# Multithreading-2

Is there any difference between binary semaphore and mutex or they are essentialy same?

They are **NOT** the same thing. They are used for different purposes!  
While both types of semaphores have a full/empty state and use the same API, their usage is very different.

**Mutual Exclusion Semaphores**  
Mutual Exclusion semaphores are used to protect shared resources (data structure, file, etc..).

A Mutex semaphore is "owned" by the task that takes it. If atask B attempts to semGive a mutex currently held by task A, task B's call will return an error and fail.

Mutexes always use the following sequence:

- SemTake

- Critical Section

- SemGive

Here is a simple example:

Thread A Thread B

Take Mutex

access data

... Take Mutex <== Will block

...

Give Mutex access data <== Unblocks

...

Give Mutex

**Binary Semaphore**  
Binary Semaphore address a totally different question:

* Task B is pended waiting for something to happen (a sensor being tripped for example).
* Sensor Trips and an Interrupt Service Routine runs. It needs to notify a task of the trip.
* Task B should run and take appropriate actions for the sensor trip. Then go back to waiting.

Task A Task B

... Take BinSemaphore <== wait for something

Do Something Noteworthy

Give BinSemaphore do something <== unblocks

Note that with a binary semaphore, it is OK for B to take the semaphore and A to give it.  
Again, a binary semaphore is NOT protecting a resource from access. The act of Giving and Taking a semaphore are fundamentally decoupled.  
It typically makes little sense for the same task to so a give and a take on the same binary semaphore.

When to use volatile with multithreading

If there are two threads accessing a global variable then many tutorials say make the variable volatile to prevent the compiler caching the variable in a register and it thus not getting updated correctly. However two threads both accessing a shared variable is something which calls for protection via a mutex isn't it? But in that case, between the thread locking and releasing the mutex the code is in a critical section where only that one thread can access the variable, in which case the variable doesn't need to be volatile?

So therefore what is the use/purpose of volatile in a multi-threaded program?

Short & quick answer: volatile is (nearly) useless for platform-agnostic, multithreaded application programming. It does not provide any synchronization, it does not create memory fences, nor does it ensure the order of execution of operations. It does not make operations atomic. It does not make your code magically thread safe. volatile may be the single-most misunderstood facility in all of C++. See[this](http://stackoverflow.com/questions/4168735/is-this-rule-about-volatile-usage-strict), [this](http://software.intel.com/en-us/blogs/2007/11/30/volatile-almost-useless-for-multi-threaded-programming/) and [this](http://stackoverflow.com/questions/4136900/what-rules-does-compiler-have-to-follow-when-dealing-with-volatile-memory-locatio) for more information about volatile

On the other hand, volatile does have some use that may not be so obvious. It can be used much in the same way one would use const to help the compiler show you where you might be making a mistake in accessing some shared resource in a non-protected way. This use is discussed by Alexandrescu in [this article](http://www.drdobbs.com/184403766). However, this is basically using the C++ type system in a way that is often viewed as a contrivance and can evoke Undefined Behavior.

volatile was specifically intended to be used when interfacing with memory-mapped hardware, signal handlers and the setjmp machine code instruction. This makes volatile directly applicable to systems-level programming rather than normal applications-level programming.

The 2003 C++ Standard does not say that volatile applies any kind of Acquire or Release semantics on variables. In fact, the Standard is completely silent on all matters of multithreading. However, specific platforms do apply Acquire and Release semantics on volatile variables.

The above all applies the the C++ language itself, as defined by the 2003 Standard. Some specific platforms however do add additional functionality or restrictions to what volatile does. For example, in Windows 2010 (at least) Acquire and Release semantics do apply to certian operations on volatilevariables

Volatile is occasionally useful for the following reason: this code:

/\* global \*/ bool flag = false;

while (!flag) {}

is optimized by gcc to:

if (!flag) { while (true) {} }

Which is obviously incorrect if the flag is written to by the other thread. Note that without this optimization the synchronization mechanism probably works (depending on the other code some memory barriers may be needed) - there is no need for a mutex in 1 producer - 1 consumer scenario.

