CSCI 4100 – Assignment 8 – Threads and Condition Variables

Learning Outcomes

Implement a multithreaded simulation using threads, locks, and condition variables

Required Reading

Saltzer & Kashoek 5.5 - 5.6

Instructions

For this programming assignment you are going to implement a simulation of Dijkstra's solution to the Dining Philosophers problem using threads, locks, and condition variables. See the lecture notes for PoCSD 5.2 for a description of the problem.

Dijkstra's Solution

Edsgar Dijkstra's original solution to the Dining Philosophers problem used semaphores, but it can be adapted to use similar mechanisms:

- Each philosopher is in one of three states: `THINKING`, `HUNGRY`, or `EATING`.
- Every philosopher starts out in the `THINKING` state.
- When a philosopher is ready to eat, her state is changed to `HUNGRY`.
- Before she can eat, she must test to see if it is safe to do so. If either of her neighbors is in the `EATING` state, it is not safe for her to eat and she must wait until the situation changes, at which point her state is changed to `EATING`.
- When a philosopher is done eating, her state is changed to `THINKING` and she must test to see if either of her neighbors is in the `HUNGRY` state and it is now safe for them to eat. If it is, she must notify that philosopher that it is now safe to start eating.

Note that the same test is used to assure two different requirements of the problem:

1. **Safety**: a philosopher only eats if it is safe to do so.
2. **Liveness**: no philosopher should go hungry if it is safe to eat.

The Dining Room

I have implemented a structure called `dining_room` that can be used to create a monitor-like object that manages a simulation of the Dining Philosophers problem.

It contains the following fields:

- `num_phils` - an integer representing the number of philosophers.
- `num_cycles` - an integer representing the number of times each philosopher tries to eat.
- `phil_state` - an array of values of type `p_state`, one for each philosopher, each of which has a value of `THINKING`, `HUNGRY`, or `EATING` depending on the state of that philosopher.
- `table_lock` - a lock to control access to the shared data in `phil_state`.
- `safe_to_eat` - an array of condition variables, one for each philosopher, each of which corresponds to the event when it is safe for that philosopher to eat.
- `phil_threads` - an array of threads, one for each philosopher.
- `phil_args` - an array of structures of type `p_args`, each of which contains three fields:
  - `phil_num` - the ID of the philosopher.
  - `num_cycles` - the number of times that philosopher should try to eat.
  - `room` - a pointer to the `dining_room` structure to use as a monitor.

I have implemented the following utility functions in the `diningRoom.c` file:

- `init_dining_room` - takes a pointer to an existing `dining_room` structure, and parameters representing the number of philosophers and the number of cycles and initializes the fields of the structure.
- `left_neighbor` - returns the ID of the left neighbor of a philosopher.
- `right_neighbor` - returns the ID of the right neighbor of a philosopher.
- `display_headings` - displays column headings for a table of state changes.
- `display_states` - displays the current state of each philosopher for a table of state changes.
- `think` - simulates the philosopher thinking.
- `eat` - simulates the philosopher eating.
- `start_philosopher` - the function to be used to start a philosopher thread. It uses information passed in a `p_args` structure to repeatedly do the following:
  1. think
  2. grab the forks
  3. eat
  4. release the forks

You must implement the following functions in the `dining_room.c` file:

- `void run_simulation(dining_room *room)` - starts a simulation of the Dining Philosophers problem:
    1. Display the headings for a table of state changes and a row of the table containing the philosophers' initial states.
    2. Start the thread for each philosopher.
    3. Wait for each philosopher's thread to complete.
- `int test(dining_room *room, int phil)` - (private) checks to see if it is safe for a philosopher to eat:
    1. If the state for `phil` is `HUNGRY` and neither of her neighbors is in the `EATING` state, set her state to `EATING`, display the current states of the philosophers and return true (1).
    2. Otherwise return false (0).
    3. The table lock must be acquired before this function is called.
    4. This function should not do anything with locks or condition variables.
- `void grab_forks(dining_room *room, int phil)` - simulates a philosopher picking up forks:
    1. Acquire the table lock.
    2. Set the state for `phil` to `HUNGRY`.
    3. Display the current states of the philosophers.
    4. Test to see if it is safe to eat.
    5. If it is not safe to eat, wait on the condition variable for `phil` in the `safe_to_eat array`.
    6. Release the table lock.
- `void release_forks(dining_room *room, int phil)` - simulates a philosopher putting down forks:
    1. Acquire the table lock.
    2. Set the state for `phil` to `THINKING`.
    3. Display the current states of the philosophers.
    4. Test to see if it is safe for each of `phil`'s neighbors to eat.
    5. If it is, notify the philosopher using the appropriate condition variable in the `safe_to_eat` array.
    6. Release the table lock.

Starting a Simulation

I have also provided the file `dpsim.c`, which contains a `main` function. This function takes two command line arguments, creates a `dining_room` structure using those values, and calls the `run_simulation` member function to start a simulation. See the **Compiling and Running Your Code** section below for more information.

POSIX Condition Variables

The POSIX library does not strictly speaking support monitors, but it does support using condition variables with locks to simulate monitors.

To create and use a condition variable you must do the following:

1. Create a variable of type `pthread_cond_t`. This is your condition variable.
2. To make the current thread wait on the condition variable use the following function call:

   ```
   pthread_cond_wait(&my_condition_variable, &my_lock)
   ```

   Note that the lock must be acquired before calling this function in order for it to work properly. The function will automatically release the lock before suspending the thread and reacquire the lock before resuming the thread.
3. To make the current thread notify a single thread waiting on the condition, use the following function call:

   ```
   pthread_cond_signal(&my_condition_variable)
   ```

   Technically, the lock does not have to be acquired to call this function, but it is a good idea to acquire the lock anyway to make sure the thread scheduling works predictably.

Compiling and Running Your Code

To create an executable file called `dpsim` use the following command:

```
gcc -lpthread -o dpsim dpsim.c dining_room.c
```

To run your code with 5 philosophers that try to eat 10 times, type:

```
./dpsim 5 10
```

If it is working correctly it should display a table showing the current state of each philosopher each time any philosopher's state changes.

A correct implementation should satisfy the safety and liveness conditions described above (remember the first and last philosophers are neighbors too.)

What to Hand In

Your source files should have comments at the top listing your name, CSCI 4100, Assignment 8, and a brief explanation of what the program does. Download `dining_room.c` to your local machine, and upload it to the D2L dropbox for Assignment 8.