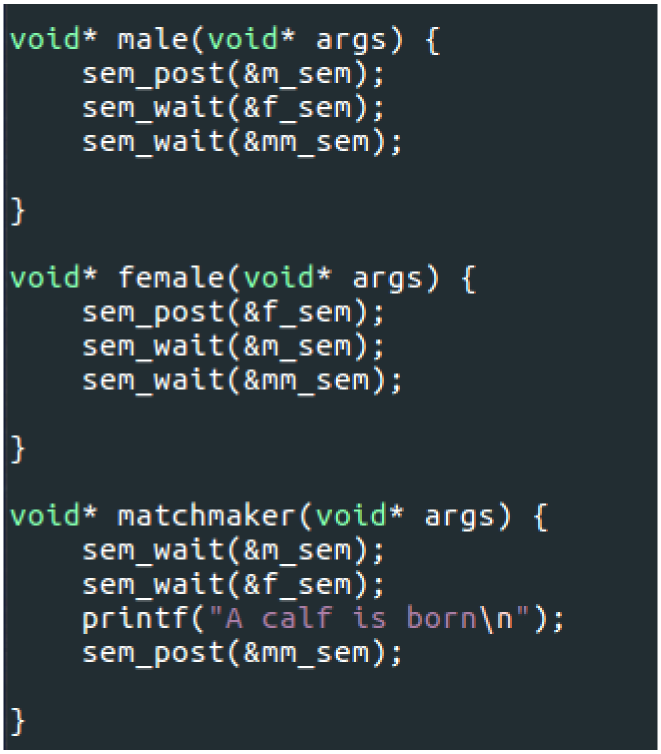
Lucas Dachman

Problem set #3

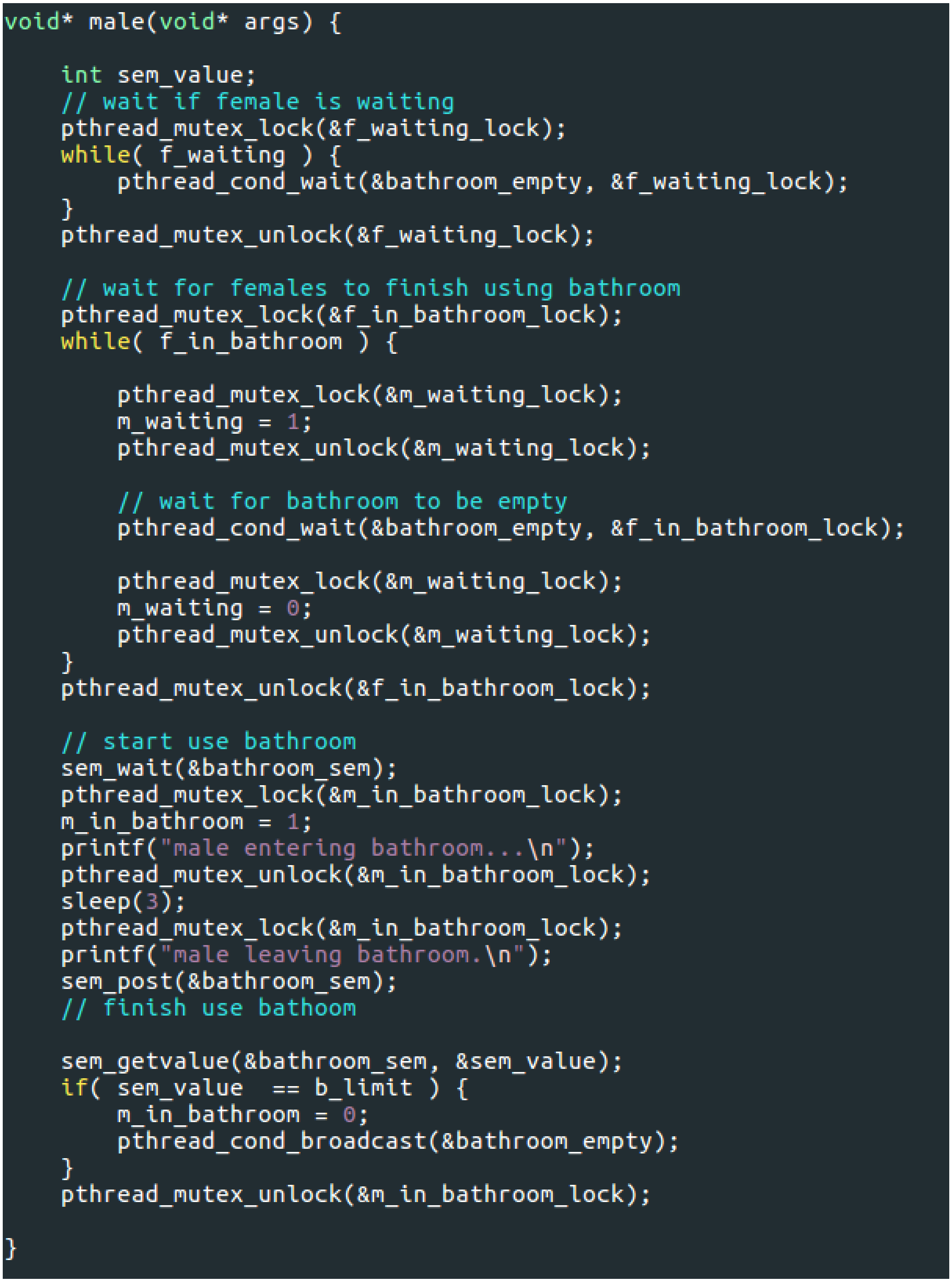
1. The swap() function is not thread safe. The function does not guarantee mutual exclusion. Multiple threads can execute the function and access global variables and shared data which can result in undefined behavior.
2. Swap is not reentrant because it uses a global variable temp which may be changed during the interrupt. The function also accepts pointers