Otherwise the volatile keyword is too weird to be useable - it does not provide any memory ordering guarantees wrt both volatile and non-volatile accesses and does not provide any atomic operations - i.e. you get no help from the compiler with volatile keyword except disabled register caching.

The problem with volatile in a multithreaded context is that it doesn't provide *all* the guarantees we need. It does have a few properties we need, but not all of them, so we can't rely on volatile *alone*.

However, the primitives we'd have to use for the *remaining* properties also provide the ones thatvolatile do, so it is effectively unnecessary.

For thread-safe accesses to shared data, we need a guarantee that

* the read/write actually happens (that the compiler won't just store the value in a register instead and defer updating main memory until much later)
* that no reordering takes place. Assume that we use a volatile variable as a flag to indicate whether or not some data is ready to be read. In our code, we simply set the flag after preparing the data, so all *looks* fine. But what if the instructions are reordered so the flag is set *first*?

volatile does guarantee the first point. It also guarantees that no reordering occurs *between different volatile reads/writes*. All volatile memory accesses will occur in the order in which they're specified. That is all we need for what volatile is intended for: manipulating I/O registers or memory-mapped hardware, but it doesn't help us in multithreaded code where the volatile object is often only used to synchronize access to non-volatile data. Those accesses can still be reordered relative to thevolatile ones.

The solution to preventing reordering is to use a *memory barrier*, which indicates both to the compiler and the CPU that *no memory access may be reordered across this point*. Placing such barriers around our volatile variable access ensures that even non-volatile accesses won't be reordered across the volatile one, allowing us to write thread-safe code.

However, memory barriers *also* ensure that all pending reads/writes are executed when the barrier is reached, so it effectively gives us everything we need by itself, making volatile unnecessary. We can just remove the volatile qualifier entirely.

Do spurious wakeups usually happen?

Seeing various locking related question and (almost) always finding the 'loop because of spurious wakeups' terms1 I wonder, has anyone experienced such kind of a wakeup (assuming a decent hardware/software environment for example)?

I know the term 'spurious' means no apparent reason but what can be the reasons for such kind of an event?

The Wikipedia [article on spurious wakeups](http://en.wikipedia.org/wiki/Spurious_wakeup) has this tidbit:

The pthread\_cond\_wait() function in Linux is implemented using the futex system call. Each blocking system call on Linux returns abruptly with EINTR when the process receives a signal. ...pthread\_cond\_wait() can't restart the waiting because it may miss a real wakeup in the little time it was outside the futex system call. This race condition can only be avoided by the caller checking for an invariant. A POSIX signal will therefore generate a spurious wakeup.

**Summary**: If a Linux process is signaled its waiting threads will each enjoy a nice, hot spurious wakeup.

I buy it. That's an easier pill to swallow than the typically vague "it's for performance" reason often given

http://www.devguli.com/blog/eng/spurious-wakeup/

Why do pthread condition variables require a mutex

t's just the way that condition variables were initially implemented in DEC threading, the precursor to pthreads.

The mutex was originally used to protect *the condition variable itself*. That's why you need it locked before you do a wait.

The wait will "atomically" unlock the mutex, allowing others access to the condition variable (for signalling). Then when the condition variable is signalled or broadcast to, one or more of the threads on the waiting list will be woken up and the mutex will be magically locked again for that thread.

You typically see the following operation with condition variables, illustrating how they work. The following example is a worker thread which is given work via a signal to a condition variable.

thread:

initialise.

lock mutex.

while thread not told to stop working:

wait on condvar using mutex.

if work is available to be done:

do the work.

unlock mutex.

clean up.

exit thread.

The work is done within this loop provided that there is some available when the wait returns. When the thread has been flagged to stop doing work (usually by another thread setting the exit condition then kicking the condition variable to wake this thread up), the loop will exit, the mutex will be unlocked and this thread will exit.

