Kevin Van
CECS 326
Project 2

Breakdown Of The Problem

- Five philosophers are represented with 5 threads
- Each philosopher needs two forks in order to eat
- Alternate between thinking and eating (thinking when picking up forks and making sure they have 2, and eating when they have 2)
- Forks can either be available or taken

Design

- Threads (pthreads): I made each philosopher a thread that simulates thinking and eating. They interact with the forks
- Mutex Lock: I used mutex lock to protect the fork array in order to prevent race conditions when being picked up by philosophers
- Condition Variables: I used condition variables to signal when philosophers can pick up forks.

Functions

- pickup_forks(int philosopher_number)
    - The left fork is assigned to the philosopher's number and the right fork is found with (philosopher_number + 1) % NUM_PHILOSOPHERS. Philosophers will be able to identify left and right forks.
    - Used pthread_mutex_lock(&mutex) to make sure only one philosopher can access the forks array and fork_owner array one at a time.
    - forks[left_fork] == 1 or forks[right_fork] == 1 checks to see if fork is already taken else it must wait until available

- If/once they are available, forks[left_fork] and forks[right_fork] are set to 1 to indicate that they are taken
- We then print the fork status using print_fork_status()
- Finally we unlock the mutex using pthread_mutex_unlock(&mutex) to allow other philosophers to access the forks array and fork_owner array

- return_forks(int philosopher_number)
  - Same as the mutex lock with pickup_forks, we lock it to make sure only one philosopher can interfere when returning the forks
  - Similar to the pickup_forks, left and right are 0 when available, and -1 to show that no philosopher is holding the forks anymore
  - pthread_cond_signal(&cond_var[i]) signals the all the philosophers to check if they can pick up the forks
  - Finally we unlock the mutex
- void * philosopher(void* arg)
  - This function is used to simulate the behavior of a philosopher in the problem
  - Int philosopher_number = *((int*) arg); is the unique identifier passed as an argument to the thread. Pointer → Int, so we can dereference to get the philosopher number
  - Used Srand to generate different sequence
  - Rest is to simulate the actions of the philosopher and use the other functions above
  - NULL indicates that it is finished

Main

- Initialize the philosopher threads and forks

- Create a thread for each philosopher in order to run the philosopher function
- Wait for all threads to complete using pthread_join
- When we are finished the program ends

Video

https://www.youtube.com/watch?v=w7xIbUw8FL0