y and z whose dereferenced values are changed in the function. This alters the state of the program outside the function which is not a characteristic of reentrant code.
3. In the first Readers/Writers solution, starvation can happen if multiple readers arrive sequentially. The third Readers/Writers solution solves this problem with the readBlock. The readBlock in the writer function blocks any readers from acquiring the wrt lock. The writers will not starve because the write function will only block when there are readers pending.
4.



```
void* male(void* args) {
    sem_post(&m_sem);
    sem_wait(&f_sem);
    sem_wait(&mm_sem);

}

void* female(void* args) {
    sem_post(&f_sem);
    sem_wait(&m_sem);
    sem_wait(&mm_sem);

}

void* matchmaker(void* args) {
    sem_wait(&m_sem);
    sem_wait(&f_sem);
    printf("A calf is born\n");
    sem_post(&mm_sem);

}
```

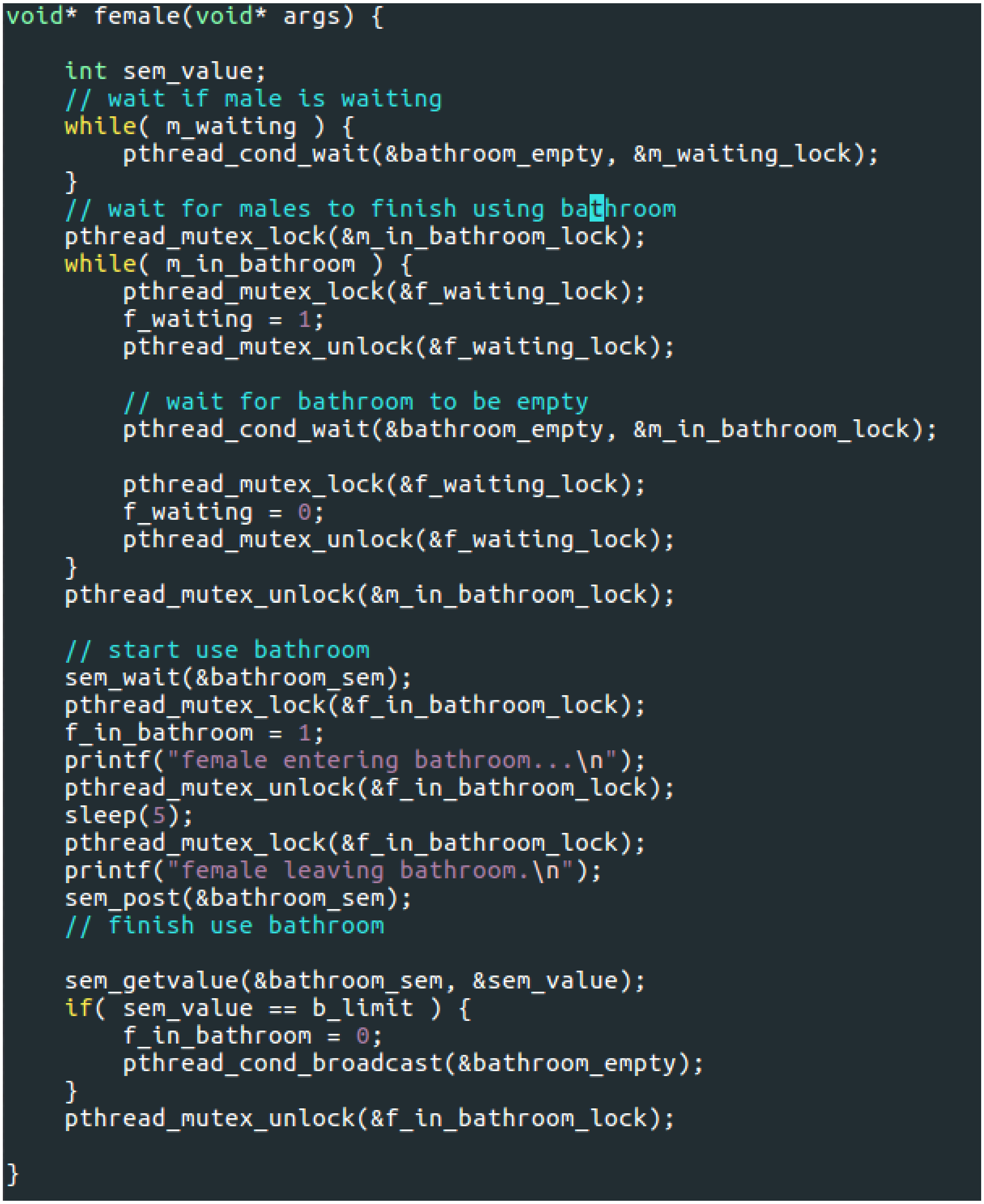
Lucas Dachman

5.



