Kernel APIs And Primitives

Travis Geiselbrecht edited this page on 21 Feb · 5 revisions

Wait Queues

See: include/kernel/wait.h

Wait Queues are a building block for higher level blocking primitives, but can be useful on their own. A wait queue is an entity upon which threads may block until another thread or interrupt handler causes them to stop blocking.

Note	The main thread spinlock must be held while manipulating a wait queue.

```
void wait_queue_init(wait_queue_t *wait);
status_t wait_queue_block(wait_queue_t *, lk_time_t timeout);
int wait_queue_wake_one(wait_queue_t *, bool reschedule, status_t wq_error);
int wait_queue_wake_all(wait_queue_t *, bool reschedule, status_t wq_error);
```

Mutexes

See: include/kernel/mutex.h

Mutexes are not usable from interrupt context, but can be used to implement traditional locking to protect data structures, etc, in thread context. They are non-recursive — you cannot acquire a mutex you already hold. Only the holding thread may release a mutex.

```
void mutex_init(mutex_t *);
status_t mutex_acquire_timeout(mutex_t *, lk_time_t);
status_t mutex_release(mutex_t *);
status_t mutex_acquire(mutex_t *m);
bool is_mutex_held(mutex_t *m);
```

Semaphores

See: include/kernel/semaphore.h

When `sem_wait()` ed upon, a semaphore's 'count' is decremented. If it became negative, the waiting thread will block. There are variants of wait that return failure instead of blocking (`sem_trywait()`), or fail after a timeout expires (`sem_timedwait()`), instead of blocking forever.

When `sem_post()` ed, a semaphore's 'count' is incremented, and if there are one or more threads blocked on it, one thread will unblock and return successfully from its `sem_wait()` call.

Like mutexes, semaphores are not usable from interrupt context.

```
void sem_init(semaphore_t *, unsigned int);
int sem_post(semaphore_t *, bool resched);
status_t sem_wait(semaphore_t *);
status_t sem_trywait(semaphore_t *);
status_t sem_timedwait(semaphore_t *, lk_time_t);
```

Timers

See: include/kernel/timer.h

Timers are used to setup one-shot or repeating callbacks in units of milliseconds.

These callbacks happen from interrupt context, so the available APIs are limited to things like signaling an event or unblocking a thread on a wait queue.

It is always safe to reprogram or cancel a timer from its own callback.

```
handler_return a_timer_callback(struct timer *, lk_time_t now, void *arg);


void timer_initialize(timer_t *);
void timer_set_oneshot(timer_t *, lk_time_t delay, timer_callback, void *arg);
void timer_set_periodic(timer_t *, lk_time_t period, timer_callback, void *arg);
void timer_cancel(timer_t *);
```

Events

See: include/kernel/event.h

Events provide a mechanism to signal threads from other threads or interrupts. They maintain a single piece of state: 'signaled' or 'unsignaled'. They come in two flavors: 'Plain' and 'AutoUnSignal' (the latter selected by passing `FLAG_AUTOUNSIGNAL` to `event_init()`).

'Plain' events will wake up any waiting threads when in the 'signaled' state and will continue to do so (no further `event_wait()` callers will block) until `event_unsignal()` is called.

'AutoUnSignal' events will wake up a single thread if any threads are waiting when signaled and they will remain in the 'unsignaled' state. If no threads are waiting, they will enter the 'signaled'

state until the next call to event_wait(), at which point the caller will not block and the state will atomically reset to 'unsignaled'. The result is precisely one thread will fall through before going back to 'unsignaled'.

```
#define EVENT_FLAG_AUTOUNSIGNAL 1
void event_init(event_t *, bool initial, uint flags);
status_t event_wait(event_t *e);
status_t event_wait_timeout(event_t *, lk_time_t);
status_t event_signal(event_t *, bool reschedule);
status_t event_unsignal(event_t *);
```

Threads

See: include/kernel/thread.h

Warning: many of the functions in here are for internal use of the kernel.

There are 32 priority levels. `DEFAULT_PRIORITY` is appropriate for typical threads. `LOW_PRIORITY` and `HIGH_PRIORITY` are relative to default priority. Threads that are set to real time will not be preempted by other threads of the same or lower priority when they are running. Use with caution.

Threads do not begin running until `thread_resume()` is called on them. Typically calls to `thread_create()` are followed immediately by `thread_resume()`. Threads that exit will wait, consuming resources like stack space until `thread_join()` is called to obtain their exit status, unless `thread_detach()` is called on them prior to exit (in which case they will silently exit and clean up). Threads may give up their quantum voluntarily by calling `thread_yield()`.

```
int a_thread_start_routine(void *arg);
```

Functions to create, start, and obtain exit status from a thread:

```
thread_t *thread_create(const char *name, thread_start_routine entry, void *arg, int
status_t thread_resume(thread_t *);
status_t thread_detach(thread_t *t);
status_t thread_detach_and_resume(thread_t *t);
status_t thread_join(thread_t *t, int *retcode, lk_time_t timeout);
```

Functions to temporarily or permanently stop executing a thread:

```
void thread_yield(void);
void thread_sleep(lk_time_t delay);
void thread_exit(int retcode);
```

Functions for a thread to modify its state:

```
void thread_set_name(const char *name);
void thread_set_priority(int priority);
status_t thread_set_real_time(thread_t *t);
```