Escola de Matemática Aplicada
Fundação Getúlio Vargas

Trabalho 1 - Computação Escalável

Dávila de Carvalho Meireles
Fernanda Luísa Silva Gomes
Igor Patricio Michels
Laura Chaves Miranda
Marcos Antônio Alves
Professor: Thiago Pinheiro de Araújo

24 de abril de 2023

1 Modelagem do problema

1.1 Mock: sistema de monitoramento de rodovias

Os dados do sistema de monitoramento de rodovias são gerados por um script Python. A cada iteração, é gerado um arquivo por rodovia. O nome do arquivo é o instante em que ele foi criado. A primeira, segunda e terceira linha do arquivo correspondem a rodovia, a velocidade máxima permitida na rodovia e a velocidade máxima possível do carro, respectivamente. A última linha contem o horário em que o arquivo foi finalizado, representando o momento de emissão. As demais linhas, apresentam uma string representando a placa do veículo e uma tupla com dois elementos,

sendo esses a localização no eixo X e a localização no eixo Y. Por simplificação, cada rodovia é representada por uma matriz, sendo cada entrada uma possível localização dos carros presentes na rodovia.

O mock define a placa de um carro aleatoriamente, seguindo o padrão Mercosul. Além disso, para cada placa gerada, armazena-se no arquivo *extraInfoCars.txt* a placa e as demais informações do carro, também geradas aleatoriamente. Esse arquivo será utilizado, pelo sistema externo, para consulta.

Por fim, outra escolha de modelagem para o mock da rodovia foi, no sistema de colisão, carros que "saltam" sobre outros, isso é, numa iteração eles permanecem na mesma pista, tendo posição inicial anterior a um outro carro que também permaneceu naquela pista e posição final posterior também colidiu.

1.2 Serviço externo

O serviço externo fornece informações adicionais sobre o carro, sendo elas obtidas no arquivo *extraInfoCars.txt*. Na implementação, criou-se uma *struct* plateData e uma classe externalAPI. A *struct* plateData contém o modelo, o proprietário e o ano do carro, obtidos pela classe. A classe externalAPI possui o arquivo a ser consultado, o tamanho máximo e atual da fila para requisições, a fila, a última placa avaliada, um booleano, indicando se a placa entrará na fila, e um registro de plateData, vinculado a placa. Além disso, possui um semáforo para auxiliar no fluxo do programa. A seguir estão os principais métodos da classe e uma breve explicação sobre cada um.

- query_next_plate(): Atualiza o registro de plateData com as informações da próxima placa que está na fila;
- query_vehicle(plate): A placa é adicionada na fila de requisições se a fila não estiver com capacidade máxima e se a placa ainda não está na fila.
- get_name(): Retorna a última placa avaliada e o proprietário do carro, se disponíveis. Caso contrário, executa o método query_next_plate() e, finalmente, retorna a placa e o proprietário;
- get_model(): Replica a ideia do método get_name(), mas com o modelo do carro;
- get_year(): Replica a ideia do método get_name(), mas com o ano do carro.

1.3 Pipeline ETL

O nome dos arquivos gerados pelo sistema de monitoramento é o horário de criação dos mesmos. Como é desejado processar primeiro os dados mais antigos (caso contrário perderíamos as informações mais recentes), os arquivos são ordenados crescentemente pelo horário e o mais antigo é lido primeiro.

Para armazenar os dados, foram criadas, além da *struct* plateData, duas outras *structs*, highwayData e carData. Em highwayData, têm-se a velocidade máxima da via, a velocidade máxima do carro, o horário de emissão do dado e um mutex. Já em carData, têm-se a pista, as três últimas posições, se o carro está na via, velocidade, aceleração, se possui risco de colisão. Além disso, existe uma variável auxiliar para informar se possuímos as informações extras do veículo. No código abaixo, está a definição dessas *structs*.

```
struct plateData
{
    string model;
    string name;
    int year;
};

struct highwayData {
    int maxSpeed = 0;
    int carMaxSpeed = 0;
    string infoTime = "";
    mutex highwayDataBlocker;
};

struct carData {
    int lane = 0;
    int actualPosition = 0;
    int lastPosition = 0;
    int penultimatePosition = 0;
    bool isInHighway = true;
    int speed = 0;
```

```
    int acceleration = 0;
    bool canCrash = false;
    bool extraInfos = false;
};
```