The code above is a single-consumer model as the mutex remains locked while the work is being done. For a multi-consumer variation, you can use, as an *example*:

thread:

initialise.

lock mutex.

while thread not told to stop working:

wait on condvar using mutex.

if work is available to be done:

copy work to thread local storage.

unlock mutex.

do the work.

lock mutex.

unlock mutex.

clean up.

exit thread.

which allows other consumers to receive work while this one is doing work.

The condition variable relieves you of the burden of polling some condition instead allowing another thread to notify you when something needs to happen. Another thread can tell that thread that work is available as follows:

lock mutex.

flag work as available.

signal condition variable.

unlock mutex.

The vast majority of what are often erroneously called spurious wakeups was generally always because multiple threads had been signalled within their pthread\_cond\_wait call (broadcast), one would return with the mutex, do the work, then re-wait.

Then the second signalled thread could come out when there was no work to be done. So you had to have an extra variable indicating that work should be done (this was inherently mutex-protected with the condvar/mutex pair here - other threads needed to lock the mutex before changing it however).

It *was* technically possible for a thread to return from a condition wait without being kicked by another process (this is a genuine spurious wakeup) but, in all my many years working on pthreads, both in development/service of the code and as a user of them, I never once received one of these. Maybe that was just because HP had a decent implementation :-)

In any case, the same code that handled the erroneous case also handled genuine spurious wakeups as well since the work-available flag would not be set for those.

* DOS and DONTs of mutlithreading
* What are some known thread issues?
* What care should be taken while using threads?
* What are good multithreading resources.
* Please provide examples.

In a multithreading environment you have to take care of **synchronization** so two threads doesn't clobber the state by simultaneously performing modifications. Otherwise you can have race conditions in your code (for an example see the [infamous Therac-25 accident](http://en.wikipedia.org/wiki/Therac-25).) You also have to **schedule** the threads to perform various tasks. You then have to make sure that your synchronization and scheduling doesn't cause a **deadlock** where multiple threads will wait for each other indefinitely.

**Synchronization**

Something as simple as increasing a counter requires synchronization:

counter += 1;

Assume this sequence of events:

* counter is initialized to 0
* thread A retrieves counter from memory to cpu (0)
* **context switch**
* thread B retrieves counter from memory to cpu (0)
* thread B increases counter on cpu
* thread B writes back counter from cpu to memory (1)
* **context switch**
* thread A increases counter on cpu
* thread A writes back counter from cpu to memory (1)

At this point the counter is 1, but both threads did try to increase it. Access to the counter has to be synchronized by some kind of locking mechanism:

lock (myLock) {

counter += 1;

}

Only one thread is allowed to execute the code inside the locked block. Two threads executing this code might result in this sequence of events:

* counter is initialized to 0
* thread A acquires myLock
* **context switch**
* thread B tries to acquire myLock but has to wait
* **context switch**
* thread A retrieves counter from memory to cpu (0)
* thread A increases counter on cpu
* thread A writes back counter from cpu to memory (1)
* thread A releases myLock
* **context switch**
* thread B acquires myLock
* thread B retrieves counter from memory to cpu (1)
* thread B increases counter on cpu
* thread B writes back counter from cpu to memory (2)
* thread B releases myLock

At this point counter is 2.

**Scheduling**

Scheduling is another form of synchronization and you have to you use thread synchronization mechanisms like events, semaphores, message passing etc. to start and stop threads. Here is a simplified example in C#:

AutoResetEvent taskEvent = new AutoResetEvent(false);

Task task;

// Called by the main thread.

public void StartTask(Task task) {

this.task = task;

// Signal the worker thread to perform the task.

this.taskEvent.Set();

// Return and let the task execute on another thread.

}

// Called by the worker thread.

void ThreadProc() {

while (true) {

// Wait for the event to become signaled.

this.taskEvent.WaitOne();

// Perform the task.

