BIRYANI SERVING

Jivitesh Jain, 2018101092, UG-2, CSE

Overview

The following is a multithreaded simulation which simulates a Biryani cooking and serving system in which the stakeholders - robot chefs, automated serving tables and the students are represented as threads. The threads use mutex-locks to stay synchronised as they communicate and augment each other through the course of the simulation.

```
Enter number of chefs, tables and students: 2 3 10
Running simulation.
CHEF 0 COOKING
CHEF 1 COOKING
TABLE 1 WAITING FOR BIRYANI
TABLE 2 WAITING FOR BIRYANI
TABLE 0 WAITING FOR BIRYANI
STUDENT 8 ARRIVED AND HUNGRY
STUDENT 1 ARRIVED AND HUNGRY
STUDENT 9 ARRIVED AND HUNGRY
...
STUDENT 2 FINISHED EATING
STUDENT 5 ASSIGNED SLOT ON TABLE 0 AND SERVED BIRYANI
STUDENT 5 FINISHED EATING
Simulation Done.
```

Implementation

The simulation uses a thread for every chef, table and student. Information about each thread (chef, table or student) is stored in a dynamically allocated struct, pointers to which are passed around to share information between threads.

The chef threads prepare Biryani vessels and conditionally wait until the number of Biryani vessels is greater than 0. Thus each Chef has a mutex lock and a conditinal variable.

The tables busy wait and randomly check the chefs, looking for one which has vessels left. They acquire it's mutex, decrement it's vessel count and conditionally signal the chef.

A similar process happens between tables that are ready to serve and the students waiting in line. The tables conditionally wait on their own conditional variables while they have slots left, and the students busy wait on the tables - continuously looking for one which has slots left - sending it a signal if they find one.

The Data Structures

```
typedef struct Chef {
    int id;
    pthread_t tid;

    pthread_mutex_t protect;
    pthread_cond_t cv_table;

    time_t cook_time;
    int left_vessels;
    int vessel_cap;
} Chef;

typedef struct Table {
    int id;
    pthread_t tid;

    int state;

    struct Chef** chefs;
    int num_chefs;

    int left_vessel_cap;

    int total_slots;
    int left_slots;

    struct Foodie* foodies[TABLE_SLOTS_LIMIT];
```

```
    pthread_mutex_t protect;
    pthread_cond_t cv_foodie;

} Table;

typedef struct Foodie {
    int id;
    pthread_t tid;

    time_t arrival_time;

    struct Table** tables;
    int num_tables;
} Foodie;
```

Each chef and table have a mutex lock and conditional variable - the former is used to protect their shared varaiables while the latter allows them to wait on those shared variables.

The Threads and Functions

Chefs

```
void* chef_run(void* args) {
    Chef* self = (Chef*)args;

    while (true) {
        printf(ANSI_CYAN "CHEF %d COOKING\n" ANSI_DEFAULT, self->id);

        self->vessel_cap = VESSEL_CAPACITY_OFFSET + rand() % VESSEL_CAPACITY_LI
MIT;

        self->cook_time = COOKING_TIME_OFFSET + rand() % COOKING_TIME_LIMIT;
        sleep(self->cook_time);

        pthread_mutex_lock(&(self->protect));
```

```
        self->left_vessels = CHEF_CAPACITY_OFFSET + rand() % CHEF_CAPACITY_LIMI
T;

        printf(ANSI_CYAN "CHEF %d READY WITH %d VESSELS OF BIRYANI WITH A CAPAC
ITY OF %d STUDENTS EACH\n" ANSI_DEFAULT, self->id, self->left_vessels, self->ve
ssel_cap); // written inside to protect left_vessels

        // mutex unlocked by cond_wait in biryani ready

        biryani_ready(self);
    }
}
```

The Chef takes a random amount of time to generate a vessel of Biryani. It updates its data under the protection of its mutex and then calls `biryani_ready()` which conditionally waits until all vessels have been picked up by tables before preparing more. This conditional waiting using `pthread_cond_wait()` prevents busy waiting.

```
void biryani_ready(Chef* self) {

    while (true) {
        if (self->left_vessels <= 0) {
            break;
        } else{
            pthread_cond_wait(&(self->cv_table), &(self->protect));
        }
    }
    self->left_vessels = 0;

    pthread_mutex_unlock(&(self->protect));

    printf(ANSI_CYAN "CHEF %d OUT OF BIRYANI\n" ANSI_DEFAULT, self->id);
}
```

