Locking wait_queues: Linux Kernel Implementation of pthread_cond_wait

Susan Paradiso

Professor Hugh Lauer
WPI – CS502
December 2011

1 Abstract

The linux system call wait_event_interruptible_locked() provides an autonomous wait-for-condition-then-wake-locked mechanism. This kernel system call is analogous to pthread's pthread_cond_wait() function. The wait_event_interruptible_locked() system call is based on linux's wait_queues, which are the underlying mechanism for several other lock/block features, including semaphores and completion variables.

2 Processes and Pthreads

A linux process is a single instance of code, its related data and memory space. Although many processes can simultaneously co-exist in Linux, the typical view from any one process is that it is the only process that exists. It has its own resources (CPU, memory, stack, etc), and other processes cannot access these resources intentionally nor accidentally.

These resources come from a shared pool, and the kernel manages the sharing and exclusive-use of the resources for processes. The kernel uses lock/block mechanisms to enable sharing of resources (the CPU or memory, for example), while preventing multiple processes from simultaneously accessing critical resources, like the scheduling queue or the data structure of allocated memory.

Pthreads are lightweight processes created and destroyed under user-code control. Pthreads share the same code, data and memory space as the parent (creating) process[1].

The pthreads/process relationship is analogous to the process/OS relationship, and a single process may have multiple pthreads. As these lightweight processes share the same memory space as their parent and sibling pthread(s), user code is responsible for ensuring that parent and sibling pthreads do not simultaneously modify the same data structures. Similar to the linux kernel, user code uses pthread lock/block mechanisms to manage accesses to shared resources.

3 Linux Blocking and wait_queues

An integral part of acquiring a lock is *blocking*: the process that does not get exclusive-access must block, or wait, until it gains access. For short-duration events, the process can hold the CPU and live-wait for the event (for example, live-wait for a lock to free). But for longer duration events, for example, for I/O events, this would waste CPU time. Instead, when a process encounters a block condition, it gives up the CPU and moves to a sleep-state until an event or condition is met, then it is moved back to the runnable state.

Note that the process itself is sleeping, so it cannot move itself back to the runnable state. Some other process must wake the blocked/sleeping process.

[1] A pthread also shares files, locks and sockets with its parent and sibling pthreads. However, each pthread has its own stack/stack pointer, registers and program counter.

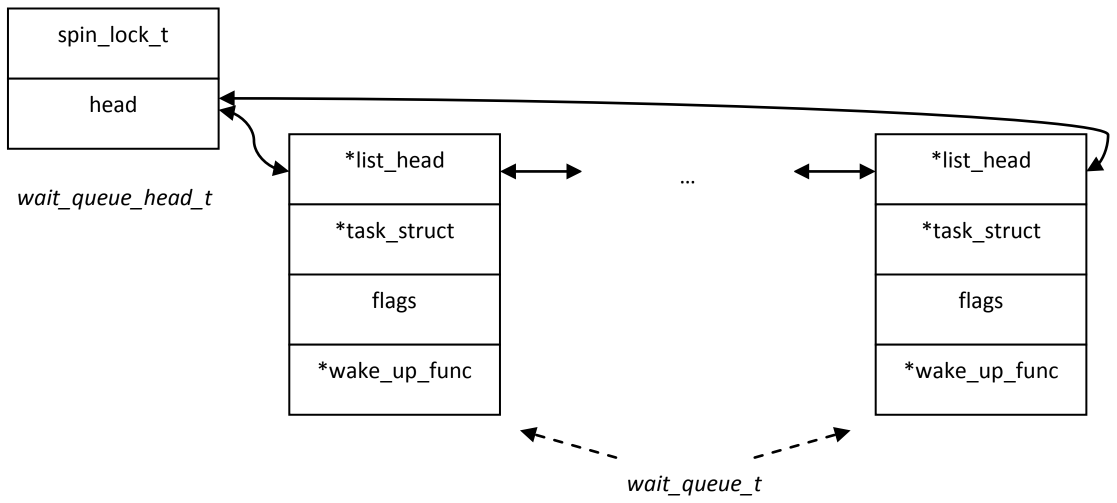
Linux has several mechanisms to support blocking/sleeping, most commonly semaphores and completion variables. Linux also has wait_queues, which are commonly used in drivers to manage I/O resources and delays.

