Lecture 22, Mar 8, 2024

Advanced Locking

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

- *Condition variables* work like semaphores; they maintain queues of threads
  - `wait` adds the calling thread to the queue for the condition variable, unlocks the mutex, and blocks
    - Calls to `wait` must already have acquired the mutex
    - One mutex can be used to protect multiple condition variables
  - When another thread calls `signal` or `broadcast`, if the thread is selected, it will get unblocked, tries to lock the mutex, and returns from `wait` if the mutex is successfully locked
    - `signal` wakes up any thread waiting, `broadcast` wakes up all of them
  - The mutex is used to protect variables that are a part of some more complex condition
  - We usually use `while` to check the condition instead of just `if` to account for the possibility of the condition updating before wakeup/locking
- Semaphores are a special case of condition variables, protecting an integer, going to sleep when the value is 0 and waking up when the value is greater than 0
  - One can be implemented with the other but it can get complex
  - Condition variables are favoured for more complex conditions since it improves code readability
- Example: producer-consumer with condition variables

```
pthread_mutex_t mutex;
int nfilled;
pthread_cond_t has_filled;
pthread_cond_t has_empty;

void producer() {
    // produce data
    pthread_mutex_lock(&mutex);
    while (nfilled == N) {
        pthread_cond_wait(&has_empty, &mutex);
    }
    // fill a slot
    ++nfilled;
    pthread_cond_signal(&has_filled);
    pthread_mutex_unlock(&mutex);
}

void consumer() {
    pthread_mutex_lock(&mutex);
    while (nfilled == 0) {
        pthread_cond_wait(&has_filled, &mutex);
    }
    // empty a slot
    --nfilled;
    pthread_cond_signal(&has_empty);
```

```
    pthread_mutex_unlock(&mutex);
    // consume data
}
```

- *Granularity* is the size of our critical sections; do we lock large sections or divide it into multiple smaller locks on smaller sections?
  - Locking can have overhead (memory, init/destruction, acquire/release time), which increases with the number of locks
  - More granular locks can increase performance through more parallelization, but increases the potential for deadlocks
- More locks increases the possibility of *deadlocks*, when threads are waiting forever
  - Deadlock conditions:
    1. Mutual exclusion
    2. Hold and wait – after acquiring a lock, attempting to acquire another lock
    3. No preemption – can't take locks away
    4. Circular wait – waiting for a lock held by another process
  - Example: two threads both need 2 locks; thread 1 acquires lock 1 first and tries to acquire lock 2, in the meantime thread 2 acquires lock 2 first and tries to acquire lock 1; now both are waiting on each other
    * This example can be prevented by enforcing order of locking and unlocking
    * Use a function that always locks them in the same order, and unlocks them in the opposite order
    * Another fix is to use `trylock` for the second one; if it doesn't succeed, give up the first lock for some time and try again