

Re: How to wait for multiple threads simultaneously?

Re: How to wait for multiple threads simultaneously?

Source: <http://linux.derkeiler.com/Newsgroups/comp.os.linux.development.apps/2008-02/msg00135.html>

- *From:* Rainer Weikusat <rweikusat@xxxxxxxxxxxx>
 - *Date:* Sun, 17 Feb 2008 20:12:49 +0100
-

David Schwartz <davids@xxxxxxxxxxxx> writes:

On Feb 15, 1:33 pm, David Schwartz <dav...@xxxxxxxxxxxx> wrote:

On Feb 15, 3:14 am, Rainer Weikusat <rweiku...@xxxxxxxxxxxx> wrote:

And this actually nicely illustrates why using established terms to express different meanings is a bad idea: Your idea of a condition variable is quite obviously not something which causes a thread to sleep until woken up, but some basically purposeless addition to a loop which is busy-waiting until some condition is true.

I honestly can't understand what you're talking about at this point.

DS

Actually, maybe I see what the issue is.

Not really.

You think that condition variables are tested in 'while' loops because of spurious wakeups and thundering herds. But again, that is not true. Condition variables are tested in 'while' loops because even with perfectly correct code, no thundering herds, and no spurious wakeups, another thread *still* can have consumed the event.

What I 'think' is basically grounded in usual synchronizations

Re: How to wait for multiple threads simultaneously?

Re: How to wait for multiple threads simultaneously?

primitives (or not-so-primitives) for [multi-processor] UNIX(*)-kernels. A 'condition variable' is the userland-equivalent of a sleep channel (UNIX(*)) or wait queue (Linux): A mechanism which thread can use to go to sleep (userland terminology: block on) until they are explicitly woken up by some other thread. A 'sleep channel' (the original UNIX(*)-kernel mechanism) was just an arbitrary memory address and processes sleeping on one would be hashed onto a set of wait queues until some other 'execution context' (eg an interrupt handler) called wakeup on this particular sleep channel, causing all processes enqueued for this particular sleep channel to be woken up. Depending on what was the reason for going to sleep, only some subset of the woken-up processes would be able to actually continue the activity which originally caused them to go to sleep, while the others would need to sleep again. This is exactly the condition you describe as 'other thread having consumed the predicate' and that IS a spurious wakeup.

Depending on how the code for a particular subsystem is designed, it is completely possible that a single wait-queue is used for multiple events, eg input or output is possible on some peripheral, if the author of the code considered the additional complication of multiple wait queue to be not worth the effort.

But nevertheless, the idea behind such a primitive is that the process/ thread will have to wait for 'a long time' (for a computer), and thus, it should not remain runnable and the scheduler should pick another process or thread to run in the meantime.

In other words, looping in the 'while' loop is still perfectly normal behavior.

Of course it is perfectly normal behaviour, taking the limitations of existing implementations and the desire to not needlessly complicate things to solve the hypothetical performance problem into account. But it is still merely an artefact of the implementation and not the 'essence' of the mechanism as it would ideally work (eg only wake a thread/ process when there actually is a reason to do so).

Consider a typical thread pool with a condition variable that threads block on when there are no jobs in the queue. Suppose one thread is working on a job while another thread puts a job on an empty queue. The thread that placed the job on the queue calls 'pthread_cond_signal' and one blocked thread is woken. Before that thread is scheduled, the thread that was working on a job may complete that job and return to its own 'while' loop, which it will leave immediately when it consumes the job.

Re: How to wait for multiple threads simultaneously?

I wouldn't use a condition variable for that, exactly to avoid the need for explicit looping. That's a nice task for a counting semaphore:

```
static sem_t posted_tasks, picked_tasks;
static struct task *tq; /* array */
static unsigned put, get, n_mask;
static pthread_mutex_t put_lock, get_lock;

[...]

static void pick_next_task(task_routine **r, void **arg)
{
    struct task *tp;

    mtx_lock(&get_lock);
    tp = tq + (get++ & n_mask);
    mtx_unlock(&get_lock);

    *r = tp->r;
    *arg = tp->arg;

    p_debug((" %s: %u: got %s", __func__, thread_id(), tp->name));

    up(&picked_tasks);
}

[...]

static void run_task(void *unused)
{
    (void)unused;

    while (1) {
        down(&posted_tasks);
        pick_and_run_task();
    }
}

[...]

void post_task(task_routine *r, void *arg, char const *name)
{
    struct task *tp;

    down(&picked_tasks);

    mtx_lock(&put_lock);
    tp = tq + (put++ & n_mask);
    mtx_unlock(&put_lock);
```

Re: How to wait for multiple threads simultaneously?

Re: How to wait for multiple threads simultaneously?

```
tp->r = r;
tp->arg = arg;
tp->name = name;

p_debug(("%s: %u: posted %s", __func__, thread_id(), name));

up(&posted_tasks);
}
[code (C) my employer and cited for educational purposes]

[...]
```

If you look at the POSIX definition of a "spin lock", nowhere does it require busy waiting. It simply requires that the lock function not return until the lock is acquired. It is perfectly reasonable to refer to acquiring such a lock as "spinning" even if the implementation does not busy wait.

This is presumably an omission, because a 'POSIX spin lock' would not be anyhow different from an ordinary mutex, then, and completely redundant.

A 'spin lock' is a synchronisation primitive intended for locks which are only held for very short times on a multiprocessor, specifically, for times which are short enough that the overhead of marking the process/ thread as no longer runnable, switching to a different 'execution context', marking the process as runnable again and switching back to it would be excessive when compared with the time the lock is actually held. Instead of going to sleep, the process/ thread 'spins' in a loop, trying to reacquire the lock until that succeeds. Combinations of both approaches exist, too, eg mutexes on which a thread spins for a short time and if it couldn't acquire the lock during this time, goes to sleep to avoid tying up a processor for no particular reason.

It is, of course, completely reasonable to describe the loop being part of the wakeup of a sleeping process/thread as 'spinning', too, if one only looks at the activity and ignores (or doesn't know) why a 'spin lock' is called 'a spin lock' to emphasize that it is something different from an ordinary mutual exclusion 'sleeping lock' (and something completely different from a sleep channel/ wait queue). Taking the existing 'other usage' in both literature (eg UNIX(*) Internal, The Design and Implementation of the xBSD Operating System) and code (Linux, *BSD, SMP-UNIX(*)) into account, this just isn't particularly sensible.