3.1 wait_queues

Linux wait_queues provide the mechanism to manage (hold) blocked processes and to wake the blocked processes at some later time.

The typedef *wait_queue_t* is a simple data structure consisting of a pointer to a task_struct and pointers to the next/previous elements in the wait_queue (using struct list_head). It also contains flags and a pointer to the function to execute to do the actual waking.

The typedef *wait_queue_head_t* contains a spinlock_t and a struct list_head containing the list of tasks on this wait_queue. The spinlock is used to protect the wait_queue list.



The wait_event() family of functions[2] place the (executing) process on a wait_queue, set the process's state to not-runnable, then call schedule(). The call to schedule() causes the executing process to relinquish the CPU.

The wake_up() family of functions are called by another process (that is, a process not blocked on a wait_queue), and these function cause process(es) to be removed from the wait_queue and moved to the runnable state.

3.1.1 wait_event()/wake_up() Modes of Operation

The wait_queue entries can be configured as exclusive. In WQ_FLAG_EXCLUSIVE mode, wake_up() wakes up only one entry marked with WQ_FLAG_EXCLUSIVE, in FIFO order. Note that all entries not marked exclusive are also woken up, and a wait_queue can have a mix of exclusive and non-exclusive entries. The wait_event() functions specify whether the exclusive flag is present or not, the wake_up() functions respond to the flag.

[2] The full list of wait_event()/wake_up() functions appears later in this document.

The wait_event()/wake_up() functions have an 'interruptible' mode, which sends the process to sleep with TASK_INTERRUPTIBLE, versus TASK_UNINTERRUPTIBLE in non-interruptible mode. TASK_INTERRUPTIBLE means the task is sleeping but can be woken by a signal or by expiry of a timer. TASK_UNINTERRUPTBLE means the task is sleeping, except it cannot be woken up by signals or timers.

3.2 wait_event()/wake_up() Contention

An important feature of wait_queues is that they are race-condition free with respect to blocking/sleeping the process and waking-up the process (makelinux.net, 2011). That is, while a process is executing wait_event(), another process may execute wake_up() and the wait_event()-process could miss the wake_up(). The implementation of wait_event() prevents this race-condition. Wait_event() is implemented as such[3]:

```
wait_event() {

    prepare_to_wait();           //add 'current' to wait-queue, set state to !runnable
    if (!condition) schedule(); //check condition, possibly give up the CPU
    finish_wait();               //remove from wait-queue, set state to runnable

}
```

wait_event() enqueues the process to the wait_queue, then checks the *condition* and calls schedule() (thus giving up the CPU) if *condition* has not occurred. This prevents a process from waiting when *condition* has just occurred.

3.2.1 wait_event() Family 'Functions' Implementation

The wait_event() family of functions are actually implemented as macros (#defines). One reason to do this is for performance: the use of macros 'in-lines' the code and therefore removes a function call/return pair, which has some (modest) performance impact. But there is another way of inlining code, namely using the *inline* directive, and this is used in many other places in the kernel, but not with these functions.

The more important reason to use macros is because of the *condition* argument, and avoiding the race-condition between wait_event() and wake_up() described in the previous section. In the pseudo-code in the previous section, the *condition* argument is checked. The *condition* argument can be a test or variable, that is, it can be "(x == 123)", or some variable "x" that evaluates to true/false, or even "(x = atomic_read(&some_counter))". It is not a pointer to either a test or variable. If wait_event() was implemented as a function, then the *condition* argument would be passed into wait_event(), and it would be 'stale' when evaluated at:

```
if (!condition) schedule();
```

That is, it would be the value from when the wait_event() function was called and not the latest value of *condition*. In order for the contention case above to be solved, wait_event() would either need to be a pointer to *condition*, or a pointer to a function to evaluate *condition*. Alternatively,

[3] Pseudo-code based on OPENSuse linux 2.6.37.6-0.5.

by implementing the wait_event() family of calls as macros, the *condition* argument is always the latest value, and the contention case can be easily solved.

The wake_up() family of functions are also implemented as macros (#defines), but this is just done as a convenience wrapper to the underlying function ___wake_up, and not to implement any functionality as is done with the wait_event() macros.

3.3 wake_up()

To awaken a process, wake_up() calls the function in the wait_queue_t entry, which was set by wait_event(). A default is supplied for this function, default_wake_function(), and this default function calls sched.c's try_to_wake_up() function.

