UNIVERSIDADE DE BRASÍLIA

Programação Concorrente - 1/2024

PROJETO DE PROGRAMAÇÃO CONCORRENTE: FESTA DA NOBREZA

DATA: 01/09/2024

Autor:

ARTUR PADOVESI PIRATELLI – MAT. 211038208

Brasília – DF

1 Introdução

O presente projeto busca explorar problemas de concorrência e condições de corrida implementando o seguinte cenário:

Uma festa da nobreza está ocorrendo. O rei e seus nobres chegam para a festa com uma sequência de ações pré-determinadas. Mas que nobre pode negar o rei quando ele o chama para conversar? E que nobre pode continuar fazendo o que está fazendo quando o rei encerra a festa? Os nobres também podem interagir entre si.

Essa ideia leva à possibilidade de formalização de um problema interessante.

2 Formalização do Problema Proposto

O projeto segue as seguintes especificações:

1. Os nobres e o rei chegam com uma lista pré-determinada de ações a serem feitas. O rei deve ter uma thread própria, e cada nobre também deve ter sua própria thread;

2. O rei tem as seguintes ações/estados:

- Falar com um nobre

    - Ele pode convocar qualquer nobre ***M***;

    - Se ele não especificar um nobre (***M = -1***), ele escolhe o próximo nobre esperando na fila;

    - Se ele não especificar um nobre e não houver nobres na fila, ele convoca um nobre aleatório;

    - Os nobres têm no máximo 1 segundo para se apresentar;

- Inativo (Idle)

    - O rei está simplesmente inativo, sem fazer nada;

- Encerrar baile

- Deve encerrar o programa;

3. Um nobre tem as seguintes ações/estados e reações:

- Falar com o rei

  - Ele entra em uma fila para falar com o rei;

  - Se o rei o selecionou para falar, ele se apresenta ao rei;

  - Aguarda a dispensa do rei;

  - Reações:

    - Se estiver na fila e o Rei Encerrar Baile, então Nobre Sai do Baile
    - Se estiver na fila e Rei Convocar Para Falar, então sair da fila e se apresentar ao rei;

- Falar com outro nobre

  - Ele pode tentar falar com o nobre ***M***;

  - Ele pode esperar até ***T*** segundos para o nobre ***M*** responder;

  - Eles conversam por ***X*** segundos;

  - Reações:

    * Se Rei Encerrar Baile, então Nobre Sai do Baile

    * Se Rei Convocar Para Conversar um dos nobres, ambos abandonam a ação, o convocado vai falar com o rei e o outro deve ir para a próxima ação;

- Inativo

  - O nobre está sem fazer nada por uma determinada duração;

  - Reações:

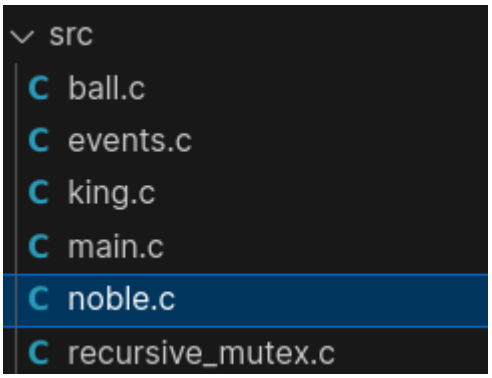
    - Se Rei Convocar Para Conversar, então Nobre deve Falar com o Rei;
    - Se Rei Encerrar Baile, então Nobre Sai do Baile;

        - Se outro Nobre deseja falar com este, então deve Falar com o outro Nobre;
- Encerrar baile

    - O nobre deve deixar o baile;

3 Descrição do Algoritmo Desenvolvido para a Solução do Problema

O problema foi resolvido em partes, cada parte com seu sub-problema. Ainda assim foram utilizadas algumas construções por todo o projeto. Observando os arquivos:



É possível notar a existência de um arquivo ***events.c***. Esse arquivo contém uma estrutura utilizada amplamente por todo o projeto para evitar condições de corrida e permitir comunicação entre as threads de maneira estruturada.

A ideia veio do JavaScript, com uma programação baseada em eventos que permite uma flexibilidade e separação de responsabilidades muito boa.

