Lecture 21, Mar 6, 2024

Semaphores

- How can we ensure some fixed order of execution between threads?
- *Semaphores* are shared values between threads/processes that are used for signaling
  - They have a `value` that is an unsigned integer, which can be initialized to anything
    - Setting this to some initial number sets the number of `wait`s that can occur at a time without `post`
  - Two fundamental operations:
    - `wait`: decrement the value atomically; if the value is 0, it waits until a `post` increments the value again before decrementing it and returning
    - `post`: increment the value atomically
- Semaphores are offered in the `<semaphore.h>` library
- Use `int sem_init(sem_t *sem, int pshared, unsigned int value);` to initialize a semaphore
  - `pshared` specifies whether the semaphore should be shared between forked processes
  - Use `sem_destroy(sem_t *sem);` to destroy
- Use `int sem_wait(sem_t *sem);` and `int sem_trywait(sem_t *sem);` for wait and `int sem_post(sem_t *sem);` for post
- If we want to make one line always execute before another, we can use a semaphore initialized to 0
  - Call `wait` before the line that executes second, which cannot return until `post` is called by the other thread
  - Call `post` after the line that executes first, to indicate that the line has been run
  - If we want a third line to execute after the previous two, we cannot reuse the same semaphore because we can't control whether the second line or third line runs first (even if we `post` twice or initialize to 1)
    - This would require a second semaphore
  - If we initialized the semaphore to 1 instead, the first thread won't block when it `wait`s initially

```c
static sem_t sem; /* New */
void* print_first(void* arg) {
    printf("This is first\n");
    sem_post(&sem); /* New */
}
void* print_second(void* arg) {
    sem_wait(&sem); /* New */
    printf("I'm going second\n");
}
int main(int argc, char *argv[]) {
    sem_init(&sem, 0, 0); /* New */
    /* Initialize, create, and join threads */
}
```

- Semaphores can be used like mutexes; instead of lock we just use `wait` and instead of unlock we use `post`, and initialize the value to 1
  - This is often a bad idea since it depends on the initialization of the semaphore, which could be far from the code that actually uses it
- Example: suppose we have a circular buffer, which producers write to and consumers read from; all consumers share an index and all producers share an index
  - The producer shouldn't write to the buffer if it's full
  - The consumer shouldn't read from the buffer if it's empty
  - To ensure this, use a semaphore to track the number of empty slots (for the producers) and another to track the number of full slots (for the consumers)

```c
void init_semaphores() {
    sem_init(&empty_slots, 0, buffer_size);
    sem_init(&filled_slots, 0, 0);
```

```
}
void producer() { while (/* ... */) {
    /* spend time producing data */
    sem_wait(&empty_slots);
    fill_slot();
    sem_post(&filled_slots); /* New */
} }
void consumer() { while (/* ... */) {
    sem_wait(&filled_slots); /* New */
    empty_slot();
    sem_post(&empty_slots);
    /* spend time consuming data */
} }
```