3.3.1 A Little Issue with Waking…

Processes move from being on the wait_queue to be woken up when another process calls wake_up(). When processes on the wait_queue are woken up, they go from a non-runnable state to a runnable state, and at some future time the scheduler will run them on the CPU. This can happen to multiple processes at the same time, if they are not marked exclusive. There is no guarantee when these waking-processes will be allocated CPU time nor in what order.

Once awoken, a process is not guaranteed that the wait_event() *condition* is true. That is, if a process was waiting for some variable 'state' to be equal to IDLE, this may not be true. Another process, one from the wait_queue or any other process in the system, may have changed the value of 'state' to something other than IDLE. In fact, the process that issues the wake_up() does not need to check for the *condition* specified in wait_event() before calling wake_up(). wake_up() can be called (blindly) and upon waking, the waking processes need to check the state of *condition* and act accordingly.

In practice, kernel code doesn't always check the value of *condition* upon return from wait_event(), because careful coding avoids the situation described above. For example, if wake_up() is always called only after setting *condition* to true, then one case of the condition-false-after-wait_event cannot occur.

3.4 wait_queue Uses in the Kernel

Wait_queues are frequently found in (I/O) drivers, where longer delays are commonplace. A process waiting for I/O is placed on a wait_queue while the I/O completes.

Wait_queues are used for the basis of the wait4() system call. Wait_queues are also used to implement completion variables. Struct completion is a wait_queue_head and a done flag.

Semephores use a wait_queue-like mechanism, and even share some lower-level functions related to waking as wake_up() uses, but semaphores are implemented independently of wait_queues.

4 Summary of Kernel and Pthread Lock/Block Mechanisms

The pthread standard defines various lock/block mechanisms to manage use of shared process resources. These mechanisms include (Barney, 2011):

pthread mutex variables;

pthread condition variables;

pthread barrier variables;

pthread read-write locks

Each of these mechanisms has supporting functions and structures/typedefs.

The linux kernel has a similar set of features to manage use of shared OS/system resources. These mechanisms include (Love, 2010):

mutexes;

spinlocks;

semaphores;

reader/writer locks;

big-kernel lock;

Again, each of these mechanisms has supporting functions/macros and structures/typedefs.

5 The Kernel’s pthread_cond_wait() Problem

For most pthread lock/block mechanisms, there is a similar or analogous kernel-process lock/block mechanism. One notable exception is the pthread_cond_wait() function.

5.1 pthread_cond_wait() Details

The pseudo-code for the pthread_cond_wait() function is:

```
//called with lock acquired
pthread_cond_wait() {
    do {
        release lock;
        wait-for-wake-up;
        acquire lock;
    } while (!condition)
}
//exits with lock acquired
```

The pthread_cond_wait() function is useful because, upon (non-error) exit, it returns with the locked acquired. It autonomously: acquires the lock, checks the *condition*, and if *condition*==true it exits with the lock still locked. Upon return from this function, the calling code is still in autonomous code because the lock is still set.

5.2 pthread_cond_wait() Usage Notes

Note that the *condition* variable itself must be of type pthread_cond_t. And "a condition variable must always be associated with a mutex, to avoid a race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it" (sourceware, 2011). The implementation of pthread_cond_wait() and pthread_cond_t does not enforce this rule through a compile-error or other notification. Poorly written code could change the condition variable without acquiring the lock (or perhaps acquiring the wrong lock) and the code will compile and run, albeit incorrectly.

Also, from the man-page for pthread_cond_wait (Berkley, 2011):

> *When using condition variables there is always a Boolean predicate, an invariant, associated with each condition wait that must be true if the thread should proceed. Spurious wakeups from the pthread_cond_wait(), pthread_cond_timedwait() or pthread_cond_reltimedwait_np() functions could occur. Since the return from pthread_cond_wait() or pthread_cond_timedwait() or pthread_reltimedwait_np() does not imply anything about the value of this predicate, the predicate should always be re-evaluated.*

In other words, upon returning from pthread_cond_wait(), the calling code must re-evaluate the predicate it requires before continuing on. Thus, pthread_cond_wait() is usually called from within a *while* or similar loop construct. For example (IBM, 2011):

```
rc = pthread_mutex_lock(&mutex);
while (!conditionMet) {
   printf("Thread blocked\n");
   rc = pthread_cond_wait(&cond, &mutex);
}
rc = pthread_mutex_unlock(&mutex);
```

Elsewhere in a different thread:

```
rc = pthread_mutex_lock(&mutex);
//The condition has occurred. Set the flag and wake up any waiting threads.
  conditionMet = 1;
  printf("Wake up all waiting threads...\n");
  rc = pthread_cond_broadcast(&cond);
rc = pthread_mutex_unlock(&mutex);
```

The pthread executing the pthread_cond_wait() does so with the mutex locked, inside a while-loop, looping until the predicate (*conditionMet*) is true. The thread executing the pthread_cond_broadcast() does so with the mutex locked, and sets *conditionMet* then issues the broadcast.