Para se utilizar de eventos é necessário criar uma thread própria para tal, e ela será a responsável por distribuir os eventos recebidos. É possível registrar uma função ***listener*** com o ***register_event_listener***, que receberá todos os eventos por alguém emitidos. É possível emitir um evento com ***emit_event***. Por fim, é possível utilizar de eventos como forma de lidar com condições de corrida utilizando duas funções em conjunto: ***wait_for_event(setup_wait_for_event(...))***.

A separação em duas funções para receber um evento permite evitar condições de corrida, como no seguinte exemplo:

```
src > C noble.c > noble_waiting_to_talk_to_noble_evaluator(event)
172   void * noble_routine(void* arg) {
258           enter_talk_to_king_queue(actions_list->id);
259
260           // @important: set this up **BEFORE** the while() to prevent race conditions
261           wait_for_event_evaluated_params *setup_wait = setup_wait_for_event_evaluation(
262             &noble_talk_to_king_event_listener_evaluator,
263             &this_noble_idle_event_listener_evaluator_params
264           );
265
266           pthread_mutex_lock(&ball_is_over_info.mutex);
267
268           if (ball_is_over_info.is_over) {
269             action->type = NOBLE_END_BALL;
270             pthread_mutex_unlock(&ball_is_over_info.mutex);
271             continue; // goto next action immediately
272           }
273
274           pthread_mutex_unlock(&ball_is_over_info.mutex);
275
276           pthread_mutex_lock(&talk_to_king_queue_info.mutex);
277
278           if (talk_to_king_queue_info.king_called_for != actions_list->id) {
279             pthread_mutex_unlock(&talk_to_king_queue_info.mutex);
280
281             wait_for_event_result* result = wait_for_event(setup_wait);
```

