- **Mutual Exclusion vs. Synchronization**
  - Mutual exclusion: Ensures that a shared object can only be accessed by one thread at a time
    - Provided by: Spinlock, binary semaphore, blocking lock
  - Synchronization: Establishes timing/order relationships among threads. For example:
    - Thread *A* should always run after thread *B*
      - Stage *i + 1* can't start until stage *i* has completed
    - Only *K* threads should be running at the same time
      - Producer/consumer problem with a bounded buffer
    - A thread should only execute if a particular condition is true
      - Wait until the stock price is above $X before selling the stock
      - Determining when a car/thread should be entering an intersection
    - Provided by: Counting semaphore and condition variable

- **Wait Channel Review**

  - Wait channel is a structure defined as follows:

```
struct wchan {
    const char *wc_name; /* name for this channel */
    struct threadlist wc_threads; /* list of waiting threads */
    struct spinlock wc_lock; /* lock for mutual exclusion */
};
```

  - *wchan_sleep* adds the current thread to the thread list
  - *wchan_wakeone* moves the first thread from the list to the end of the ready queue
  - *wchan_wakeall* moves all of the threads from the list to the ready queue
  - They must acquire and release *wc_lock*

- *wchan_wakeone and wchan_wakeall* acquire and release wc_lock on their own:

```
void
wchan_wakeone(struct wchan *wc)
{
   struct thread *target;

   /* Lock the channel and grab a thread from it */
   spinlock_acquire(&wc->wc_lock);
   target = threadlist_remhead(&wc->wc_threads);
```

- *wchan_sleep* does not acquire *wc_lock* on its own
  - Must explicit call *wchan_lock* before calling *wchan_sleep*

- This is because:

  - Wait channels are almost never used on their own
    - Too easy to miss a wakeup signal due to unpredictable thread ordering
  - Usually paired with some kind of state variable
    - A counter in the case of a semaphore
  - State variable is protected by its own spinlock
  - Must ensure the state variable is modified together with the wait channel
    - Thread must therefore always be holding at least one of the two locks
  - Must also ensure that all locks are released before going to sleep
    - Locks must be held to wake up a sleeping thread
  - Therefore, if *wchan_sleep* acquires *wc_lock* on its own, then it is difficult to:
    1. Release the state variable's lock
    2. while holding wc_lock and
    3. before the thread goes to sleep

- **Semaphore Review**
  - A semaphore counter should never be negative (not even temporarily)
  - P() must therefore block when the counter value is 0
  - Important details of **our** P() implementation:

```
1  P(struct semaphore* sem) {
2     spinlock_acquire(&sem->sem_lock);
3     while (sem->sem_count == 0) {
4         wchan_lock(sem->sem_wchan);
5         spinlock_release(&sem->sem_lock);
6         wchan_sleep(sem->sem_wchan);
7         spinlock_acquire(&sem->sem_lock);
8     }
9     sem->sem_count--;
10    spinlock_release(&sem->sem_lock);
11 }
```

  - Does not provide FIFO ordering. Example:
    - Semaphore count is 1 after waking up a sleeping thread
    - Another thread can decrement it back to 0 before the previously sleeping thread gets to run
  - Must check the semaphore count after waking up
    - *While* loop instead of an *if* statement at line 3
  - *sem_lock* must be released after acquiring the wait channel lock and before *wchan_sleep*