5.3 The Kernel's wait_event_locked()/wake_up_locked() Functions

Until recently, the kernel did not have a function analogous to pthread_cond_wait. The kernel had the above-mentioned wait_event() family of functions, but these functions did not leave a lock acquired upon exit, and *condition* was not necessarily true when a wait_event() completed. Also, the original wait_event() functions could not be called while holding a lock, because these functions cause the CPU to be released and the processes to block/sleep (the kernel is not allowed to block/sleep while holding a spin lock (Love, 2010)).

In 2010[4] (Nazarewicz, 2010), the kernel's wait_queue family of functions was expanded to include _locked versions. These _locked functions operate the same as the existing family of functions, but they are entered and exited with a spin_lock_t set. The wait_queue_head_t's spin-lock is used as the locked-entity, so no new lock is required.

5.4 wait_event_locked()

There are four wait_event locked functions: wait_event_interruptible_locked(), wait_event_interruptible_locked_irq(), wait_event_interruptible_exclusive_locked(), and wait_event_interruptible_exclusive_locked_irq().

The pseudo-code for wait_event_locked()[5] is:

```
//called with lock acquired
wait_event_locked() {
    if (condition) return;
    enqueue current to wait queue;
    do {
       release lock;
       schedule(); //releases CPU, sleeps 'currrent' process
       acquire lock;
    }  while (!condition)
    remove from wait_queue;
    set state to runnable;
}
//exits with lock acquired
```

The code is restructured from the wait_event() form to the wait_event_locked() form: a do-while loop is added so the lock can be released and re-acquired on either side of the blocking (schedule) call. The concept of the code is the same though, with the added knowledge that when the call returns, the lock is acquired *and* *condition* is true. Recall that *condition*-being-

[4] 2010 is the first full reference to wait_queue lock functions that I could find, but they were at least partially implemented as early as 2008, although not in an official patch/release.

[5] Pseudo-code based on OPENSuse linux 2.6.37.6-0.5, include/linux/wait.h, macro __wait_event_interruptible_locked.

true is not a guarantee with the (not locked) wait_event() functions, because some other process may have changed the state of *condition* before the waking process is able to evaluate it, or even lock-then-evaluate it. With the _locked functions, *condition* must be true and it is checked while lock is set.

However, the wait_event_locked() functions require that *condition* be changed (written) carefully, under protection of the same wait_queue's spin-lock, so that a waiting process that awakes and acquires the lock sees a valid and coherent value for *condition* with respect to the lock. A weakness in the current implementation is that nothing enforces the code to protect *condition* with the same spin-lock as the wait_queue. This is analogous to the weakness with the pthread_wait_cond code (see pthread_cond_wait() Usage Notes).

5.5 wake_up_locked()

There are two wake_up locked functions: wake_up_locked() and wake_up _locked_poll().

The pseudo-code for wake_up_locked()[6] is:

```
//called with lock acquired
wake_up_locked() {
        __wake_up_common();
}
//exits with lock still acquired
```

This is the same as wake_up(), except wake_up() sandwiches the call to __wake_up_common() between acquiring and releasing the wait_queue's lock. Wake_up_locked() doesn't change the lock state, so the unlock/lock is removed.

5.6 irq Variants

The wait_event_locked() functions support two spin lock variants, spin_lock()/spin_unlock() and spin_lock_irq()/spin_unlock_irq(). The variant used in wait_event_locked() must match the variant that was used to acquire the lock before wait_event_locked() was called.

There are no 'irq' variants for wake_up_locked() functions, because they do not change the state of the spin-lock.