}

}

You will notice that access to this.task probably isn't synchronized correctly, that the worker thread isn't able to return results back to the main thread, and that there is no way to signal the worker thread to terminate. All this can be corrected in a more elaborate example.

**Deadlock**

A common example of deadlock is when you have two locks and you are not careful how you acquire them. At one point you acquire lock1 before lock2:

public void f() {

lock (lock1) {

lock (lock2) {

// Do something

}

}

}

At another point you acquire lock2 before lock1:

public void g() {

lock (lock2) {

lock (lock1) {

// Do something else

}

}

}

Let's see how this might deadlock:

* thread A calls f
* thread A acquires lock1
* **context switch**
* thread B calls g
* thread B acquires lock2
* thread B tries to acquire lock1 but has to wait
* **context switch**
* thread A tries to acquire lock2 but has to wait
* **context switch**

At this point thread A and B are waiting for each other and are deadlocked.

When should one use a semaphore and when should one use a conditional variable (CondVar) ?

Locks are used for mutual exclusion. When you want to ensure that a piece of code is atomic, put a lock around it. You could theoretically use a binary semaphore to do this, but that's a special case.

Semaphores and condition variables build on top of the mutual exclusion provide by locks and are used for providing synchronized access to shared resources. They can be used for similar purposes.

A condition variable is generally used to avoid busy waiting (looping repeatedly while checking a condition) while waiting for a resource to become available. For instance, if you have a thread (or multiple threads) that can't continue onward until a queue is empty, the busy waiting approach would be to just doing something like:

//pseudocode

while(!queue.empty())

{

sleep(1);

}

The problem with this is that you're wasting processor time by having this thread repeatedly check the condition. Why not instead have a synchronization variable that can be signaled to tell the thread that the resource is available?

//pseudocode

syncVar.lock.acquire();

while(!queue.empty())

{

syncVar.wait();

}

//do stuff with queue

syncVar.lock.release();

Presumably, you'll have a thread somewhere else that is pulling things out of the queue. When the queue is empty, it can call syncVar.signal() to wake up a random thread that is sitting asleep onsyncVar.wait() (or there's usually also a signalAll() or broadcast() method to wake up all the threads that are waiting).

I generally use synchronization variables like this when I have one or more threads waiting on a single particular condition (e.g. for the queue to be empty).

Semaphores can be used similarly, but I think they're better used when you have a shared resource that can be available and unavailable based on some integer number of available things. Semaphores are good for producer/consumer situations where producers are allocating resources and consumers are consuming them.

Think about if you had a soda vending machine. There's only one soda machine and it's a shared resource. You have one thread that's a vendor (producer) who is responsible for keeping the machine stocked and N threads that are buyers (consumers) who want to get sodas out of the machine. The number of sodas in the machine is the integer value that will drive our semaphore.

Every buyer (consumer) thread that comes to the soda machine calls the semaphore down() method to take a soda. This will grab a soda from the machine and decrement the count of available sodas by 1. If there are sodas available, the code will just keep running past the down() statement without a problem. If no sodas are available, the thread will sleep here waiting to be notified of when soda is made available again (when there are more sodas in the machine).

The vendor (producer) thread would essentially be waiting for the soda machine to be empty. The vendor gets notified when the last soda is taken from the machine (and one or more consumers are potentially waiting to get sodas out). The vendor would restock the soda machine with the semaphoreup() method, the available number of sodas would be incremented each time and thereby the waiting consumer threads would get notified that more soda is available.

The wait() and signal() methods of a synchronization variable tend to be hidden within thedown() and up() operations of the semaphore.

Certainly there's overlap between the two choices. There are many scenarios where a semaphore or a condition variable (or set of condition variables) could both serve your purposes. Both semaphores and condition variables are associated with a lock object that they use to maintain mutual exclusion, but then they provide extra functionality on top of the lock for synchronizing thread execution. It's mostly up to you to figure out which one makes the most sense for your situation.