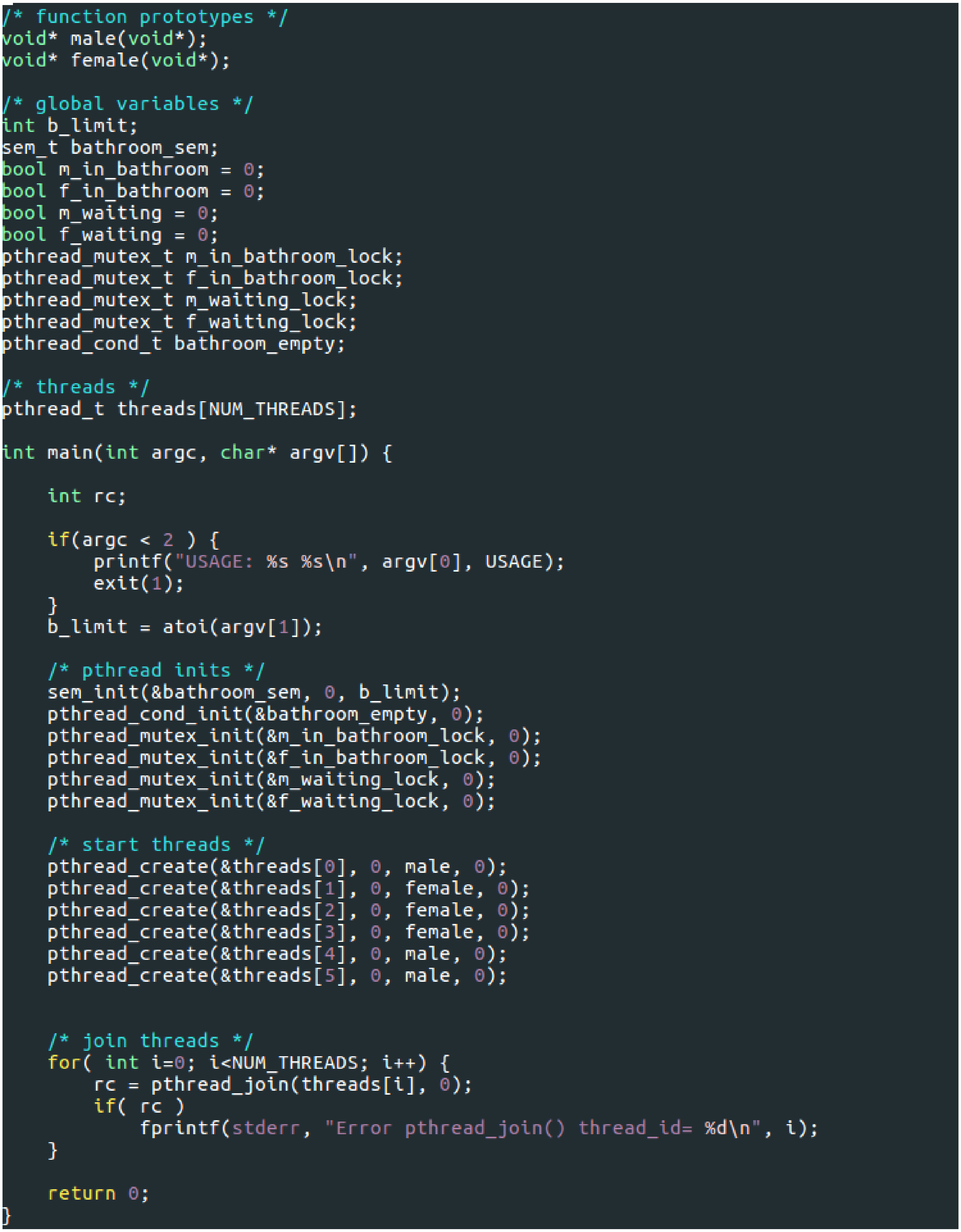
```
void* male(void* args) {

    int sem_value;
    // wait if female is waiting
    pthread_mutex_lock(&f_waiting_lock);
    while( f_waiting ) {
        pthread_cond_wait(&bathroom_empty, &f_waiting_lock);
    }
    pthread_mutex_unlock(&f_waiting_lock);

    // wait for females to finish using bathroom
    pthread_mutex_lock(&f_in_bathroom_lock);
    while( f_in_bathroom ) {

        pthread_mutex_lock(&m_waiting_lock);
        m_waiting = 1;
        pthread_mutex_unlock(&m_waiting_lock);

        // wait for bathroom to be empty
        pthread_cond_wait(&bathroom_empty, &f_in_bathroom_lock);

        pthread_mutex_lock(&m_waiting_lock);
        m_waiting = 0;
        pthread_mutex_unlock(&m_waiting_lock);
    }
    pthread_mutex_unlock(&f_in_bathroom_lock);

    // start use bathroom
    sem_wait(&bathroom_sem);
    pthread_mutex_lock(&m_in_bathroom_lock);
    m_in_bathroom = 1;
    printf("male entering bathroom...\n");
    pthread_mutex_unlock(&m_in_bathroom_lock);
    sleep(3);
    pthread_mutex_lock(&m_in_bathroom_lock);
    printf("male leaving bathroom.\n");
    sem_post(&bathroom_sem);
    // finish use bathoom

    sem_getvalue(&bathroom_sem, &sem_value);
    if( sem_value  == b_limit ) {
        m_in_bathroom = 0;
        pthread_cond_broadcast(&bathroom_empty);
    }
    pthread_mutex_unlock(&m_in_bathroom_lock);

}
```

Lucas Dachman



```
void* female(void* args) {

    int sem_value;
    // wait if male is waiting
    while( m_waiting ) {
        pthread_cond_wait(&bathroom_empty, &m_waiting_lock);
    }
    // wait for males to finish using bathroom
    pthread_mutex_lock(&m_in_bathroom_lock);
    while( m_in_bathroom ) {
        pthread_mutex_lock(&f_waiting_lock);
        f_waiting = 1;
        pthread_mutex_unlock(&f_waiting_lock);

        // wait for bathroom to be empty
        pthread_cond_wait(&bathroom_empty, &m_in_bathroom_lock);

        pthread_mutex_lock(&f_waiting_lock);
        f_waiting = 0;
        pthread_mutex_unlock(&f_waiting_lock);
    }