5.7 exclusive Variants

The wait_event_locked() functions support two variants for how many processes to awaken: exclusive and non-exclusive. These operate analogously to the non-locked version, see wait_event()/wake_up() Modes of Operation (above) for more information. Note that in non-exclusive mode, all waiting processes will be awoken, but because of the spin-lock, only one newly-awoken process can execute the code following the wait_event() at one time.

There are no 'exclusive' variants for wake_up_locked() functions, because they read the exclusive-state flag from the wait_queue entry.

[6] Pseudo-code based on OPENSuse linux 2.6.37.6-0.5, include/linux/wait.h, macro wake_up_locked(), and kernel/sched.c __wake_up_locked() function.

5.8 interruptible Variants

All wait_queue _locked functions are interruptible. There are no non-interruptible modes. The interrupt/non-interruptible versions of the (non-locked) wait_queue functions affect how the task is put to sleep (TASK_INTERRUPTIBLE vs TASK_NONINTERRUPTIBLE mode). Non-interruptible mode is for use inside interrupt handlers, not for general driver and other kernel code. It appears there has been no need to offer non-interruptible, lockable wait_queue functions (yet), and there may never be, since interrupt handlers are highly-specialized, short-duration events.

The wait_event_locked() functions are all of the interruptible variant. The wake_up_locked() functions do not specify the interruptible variant, but they will affect all wait_queue processes, both interruptible and non-interruptible.

6 Kernel wait_queue Implementation Resources

6.1 Files

The kernel's wait_queue mechanism is implemented in three files, wait.h, wait.c and sched.c.

6.1.1 include/linux/wait.h

The file wait.h is the header files for wait_queues. This file contains the #defines, structures and function prototypes for most functions associated with wait_queues.

Note this file also contains the macros (#defines) for the wait_event() family of macro-functions. See wait_event() Family 'Functions' Implementation above, for information on why macros are used.

6.1.2 kernel/wait.c

The file wait.c contains the definition (code) of wait-related functions, including the prepare_to_wait() family of functions, as well as the finish_wait() function.

6.1.3 kernel/sched.c

The file sched.c contains the __wake_up_common() function and the functions ___wake_up() and __wake_up_locked(). It also contains default_wake_function().

6.2 Overview

Using wait_queues with locks is an easy way to synchronize multiple processes in the kernel where locking is needed around the blocking event or condition. The function is comparable to semaphores and also (obviously) to wait_queues (non-locking), but the ability to block and awaken autonomously with a lock-acquired is powerful. It appears to the calling code that the block was done with the lock held.

The wait_event()-locked functions are entered with the wait_queue's spin-lock acquired. The lock is released by wait_event() before the block occurs (meeting the requirement that spin-locks not be held while blocked). When the blocking code unblocks, the lock is re-acquired and the condition is re-checked. If the condition is not met, the code unlocks/blocks; if the condition is met, the wait_event() function returns with the lock acquired.

Wait_queues are represented by the wait_queue_head_t and wait_queue_t typedefs (structures), and are defined in include/linux.wait.h.

6.2.1 Terms

wq : wait_queue_head_t, passed by value to wait_event() macros and passed as a pointer to wake_up() functions
condition : a boolean expression, may be evaluated multiple times so it needs to be safe from side effects.

6.2.2 Initialization

A wait_queue can be declared statically:

```
DECLARE_WAIT_QUEUE_HEAD(wq);
```

or dynamically:

```
wait_queue_head_t wq;
init_waitqueue_head(&wq);
```

6.2.3 wait_event() Functions (macros):

These functions are for the most part autological (self-descriptive). The *locked* functions are italicized.

wait_event(wq, condition);
wait_event_timeout(wq, condition, timeout);
wait_event_interruptible(wq, condition);
wait_event_interruptible_timeout(wq, condition, timeout);
wait_event_interruptible_exclusive(wq, condition);
wait_event_interruptible_locked(wq, condition);
wait_event_interruptible_locked_irq(wq, condition);
wait_event_interruptible_exclusive_locked(wq, condition);
wait_event_interruptible_exclusive_locked_irq(wq, condition);

6.2.4 wake_up() Functions (macros):

wake_up(wq);
wake_up_nr(wq, num); //num == how many wake-one or wake-many processes to wake
wake_up_all(wq);
wake_up_locked(wq);
wake_up_interruptible(wq);
wake_up_interruptible_nr(wq, num);
wake_up_interruptible_all(wq);
wake_up_interruptible_sync(wq, m);
wake_up_poll(wq, m);

wake_up_locked_poll(wq, m);
wake_up_interruptible_poll(wq, m);
wake_up_interruptible_sync_poll(wq, m);

The _sync variants of the wake_up() functions are used in cases where the waker knows that it will be scheduled away soon, and the target (waking) process will not be migrated to another CPU. That is, the two processes are 'synchronized' with each other. This prevents needless bouncing between CPUs.

