated with s > 0 */
  private Mutex myLock /* Lock on "s" */

  public method P()
  {
    myLock.acquire()
    while s = 0:
      wait(myLock, sIsPositive)
    assert s > 0
    s := s - 1
    myLock.release()
  }

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

Monitor implemented using semaphores

Conversely, locks and condition variables can also be derived from semaphores, thus making monitors and semaphores reducible to one another:

The implementation given here is incorrect. If a thread calls wait() after signal() has been called it may be stuck indefinitely, since signal() increments the semaphore only enough times for threads already waiting.

```
public method wait(Mutex m, ConditionVariable c){

    assert m.held;

    c.internalMutex.acquire();

    c.numWaiters++;
    m.release(); // Can go before/after the neighboring lines.
    c.internalMutex.release();

    // Another thread could signal here, but that's OK because of how
    // semaphores count.  If c.sem's number becomes 1, we'll have no
    // waiting time.
    c.sem.Proberen(); // Block on the CV.
    // Woken
    m.acquire(); // Re-acquire the mutex.
}

public method signal(ConditionVariable c){

    c.internalMutex.acquire();
    if (c.numWaiters > 0){
        c.numWaiters--;
        c.sem.Verhogen(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();

}

public method broadcast(ConditionVariable c){

    c.internalMutex.acquire();
    while (c.numWaiters > 0){
        c.numWaiters--;
        c.sem.Verhogen(); // (Doesn't need to be protected by c.internalMutex.)
    }
    c.internalMutex.release();

}
```

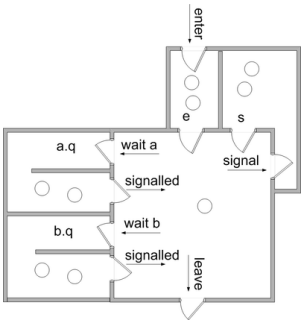
```
class Mutex {

    protected boolean held=false; // For assertions only, to make sure sem's number never goes > 1.
    protected Semaphore sem=Semaphore(1); // The number shall always be at most 1.
                                         // Not held <--> 1; held <--> 0.

    public method acquire(){

        sem.Proberen();
        assert !held;
        held=true;

    }

    public method release(){

        assert held; // Make sure we never Verhogen sem above 1.  That would be bad.
        held=false;
        sem.Verhogen();

    }

}

class ConditionVariable {

    protected int numWaiters=0; // Roughly tracks the number of waiters blocked in sem.
                                // (The semaphore's internal state is necessarily private.)
    protected Semaphore sem=Semaphore(0); // Provides the wait queue.
    protected Mutex internalMutex; // (Really another Semaphore.  Protects "numWaiters".)

}
```

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* give priority to a signaled thread.
- *Nonblocking condition variables* 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*. 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. Monitors using blocking condition variables are often called *Hoare-style* monitors or *signal-and-urgent-wait* monitors.



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

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 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.)

```
 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[4] and Buhr *et al*.,[5] 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.

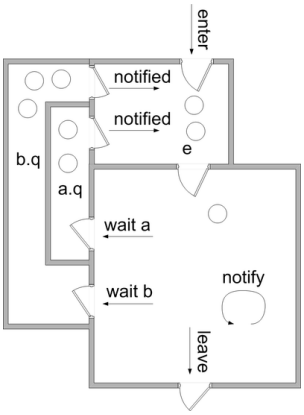
We conclude this section with an example of a thread-safe class using a blocking monitor that implements a bounded, thread-safe stack.

```
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]
  }
}
```

Note that, in this example, the thread-safe stack is internally providing a mutex, which, as in the earlier producer/consumer example, is shared by both condition variables, which are checking different conditions on the same concurrent data. The only difference is that the producer/consumer example assumed a regular non-thread-safe queue and was using a standalone mutex and condition variables, without these details of the monitor abstracted away as is the case here. In this example, when the "wait" operation is called, it must somehow be supplied with the thread-safe stack's mutex, such as if the "wait" operation is an integrated part of the "monitor class". Aside from this kind of abstracted functionality, when a "raw" monitor is used, it will *always* have to include a mutex and a condition variable, with a unique mutex for each condition variable.

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.



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

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.)

```
 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[4] and Buhr *et al*.[5] 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
```

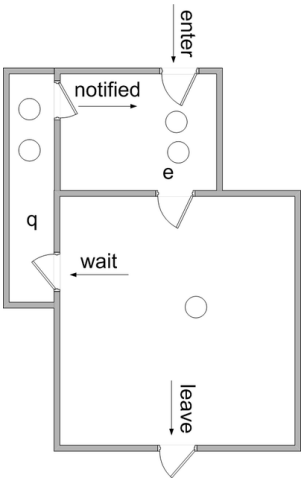
where P is some condition stronger than P_c. The operations **`notify c`** and **`notify all c`** are treated as "hints" that P may be true for some waiting thread. Every iteration of such a loop past the first represents a lost notification; thus with nonblocking monitors, one must be careful to ensure 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



A Java style monitor

In the Java language, each object may be used as a monitor. Methods requiring mutual exclusion must be explicitly marked with the **synchronized** keyword. Blocks of code may also be marked by **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 **notifyAll** operations apply to this queue. This approach has been adopted in other languages, for example C#.

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

Brinch Hansen and Hoare developed the monitor concept in the early 1970s, based on earlier ideas of their own and of Edsger Dijkstra.[6] Brinch Hansen published the first monitor notation, adopting the class concept of Simula 67,[1] and invented a queueing mechanism.[7] Hoare refined the rules of process resumption.[2] Brinch Hansen created the first implementation of monitors, in Concurrent Pascal.[6] Hoare demonstrated their equivalence to semaphores.

Monitors (and Concurrent Pascal) were soon used to structure process synchronization in the Solo operating system.[8][9]

Programming languages that have supported monitors include

- Ada since Ada 95 (as protected objects)
- C# (and other languages that use the .NET Framework)
- C++ since C++11
- 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 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.

See also

- Mutual exclusion
- Communicating sequential processes - a later development of monitors by C. A. R. Hoare