



Trabalho 1

Programação Concorrente

Alex Davis Neuwiem da Silva (21202103)



Paralelismo - Mutex

Foi usado um vetor de mutexes com o tamanho da esteira, sua função é impedir que os clientes e o chef interajam na mesma posição da esteira ao mesmo tempo. Ele também garante que a esteira não se mova enquanto alguém mexe nela.

Imagens a seguir: `conveyor_belt.c` e `customer.c` , respectivamente.

conveyor_belt_run:

```
/* O próximo "for" bloqueia todos os mutexes do vetor _pratos_mutex */
/* Dessa forma, a esteira só irá se mover assim que mais ninguém tiver acesso à região crítica */

/* Bloqueando os mutexes */
for (int i=0; i<self->_size; i++) {
    pthread_mutex_lock(&self->_pratos_mutex[i]);
}

/* Movendo a esteira */
int last = self->_food_slots[0];
for (int i=0; i<self->_size-1; i++) {
    self->_food_slots[i] = self->_food_slots[i+1];
}
self->_food_slots[self->_size-1] = last;

/* Desbloqueando os mutexes */
for (int i=0; i<self->_size; i++) {
    pthread_mutex_unlock(&self->_pratos_mutex[i]);
}
```

customer_run:

```
/* Como há um mutex em cada posição da esteira, os clientes podem pegar a comida paralelamente */
/* Esses mutexes também são disputados pelo sushi_chef e pela conveyor_belt */

pthread_mutex_lock(&conveyor_belt->pratos_mutex[self->_seat_position]);
if (conveyor_belt->food_slots[self->_seat_position] == i) {
    food = customer_pick_food(self->_seat_position);
    food_found = 1;
}
pthread_mutex_unlock(&conveyor_belt->pratos_mutex[self->_seat_position]);

if (food_found == 0) {
    pthread_mutex_lock(&conveyor_belt->pratos_mutex[self->_seat_position - 1]);
    if (conveyor_belt->food_slots[self->_seat_position - 1] == i) {
        food = customer_pick_food(self->_seat_position - 1);
        food_found = 1;
    }
    pthread_mutex_unlock(&conveyor_belt->pratos_mutex[self->_seat_position - 1]);
}

if (food_found == 0 && (self->_seat_position + 1) != conveyor_belt->size) {
    pthread_mutex_lock(&conveyor_belt->pratos_mutex[self->_seat_position + 1]);
    if (conveyor_belt->food_slots[self->_seat_position + 1] == i) {
        food = customer_pick_food(self->_seat_position + 1);
        food_found = 1;
    }
    pthread_mutex_unlock(&conveyor_belt->pratos_mutex[self->_seat_position + 1]);
}
```



Paralelismo - Semáforo

Usaram-se 3 semáforos:

- Um semáforo para impedir a espera ocupada do customer;

- Um semáforo para impedir a espera ocupada do hostess;

- Um semáforo para impedir a espera ocupada do chef.


Imagens a seguir: `customer.c` e `hostess.c` , respectivamente.

customer_run:

```
customer_t* self = (customer_t*) arg;

/* O semáforo abaixo é inicializado com 0 e apenas o hostess e a Queue podem fazer o "post" */
/* Assim, o cliente só continuará sua execução quando estiver sentado ou quando o restaurante fechar */

/* Semáforo para evitar espera ocupada do cliente */
sem_wait(&self->espera);
```




hostess_guide_first_in_line_customer_to_conveyor_seat:


```
conveyor_belt_t* conveyor = globals_get_conveyor_belt();
queue_t* queue = globals_get_queue();

customer_t* customer = queue_remove(queue);
conveyor->_seats[seat] = 1;
customer->_seat_position = seat;


/* Semáforo que impede a espera ocupada do cliente */
sem_post(&customer->espera);
```