The _poll variants of the wake_up() functions support the *poll* (SystemV), *select* (BSD Unix) and *epoll* system calls. These functions are used to support non-blocking I/O operations. The data structures supporting poll/select/epoll include wait_queues as well as other state/defines/etc. A full description of the _poll variants is outside the scope of this document. Briefly, the _poll variants pass an opaque key (void *) to their (non-default) wake-up function. For further details on the poll/select/epoll functions, see Linux Device Drivers, Chapter 6: Advanced Char Driver Operations (Jonathan Corbet, 2005)

6.2.5 Pairing wait_event() and wake_up() calls

Typically, the variant of wait_event() and the variant of wake_up() are the same. That is, wake_up_interruptible() is used for processes waiting via wait_event_interruptible(). The exception is the lock functions: the wait_event_locked() functions all specify 'interruptible', but the wake_up_locked() functions do not, although the wake_up_locked() functions affect the interruptible processes also.

6.3 Example Usage: timerfd.c

The linux file fs/timerfd.c[7] uses the wait_event_interruptible_locked_irq() and wake_up_locked() functions. The details of timerfd is beyond the scope of this document, but briefly, file timerfd.c implements a timer that delivers timer expiration notifications via a file descriptor, unlike timer.c functions, which also deliver timer notifications, but through a (local) data structure. Timerfd.c has the advantage that select/poll/epoll functions may be used to monitor the file descriptor.

The data structure for the timerfd, *ctx*, contains a wait_queue, *wqh*, as well as time support structures, the *expired* flag and the *ticks* counter. The wait_queue's lock is used to protect *expired* and *ticks* as well as the wait_queue *wqh* itself.

6.3.1 timerfd_read()

The timer information is 'reported' to the calling code via a file-read operation, timerfd_read(), which uses wait_event_interruptible_locked_irq() to wait for the timer to change. After wait_event_..._locked() wakes up, the lock is still acquired and the code saves the number of *ticks* that have expired, set the *expired* flag, resets the *ticks* counter and the expired flag, then unlocks the lock.

[7] Code based on OPENSuse linux 2.6.37.6-0.5.

```
static ssize_t timerfd_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
        struct timerfd_ctx *ctx = file->private_data;
        ssize_t res;
        u64 ticks = 0;

        if (count < sizeof(ticks))
                return -EINVAL;
        spin_lock_irq(&ctx->wqh.lock);
        if (file->f_flags & O_NONBLOCK)
                res = -EAGAIN;
        else
                res = wait_event_interruptible_locked_irq(ctx->wqh, ctx->ticks);
        if (ctx->ticks) {
                ticks = ctx->ticks;
                if (ctx->expired && ctx->tintv.tv64) {
                        /*
                         * If tintv.tv64 != 0, this is a periodic timer that
                         * needs to be re-armed. We avoid doing it in the timer
                         * callback to avoid DoS attacks specifying a very
                         * short timer period.
                         */
                        ticks += hrtimer_forward_now(&ctx->tmr,
                                                     ctx->tintv) - 1;
                        hrtimer_restart(&ctx->tmr);
                }
                ctx->expired = 0;
                ctx->ticks = 0;
        }
        spin_unlock_irq(&ctx->wqh.lock);
        if (ticks)
                res = put_user(ticks, (u64 __user *) buf) ? -EFAULT: sizeof(ticks);
        return res;
}
```

6.3.2 timerfd_tmrproc()

The function timerfd_tmrproc() handles timer expirations. The function acquires the lock, and sets the *expired* flag and updates the *ticks* count. Any processes waiting on the wait_queue for the timer to expire are woken up via wake_up_locked(). The lock is then released.

```
/*
 * This gets called when the timer event triggers. We set the "expired"
 * flag, but we do not re-arm the timer (in case it's necessary,
 * tintv.tv64 != 0) until the timer is accessed.
 */
```

```
static enum hrtimer_restart timerfd_tmrproc(struct hrtimer *htmr)
{
        struct timerfd_ctx *ctx = container_of(htmr, struct timerfd_ctx, tmr);
        unsigned long flags;

        spin_lock_irqsave(&ctx->wqh.lock, flags);
        ctx->expired = 1;
        ctx->ticks++;
        wake_up_locked(&ctx->wqh);
        spin_unlock_irqrestore(&ctx->wqh.lock, flags);
        return HRTIMER_NORESTART;
}
```