That's not necessarily the most technical description, but that's how it makes sense in my head.

isn't it OK to call **pthread\_cond\_signal** or **pthread\_cond\_broadcast** methods without locking the mutex?

If you do not lock the mutex in the codepath that changes the condition and signals, you can lose wakeups. Consider this pair of processes:

**Process A:**

pthread\_mutex\_lock(&mutex);

while (condition == FALSE)

pthread\_cond\_wait(&cond, &mutex);

pthread\_mutex\_unlock(&mutex);

**Process B (incorrect):**

condition = TRUE;

pthread\_cond\_signal(&cond);

Then consider this possible interleaving of instructions, where condition starts out as FALSE:

Process A Process B

pthread\_mutex\_lock(&mutex);

while (condition == FALSE)

condition = TRUE;

pthread\_cond\_signal(&cond);

pthread\_cond\_wait(&cond, &mutex);

The condition is now TRUE, but Process A is stuck waiting on the condition variable - it missed the wakeup signal. If we alter Process B to lock the mutex:

**Process B (correct):**

pthread\_mutex\_lock(&mutex);

condition = TRUE;

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mutex);

...then the above cannot occur; the wakeup will never be missed.

(Note that you *can* actually move the pthread\_cond\_signal() itself after thepthread\_mutex\_unlock(), but this can result in less optimal scheduling of threads, and you've necessarily locked the mutex already in this code path due to changing the condition itself).

s there a relationship between kernel/user thread? some OS textbook said that "**maps** one(many) user thread to one(many) kernel thread",what does **map** means here?

When they say map, they mean that each kernel thread is assigned to a certain number of user mode threads.

Kernel threads are used to provide privileged services to applications (such as system calls ). The are also used by the kernel to keep track of what all is running on the system, how much of which resources are allocated to what process, and to do scheduling.

If your applications make a heavy use of system calls, the more user threads per kernel thread, the slower your applications will run, because the kernel thread will become a bottleneck, since all system calls will pass through it.

On the flip side though, if you're programs rarely use system calls (or other kernel services), you can assign a large number of user threads to a kernel thread without much performance penalty, other than overhead.

You can increase the number of kernel threads, but this adds overhead to the kernel in general, so while individual threads will be more responsive with respect to system calls, the system as a whole will become slower.

That is why it is important to find a good balance between the number of kernel threads and the number of user threads per kernel thread.

User threads are managed in userspace - that means scheduling, switching, etc. are not from the kernel.

Since, ultimately, the OS kernel is responsible for context switching between "execution units" - your user threads must be associated (ie., "map") to a kernel schedulable object - a kernel thread[+1].

So, given N user threads - you could use N kernel threads (a 1:1 map). That allows you to take advantage of the kernel's hardware multi-processing (running on multiple CPUs) and be a pretty simplistic library - basically just deferring most of the work to the kernel. It does, however, make your app portable between OS's as you're not directly calling the kernel thread functions. I believe that POSIX Threads ([PThreads](http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html)) is the preferred \*nix implementation, and that it follows the 1:1 map (making it virtually equivalent to a kernel thread). That, however, is not guaranteed as it'd be implementation dependent (a main reason for using PThreads would be portability between kernels).

Or, you could use only 1 kernel thread. That'd allow you to run on non multitasking OS's, or be completely in charge of scheduling. Windows' [User Mode Scheduling] is an example of this N:1 map.

Or, you could map to an arbitrary number of kernel threads - a N:M map. Windows has [Fibers](http://msdn.microsoft.com/en-us/library/ms682661%28VS.85%29.aspx), which would allow you to map N fibers to M kernel threads and cooperatively schedule them. A threadpool could also be an example of this - N workitems for M threads.

[+1] A process has at least 1 kernel thread, which is the actual execution unit. Also, a kernel thread must be contained in a process. OS's must schedule the thread to run - not the process.