hostess_check_for_a_free_conveyor_seat:

```
/* O semáforo a seguir é inicializado com a quantidade de assentos livres */  
/* Com isso, sempre que algum cliente se sentar, o hostess fará um "wait" */  
/* O cliente fará um "post" nesse semáforo assim que sair */  
  
/* Semáforo que impede a espera ocupada do hostess */  
sem_wait(&conveyor->vagas_sem);   
  
print_virtual_time(globals_get_virtual_clock());  
fprintf(stdout, GREEN "[INFO]" NO_COLOR " O Hostess está procurando por um assento livre...\n");  
print_conveyor_belt(conveyor);
```

customer_leave:

```
if (self->_seat_position != -1) {  
    conveyor_belt_t* conveyor_belt = globals_get_conveyor_belt();  
    conveyor_belt->_seats[self->_seat_position] = -1;  
  
    /* Semáforo que impede a espera ocupada do hostess */  
    sem_post(&conveyor_belt->vagas_sem);   
}  
  
customer_finalize(self);
```



Implementação do Fechamento do Sushi Shop


Utilizou-se uma variável global, conectada ao `virtual_clock` , que determina se o restaurante está aberto ou fechado.

Todas as funções de “run” funcionam apenas enquanto o restaurante estiver aberto. Quando o restaurante fecha, as funções “finalize” são ativadas.

Imagens a seguir: `virtual_clock.c` e `queue.c` , respectivamente.


virtual_clock_run:

```
void* virtual_clock_run(void* arg) {
    /* ESSA FUNÇÃO JÁ POSSUÍ A LÓGICA BÁSICA DE FUNCIONAMENTO DO RELÓGIO VIRTUAL */
    virtual_clock_t* self = (virtual_clock_t*) arg;
    while (TRUE) {
        if (self->current_time >= self->closing_time) {

            /* Indica o fechamento do restaurante */
            globals_set_sushi_shop_fechado(1); 

            print_virtual_time(self);
            fprintf(stdout, GREEN "[INFO]" RED " RESTAURANT IS CLOSED!!!\n");
            break;
        }
        self->current_time += 1;
        msleep(1000/self->clock_speed_multiplier);
    }
    pthread_exit(NULL);
}
```

queue_run:

```
void* queue_run(void *arg) {  
    /* NÃO PRECISA ALTERAR ESSA FUNÇÃO */  
    queue_t* self = (queue_t*) arg;  
    virtual_clock_t* clock = globals_get_virtual_clock();  
    while (globals_get_sushi_shop_fechado() == 0) {   
        customer_t* customer = customer_init();  
        queue_insert(self, customer);  
        print_virtual_time(clock);  
        fprintf(stdout, GREEN "[INFO]" NO_COLOR " Customer %d arrived at the Sushi Shop queue!\n",  
            print_queue(self);  
        msleep((rand() % 120000)/clock->clock_speed_multiplier);  
    }  
    pthread_exit(NULL);  
}
```



Dificuldades


- A função `queue_finalize()` não funcionava corretamente, causando um sinal de erro de “heap overflow”.
- Sincronização!!!

Imagem a seguir: `queue.c` .

queue_finalize:

```
void queue_finalize(queue_t* self) {
    /* NÃO PRECISA ALTERAR ESSA FUNÇÃO */
    virtual_clock_t* clock = globals_get_virtual_clock();
    struct queue_item *item = NULL;

    /* Esta função não desaloca os clientes diretamente, apenas faz um "post" no semáforo de espera ocupada */
    /* Com isso, o cliente que se desaloca sozinho enquanto a queue espera com um "sleep" */

    for (int i=0; i<self->_length; i=i+1) {
        item = self->_first;
        self->_first = self->_first->_next;
        sem_post(&item->_customer->espera); 
        msleep(50/clock->clock_speed_multiplier);
        free(item);
    }
    pthread_join(self->thread, NULL);
    free(self);
}
```



Observações

- O código sequencial é um pouco mais rápido que o paralelo, já que no paralelo a esteira tem que bloquear os mutexes de todas as posições.
- Os clientes mais próximos do chef se satisfazem mais rápido que os outros.
- O chef produz relativamente pouco para a quantidade de clientes.
- É possível lotar a esteira com um mesmo prato, sendo que ninguém come o alimento disponível.