6.3.3 Other timerfd Functions Using wqh.lock

Several other functions in timerfd.c that access *expired* or *ticks* lock their accesses with the wait_queue lock even though they do not use any wait_queue functions. The wait_queue lock is used to protect the data structure holding the wait_queue as well as the wait_queue itself.

7 mailbox.c Implemented with wait_queues

7.1 Weakness with Semaphores in mailbox.c

In project4, a process-level mailbox was implemented. The mailbox supports Send, Receive, and Manage(stop) functions. The assignment suggested we use semaphores, empty and full, and use empty/full up()/down() to control the sending and receiving of messages. This method has two related issues.

The first issue is that down() blocks, and like all things that block, it can't hold a lock while it is blocked. Thus, the code must release any locks before calling down(), and re-acquire the lock after the down() returns.

The second issue is that there is no up_all() operation. When a mailbox is being stopped or deleted, any senders/receivers blocked on a down() call must be unblocked and allowed to complete before the mailbox can be stopped/deleted. The stop/delete code does this by issuing up() operations to unblock all down()-waiters, which means it needs to know how many down() calls are blocked. Reference counts must be maintained, one for each semaphore, to count how many down() operations are currently blocked.

Even with the reference counts, the stopping function cannot accurately determine how many blocked senders/receivers there are, because the down() operation cannot be executed with a lock set. The pseudo-code for the down() logic of the send function is:

```
acquire lock;
tx_ref_cnt++;
release lock;
//what if the process switches here (A)?
```

```
down(); //may block
//what if the process switches here (B)?
acquire lock;
tx_ref_cnt--;
release lock;
```

Just before and just after the down() operation, tx-ref-cnt is inaccurate. In the 'before' case, it indicates the process is in down() code that is more accurately 'about-to-down()', but is not yet in down(). And in the 'after' case, tx_ref_cnt indicates that the process is still in down() code when it has returned, but not yet decremented tx_ref_cnt.

The implementation of the semaphore includes a spin-lock. Ideally it would be useful to have an up_all() function, which would lock the semaphore's spin-lock, issue an up() call for all blocked down() operations, then unlock the semaphore's spin-lock. This would remove the need for the reference counts around the two semephores.

Alternatively, if down() could be issued with the lock set, the calling code would use the down()'s spin-lock to protect the increment and decrement of the reference counts, making them autonomous with the down() operation.

7.2 Using wait_queues With Locks in mailbox.c

The mailbox code for project4 can be simplified if wait_queues are used, and further simplified if wait_queues with locks are used.

Using wait_queues removes one of the issues with the semaphore approach: it solves the up_all() problem. The wait_event() function can be used in non-exclusive mode, then the wake_up_all() function can be used to wake up all waiters, without the need for reference counts on send/receive.

Using locked variants of wait_queues remove the other issue with the semaphore approach: it allows the wait_queue's spin-lock to be used to protect all data structures in the mailbox. This removes the need for other locks.

7.3 mailbox.c ReVisited

I modified my mailbox.c[8] implementation to use wait_queues with locks. My original implementation of mailbox.c worked, and passed all of my tests as well as the professor's tests. My implementation involved several locks, and I was able to remove some locks and simplify the code by using wait_queues with locks.

SendMsg() required locking the task-list (tasklist_lock), and this is still required. All functions (send/receive/manage) require the mailbox_lock to protect access to the mailbox lock itself. This too is still required.

My original implementation had two other locks: one to protect 'state variables', like reference counters, and one to protect the message queue itself. These multiple locks allowed multiple

[8] New mailbox code submitted as part of this term project, including my test program.

processes to simultaneously access different parts of the mailbox data structure, improving performance at the cost of making the code more complicated. The two semaphores also have spin-locks built into them.

