LAB 6 - CSB

Q1 - What happens to a thread when it calls thread_exit? What about when it sleeps?

Exiting

- Thread is removed from the process.
- The thread count is decremented.
- `thread_switch` is called which changes state to `S_ZOMBIE`

Sleeping

- The thread state is changed to `S_SLEEP`
- Thread is added to the tail of the wait channel

Q2 - What function—or functions—handle(s) a context switch?

`thread_switch`

Q3 - What does it mean for a thread to be in each of the possible thread states?

```
/* States a thread can be in. */
typedef enum {
  S_RUN,    /* running */
  S_READY,  /* ready to run */
  S_SLEEP,  /* sleeping */
  S_ZOMBIE, /* zombie; exited but not yet deleted */
} threadstate_t;
```

Q4 - What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?

Turning off interrupts means the CPU can't be interrupted. The function `splhigh()` is used to turn interrupts off. It is important to turn off interrupts in thread subsystem code as an interrupt may stop the CPU's code in a state where it will cause a deadlock or not allow a thread to switch properly.

Q5- What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

The thread is removed from the wait channel and made runnable i.e `S_READY` in functions `wchan_wakeone` or `wchan_wakeall`. Both these functions call the `thread_make_runnable` function.

Q6 - What function (or functions) choose the next thread to run?

`thread_consider_migration` and `schedule` . Currently, `schedule` is not implemented. Both of these functions are called from function `hardclock`

Q7 - How is the next thread to run chosen?

Currently, the `head` of `cpu_runqueue` is chosen to be run next. `cpu_runqueue` is a LinkedList.

Q8 - What role does the hardware timer play in scheduling?

Switching threads in `round-robin` manner.

Q9 - What hardware-independent function is called on a timer interrupt?

`thread_yield`

Q10 - Describe how wchan_sleep and wchan_wakeone are used to implement semaphores.

The semaphore is initialised with some initial count, which is the number of threads that can be running at one time (usually through some critical section). The `P` function is called when a thread wants to request entry to the critical section. If the semaphore count > 0, the thread is allowed through and the count is decremented. Otherwise, the thread is put to sleep with `wchan_sleep()`. When function `V` is called count is incremented and one sleeping thread is awakened by the `wchan_wakeone` function.

Lock

```
// Lock
struct lock
{
      char *lk_name;
      volatile bool acquired;
      struct wchan* lk_wchan;
      struct spinlock lk_spn_lock;
      struct thread *lk_thr;
      HANGMAN_LOCKABLE(lk_hangman);
};

struct lock *
lock_create(const char *name)
{
  struct lock *lock;

  lock = kmalloc(sizeof(*lock));
  if (lock == NULL) {
    return NULL;
  }

  lock->lk_name = kstrdup(name);
  if (lock->lk_name == NULL) {
    kfree(lock);
    return NULL;
  }

  HANGMAN_LOCKABLEINIT(&lock->lk_hangman, lock->lk_name);

  // add stuff here as needed
  lock->acquired = false;
  lock->lk_thr = NULL;

  lock->lk_wchan = wchan_create(lock->lk_name);
  if (lock->lk_wchan == NULL) {
    kfree(lock->lk_name);
    kfree(lock);
    return NULL;
  }
  spinlock_init(&lock->lk_spn_lock);
  return lock;
}

void
lock_destroy(struct lock *lock)
{
  KASSERT(lock != NULL);
  KASSERT(lock->acquired != true);
  lock->lk_thr = NULL;
  kfree(lock->lk_name);
  spinlock_cleanup(&lock->lk_spn_lock);
  wchan_destroy(lock->lk_wchan);
```

```
  kfree(lock);
}

void
lock_acquire(struct lock *lock)
{
  KASSERT(lock != NULL);
  /* Call this (atomically) before waiting for a lock */
  HANGMAN_WAIT(&curthread->t_hangman, &lock->lk_hangman);
  spinlock_acquire(&lock->lk_spn_lock);
  while (lock->acquired) {
    wchan_sleep(lock->lk_wchan, &lock->lk_spn_lock);
  }
  lock->lk_thr = curthread;
  lock->acquired = true;
  KASSERT(lock->lk_thr == curthread);
  spinlock_release(&lock->lk_spn_lock);
  /* Call this (atomically) once the lock is acquired */
  HANGMAN_ACQUIRE(&curthread->t_hangman, &lock->lk_hangman);
}

void
lock_release(struct lock *lock)
{
  KASSERT(lock != NULL);
  KASSERT(lock_do_i_hold(lock));
  spinlock_acquire(&lock->lk_spn_lock);
  lock->lk_thr = NULL;
  lock->acquired = false;
  wchan_wakeone(lock->lk_wchan, &lock->lk_spn_lock);
  spinlock_release(&lock->lk_spn_lock);
  /* Call this (atomically) when the lock is released */
  HANGMAN_RELEASE(&curthread->t_hangman, &lock->lk_hangman);
}

bool
lock_do_i_hold(struct lock *lock)
{
  KASSERT(lock != NULL);
  return (lock->acquired && lock->lk_thr == curthread);
}
```

CV

```
// CV
struct cv
{
      char *cv_name;
      struct wchan* cv_wchan;
      struct spinlock cv_spn_lock;
```

```
};

struct cv *
cv_create(const char *name)
{
  struct cv *cv;

  cv = kmalloc(sizeof(*cv));
  if (cv == NULL) {
    return NULL;
  }

  cv->cv_name = kstrdup(name);
  if (cv->cv_name==NULL) {
    kfree(cv);
    return NULL;
  }

  cv->cv_wchan = wchan_create(cv->cv_name);
  if (cv->cv_wchan == NULL) {
    kfree(cv->cv_name);
    kfree(cv);
    return NULL;
  }
  spinlock_init(&cv->cv_spn_lock);
  return cv;
}

void
cv_destroy(struct cv *cv)
{
  KASSERT(cv != NULL);
  wchan_destroy(cv->cv_wchan);
  spinlock_cleanup(&cv->cv_spn_lock);
  kfree(cv->cv_name);
  kfree(cv);
}

void
cv_wait(struct cv *cv, struct lock *lock)
{
  KASSERT(cv != NULL);
  KASSERT(lock_do_i_hold(lock));
  spinlock_acquire(&cv->cv_spn_lock);
  lock_release(lock);
  wchan_sleep(cv->cv_wchan, &cv->cv_spn_lock);
  spinlock_release(&cv->cv_spn_lock);
  lock_acquire(lock);
}

void
cv_signal(struct cv *cv, struct lock *lock)
{
  KASSERT(cv != NULL);
```

```
  KASSERT(lock_do_i_hold(lock));
  spinlock_acquire(&cv->cv_spn_lock);
  wchan_wakeone(cv->cv_wchan, &cv->cv_spn_lock);
  spinlock_release(&cv->cv_spn_lock);
}

void
cv_broadcast(struct cv *cv, struct lock *lock)
{
  KASSERT(cv != NULL);
  KASSERT(lock_do_i_hold(lock));
  spinlock_acquire(&cv->cv_spn_lock);
  wchan_wakeall(cv->cv_wchan, &cv->cv_spn_lock);
  spinlock_release(&cv->cv_spn_lock);
}
```