Programação Concorrente
Implementação de um controlador de elevadores com *threads*

Oseias Romeiro Magalhães / 211036123
Prof. Eduardo Adilio Pelinson Alcheri

Junho de 2023

Resumo

Neste trabalho de Programação Concorrente, foi desenvolvido um controlador para lidar com as chamadas em cada andar de um prédio. Baseando-se no problema clássico do produtor-consumidor abordado em sala de aula [1].

1 Introdução

Um controlador de elevadores será abordado neste relatório como um exemplo prático de concorrência entre processos com memória compartilhada. O controlador deve receber chamadas vindo de vários andares e "desenhar"o melhor trajeto para o elevador seguir, no qual foi utilizado o algoritmo SCAN para esta tarefa. Mais detalhes será abordado na seção (4) de descrição.

2 Ambiente

Para o desenvolvimento desse projeto, foi utilizado a linguagem C em ambiente Linux e compilado com GCC na versão 12.2. Utilizando as bibliotecas *POSIX Pthreads* para utilização de estruturas de concorrências.

3 Formalização do problema

O problema aborda o cenário onde vários usuários precisam se locomover por andares de um prédio de forma rápida. Para isso, existe um ou mais elevadores para atender as requisições em uma determinada quantidade de andares. Cada requisição é única, ou seja, só pode existir uma requisição por andar. Quando já existir uma requisição para todos os

andares, qualquer outra requisição deve esperar até poder fazer sua chamada. Os elevadores também ficam em espera, caso não haja requisições para serem atendidas. Para lidar com o melhor trajeto que o elevador deva seguir, existe um decisor que decide o trajeto por meio de algum algoritmo de *scheduler*.

4 Descrição

O problema foi dividido em 3 (três) funções principais: *elevator*, *callsHandler* e *decider*. Onde cada um é responsável por uma parte importante da implementação, contendo saídas de texto no terminal para acompanhar o processo.

```
void* elevator(void* arg);
void* callsHandler(void* arg);
void* decider(void* arg);
void SCAN(int head, int direction);
```

Figura 1: Funções

Foi utilizado dois bufferes: *callBuffer* e *trackBuffer*, implementando assim dois cenários de produtor-consumidor utilizando *mutex* e e variáveis de condições [2]. Ambos bufferes são incicializados na **main** com o número de andares (valor impossível).

Além disso, existe as seguintes declarativas que configuram o cenário do problema:

```
#define ELEVATORS 2  // numero de elevadores
#define CALLERS   5  // numero de chamadores
#define FLOORS    20 // quantidade de andares
```

Figura 2: Declarativas

4.1 Controle de chamadas

A função *callsHandler* é responsável por geração de chamadas forma randômica, garantindo também que cada chamada seja única, ou seja, não tenha mais de uma chamada por andar. Respaldo no cenário de que o elevador vai atender o andar independente da quantidade de usuários lá esperando, caso haja um número que exceda a quantidade permitida, os usuário devem se dividir. De qualquer forma, este cenário não é o foco do problema e nem é abordado pelo programa (também não sei por que o citei).

Cada chamada é gerada em uma thread e colocado em *callBuffer*. Quando o buffer chegar no limite, as threads entram em espera. Como mencionado, as chamadas são únicas, logo o *callBuffer* é limitado por NUM_FLOORS (Figura 2). Caso, seja a primeira a adicionar uma chamada no buffer, então acorda o decisor (4.3), que deve estar esperando por novas chamadas.

4.2 Elevadores

A função *elevators* é responsável por consumir a chamadas em *trackBuffer*, ou seja, atender as solicitações que o decisor (4.3) o passou, sendo consumido como uma fila. As regras no tocante ao limite do buffer e a lógica de concorrência é semelhante ao *callBuffer* explicado na seção anterior.

4.3 Decisor

O decisor tem como principal função, consumir as chamadas em *callBuffer*, calcular a melhor rota para as chamadas até o momento e repassar para o *trackBuffer* que será consumido pelos elevadores. Para traçar a melhor rota que o elevador deve assumir, foi utilizado o algoritmo de SCAN [3] que considera a posição do elevador e o sentido, subindo ou descendo. O sentido é alternado a cada vez que o decisor executa e quanto a posição, é considerado a do primeiro elevador. O decisor, fica em estado de espera quando o o buffer está vazio, e libera o *mutex* de chamada assim que a rota é calculada.

4.4 SCAN

O algoritmo usado no decisor (4.3) para "desenhar"a melhor rota é o SCAN. Algoritmo de escalonamento que consegue encontrar uma rota com menor custo de deslocamento e tempo ocioso dos passageiros, com base na posição do elevador e o sentido de prioridade (descida/subida), ordenando e dividindo as chamadas em duas sub listas (andares abaixo e acima da posição do elevador), criando um trajeto para percorrer as duas sub listas da melhor forma. O algoritmo foi traduzido para C e adaptado para a o problema, com uso dos bufferes que são variáveis globais, sendo assim, acessível pela função SCAN (Figura 1).

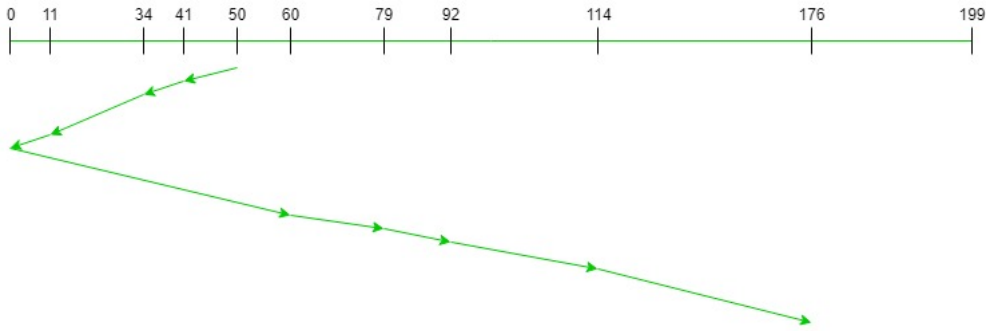

Figura 3: SCAN

5 Fluxo de execução

Assim que executado o programa, os elevadores e o decisor entrará em espera enquanto é feita as chamadas, então o decisor é acordado, realizando as operações necessárias e assim, liberando os chamadores e acordando os elevadores. O que é mostrado na tela do terminal por meios de *prints*. A quantidade de elevadores, andares e usuaraios que fazem a chamada podem ser editados pelas declarativas mostradas na Figura 2.

6 Conclusão

Portanto, foi desenvolvido assim, um programa que aborda um problema prático envolvendo comunicação entre processos através de memória compartilhada. Assim como requisitado para o trabalho final da disciplina.

Referências

[1] Eduardo Adilio Pelinson Alcheri. *CIC0202 - PROGRAMAÇÃO CONCORRENTE - Turma 01 - 2023/1*. `https://aprender3.unb.br/course/view.php?id=18786`.

[2] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming 2ed*. 2005.

[3] sid779. *SCAN (Elevator) Disk Scheduling Algorithms*. `https://www.geeksforgeeks.org/scan-elevator-disk-scheduling-algorithms/`.