As conexões entre rodovias e carros foram elaboradas via *maps*. Nesse sentido, temos um *map* com o número da rodovia levando a *struct* highwayData correspondente. Além desse, existe outro *map*, também chaveado pelo número da rodovia, que leva a um *map* chaveado pelas placas dos veículos, onde são armazenadas as informações do mesmo dentro da *struct* carData. A escolha pela criação desses dois maps foi puramente por simplicidade de acesso. Por fim, criou-se um terceiro *map* global. Este último é chaveado pela placa do veículo e armazena *structs* plateData com as informações extras do respectivo carro.

Os transformadores, responsáveis pelas análises, são funções *void* que imprimem na tela os resultados. Cada transformador é executado em uma thread. Além dos resultados, é informado o tempo entre a leitura do dado e a impressão na tela.

2 Decisão de projeto

- Para realizar as análises, são necessárias a posição, a velocidade e aceleração do carro. Como essas informações são obtidas pelas três últimas posições ocupadas por um carro, posições anteriores são descartadas;
- Devido a avaliação de eficiência, apenas uma thread é responsável pela leitura e atualização dos dados gerados pelo mock;
- Com o intuito de não gerar um mutex por carro, optou-se pela criação de um mutex por rodovia. Assim, não é possível atualizar simultaneamente as informações de dois ou mais carros presentes em uma rodovia, uma vez que a mesma é travada sempre que formos ler ou acessar a informação de um carro;
- Cada thread é inicializada com um transformador. Assim, a prioridade das análises é definida pela prioridade das threads, por meio do módulo native_handle();
- Sobre a priorização, definimos que as funções de leitura, atualização de velocidade e análise de colisão recebem prioridade máxima. Análises secundárias como número de rodovias e veículos

ou veículos acima do limite de velocidade recebem baixa prioridade. Por fim, como os dados da API externa são gerados por um sistema legado, sendo opcionais sua posse, os mesmos recebem baixa prioridade, junto a função de impressão da informação, a qual gera muito I/O e, com isso, um menor desempenho.

3 Problemas de concorrência

A seguir estão listados os problemas de concorrência observados e as soluções propostas.

- Se o mock (python) cria e atualiza apenas um arquivo com todas as iterações, teríamos problema de dois programas tentarem acessar ao mesmo tempo o arquivo. Para contornar esse aspecto, cada arquivo armazena uma iteração de uma rodovia, liberando o mesmo assim que concluir sua escrita;

- A paralelização da leitura dos arquivos gera um problema de concorrência. Porém, após alguns testes, percebeu-se que a leitura dos arquivos em uma thread é mais eficiente, uma vez que os arquivos não são grandes. Logo, a vantagem que teríamos em utilizar mais de uma thread para a escrita é perdida com o split dos dados e com a excessiva troca de contexto, uma vez que, para atualizar as informações de um carro a rodovia toda é bloqueada. Tendo em vista isso, optou-se pela não paralelização;

- O serviço externo é capaz de processar uma única requisição por vez. Com o objetivo de satisfazer essa exigência, utilizou-se um semáforo. O semáforo impede que duas placas sejam processadas simultaneamente, além de liberar o processamento da próxima placa apenas após o consumo da informação atual;

- No geral, os transformadores realizam leitura, análise ou escrita sobre os mesmos *maps* e *structs*. Para garantir que o programa reproduza o resultado desejado, sempre que um transformador entra em uma região em que haverá algum acesso supracitado é dado um lock no mutex da rodovia correspondente;

- Além dos *maps* acima, com as informações das rodovias, o *map* com as informações extras dos carros também sofre de problema de concorrência, com acessos simultâneos entre a função que recebe os dados da API e a função que irá imprimir a informação em tela. Nesse sentido, também criou-se um mutex para coordenar tais acessos.