The wait_queue-with-locks version of the code replaces the state-lock, q-lock and semaphore-locks with a single lock, the wait_queue's lock. The wait_queue lock is used to protect the all mailbox data structures. The semaphores are not needed, replaced by wait_event_interruptible_locked() functions. The reference counts are also not needed, solved by wake_up(). The mailbox data structure is now treated as one data structure, and different parts of it cannot be accessed by different processes simultaneously as before. This does not impact performance/parallelism though, because there is less data to protect. The reference counts are gone, as are the semaphores.

The 'state' is also simplified. In my original mailbox implementation, I supported a 'stopping' state. Because of less parallelism and the autonomous nature of the wait_event_lock() function, the 'stopping' state is no longer needed. In the original code, send/receive could wake from their blocked-down() functions and the mailbox could be in a stopping state: not running but not yet fully stopped. The issue was that an unblocking receive function could be unblocked because a message had arrived or because of a stop's up(), an up() issued by a Manage()-stop. Now, the wake_up_locked() wakes-up all waiters and completes before any waiters can attain the lock and unblock. When a receiver does wake-up and get the lock, the mailbox data structure is coherent and accurate. The num_msgs count is accurate for the number of messages in the queue, and the receiver has the lock, so the receiver can read a message if present, or return empty if no message is present.

8 Application to Monitor Concept

A monitor is construct that encapsulates and controls access to a shared data structure. Monitors make all functions that affect the data structure be part of one critical section, and allow for well-defined invariants. This usually involves a lock/mutex, and the ability for an executing function to release the lock, block and re-acquire the lock when some condition occurs (see pthread_cond_wait() for example).

The wait_queue-locked family of functions supports the monitor concept. The wait_queue structure itself contains a spin-lock that can be used by all functions protecting the monitor data structures. The wait_queue-locked functions provide a method to release the lock and block-on-condition/event, then unblock and re-acquire the lock later when the condition is met. All data protected by the wait_queue's spin-lock and access functions become invariants and are easy to understand and code.

Wait_queues do not enforce proper coding and do not encapsulate the data in a structure that can *only* be accessed after the lock is attained. A poor/malicious coder can still access the protected data structure directly without the lock. However, the coder's, reviewer's and support's task are simplified because the single structure, the wait_queue with lock, provides both the lock and the ability to unlock/block-condition/unblock.

The new mailbox code I implemented with wait_queues-with-locks follows the monitor concept and is much cleaner and simpler to read/understand/support than my original mailbox code. The number of variables in the data structure has been reduced significantly. The invariants are obvious to understand, partially because the mailbox data structure itself is now simplified with fewer variables.

The invariants in the new wait_queue-locked mailbox implementation are:

When SendMsg()/RcvMsg()/ManageMailbox() are inactive (no one is executing them, or someone is on a wait_queue in the middle of executing one of these functions), and the process exists (no in create/destroy), the state of the mailbox is as such:

1. No locks are held;
2. The mailbox's msg_q (type list head) can be empty or will have a linked list of msg_t objects.
3. The atomic_t variable num_msgs accurately accounts for the number of objects in msg_q.
4. The state mbox_state is either running or stopped.
5. Any processes blocked on send (waiting for space in the mailbox) or receive (waiting for a message to arrive) are blocked on the wait_queue.

9 Bibliography

Barney, B. (2011). *POSIX Threads Programming*. Retrieved from Lawrence Livermore National Laboratory: https://computing.llnl.gov/tutorials/pthreads

Berkley. (2011). *Standard C Library Functions, pthread_cond_wait*. Retrieved from www.compute.cnr.berkelet.edu: http://compute.cnr.berkeley.edu/cgi-bin/man-cgi?pthread_cond_wait+3

IBM. (2011). *pthread_cond_wait()* -- *Wait for Condition*. Retrieved from publib.boulder.ibm.com: http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Fapis%2Fusers_78.htm

Jonathan Corbet, A. R.-H. (2005). *Linux Device Drivers.* New York: O'Reilly Media.

Love, R. (2010). *Linux Kernel Development.* New York: Addison-Wesley.

makelinux.net. (2011). *Blocking I/O*. Retrieved from makelinux.net: http://www.makelinux.net/ldd3/chp-6-sect-2

Nazarewicz, M. (2010, April 9). *LKML.org*. Retrieved from LKML.org malist archive: https://lkml.org/lkml/2010/4/9/264

sourceware. (2011). *sourceware.org*. Retrieved from www.sourceware.org: http://sourceware.org/pthreads-win32/manual/pthread_cond_init.html

10 Contents