Is it safe to read from a file as soon as write returns

Yes, it is safe to read immediately after writing. In a Unix-like system, the data is safely in the kernel buffer pool when a write() returns and will be returned to other processes that need to read the data. Similar comments apply when using O\_SYNC, O\_DSYNC, O\_FSYNC (which ensure that data is written to disk) and to Windows systems. Clearly, an asynchronous write will not be complete when theaio\_write() call returns, but it will be complete when the completion is signalled.

However, your problem arises because you are not ensuring that you have a single process or thread accessing the file at a time. You *must* ensure that you get serial access so that you don't get two processes (or threads) reading from the file at the same time. This is the 'lost update' problem in DBMS terms.

You need to ensure that only one process has access at a time. If your processes cooperate, you can use advisory locking (via fcntl() on POSIX systems). If your processes don't cooperate, or you're not sure, you may need to go for mandatory locking, or use some other technique altogether.

I have the following situation:

There is a thread that reads from a device with a fread call. This call is blocking as long as there is no data send from the device. When I stop this thread it remains hanging inside this thread.

Now I found the following inside the man page of fread:

**ERRORS**

On all systems that conform to the Single UNIX Specification, the fread() function sets errno as listed for the following conditions:

[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

That would mean there is a way to interrupt the call from a different thread. But I have no idea how. Can someone tell me how to send a signal to interrupt the fread call? And what signal I need to send?

|  |  |
| --- | --- |
| 7down vote | **1. Signals:**  Using signals, as many others pointed out, would work. However, as many others also pointed out, the approach has its disadvantages.  **2. Select():**  Using select() (or other multiplexing function), you can block waiting for data to arrive from more than one file descriptor, and specify a timeout.  Use the timeout to your advantage. Whenever select() returns, check a global variable to see if you must terminate. If you want immediate reaction, keep reading.  **3. Select() and pipes:**  Multiple fds means you can wait for data arriving through the device you mentioned and, say, a pipe.  Before you create the thread, create a pipe, and then have the thread block on select() monitoring both the device and the pipe. Whenever you want to unblock select whether the device has new data or not, send a byte down the pipe.  If select() tells you it unblocked due to data arriving through the pipe, you can clean up and terminate. Note this method is much more flexible than the signaling method, since you can, besides just using the pipe as a wake-up method, use it to pass useful information or commands.  **4. Select(), pipes and signals:**  If you are using multiple processes and don't want to/can't pass around a pipe, you can combine both solutions. Create a pipe and install a signal handler for, say, SIGUSR1. In the signal handler, send a byte down the pipe.  Whenever a process sends SIGUSR1, the handler will be called and unblock select(). By examining the fdsets, you will know it was for no other reason than your own program signaling itself. |

Recusrive versus non recursive mutex

<http://stackoverflow.com/questions/187761/recursive-lock-mutex-vs-non-recursive-lock-mutex/189778#189778>

Deadlocks are hard to find and very uncomfortable to remove.

How can I find error sources for deadlocks in my code? Are there any "deadlock patterns"?

In my special case, it deals with databases, but this question is open for every deadlock.

Update: This recent MSDN article, [Tools And Techniques to Identify Concurrency Issues](http://msdn.microsoft.com/en-us/magazine/cc546569.aspx), might also be of interest

Stephen Toub in the MSDN article [Deadlock monitor](http://msdn.microsoft.com/en-us/magazine/cc163352.aspx) states the following four conditions necessary for deadlocks to occur:

* A limited number of a particular resource. In the case of a monitor in C# (what you use when you employ the lock keyword), this limited number is one, since a monitor is a mutual-exclusion lock (meaning only one thread can own a monitor at a time).
* The ability to hold one resource and request another. In C#, this is akin to locking on one object and then locking on another before releasing the first lock, for example:

lock(a)

{

...

lock(b)

{

...

}

}

* No preemption capability. In C#, this means that one thread can't force another thread to release a lock.
* A circular wait condition. This means that there is a cycle of threads, each of which is waiting for the next to release a resource before it can continue.

