Goals

In this MP, you will:

- learn about multithreaded programming in C
- create thread-safe data structure using mutex, condition variable, etc.
- implement a wallet structure that holds resources
Multithreading
Thread

A thread is a single sequential flow of control within a program.

A program can have multiple threads running concurrently.
Implement wallet

In this MP, you will create a wallet data structure that will be accessed by multiple threads at the same time.

Thread 1:
1. Add 10 🍎
2. Sub 5 ☘️
3. Sub 2 ⭐️

Thread 2:
1. Add 7 ☘️
2. Sub 10 ⭐️
3. Add 1 ✨
Synchronization

Threads should be synchronized to avoid critical resource use conflicts

Race conditions happen when an operation touches a piece of shared memory at the same time as another thread

Critical section: a section of code that can only be executed by one thread at a time if the program is to function correctly.
## Race Condition

A wallet 🎩 contains 10 🍃

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>access 🍃 (= 10)</td>
<td>access 🍃 (= 10)</td>
</tr>
<tr>
<td>🍃 += 5 (10 + 5 = 15)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>🍃 += 10 (10 + 10 = 20)</td>
</tr>
<tr>
<td></td>
<td>🍃 = 20</td>
</tr>
<tr>
<td>🍃 should be 25! (10 + 10 + 5)</td>
<td></td>
</tr>
</tbody>
</table>
Mutex

Ensure only one thread is inside the critical section at one time

- `pthread_mutex_init` - create a new mutex in the “unlocked” state
- `pthread_mutex_lock` - lock the mutex; if the mutex is already locked by another thread, block execution until the mutex is unlocked
- `pthread_mutex_unlock` - unlock the mutex
- `pthread_mutex_destroy` - destroy the mutex
Wallet resource

A user will interact with your wallet by adding/subtracting resources to/from it.

You must not allow the wallet to ever go negative. The function must wait until there are enough resources to subtract from.

Thread 1:
Sub 50 ✨ Blocked! (contain 10 ✨)
Wallet resource

A user will interact with your wallet by adding/subtracting resources to/from it.

You must not allow the wallet to ever go negative. The function must wait until there are enough resources to subtract from.

Thread 1:
Sub 50 ✨ Blocked!
(contain 10 ✨)

Thread 2:
Add 100 ✨ Proceed
Wallet resource

A user will interact with your wallet by adding/subtracting resources to/from it.

You must not allow the wallet to ever go negative. The function must wait until there are enough resources to subtract from.

Thread 1:

Sub 50 ✨ Proceed (contain 110 ✨)
Avoid Busy Waiting

A naive approach: repeatedly check if the condition is satisfied in a loop before proceeding with its execution.

It is considered bad practice because:

1. errors may occur due to race conditions
2. system resources are wasted

// DON'T DO THIS!
while (condition not met) {
    sleep for a little
    wake up and check again
}
Condition Variable

Condition variables allow a set of threads to sleep until woken up

- `pthread_cond_init` - create a new condition variable
- `pthread_cond_wait` - release mutex and cause the calling thread to block on the condition variable
- `pthread_cond_signal` - unblock at least one thread that is blocked on the condition variable
Condition Variable

Condition variables allow a set of threads to sleep until woken up

- *pthread_cond_broadcast* - unblock all threads that are blocked on the condition variable
- *pthread_cond_destroy* - destroy the condition variable
Spurious Wakeup

Occasionally, a waiting thread may appear to wake up for no reason. This is called a spurious wakeup.

It usually happens due to race condition, where another thread changes the condition before the waiting thread finally runs.

You want to call `pthread_cond_wait` on the thread again if that happens.

```c
// mutex is locked...
while(condition not met)
{
  pthread_cond_wait();
}
// condition is met
```
structs in wallet

In your *lib/wallet.h*:

- `wallet_t` - maintain the state of a wallet
- `wallet_resource` - represent the resource in a wallet

Add any additional variable you may need

Example 🎒: 10 🍀 → 2 💍 → 1 🎫
functions in wallet

Implement these functions in *lib/wallet.c*:

- **wallet_init** - initialize the wallet
  - the wallet starts out empty, with 0 of all resources
- **wallet_get** - return the amount of a given resource
  - ensure accesses to your wallet are properly synchronized
functions in wallet

Implement these functions in `lib/wallet.c`:

- `wallet_change_resource` - change the amount of a resource by a certain `delta`
  - the resource amount cannot go negative
  - must wait until the request can be satisfied (e.g. another thread add to the resource)
- `wallet_destroy` - destroy a wallet and free any memory associated with it
Question