Tables

```
void* table_run(void* args) {
    Table* self = (Table*)args;

    while(true) {
        // OVERALL SERVING CYCLE

        pthread_mutex_lock(&(self->protect));
        // just to protect state change from Foodies
        self->state = TABLE_ST_PREPARING;
        pthread_mutex_unlock(&(self->protect));

        printf(ANSI_RED "TABLE %d WAITING FOR BIRYANI\n" ANSI_DEFAULT, self->i
d);

        while(true) {
            // JUST GET A VESSEL
            int i = rand() % self->num_chefs;
            Chef* chef = self->chefs[i];
            pthread_mutex_lock(&(chef->protect));

            if (chef->left_vessels > 0) {
                chef->left_vessels--;
                self->left_vessel_cap = chef->vessel_cap;

                printf(ANSI_RED "TABLE %d GOT VESSEL OF CAPACITY %d FROM CHEF %
d\n" ANSI_DEFAULT, self->id, self->left_vessel_cap, chef->id);

                pthread_cond_signal(&(chef->cv_table));
                pthread_mutex_unlock(&(chef->protect));
                break;
            }

            pthread_cond_signal(&(chef->cv_table));
            pthread_mutex_unlock(&(chef->protect));
        }
```

```
        // NOW WE HAVE A VESSEL

        while (true) {
            // THIS IS THE SLOT GENERATION LOOP

            if (self->left_vessel_cap <= 0) { // NOBODY ELSE USES LEFT_VESSEL_C
APACITY
                self->left_vessel_cap = 0;
                break;
            }

            pthread_mutex_lock(&(self->protect));
            // just to protect state change and left_slots from Foodies
            self->state = TABLE_ST_INTERMEDIATE;

            self->total_slots = TABLE_SLOTS_OFFSET + rand() % TABLE_SLOTS_LIMI
T;
            if (self->total_slots > self->left_vessel_cap) {
                self->total_slots = self->left_vessel_cap;
            }

            self->left_slots = self->total_slots;

            self->left_vessel_cap = self->left_vessel_cap - self->total_slots;

            self->state = TABLE_ST_SERVING;

            printf(ANSI_RED "TABLE %d READY TO SERVE %d SLOTS. VESSEL CAPACITY
 REDUCED TO %d\n" ANSI_DEFAULT, self->id, self->left_slots, self->left_vessel_c
ap);

            ready_to_serve_table(self);

            printf(ANSI_RED "TABLE %d OUT OF SLOTS\n" ANSI_DEFAULT, self->id);
        }

        printf(ANSI_RED "TABLE %d OUT OF BIRYANI\n" ANSI_DEFAULT, self->id);
```

```
    }
}
```

The table thread first randomly looks over the chefs, looking for one with available vessels. After finding one, it acquires its mutex, decrements its number of available vessels and signals the chef.

Once the table has a vessel, it randomly generates some slots and then conditionally waits on those slots until students fill in by calling `ready_to_serve_table()`.

```
void ready_to_serve_table(Table* self) {
    while (true) {
        if (self->left_slots <= 0) {
            break;
        } else {
            pthread_cond_wait(&(self->cv_foodie), &(self->protect));
        }
    }

    self->left_slots = 0;

    self->state = TABLE_ST_INTERMEDIATE;
    pthread_mutex_unlock(&(self->protect));
}
```

If the table is out of slots, it proceeds to randomly generate more. If it is out of Biryani, it proceeds to get more from the chefs.

Students (aka Foodies)

```
void* foodie_run(void* args) {
    Foodie* self = (Foodie*)args;

    self->arrival_time = FOODIE_DELAY_OFFSET + rand() % FOODIE_DELAY_LIMIT;
    sleep(self->arrival_time);

    printf(ANSI_MAGENTA "STUDENT %d ARRIVED AND HUNGRY\n" ANSI_DEFAULT, self->i
```

```
d);

    wait_for_slot(self);
    student_in_slot(self);

    return NULL;
}
```

The student thread simply randomly looks over the tables, trying to find one where a slot is available. As soon as it finds one, it acquires its mutex, decreases the number of available slots and signals the table. The student is immediately served Biryani and leaves, as per the assignment requirements and clarifications.

```
void wait_for_slot(Foodie* self) {
    while (true) {
        int i = rand() % self->num_tables;
        Table* table = self->tables[i];
        pthread_mutex_lock(&(table->protect));

        if (table->state != TABLE_ST_SERVING) {
            pthread_cond_signal(&(table->cv_foodie));
            pthread_mutex_unlock(&(table->protect));
            continue;
        }

        if (table->left_slots <= 0) {
            pthread_cond_signal(&(table->cv_foodie));
            pthread_mutex_unlock(&(table->protect));
            continue;
        }

        table->left_slots--;
        int n = table->total_slots - table->left_slots;
        table->foodies[n - 1] = self;

        pthread_cond_signal(&(table->cv_foodie));
        pthread_mutex_unlock(&(table->protect));
```

```
        break;
    }
}

void student_in_slot(Foodie* self) {
    printf(ANSI_YELLOW "STUDENT %d ASSIGNED SLOT ON TABLE %d AND SERVED BIRYANI
\n" ANSI_GREEN "STUDENT %d FINISHED EATING\n" ANSI_DEFAULT, self->id, table->i
d, self->id);
    return;
}
```

Others

The simulation uses several utility functions etc. to run smoothly.

Conclusion

Thus this simulation demonstrates thread synchronisation using mutexes.