    pthread_mutex_unlock(&m_in_bathroom_lock);

    // start use bathroom
    sem_wait(&bathroom_sem);
    pthread_mutex_lock(&f_in_bathroom_lock);
    f_in_bathroom = 1;
    printf("female entering bathroom...\n");
    pthread_mutex_unlock(&f_in_bathroom_lock);
    sleep(5);
    pthread_mutex_lock(&f_in_bathroom_lock);
    printf("female leaving bathroom.\n");
    sem_post(&bathroom_sem);
    // finish use bathroom

    sem_getvalue(&bathroom_sem, &sem_value);
    if( sem_value == b_limit ) {
        f_in_bathroom = 0;
        pthread_cond_broadcast(&bathroom_empty);
    }
    pthread_mutex_unlock(&f_in_bathroom_lock);

}
```

Lucas Dachman



```
/* function prototypes */
void* male(void*);
void* female(void*);

/* global variables */
int b_limit;
sem_t bathroom_sem;
bool m_in_bathroom = 0;
bool f_in_bathroom = 0;
bool m_waiting = 0;
bool f_waiting = 0;
pthread_mutex_t m_in_bathroom_lock;
pthread_mutex_t f_in_bathroom_lock;
pthread_mutex_t m_waiting_lock;
pthread_mutex_t f_waiting_lock;
pthread_cond_t bathroom_empty;

/* threads */
pthread_t threads[NUM_THREADS];

int main(int argc, char* argv[]) {

    int rc;

    if(argc < 2 ) {
        printf("USAGE: %s %s\n", argv[0], USAGE);
        exit(1);
    }
    b_limit = atoi(argv[1]);

    /* pthread inits */
    sem_init(&bathroom_sem, 0, b_limit);
    pthread_cond_init(&bathroom_empty, 0);
    pthread_mutex_init(&m_in_bathroom_lock, 0);
    pthread_mutex_init(&f_in_bathroom_lock, 0);
    pthread_mutex_init(&m_waiting_lock, 0);
    pthread_mutex_init(&f_waiting_lock, 0);

    /* start threads */
    pthread_create(&threads[0], 0, male, 0);
    pthread_create(&threads[1], 0, female, 0);
    pthread_create(&threads[2], 0, female, 0);
    pthread_create(&threads[3], 0, female, 0);
    pthread_create(&threads[4], 0, male, 0);
    pthread_create(&threads[5], 0, male, 0);


    /* join threads */
    for( int i=0; i<NUM_THREADS; i++) {
        rc = pthread_join(threads[i], 0);
        if( rc )
            fprintf(stderr, "Error pthread_join() thread_id= %d\n", i);
    }

    return 0;
}
```