Some more C functions for you:
`sigprocmask pthread_sigmask pthread_self() atexit sigaction`
The big problem: How to implement the mutex lock

Hardware CPU instruction simplified solution

('*Atomic_Exchange'* swaps values at two addresses as an *uninterruptable* operation)

```
typedef p_mutex_t int;
pthread_mutex_init(p_mutex_t* m)        { *m = 1; }
pthread_mutex_lock(p_mutex_t* m)        { int local=0;
                                          do {
                                                ATOMIC_EXCHANGE(m, &local);
                                            } while(!value);
                                        }

pthread_mutex_unlock(p_mutex_t* m)      { *m = 1; }
```

C-Code Candidate # 0 (Review) Protect our critical section with a mutex. But how should it work!?

```
pthread_mutex_lock(p_mutex_t* m)        { while(m->lock) {}; m->lock = 1;}
pthread_mutex_unlock(p_mutex_t* m)      { m->lock = 0; }
```

Problems?

Psuedo code Candidate # 1

wait until your flag is lowered raise my flag *// Do Critical Section stuff* lower my flag	wait until your flag is lowered raise my flag *// Do Critical Section stuff* lower my flag

// Threads do other stuff and then will repeat at sometime in the future

Problems with 1b? Fix?

Candidate #2

raise my flag wait until your flag is lowered *// Do Critical Section stuff* lower my flag	raise my flag wait until your flag is lowered *// Do Critical Section stuff* lower my flag

// Threads do other stuff and then will repeat at sometime in the future

Problems with 2?

Candidate #3

wait until my turn (turn==id?) *// Do Critical Section stuff* turn = *yourid*	wait until my turn (turn==id?) *// Do Critical Section stuff* turn = *yourid*

// Threads do other stuff and then will repeat at sometime in the future

Problems with 3?

Three desirable properties of the solution to the Critical Section Problem?

```
//Ex 1 Make these functions thread-safe
// int thread_mutex_lock(pthread_mutex_t *mutex); ("p_m_lock")
// int pthread_mutex_unlock(pthread_mutex_t *mutex); ("p_m_unlock") will be useful
// Compile with -pthread

// Create a mutex that is ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

float sensor[2];
float old_vals[2];

void on_sensor_update(float x,float y) {
  // memcpy - fastest way to copy memory regions that do not overlap.
  memcpy(  old_vals, sensor, sizeof( sensor ) );
  sensor[0]=x;
  sensor[1]=y;
}

float moved2() {
  float dx = (sensor[0] - old_vals[0]);
  float dy = (sensor[1] - old_vals[1]);
  return dx*dx + dy*dy;
}
//#Ex 2
// Use a counting semaphore to ensure a maximum of 20 threads are running at a time. Threads that cannot acquire a
music pass must wait (block) until one is released.
// hint: int sem_init(sem_t *sem, int pshared, unsigned int value);
// int sem_destroy(sem_t *sem); "The effect of destroying a semaphore upon which other threads are currently
blocked is undefined."
// int sem_post(sem_t *sem);
// int sem_wait(sem_t *sem); will be useful

sem_t s;

void init() { ?

void acquireMusicPass() { ?

void releaseMusicPass() { ?

//Ex3 Carefully explain when and how the following code can copy more than size+1 bytes to the target address
// when used with more than one thread. Your answer should include the interleaving of the two threads' actions.
char *buffer = calloc(1,1);
size_t size =1;
void append(char c) { buffer = realloc(buffer, ++size); buffer[size-2] = c; buffer[size-1]='\0'; }
```