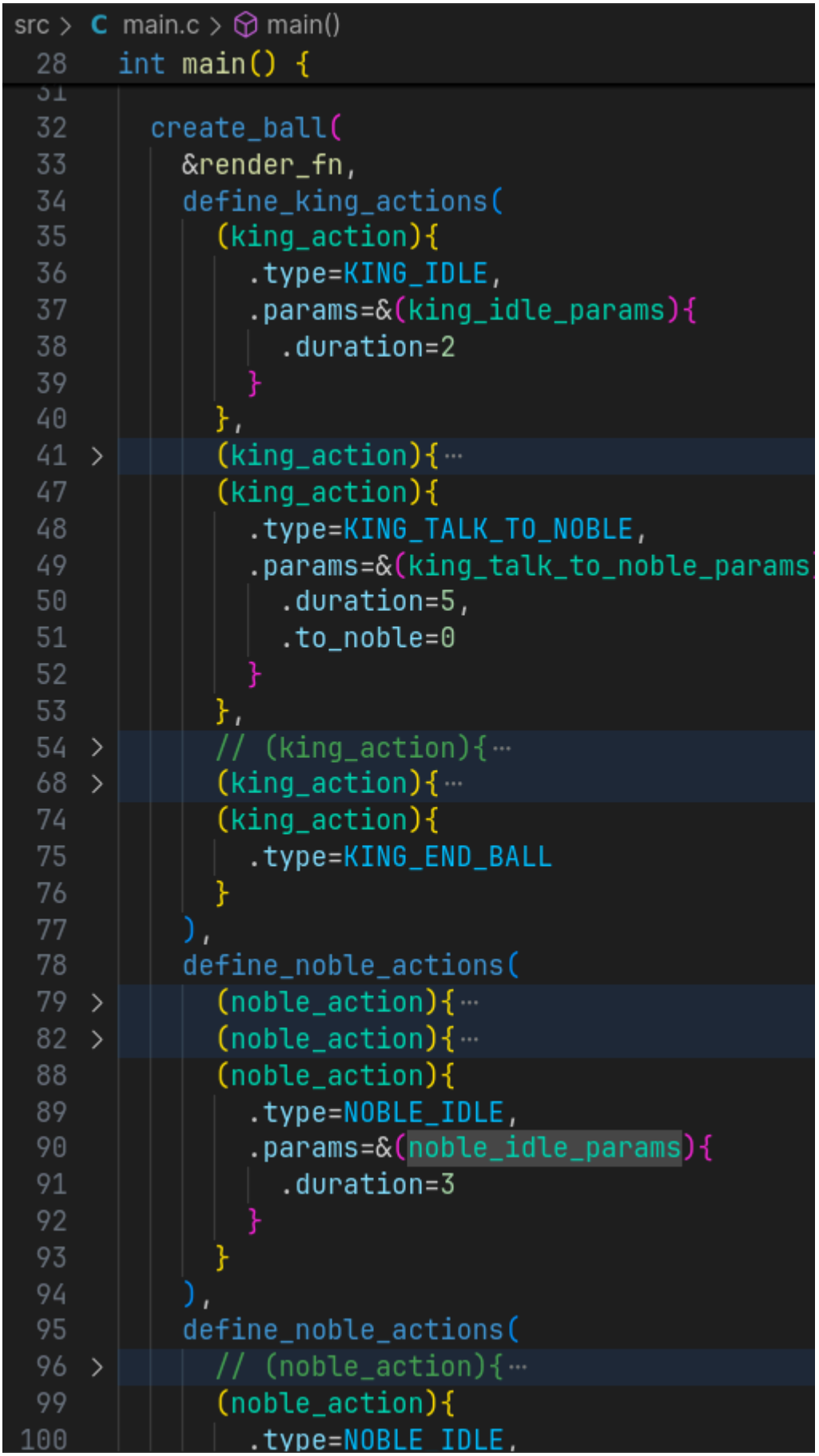
Onde a função utilizada como *event listener* é a seguinte:

```
87  int noble_talk_to_king_event_listener_evaluator(event ev) {
88    if (ev.type & KING_EMITTED_BALL_IS_OVER) return 1;
89
90    if (ev.type & KING_EMITTED_NEXT_ON_QUEUE) {
91      noble_idle_event_listener_evaluator_params* params = ((wait_for_event_evaluated_params*)ev.curried_params)->evaluator_curried_params;
92      pthread_mutex_lock(&talk_to_king_queue_info.mutex);
93      if (talk_to_king_queue_info.king_called_for == params->noble_id) {
94        pthread_mutex_unlock(&talk_to_king_queue_info.mutex);
95        return 1;
96      }
97      pthread_mutex_unlock(&talk_to_king_queue_info.mutex);
98    }
99
100   return 0;
101 }
```

É possível ver neste mesmo código que existem estruturas de acesso compartilhadas entre o Rei e os Nobres, que estão sendo regiões de exclusão mútua devido aos *mutexes* existentes.

Foi considerada a utilização do algoritmo de leitores e escritores nesses casos, onde vários leitores podem ler ao mesmo tempo, mas o overhead dessa implementação não valeria nada a pena.

Além disso, foi criada uma interface declarativa para inicializar os nobres e o rei, incluindo uma função de renderização a ser utilizada como desejado:


src > C main.c > main()
int main() {
create_ball(
&render_fn,
define_king_actions(
(king_action){
.type=KING_IDLE,
.params=&(king_idle_params){
.duration=2
}
},
(king_action){
.type=KING_TALK_TO_NOBLE,
.params=&(king_talk_to_noble_params
.duration=5,
.to_noble=0
}
},
// (king_action){
(king_action){
(king_action){
.type=KING_END_BALL
}
),
define_noble_actions(
(noble_action){
(noble_action){
(noble_action){
.type=NOBLE_IDLE,
.params=&(noble_idle_params){
.duration=3
}
}
),
define_noble_actions(
// (noble_action){
(noble_action){
.type=NOBLE_IDLE,

4 Conclusão

O problema foi solucionado de uma maneira incrivelmente interessante.

É importante notar que a utilização de eventos se mostrou necessário, visto que em certas partes, para evitar condições de corrida, usavam-se dois locks, e a utilização de ***pthread_cond_t*** não bastava.

Sendo assim, utilizar semáforos e uma thread de eventos para contornar o tipo de situação citada é uma maneira bem elegante de resolver o problema.

5 Referências

Sem referências.
Código em: https://github.com/artistrea/concurrency-project