He goes on to explain that the way to avoid deadlocks is to avoid (or thwart) condition four.

[Joe Duffy discusses several techniques](http://msdn.microsoft.com/msdnmag/issues/06/04/Deadlocks) for avoiding and detecting deadlocks, including one known as lock leveling. In lock leveling, locks are assigned numerical values, and threads must only acquire locks that have higher numbers than locks they have already acquired. This prevents the possibility of a cycle. It's also frequently difficult to do well in a typical software application today, and a failure to follow lock leveling on every lock acquisition invites deadlock.

Without keeping a list of current threads, I'm trying to see that a realtime signal gets delivered to all threads in my process. My idea is to go about it like this:

* Initially the signal handler is installed and the signal is unblocked in all threads.
* When one thread wants to send the 'broadcast' signal, it acquires a mutex and sets a global flag that the broadcast is taking place.
* The sender blocks the signal (using pthread\_sigmask) for itself, and enters a loop repeatedly calling raise(sig) until sigpending indicates that the signal is pending (there were no threads remaining with the signal blocked).
* As threads receive the signal, they act on it but wait in the signal handler for the broadcast flag to be cleared, so that the signal will remain masked.
* The sender finishes the loop by unblocking the signal (in order to get its own delivery).
* When the sender handles its own signal, it clears the global flag so that all the other threads can continue with their business.

The problem I'm running into is that pthread\_sigmask is not being respected. Everything works right if I run the test program under strace (presumably due to different scheduling timing), but as soon as I run it alone, the sender receives its own signal (despite having blocked it..?) and none of the other threads ever get scheduled.

Any ideas what might be wrong? I've tried using sigqueue instead of raise, probing the signal mask, adding sleep all over the place to make sure the threads are patiently waiting for their signals, etc. and now I'm at a loss.

**Edit:** Thanks to psmears' answer, I think I understand the problem. Here's a potential solution. Feedback would be great:

* At any given time, I can know the number of threads running, and I can prevent all thread creation and exiting during the broadcast signal if I need to.
* The thread that wants to do the broadcast signal acquires a lock (so no other thread can do it at the same time), then blocks the signal for itself, and sends num\_threads signals to the process, then unblocks the signal for itself.
* The signal handler atomically increments a counter, and each instance of the signal handler waits until that counter is equal to num\_threads to return.
* The thread that did the broadcast also waits for the counter to reach num\_threads, then it releases the lock.

One possible concern is that the signals will not get queued if the kernel is out of memory (Linux seems to have that issue). Do you know if sigqueue reliably informs the caller when it's unable to queue the signal (in which case I would loop until it succeeds), or could signals possibly be silently lost?

**Edit 2:** It seems to be working now. According to the documentation for sigqueue, it returns EAGAIN if it fails to queue the signal. But for robustness, I decided to just keep calling sigqueue untilnum\_threads-1 signal handlers are running, interleaving calls to sched\_yield after I've sentnum\_threads-1 signals.

There was a race condition at thread creation time, counting new threads, but I solved it with a strange (ab)use of read-write locks. Thread creation is "reading" and the broadcast signal is "writing", so unless there's a thread trying to broadcast, it doesn't create any contention at thread-creation.

raise() sends the signal to the current thread (only), so other threads won't receive it. I suspect that the fact that strace makes things work is a bug in strace (due to the way it works it ends up intercepting all signals sent to the process and re-raising them, so it may be re-raising them in the wrong way...).

You can probably get round that using kill(getpid(), <signal>) to send the signal to the current process as a whole.

However, another potential issue you might see is that sigpending() can indicate that the signal is pending on the process before all threads have received it - all that means is that there is at least one such signal pending for the process, and no CPU has yet become available to run a thread to deliver it...

Can you describe more details of what you're aiming to achieve? And how portable you want it to be? There's almost certainly a better way of doing it (signals are almost always a major headache, especially when mixed with threads...)